Completeness in Languages for Attribute-Based Access Control

by

Conrad Eason Williams

A thesis submitted in fulfilment of the requirements for the Degree of
Doctor of Philosophy

August 2017
Information Security Group
Royal Holloway, University of London

Declaration of Authorship

I, Conrad Eason Williams, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed: _____

Date: _____

Abstract

Access control restricts the interactions that are possible between users (or programs operating under the control of users) and sensitive resources, and is an essential component of any security architecture in multi-user computing systems. The most common means of implementing access control is to define an *authorization policy*, specifying which *requests* (that is, attempted user-resource interactions) are authorized and can thus be allowed. In recent years, we have seen the emergence of *attribute-based* access control (ABAC), in part to cater for open, distributed computing environments where it is not necessarily possible to authenticate all entities directly. The primary goal of this thesis is to improve the understanding and specification of ABAC languages.

Our approach focuses on the connection between *multi-valued logics* (MVLs) and many ABAC languages present in the literature. We introduce the necessary theoretical foundations to analyse and reason about various properties of ABAC languages. This enables us to show that XACML, the predominant language for authoring ABAC policies, exhibits a number of shortcomings. We present extensions to the ABAC language PTaCL, and demonstrate how it may be modified to address the shortcomings identified in XACML. Later, we extend our foundations to lattice-based logics and languages, establishing new results about Belnap logic and its associated ABAC languages.

Another major difficulty encountered in many ABAC languages is how to construct a desired policy using the operators defined in the given language. Even in languages that are known to be *functionally complete*, this is in general a non-trivial task. We present a novel solution to this problem: specifying policies in a tabular form. We demonstrate why representing policies in this manner is convenient, intuitive and flexible for policy authors, and provide a method for automatically compiling policy tables into a form that is machine-enforceable.

To my grandparents
Ray and Patricia Bowler.
Words cannot describe the appreciation
I have of your continual love, support and
encouragement, without which, this thesis
would never have been written.

Contents

List of Figures

List of Tables

Acknowledgements

I must first thank my PhD supervisor, Professor Jason Crampton, for expressing an interest in taking charge of my supervision, and for persuading me to reconsider my initial rejection. The guidance, support, wisdom and attention to detail that Jason has shared throughout the duration of this thesis has been instrumental in the improvement of my ability to reason, think and write about the exciting research that I have been involved in. The past four years have been both challenging and rewarding, and filled with a great deal of personal growth and development, to which I owe Jason a huge amount of thanks for the role he played as my mentor. The quality of my work has been constantly refined by the input of Jason, through countless hours of proof reading, and I'm sure, many red pens. I am grateful for the patience he showed as I slowly found my feet and developed my writing capabilities.

In addition, I would like to thank my parents Beth and Derek Williams for their help in proof reading and comments on this thesis. I am extremely fortunate to have a loving and supportive family, who take their belief in my abilities to a somewhat absurd level. For them, there was never any doubt that I was capable of completing a PhD, the four years were just a formality required to become Dr Williams. This belief and encouragement played a fundamental role in my success and for that, I am eternally thankful. Particular thanks goes to my grandfather Ray Bowler, for his compelling arguments that convinced me to pursue the opportunity to study towards a PhD.

I would like to thank Erik Rissanen for hosting me at Axiomatics. The time I spent at Axiomatics provided me with a fresh, practical perspective on the research I was conducting, and led to a number of novel contributions presented in this thesis. I would also like to thank EPSRC for the funding and support given to me during my time as a member of the Center for Doctoral Training in Cyber Security at Royal Holloway.

Finally, I would like to thank all the academic and support staff at Royal Holloway for their stimulating conversations and assistance.

Chapter 1

Introduction

In multi-user computing systems, the security of information and resources is an essential component of any security architecture. As a result, a field of research has grown around the design and implementation of methods to secure information in a multi-user setting. This field of research is commonly known in the community as *access control*. Access control then, restricts the interactions that are possible between users (or programs operating under the control of users) and sensitive resources.

The most common means of implementing access control is to define an *authorization policy*, specifying which *requests* (that is, attempted user-resource interactions) are authorized and can thus be allowed. In a typical implementation, all requests are intercepted and evaluated with respect to the policy by trusted software components, often known as the *policy enforcement point* and *policy decision point*, respectively.

Thus, in general terms, an authorization policy may be viewed as a function $P : Q \to D$, where Q is the set of requests and D is the set of authorization decisions. We assume 0 and 1 belong to D, representing the "deny" and "allow" decisions, respectively. Traditionally, Q was modelled as a set of triples of the form (s, o, a), where s is a subject, o is an object, and a is an action: a subject represents an authenticated entity, an object represents a protected resource, and an action is the means by which the subject wishes to interact with the object [6, 8, 27, 41, 43].

In recent years, we have seen the emergence of *attribute-based* access control (ABAC), in part to cater for open, distributed computing environments where it is not necessarily possible to authenticate all entities directly [11, 12, 29, 37, 39]. (Opposed to the more traditional "closed", centralised systems in which the set of users was assumed to be known in advance.) Subjects and objects are associated with attributes, requests are collections of attributes associated with the subjects and objects, and these attributes determine whether a request is authorized or not through evaluation of the authorization policy, which is defined in terms of user and resource attributes.

The main goal of this thesis is to improve the understanding and specifi-

cation of ABAC languages. We demonstrate that *multi-valued logics* (MVLs) are the fundamental "building blocks" of many ABAC languages, and through the careful choice of multi-valued logic, desirable properties can be guaranteed for new ABAC languages. We thereby provide a theoretical framework for the development of new ABAC languages, which eliminate the disadvantages of existing ABAC languages.

1.1 Attribute-based access control

In the context of attribute-based access control (ABAC), we assume there exists a set of attributes, each of which can take a range of values. An authorization request is specified in terms of attribute name-value pairs, associating the subject and object with the relevant attributes. Given a set of requests, an ABAC policy specifies whether each request is authorized or not.

Much of the research on ABAC policies assumes that policies are constructed from sub-policies. One sub-policy might, for example, specify that some subset of requests is allowed, while another sub-policy specifies that some subset of requests is denied. Defining policies in this way inevitably means that the sub-policies may "clash", so research in this area has focused on ways of resolving the conflicts that may arise when combining policies [9, 34, 37].

There are two broad approaches, which we may label as "policy algebras" [9, 34, 40, 46] and "tree-structured languages" [11, 12, 29, 37]. A policy algebra defines the semantics of a policy in terms of the sets of requests it allows and denies. Then sub-policies are combined by defining policy operators that are defined in terms of set operations (such as intersection, union and set difference) on the sets of allowed and denied requests. In contrast, a tree-structured language defines what decision to return for each sub-policy and then combines the decisions arising from the evaluation of sub-policies using decision-combining algorithms (or decision operators).

Of course, there are strong parallels between the two approaches, and it is often possible to define exact correspondences between policy operators and decision-combining algorithms. Nevertheless, the popularity and widespread use of XACML [37] has led to more research on tree-structured languages in recent years (in comparison to policy algebras) [11, 12, 29, 37]. As a result, in this thesis we restrict our attention to tree-structured languages, and the literature surrounding these languages. We formally define how we classify ABAC languages as tree-structured in the preliminaries (Chapter 2), and introduce a number of tree-structured languages from the literature, including XACML [37], PTaCL [12] and PBel [10].

Despite the extensive research on tree-structured languages [11, 12, 29, 37], we have identified three general issues with existing tree-structured languages. First, we explore the issue of *expressivity*. Informally, we gauge the expressive power of a language by the number of different policies that may be constructed

in it. We say a language is *functionally complete* if any arbitrary policy may be constructed in the language. It is known that PTaCL [12] and PBel [10] are functionally complete languages, and one of the contributions of this thesis is to show that XACML [37] is not functionally complete. We believe that the inability to express desired policies is a major drawback, as policy authors are forced to either approximate the desired policy, or introduce custom combining algorithms. Approximation of policies is clearly undesirable, as unintended scenarios may be introduced. While XACML supports the specification of custom combining algorithms, there is little to no support or guidance given on how custom combining algorithms should be defined and used, and we believe allowing policy authors unhindered freedom to define their own custom combining algorithms in the native XACML framework is risky. For instance, rigorous testing of any custom combining algorithms should be carried out, to ensure these algorithm behave as intended. Both scenarios may lead to policy misconfigurations, and ultimately, unauthorized access.

Secondly, we identify a problem that is inherent to the nature of tree-structured languages. Policies in tree-structured languages are expressed in a tree-like structure, where leaf nodes are attribute-decision pairs and interior nodes are attribute-operator pairs.[1] Essentially this means policies must be constructed in a bottom-up fashion. In particular, if it is not possible to express a policy using a single target and decision, the policy author must engineer the desired policy by combining sub-policies using the set of operators specified in the given language. This is a non-trivial task in general and makes policy specification in tree-structured ABAC languages a challenging task, and it can be difficult to anticipate how a policy will evaluate for all access requests (due to the complex structure of policies).

In addition, this issue is further exasperated by the lack of a universal or standard method for constructing policies in tree-structured languages such as XACML. This means that, in practice, if two policy authors are given a policy expressed in natural language and are asked to construct the policy in XACML, it is possible that they will construct two different policies.

In this thesis we propose an alternative method for policy specification, through specifying policies in tables indexed by sub-policies (and later, attributes). We demonstrate that policies expressed in this manner can be automatically converted into a *normal form*, which is machine-enforceable. In order to achieve this, we first note a similar method employed in traditional 2-valued propositional logic. One can use the truth table for an arbitrary Boolean function to write down a logically equivalent formula in disjunctive normal form. Another contribution of this thesis is the generalisation of this method to *multi-valued logics*, and its application in generating policies for ABAC languages.

[1] This is something of a simplification, but a good approximation of how such policies are structured.

A further shortcoming of existing work on languages for attribute-based access control is the way in which requests and attributes are matched via targets. Suppose we have an attribute name-value pair (n, v) and a request that contains multiple name-value pairs, including (n, v) and (n, v'), where $v' \neq v$. Then one might argue the request matches the attribute (since it contains (n, v)); on the other hand, one might argue it doesn't match the attribute (since it also contains (n, v')). XACML always assumes the former interpretation, which may be inappropriate if, for example, the policy author wishes to insist that the request contains exactly one name-value pair for the named attribute. Although PTaCL has a slightly more complex match semantics for requests and attributes, it ignores several possible match semantics that might be relevant in practice.

Furthermore, existing ABAC languages essentially have two layers of abstraction, in the form of targets and policies. The former must be evaluated first, to determine the applicability of policies, which are then evaluated. We believe that this separation is unnecessary, and can over-complicate policy specification. Chapter 7 explores this issue in more depth, where we propose an alternative method for matching requests and attributes, and directly specify policies in terms of attributes, removing the need for targets.

1.2 Multi-valued logics

The underlying foundations of this thesis are built on the study of *multi-valued logics*. In the past, logic systems contained only two truth values "true" and "false", which correspond to our intuitive understanding of truth and falsity. However, these systems rapidly expanded to incorporate more values, used to represent values such as "unknown", "lack of information" and "conflict" [7, 26, 38]. The truth values introduced in multi-valued logics often have intuitive interpretations in access control, the "unknown" value may be interpreted as "not-applicable" in the context of policies [10, 12]. Likewise, "conflict" can be used to represent conflicting access control decisions in a policy [10]. As a result, many tree-structured ABAC languages are based on well-known multi-valued logics, and use operators from these logics as policy operators [10, 12, 32, 45]. We shall show that, in tree-structured languages, policies are essentially terms in a logic-based formalism.

Certain tree-structured ABAC languages such as PTaCL [12] and PBel [10] are known to be functionally complete, and the proofs of their functional completeness rely on the underlying functional completeness of the multi-valued logic on which they are based. In other words, there is a strong relationship between the properties of an ABAC language and the multi-valued logic on which it is built.

In this thesis we expand upon this relationship, by introducing the theoretical foundations, based on results of Jobe [24], for characterising properties

of multi-valued logics. We define concepts such as *canonical suitability, selection operators, normal form* and *canonical completeness* for multi-valued logics, and show the application of these concepts to ABAC languages. We demonstrate in Chapter 4 the value of canonical completeness in ABAC languages, and how this helps us to overcome the challenges discussed earlier for tree-structured languages.

The results of Jobe are limited to logics in which the truth values are assumed to be totally ordered; we call logics of this type *total-ordered logics*. However, there are many partially ordered logics of interest, such as Belnap logic [7], on which a number of tree-structured languages are based. We extend Jobe's results to lattice-based multi-valued logics in Chapter 5, and prove that Belnap logic is not canonically complete. We then construct a 4-valued lattice-based multi-valued logic that is canonically complete, and demonstrate its use as the underlying logic for an ABAC language in Chapter 6.

1.3 Outline of thesis

In Chapter 2 we introduce some prerequisite material on tree-structured attribute-based access control (ABAC) languages and multi-valued logics (MVLs). In Section 2.1 we formally define the generic structure and evaluation semantics for targets and policies in tree-structured languages. We also introduce the notions of obligations and indeterminacy, which we explore in more detail in Chapters 3 and 4. Section 2.2 introduces the OASIS standard language XACML [37], which is referenced and used as a comparative language throughout this thesis. We then formally define PTaCL [12] in Section 2.3, which we will analyse and adapt extensively in this thesis. We conclude the chapter with an introduction to multi-valued logics, highlighting why they are fundamental in establishing properties of many ABAC languages. We formally define various properties of multi-valued logics such as *functional completeness*, *canonical suitability*, *normal form* and *canonical completeness*, based on work by Jobe [24]. These properties will play a crucial role in this thesis, and in the development of ABAC languages with desirable properties.

Chapter 3 is a comprehensive review of XACML. We summarise the historical development of the "indeterminate" decision, and draw attention to the ambiguous behaviour of this decision in the XACML standard in Section 3.1. In Section 3.2 we prove there are numerous dependencies and redundancies between the XACML rule-combining algorithms, ultimately showing only two algorithms are required to express the entire suite of XACML rule-combining algorithms. We then extend our investigation to the overall expressivity of XACML in Section 3.3, demonstrating that XACML is not a functionally complete language: that is, there are policies of practical relevance that cannot be constructed in XACML. We consider which binary operators may be constructed in XACML in Section 3.4, and discuss the advantages of repla-

cing the XACML combining algorithms with the PTaCL policy operators in Section 3.5.

Motivated by addressing the shortcomings exhibited in XACML standard, we expand our attention to an ABAC language that is known to be functionally complete, PTaCL. However, we demonstrate in Chapter 4 that, while functional completeness implies any conceivable policy may be constructed, doing so in PTaCL is a non-trivial task. In Section 4.1 we formalise this observation, showing that PTaCL is not *canonically complete*, which means PTaCL does not permit a *normal form* for policies. Naturally, we then explore how PTaCL may be modified to make it canonically complete. Section 4.2 shows, through a simple substitution of operators, how we may produce a canonically complete version of PTaCL, called $\text{PTaCL}_3^{\leq}$. We discuss the advantages of canonically complete ABAC languages in Section 4.3. In Sections 4.4 and 4.5, we define precise syntax and semantics for handling indeterminacy and obligations in $\text{PTaCL}_3^{\leq}$, and provide a comparison with XACML and other languages in the literature in Section 4.6.

Chapters 3 and 4 focus on ABAC languages defined over a 3-valued decision set. However a number of other languages in the literature are defined over 4 values, in which the decision set is partially ordered [10, 32, 45]. In Chapter 5 we expand our theoretical foundations to consider these 4-valued languages. We formally define the necessary prerequisite material on partially ordered sets, lattices and bilattices in Section 5.1. In Section 5.2 we review Belnap logic [7], the logic on which the languages mentioned above are based. We extend the definitions of canonical suitability, normal form and canonical completeness to lattice-based logics in Section 5.3, enabling us to determine whether Belnap logic and its associated ABAC languages have these properties. We show that Belnap logic, and thus any language based on Belnap logic, is not canonically complete in Section 5.4. In Section 5.5 we identify connections between the symmetric group and unary operators on the set of authorization decisions, which enables us to construct a canonically complete 4-valued lattice-based logic. Finally, we conclude the chapter by demonstrating how our construction may be applied to total-ordered logics, showing a construction for a canonically complete m-valued total-ordered logic.

Chapter 6 demonstrates how we may use the canonically complete 4-valued lattice-based logic from Chapter 5 as the foundation for an ABAC language. We formally define a new 4-valued canonically complete ABAC language, $\text{PTaCL}_4^{\leqslant}$, in Section 6.1, and present a novel method for constructing policies. Sections 6.2 and 6.3 extend methods for handling indeterminacy and obligations to $\text{PTaCL}_4^{\leqslant}$, based on work from Chapter 4. In Section 6.4 we demonstrate how we may leverage the well-defined parts of XACML, in particular its architecture, and combine these with $\text{PTaCL}_4^{\leqslant}$. Thus, we produce a canonically complete ABAC language with a standardized, well-defined ar-

chitecture. Furthermore, we develop an algorithm which takes an arbitrary policy expressed as a decision table as input, and outputs an equivalent normal form for the policy in Section 6.4.1. This may be implemented alongside the customized XACML architecture, providing a means for simplified policy specification, which produces policies that are machine-enforceable.

In Chapter 7 we define a new ABAC language, Attribute Expression Policy Language (AEPL). AEPL is based on the accumulation of insights, intuitions and results from the previous chapters. Section 7.1 formally defines the AEPL language. We introduce the idea of an *attribute expression*, arguing why they are more expressive than traditional targets, and provide semantics for evaluating requests with respect to attribute expressions. We then define AEPL policies, and demonstrate why constructing policies in AEPL is intuitive and preferable to the tree-structured policies found in XACML and PTaCL. In Section 7.2 we demonstrate a number of methods for reducing the size of policy tables in AEPL. We provide a comparison of AEPL with XACML and PTaCL in Section 7.3, and show that AEPL is more expressive and intuitive than both of these languages. Finally in Section 7.4, we show how existing access control paradigms such as role-based access control and access control lists may be enhanced through the use of policy tables, to make them "attribute-aware".

Finally in Chapter 8 we review the contributions of the thesis and discuss the various opportunities for future research.

The contributions of this thesis are summarized in Table 1.1. Figures, tables, commands and equations are numbered sequentially within each of the eight chapters. Theorems, definitions and similar environments are numbered sequentially within each section. Full details of the references are given at the end of the thesis in alphabetical order by author.

1.3.1 Thesis audience

The work in this thesis tackles a number of different areas and challenges encountered in the design, specification and implementation of ABAC languages. In the interests of clarity and to assist readers whose interests lay in specific regions, we loosely group together related issues below, and summarise the portions of the thesis that address them.

First, we address the syntactic and semantic definitions used to express targets, policies and combining operators in ABAC languages. Chapter 3 extensively covers the syntax and semantics of XACML, shows how decisions can be used to handle errors, how redundancies can arise from overlapping definitions of combining algorithms, and ultimately that poor specification of combining operators leads to a functionally incomplete language. In contrast, PTaCL is an ABAC language which as formal, well-defined syntax and semantics, which provides a firm foundation for developing new ABAC languages.

Section 3.2	Analysis of dependencies in XACML combining algorithms
Section 3.3	Proof that XACML is not functionally complete
Section 3.4	Construction of all possible XACML binary operators
Section 4.1	Proof that PTaCL is not canonically complete
Section 4.2	Definition of $\text{PTaCL}_3^{\leq}$
Section 4.5	Syntax and semantics for computing obligations in $\text{PTaCL}_3^{\leq}$
Section 5.3	Extension of canonical completeness to lattice-based logics
Section 5.4	Completeness results for Belnap logic and associated ABAC languages
Section 5.5	Definition of a canonically complete 4-valued logic
Section 5.6	Construction of a canonically complete m-valued total-ordered logic
Chapter 6	Definition of $\text{PTaCL}_4^{\preccurlyeq}$
Section 6.1.3	A novel method for policy specification
Section 6.4	Method for leveraging the XACML architecture
Section 6.4.1	An algorithm for automatic policy generation
Chapter 7	A novel ABAC language based on policy tables
Section 7.1	Definition of AEPL
Section 7.2	Methods for policy compression in AEPL
Section 7.3	Comparison of AEPL with XACML and PTaCL
Section 7.4	Applications of AEPL to role-based access control and ACLs

Table 1.1: Contributions of the thesis

We demonstrate in Chapter 4, through minor modification of PTaCL, how we can construct a canonically complete ABAC language. In Chapter 7, we presented revised syntax and semantics for targets, allowing greater control over evaluation of requests.

The second focus of this thesis is the underlying multi-valued logics of ABAC languages [10, 12, 32, 45]. We utilise theoretical foundations based on results of Jobe [24], and show how these concepts may be leveraged to construct ABAC languages with desirable properties, such as functional and canonical completeness. This is covered primarily in Chapters 4,5 and 6.

Thirdly, we investigate the syntax and semantics for automatically computing obligations in Section 4.5. While our attention is restricted to an abstract set of obligations and we do not consider how conflicting obligations, or temporal constraints are handled, the syntax and semantics are necessary precursors and pave the way for future work to explore the other issues.

Finally, a recurring theme throughout this thesis is the focus on policy specification and composition, primarily from the viewpoint of a policy author. We present extensive arguments that policy specification is challenging in existing languages such as XACML and PTaCL. Engineering an arbitrary policy using the operators specified in the given language is a non-trivial task, and little support or guidance is given on this matter. To counteract this problem, we introduce the notion of policy tables, an intuitive, simple method for specifying arbitrary policies, and build back end support behind this idea to

enable automatic conversion from policy table to machine-enforceable policy. Chapters 4, 6 and 7 contribute the majority of this work.

1.4 Publications

In the duration of this thesis we have published a total of four peer-reviewed papers: one at the International Workshop on Security and Trust Management (STM) [14], two at the ACM Symposium on Access Control Models and Technologies (SACMAT) [15, 16], and one at the ACM Conference on Data and Application Security and Privacy (CODASPY) [17]. Chronologically listed:

- Crampton and Williams, "Obligations in PTaCL" [14], provides the basis for Sections 4.5 – 4.6.
- Crampton and Williams, "On Completeness in Languages for Attribute-Based Access Control" [15], provides the basis for Sections 3.2 – 3.5 and Sections 4.2 – 4.3.
- Crampton and Williams, "Canonical Completeness in Lattice-Based Languages for Attribute-Based Access Control" [17], provides the basis for Chapters 5 and 6.
- Crampton and Williams, "Attribute Expressions, Policy Tables and Attribute-Based Access Control" [16], provides the basis for Chapter 7.

Chapter 2

Preliminaries

The purpose of this chapter is to familiarize the reader with the prerequisite material on tree-structured attribute-based access control (ABAC) languages and multi-valued logics (MVLs).

Section 2.1 introduces the basic structure and evaluation semantics for targets and policies in *tree-structured* languages. We also introduce the notions of *obligations* and *indeterminacy*, which are fundamental to the material in Chapters 3 and 4. Section 2.2 introduces the XACML standard [35, 36, 37], which will be referenced and used as a comparative language throughout this thesis. We describe how rules, policies and policy sets are evaluated, and provide a brief introduction to the rule- and policy-combining algorithms, which are investigated in more detail in Chapter 3. In Section 2.3 we formally define PTaCL, an ABAC language that we will analyse extensively in this thesis, later we adapt and develop variants of PTaCL.

We conclude the chapter with an introduction to multi-valued logics, identifying how they are fundamentally related to many ABAC languages. We present the theoretical foundations, based on work by Jobe [24], for characterising properties of multi-valued logics, which will be applied throughout this thesis.

2.1 Tree-structured languages

Informally, we say a language is *tree-structured* if a policy is specified by a policy operator (decision combining algorithm) and a set of child policies. A request is evaluated with respect to a policy by first computing a decision for each of the child policies and then combining those decisions using the policy operator (decision-combining algorithm).

More formally, we assume the existence of a set of requests, defined in terms of attributes. Each policy specifies a target predicate defining, in terms of attribute values, the set of requests to which a policy applies. A target t is evaluated with respect to a request q, and returns either "no-match" or "match", denoted by 0_{m} and 1_{m} respectively. We write $\tau_q(t) \in \{0_{\mathrm{m}}, 1_{\mathrm{m}}\}$ to

indicate the result of evaluating target t with respect to request q, where

$$\tau_q(t) = \begin{cases} 1_{\mathrm{m}} & \text{if the target is applicable,} \\ 0_{\mathrm{m}} & \text{otherwise.} \end{cases}$$

We do not discuss here how target applicability is determined; the reader is referred to the literature for further details [12, 37]. Throughout this thesis we will use these semantics for target evaluation, later adding a third evaluation possibility $?_{\mathrm{m}}$ when discussing indeterminacy (in languages such as PTaCL and XACML). An example of a simplistic target that contains only one name-value pair is $t = \{(\mathit{deparment}, \mathit{purchasing})\}$. We say this target is matched if user who is in the purchasing department makes an access request, that is, the user supplies the attribute name-value pair $(\mathit{department}, \mathit{purchasing})$ as part of their request.

Let D be a set of *authorization decisions*. Typically, we assume D contains the values 0, 1 and $\perp$ representing "deny", "allow" and "not-applicable", respectively. We call 0 and 1 *conclusive* decisions. Let $\oplus$ be an associative binary operator defined on D and $-$ be a unary operator defined on D, and let p and p' be policies. Then

- d is an *atomic policy*;
- $p \oplus p'$ and $-p$ are policies;
- (t, p) is a policy.

The first stage in policy evaluation for a request q is to determine whether a policy is "applicable" to q or not. Every (well-formed) request q, allows us to assign a truth value to the target t. Specifically, if t evaluates to 1_{m}, we say the associated policy is *applicable*; otherwise the policy is *not applicable*. Then, writing $\rho_q(p)$ to denote the decision assigned to policy p for request q, we define:

$$\begin{aligned} \rho_q(d) &= d; \\ \rho_q(-p) &= -\rho_q(p); \\ \rho_q(p \oplus p') &= \rho_q(p) \oplus \rho_q(p'); \\ \rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \perp & \text{otherwise.} \end{cases} \end{aligned}$$

It is easy to see that we may represent a policy as a tree. Hence, we describe policy languages of this nature as *tree-structured*. The first stage in policy evaluation corresponds to labelling the nodes of the tree applicable or not applicable. We then compute a decision for non-leaf nodes in the tree by combining the decisions assigned to their respective children.

In the interests of clarity, we now demonstrate a simple policy expressed in natural language, followed by an abstract policy represented as logical terms. Consider the policy:

> *A user in the purchasing department is authorized to raise a purchase order, on the condition that they located on-site.*

We express the attributes from this policy this in the following targets:

- $t_1 = \{$(department,purchasing),(purchase order,raise)$\}$
- $t_2 = \{$(location,offsite)$\}$

Note that t_2 uses the location value "offsite", which we use here as syntactic sugar for the logical inverse of "on-site". (An alternative method would be to define $t_2 = \{$(location,not(on-site)$\}$.) Then we can define the following policies:

$$\begin{aligned} p_1 &= (t_1, 1), \\ p_2 &= (t_2, 0), \\ p_3 &= p_1 \oplus p_2. \end{aligned}$$

When a user makes a request to this policy, they supply a set of attributes, which are first evaluated against the targets in each policy. If the user provides the two attribute name-value pairs (department,purchasing) and (purchase order,raise) in their request, target t_1 is matched. Likewise, if a user provides an attribute value for location that is offsite (not on-site), then target t_2 is matched. Hence, policy p_1 will return an allow decision, and p_2 will return a deny decision. As the intention is to deny requests which originate from locations that are not on-site, the combination of $p_1 \oplus p_2 = 1 \oplus 0$ should return a deny decision. If policy p_2 had returned a not-applicable decision, representing that a user is on-site, then the combination of $p_1 \oplus p_2 = 1 \oplus \perp$ should return allow. We shall see later that XACML's deny-override operator implements these precise semantics. Visually, we can imagine this policy as a tree, shown in Figure 2.1.

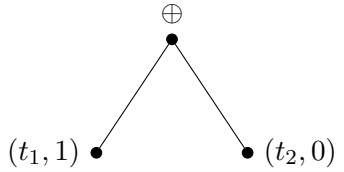
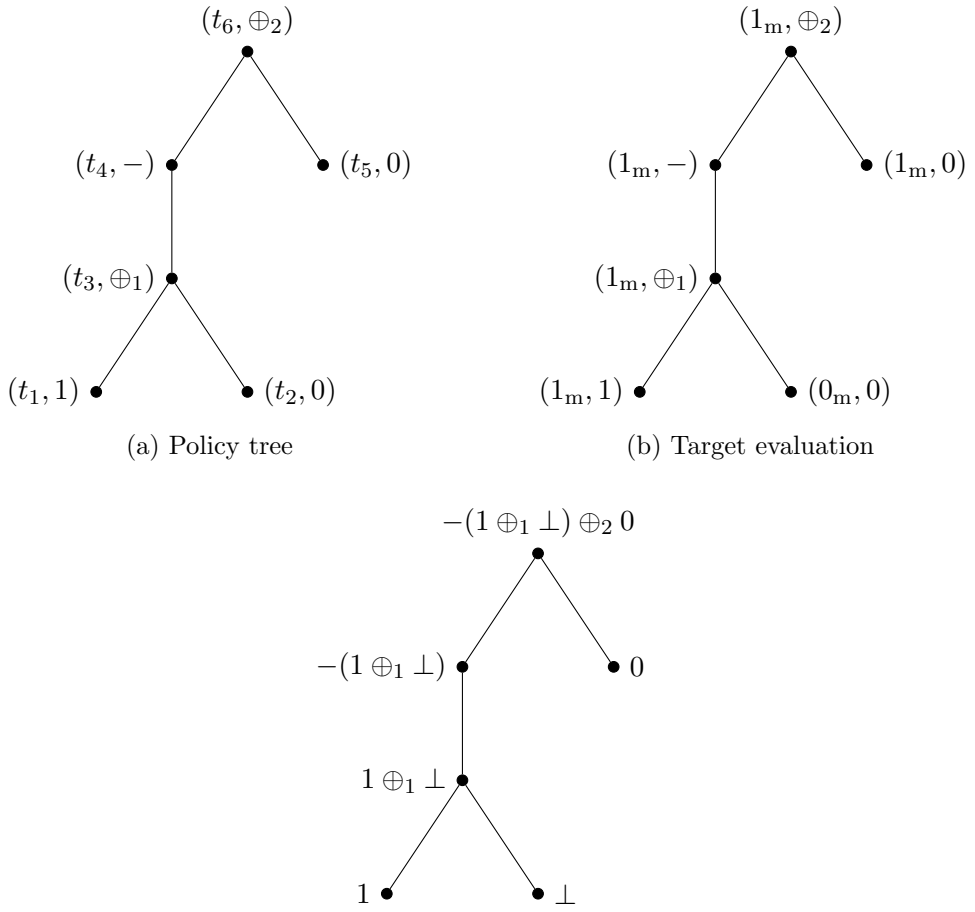


Figure 2.1: Simple policy tree

Figure 2.2 shows the tree for the more complex, abstract policy

$$\Big(t_6, \big(t_4, -\big(t_3, (t_1, 1) \oplus_1 (t_2, 0)\big) \oplus_2 (t_5, 0)\big)\Big)$$

and the evaluation of that policy for a request q such that $\tau_q(t_i) = 1_{\mathrm{m}}$ for all i except $i = 2$.



(a) Policy tree

(b) Target evaluation

(c) Policy evaluation

Figure 2.2: Evaluating a policy

We will make use of the following terminology [11] when describing policy (decision) operators.

Definition 2.1.1. *Let $\oplus : D \times D \to D$ be a policy operator. Then*

- $\oplus$ *is* commutative *if $d \oplus d' = d' \oplus d$ for all $d \in D$;*
- $\oplus$ *is* idempotent *if $d \oplus d = d$ for all $d \in D$, and* quasi-idempotent *if $d \oplus d = d$ for all $d \in \{0, 1\}$;*
- $\oplus$ *is* conclusive *if $d \oplus d' \in \{0, 1\}$ for all $d, d' \in D$, and* quasi-conclusive *if $d \oplus d' \in \{0, 1\}$ for all $d, d' \in \{0, 1\}$;*
- $\oplus$ *is a* $\cup$-operator *if $d \oplus \bot = d = \bot \oplus d$ for all $d \in D$;*
- $\oplus$ *is a* $\cap$-operator *if $d \oplus \bot = \bot = \bot \oplus d$ for all $d \in D$;*
- $\oplus$ *is* well-behaved *if it is either a $\cup$- or an $\cap$-operator.*

We say two policies p and p' are *equivalent*, denoted by $p \equiv p'$, if $\rho_q(p) = \rho_q(p')$ for all requests q. We now briefly introduce the concepts of *obligations* and *error handling (indeterminacy)* in the context of tree-structured ABAC languages.

2.1.1 Obligations

In addition to the specification of access control requirements, such as which users are allowed to access which objects, access control mechanisms can impose obligation requirements that specify what actions a user or system is obliged to perform. An *obligation* then, is a mandate on what must be carried out before or after an access is approved or denied, and they are used to meet formal requirements of systems such as non-repudiation [22, 23]. One example of an obligation would be to log access to a resource when an access request is made.

Usually, each access control policy will have associated obligations, rather than having obligations as separate functions, so obligations may be thought of as a function of the access request. Due to this relationship between access control policies and obligations, methods for evaluating obligations alongside policies have been proposed in the literature [3, 12, 29, 37]. We discuss methods for computing obligations in more detail in Chapter 4.

2.1.2 Error handling

Given the distributed, connected nature of modern access control mechanisms, it is important to consider scenarios where policy evaluation may fail. This may occur for a number of reasons: (i) missing attributes in the request or policy; (ii) network errors while attempting to retrieve policies or attributes; (iii) division by zero during policy evaluation; and (iv) syntax errors in the request or policy. Broadly speaking, we refer to these cases as encountering "indeterminacy", representing policy evaluation is unsure on how to proceed. An important aspect of authorization languages is how they handle indeterminacy, and there are varying methods employed in the literature [11, 12, 29, 33, 37]. We present and review these methods in Chapters 3 and 4.

2.2 XACML

XACML [35, 36, 37] is a commonly used authorization language for implementing attribute-based access control in the real world. XACML is a standardized XML-based language, currently in its 3rd version [37]. Due to its widespread use, XACML has been the focus of a number of works in the literature [29, 34, 40], and is often used as a comparative language. Throughout this section we will demonstrate why XACML may be viewed as a tree-structured language.

We begin by formally defining the evaluation semantics for XACML rules, policies and policy sets and introduce the rule and policy-combining algorithms specified in the XACML standard. In this thesis we do not consider XACML "conditions", mostly because this notion is inadequately constrained in XACML 3.0. Indeed, a condition can be any Boolean expression, including arbitrarily complex functions, which in practice means that it is possible to write a full program in the condition of a rule. In particular, the notion of condition makes the notion of target redundant, because any target can be expressed as a condition. The extent to which conditions are used in practice is unclear and their generality means they are rarely, if ever, discussed in the academic literature. Furthermore, during the preliminaries, we do not consider indeterminacy in XACML. We dedicate a section of Chapter 3 to the discussion and analysis of how XACML handles indeterminacy.

2.2.1 Rule, policy and policy set evaluation

An XACML *rule* is defined by a *target* and an *effect*: the target determines whether a policy is "applicable" to a request;[1] and the effect is either "deny" or "permit". For brevity, we write 0 and 1 to denote "deny" and "permit", respectively. An XACML *policy* is defined by a target, one or more rules and a rule-combining algorithm: the target determines when a policy is "applicable" to a request or not; and the decision returned for request q with respect to a policy is obtained by computing a decision for q with respect to each rule, and combining those decisions using the rule-combining algorithm.

More formally, for an XACML target t, we write $\tau_q(t)$ to indicate the result of evaluating target t with respect to request q, where

$$\tau_q(t) = \begin{cases} 1_{\mathrm{m}} & \text{if the target is applicable,} \\ 0_{\mathrm{m}} & \text{otherwise.} \end{cases}$$

We then write $\rho_q(r)$ to denote the decision returned by evaluating rule r for request q. An XACML rule may be represented as a pair (t, e), where t is a target and $e \in \{0, 1\}$ is the effect of the rule. Then we define

$$\rho_q(t, e) = \begin{cases} e & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \bot & \text{otherwise.} \end{cases}$$

where $\bot$ denotes the "not-applicable" decision. An XACML policy may be represented as a triple $(t, A, \overline{r})$, where t is a target, A is a rule-combining

[1] We do not discuss how target applicability is determined; the reader is referred to the XACML standard [37] for further details.

algorithm and $\overline{r} = \langle r_1, \ldots, r_k \rangle$ is a tuple of rules. Then we define

$$\rho_q(t, A, \overline{r}) = \begin{cases} A(\rho_q(r_1), \ldots, \rho_q(r_k)) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \bot & \text{otherwise.} \end{cases}$$

An XACML policy set is a triple of the form $(t, A, \overline{p})$, where $\overline{p} = \langle p_1, \ldots, p_k \rangle$ is a tuple of policies, t is a target, and A is a policy-combining algorithm. Then

$$\rho_q(t, A, \overline{p}) = \begin{cases} A(\rho_q(p_1), \ldots, \rho_q(p_k)) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \bot & \text{otherwise.} \end{cases}$$

In other words, an XACML policy set is evaluated in exactly the same way as an XACML policy: (recursively) evaluate the "child" policies and then combine the decisions. It it worth noting that the semantics for target evaluation in XACML are similar to those of generic tree structured languages defined in Section 2.1. In addition, the evaluation semantics for rules are identical to those for "atomic policies" (with a target t).

2.2.2 Rule- and policy-combining algorithms

XACML 3.0 defines 11 rule-combining algorithms, summarized in Table 2.1. XACML defines ordered and unordered versions of most of the algorithms and also provides backward compatibility with previous versions of XACML [35, 36]. The one algorithm that is non-commutative (that is, the order of rule evaluation matters) is first-applicable (fa).

Algorithm	Ordered?	Order-dependent eval?
Deny-overrides (do)	No	No
Permit-overrides (po)	No	No
Legacy Deny-overrides (ldo)	No	No
Legacy Permit-overrides (lpo)	No	No
Deny-unless-permit (dup)	No	No
Permit-unless-deny (pud)	No	No
Ordered-deny-overrides (odo)	Yes	No
Ordered-permit-overrides (opo)	Yes	No
Legacy Ordered-deny-overrides (lodo)	Yes	No
Legacy Ordered-permit-overrides (lopo)	Yes	No
First-applicable (fa)	Yes	Yes

Table 2.1: The XACML 3.0 rule-combining algorithms

The XACML rule-combining algorithms take an arbitrary number of inputs but process the inputs sequentially. As such, an XACML rule-combining algorithm can be thought of as a family of k-ary operators, $k \geqslant 2$, on the decision set $\{0, 1, \bot\}$. It is convenient, in terms of formal exposition, to assume that a decision-combining algorithm is implemented using binary *decision operators*.

do	0	1	⊥
0	0	0	0
1	0	1	1
⊥	0	1	⊥

po	0	1	⊥
0	0	1	0
1	1	1	1
⊥	0	1	⊥

dup	0	1	⊥
0	0	1	0
1	1	1	1
⊥	0	1	0

pud	0	1	⊥
0	0	0	0
1	0	1	1
⊥	0	1	1

fa	0	1	⊥
0	0	0	0
1	1	1	1
⊥	0	1	⊥

(a) Decision tables

Operator	Idempotent	∪-operator	Commutative	Conclusive
do, po	Yes	Yes	Yes	No
dup, pud	No	No	Yes	Yes
fa	Yes	Yes	No	No

(b) Properties

Figure 2.3: XACML rule-combining algorithms do, po, dup, pud and fa

(Thus, we would apply a binary decision operator $k-1$ times to evaluate a call to a decision-combining algorithm with k inputs.)

Any binary operator on the decision set $\{0, 1, \bot\}$ can be represented as a 3×3 array, as shown in Figure 2.3a for the deny-overrides (do), permit-overrides (po), deny-unless-permit (dup), permit-unless-deny (pud), and first-applicable (fa) operators. The decision tables for the ordered, unordered and legacy versions of do, po, dup and pud are identical for the decision set $\{0, 1, \bot\}$. Henceforth, we will treat the XACML rule-combining algorithms as decision-combining binary operators, as defined in Figure 2.3a.

All five of the XACML decision operators in Figure 2.3a are ∪-operators and quasi-conclusive (and thus well-behaved). Other properties of the XACML decision operators are summarized in Figure 2.3b.

All of the XACML rule-combining algorithms have associated policy-combining algorithms which have identical properties and produce the same decision tables[2], the only difference is that they act on policies as opposed to rules. Thus, the features and properties shown in Figure 2.3 also hold true for the XACML policy-combining algorithms. There is one additional policy-combining algorithm – only-one-applicable – which does not have an associated rule-combining algorithm. We do not consider this algorithm here as it introduces a fourth indeterminate decision when combining decisions in the set $\{0, \bot, 1\}$, and for reasons we will discuss in Chapter 3.

To summarise, we express the XACML rule-combining algorithms as (binary) decision operators on the set $D = \{0, 1, \bot\}$, corresponding to the decisions "deny", "permit" and "non-applicable", respectively. And we compute a

[2]For reference see Appendix C in the XACML standard [37].

k-ary operator by applying a binary operator $k-1$ times. Hence, we assume that XACML policies can be represented in the form $(t, r \oplus r')$, where $\oplus$ is a binary operator on D and

$$\rho_q(t, r \oplus r') = \begin{cases} \rho_q(r) \oplus \rho_q(r') & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \perp & \text{otherwise.} \end{cases}$$

This representation of XACML policies is equivalent to the representation of policies introduced in Section 2.1, thus justifying our classification of XACML as a tree-structured language.

2.2.3 Architecture

The reference architecture for evaluating access requests forms a significant portion of the XACML standard. We now provide a high level summary of the XACML architecture, describing the key components and interactions.[3]

First, we describe the main components of the XACML architecture:

- The Policy Administration Point (PAP) is the administration interface, which provides a means for authoring and deploying policies, and acts as a repository for the access control policies.
- The Policy Enforcement Point (PEP) is the interface of the whole environment to the outside world. The PEP receives the access requests and enforces the decision of the PDP with the help of the other main components, and either permits or denies access.
- The Policy Decision Point (PDP) is the main decision point for access requests. The PDP collects all the necessary attributes and carries out the policy evaluation.
- The Policy Information Point (PIP) is the main point where attributes for policy evaluation are retrieved, from internal and external sources.
- The Context Handler acts as an interface between the components listed above.

When an access request is made by a user, it intercepted by the PEP. The PEP then sends the request to the context handler, which transforms the request (which may be in the format specific to the application environment) into an XACML request comprising attributes and sends the request to the PDP. Upon receiving a request, the PDP will look up the policies deployed on it and establish which policies are applicable to the specific request. During

[3]The focus of this thesis is primarily on policy languages, rather than the architecture that implements such languages. However, a basic understanding of how access control policies are enforced will provide some benefit to the reader.

policy evaluation, the PDP may require additional attributes, and will occasionally send attribute queries to the context handler. The context handler will retrieve these attributes via the PIP, and return them to the PDP. The PDP will then decide if the access request is allowed, denied, not-applicable or report an error. This decision is returned to the PEP, which enforces the decision and fulfils any obligations if they exist.

The major components in the XACML domain are shown in the data-flow diagram of Figure 2.4.

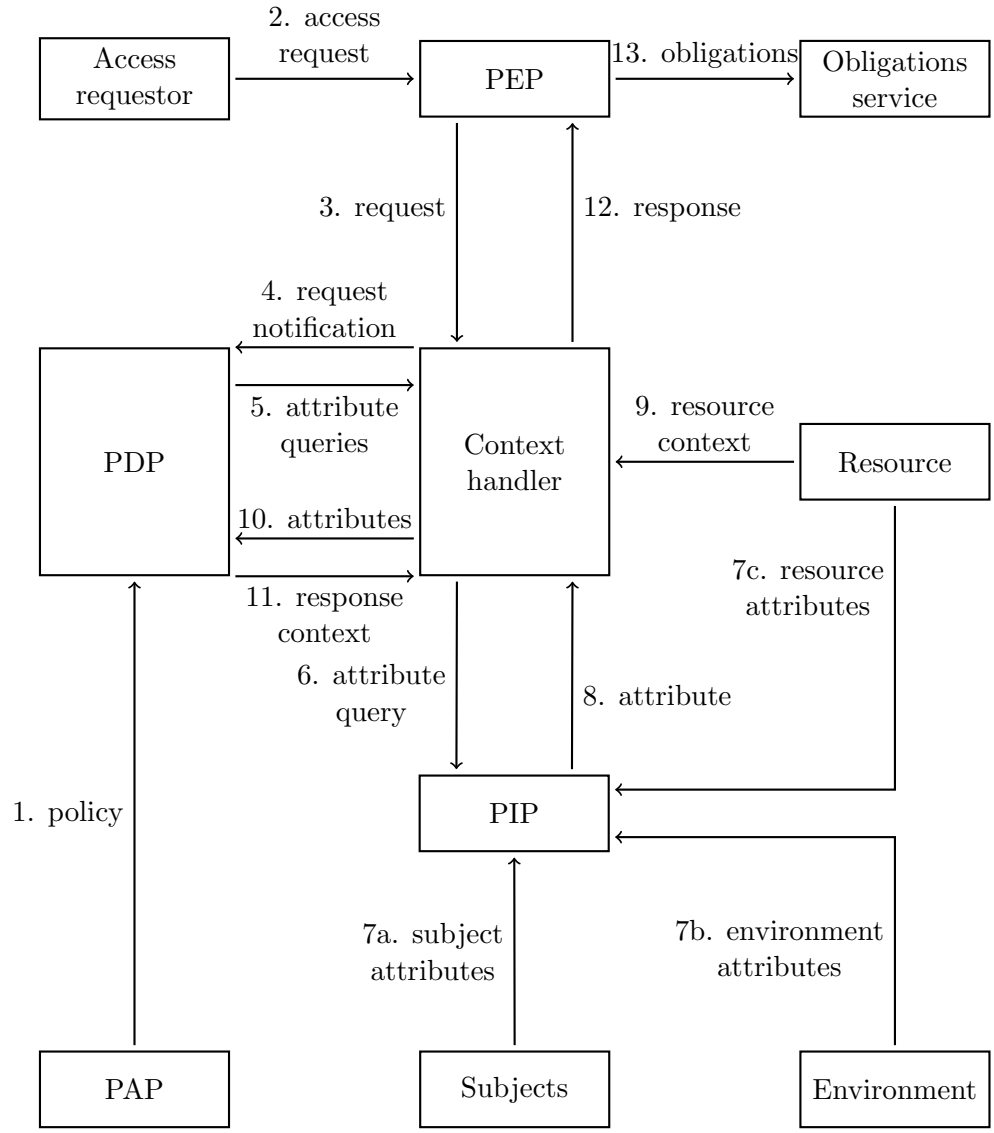


Figure 2.4: XACML data-flow model

2.3 PTaCL

PTaCL [12] is a tree-structured language, intended to provide a generic framework for specifying target-based policy languages. Like XACML, it is defined (without indeterminacy) over a three-valued decision set. Hence, PTaCL may

be used to provide a more formal point of comparison with XACML, thereby facilitating the analysis of XACML and identifying weaknesses in its specification. PTaCL defines a *policy target language* (PTL), for specifying targets in terms of attributes (of users and resources), and a *policy combining language* (PCL), for combining (the decisions associated with the evaluation of) sub-policies. As previously mentioned, in this thesis we do not discuss how target applicability is determined. In the remainder of this section we discuss the policy combining language of PTaCL, the reader is referred to the literature for further details on the policy target language of PTaCL [12].

2.3.1 Syntax and semantics.

PTaCL policies are defined inductively with respect to a set of policy decisions D. We assume that D contains decisions 0 and 1, corresponding to "deny" and "allow", respectively. For now, we assume $D = \{0, 1, \bot\}$, where the decision $\bot$ denotes that a policy is not applicable to a request. (We consider more complex decision sets in Chapter 4). Then 0 and 1 are (atomic) policies. Moreover, if p, p_1 and p_2 are policies and t is a target, then the following are policies:

$$\begin{array}{llll} \neg\, p & \text{(negation)} & p_1 \wedge_{\mathrm{p}} p_2 & \text{(join)} \\ \sim p & \text{(deny-by-default)} & (t, p) & \text{(selection)} \end{array}$$

The semantics of a PTaCL policy are defined by applying the operators $\neg$, $\sim$ and $\wedge_{\mathrm{p}}$ (defined on the set of decisions) to the decisions returned by the evaluation of sub-policies.[4] The evaluation tables for $\neg$, $\sim$ and $\wedge_{\mathrm{p}}$ are shown in Figure 2.5a.

d	$\neg\, d$	$\sim d$
0	1	0
$\bot$	$\bot$	0
1	0	1

$\wedge_{\mathrm{p}}$	0	$\bot$	1
0	0	0	0
$\bot$	0	$\bot$	$\bot$
1	0	$\bot$	1

(a) $\neg, \sim$ and $\wedge_{\mathrm{p}}$

$$\begin{aligned} \rho_q(0) &= 0; \\ \rho_q(1) &= 1; \\ \rho_q(\neg\, p) &= \neg\, \rho_q(p); \\ \rho_q(\sim p) &= \sim \rho_q(p); \\ \rho_q(p_1 \wedge_{\mathrm{p}} p_2) &= \rho_q(p_1) \wedge_{\mathrm{p}} \rho_q(p_2); \\ \rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \bot & \text{otherwise.} \end{cases} \end{aligned}$$

(b) Policy semantics

Figure 2.5: Decision operators and policy semantics in PTaCL

The unary operators $\neg$ and $\sim$ simply modify the decision: the former switches the values of 0 and 1, leaving $\bot$ unchanged; the latter transforms $\bot$ to 0, leaving 0 and 1 unchanged. These operators are used to implement policy negation and deny-by-default policies, respectively. The binary operator $\wedge_{\mathrm{p}}$

[4] We use the subscript p to distinguish $\wedge_{\mathrm{p}}$ from other $\wedge$ operators that will be used throughout this thesis.

is strong conjunction in the Kleene three-valued logic [25]. It returns 0 if at least one of the operands is 0, 1 if both operands are 1, and $\perp$ otherwise. (The operator $\wedge_{\mathrm{p}}$ may also be considered as the greatest lower bound operator under the ordering $0 < \perp < 1$.) Given a request q, we write $\rho_q(p)$ to denote the decision returned by policy p for request q. The semantics of PTaCL policies are defined in Fig. 2.5b.

Any PTaCL policy may be represented as a *policy tree*, in which leaf nodes are 0 or 1 decisions and internal nodes may be a target, one of the unary operators $\neg$ or $\sim$, or the binary operator $\wedge_{\mathrm{p}}$. We introduce a special target all where

$$\tau_q(\mathsf{all}) = 1_{\mathrm{m}} \quad \text{for all requests } q.$$

We assume that targets on the operators $\neg$ and $\sim$ always use the target all as these operators only change decision values. Thus, we may assume that every policy has the form (t, p). Figure 2.6a shows the policy tree representing the policy

$$\sim \Big(t_5, \big(\neg(t_3, (t_1, 1) \wedge_{\mathrm{p}} (t_2, 0)\big) \wedge_{\mathrm{p}} (t_4, 1)\big)\Big).$$

Request evaluation may be described in terms of policy trees and comprises two phases. The first phase *evaluates the targets*. The second phase *propagates the decisions* of sub-policies up to the root of the policy tree using the policy-combining operators at the internal nodes and the semantics defined in Fig. 2.5.

Consider the policy depicted in Fig. 2.6a and suppose that

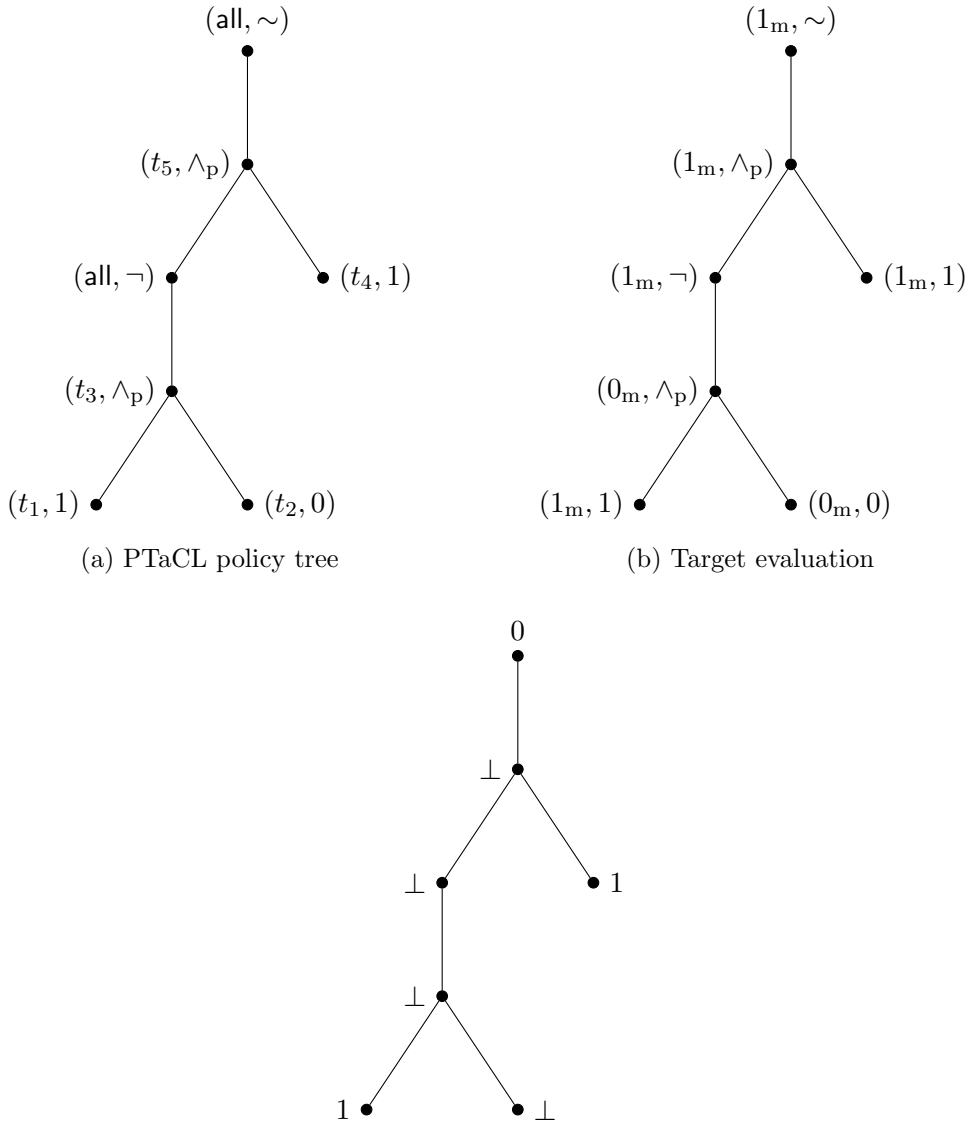
$$\tau_q(t_1) = \tau_q(t_4) = \tau_q(t_5) = 1_{\mathrm{m}} \text{ and } \tau_q(t_2) = \tau_q(t_3) = 0_{\mathrm{m}}.$$

The first phase of request evaluation results in the tree shown in Fig. 2.6b; recall that the targets for $\sim$ and $\neg$ are all and thus necessarily evaluate to 1_{m}. The second phase of policy evaluation is shown in Fig. 2.6c. Note that the evaluation of the sub-trees with roots t_3 and t_5 consider the combination of a 1 and $\perp$ decision, and $\perp \wedge_{\mathrm{p}} 1 = \perp$. At the root, the $\sim$ operator converts the $\perp$ decision into a 0 decision, which is the final decision returned for this policy.

PTaCL defines syntax and semantics for handling indeterminacy, that is, errors encountered during target evaluation. Informally, when target evaluation fails, PTaCL assumes that either $\tau_q(t) = 1_{\mathrm{m}}$ or $\tau_q(t) = 0_{\mathrm{m}}$ could have been returned, and returns the unions of the (sets of) decisions that would have been returned in both cases. We omit the formal definition, syntax and semantics of indeterminacy in PTaCL in the preliminaries; we will cover these in Chapter 4.

2.3.2 Additional operators

Crampton and Morisset showed that PTaCL is functionally complete [12]. In practical terms, this means we can introduce new binary operators to combine policies, which act as syntactic sugar, knowing that any such operator may be



(a) PTaCL policy tree

(b) Target evaluation

(c) Policy evaluation

Figure 2.6: Evaluating a PTaCL policy

constructed using the PTaCL operators. In particular, we define three new operators: $\vee_{\mathrm{p}}$, po and do, where

$$d \vee_{\mathrm{p}} d' \stackrel{\mathrm{def}}{=} \neg((\neg d) \wedge_{\mathrm{p}} (\neg d'));$$
$$d \,\mathsf{po}\, d' \stackrel{\mathrm{def}}{=} (d \vee_{\mathrm{p}} (\sim d')) \wedge_{\mathrm{p}} ((\sim d) \vee_{\mathrm{p}} d');$$
$$d \,\mathsf{do}\, d' \stackrel{\mathrm{def}}{=} \neg((\neg d) \,\mathsf{po}\, (\neg d')).$$

It is easy to show that these operators have the evaluation tables shown in Fig. 2.7. The operators po and do correspond to the XACML permit-overrides and deny-overrides combining algorithms, respectively (over the 3-valued decision set $\{0, 1, \bot\}$.

$\vee_p$	0	$\perp$	1
0	0	$\perp$	1
$\perp$	$\perp$	$\perp$	1
1	1	1	1

po	0	1	$\perp$
0	0	1	0
1	1	1	1
$\perp$	0	1	$\perp$

do	0	1	$\perp$
0	0	0	0
1	0	1	1
$\perp$	0	1	$\perp$

Figure 2.7: Supplementary decision operators for PTaCL

2.4 Other notable tree-structured languages

We now briefly introduce other notable tree-structured languages in the literature that we will discuss and review throughout this thesis. Li *et al.* [29] proposed a policy combining language (PCL), with the intention of providing a formal method for specifying new policy-combining algorithms for languages such as XACML [37]. While XACML supports the specification of custom combining algorithms, little to no help is given about the structure, syntax or semantics that custom combining algorithms should use or conform to. The choice is left entirely to the policy author, which can lead to policy misconfigurations. PCL addresses this problem, providing semantics for constructing additional policy combining algorithms through the use of policy combining operators (binary policy operators) and linear constraints. However, Li *et al.* admit that PCL is not fully expressive, that is, there are policy combining algorithms that cannot be constructed using either policy combining operators or linear constraints. This provides us with motivation to investigate ABAC languages that are known to be functionally complete, such as PTaCL [12].

In Chapters 3 and 4, we restrict our attention to 3-valued ABAC languages; primarily XACML and PTaCL. Later, in Chapters 5 and 6, we expand our research to consider ABAC languages that contain four decisions. A number of these languages are based on Belnap's 4-valued logic [7], including PBel [10], BelLog [45] and Rumpole [32]. We focus primarily on PBel, developed by Bruns and Huth [10]. Like PTaCL, PBel is functionally complete, since the policy operators in PBel are based on a logic that is known to be functionally complete. Furthermore, PBel assumes a lattice-ordering on its decisions, in contrast to the previously discussed 3-valued languages. We will discuss PBel and Belnap logic in more detail in Chapter 5.

2.5 Multi-valued logics

Traditionally, logic systems contained only two truth values: true or false; which correspond to our intuitive understanding of truth and falsity. However, the need to reason logically about systems that contain more than two truth values became apparent, leading to the inception of *multi-valued logics (MVLs)*. Multi-valued logics then, are logical calculi in which there are more than two possible truth values. The first multi-valued logics were proposed by

Łukasiewicz [30], which included a third truth value "unknown". Various work in the literature extended Łukasiewicz's work; perhaps the most well known 3-valued logics being Kleene's logic [26] and Priest's "Logic of Paradox" [38]. For now, we restrict our attention to 3-valued multi-valued logics, as both PTaCL and XACML operate over a 3-valued decision set. We expand our scope to more general multi-valued logics from Chapter 5 onwards.

Multi-valued logics are the fundamental "building blocks" for many ABAC languages, and the choice of multi-valued logic plays a crucial role in establishing properties of a new ABAC language. The choice of operators in ABAC languages is important, since we can reason about the *functional completeness* of a set of operators and truth values, and thus the functional completeness of the associated language. This is because, in tree-structured languages, policies are, essentially, terms in a logic-based formalism. For example, PTaCL is defined over the 3-valued decision set $\{0, \bot, 1\}$, and uses operators from Kleene's 3-valued logic.

The main purpose of this section is to introduce the theoretical foundations, based on results of Jobe [24], for characterising properties of multi-valued logics, such as *canonical suitability, functional completeness, selection operators, normal form* and *canonical completeness*. A number of the contributions of this thesis will leverage the completeness properties of multi-valued logics introduced by Jobe, and utilise them in the context in ABAC languages to construct languages with desirable properties (in terms of completeness, ease of policy specification and automatic generation of policies). Later in Chapter 5, we will extend these concepts to lattice-based multi-valued logics. This will allow us to analyse Belnap's 4-valued logic [7], which forms the basis for a number of ABAC languages including PBel [10], BelLog [45] and Rumpole [32].

2.5.1 Theoretical foundations of canonical completeness

Let $L = (V, \mathsf{Ops})$ be a logic associated with a set of truth values V and a set of logical operators Ops. We assume that V is a totally ordered set of m truth values, $\{0, \ldots, m-1\}$, with $0 < 1 < \cdots < m-1$. We omit V and Ops when no ambiguity can occur. We write $\Phi(L)$ to denote the set of formulae that can be written in the logic L.

We say that L is *canonically suitable* if and only if there exist formulae $\phi_{\max}$ and $\phi_{\min}$ of arity 2 in $\Phi(L)$ such that $\phi_{\max}(x, y)$ returns $\max\{x, y\}$ and $\phi_{\min}(x, y)$ returns $\min\{x, y\}$. If a logic is canonically suitable, we will write $\phi_{\max}$ and $\phi_{\min}$ using infix binary operators as $\curlyvee$ and $\curlywedge$ respectively.

Example 2.5.1. *Standard propositional logic with truth values* 0 *and* 1*, and operators* $\vee$ *and* $\neg$*, representing disjunction and negation, respectively, is canonically suitable:* $\phi_{\max}(x, y)$ *is simply* $x \vee y$*, while* $\phi_{\min}(x, y)$ *is* $\neg(\neg x \vee \neg y)$ *(that is, conjunction).*

A function $f : V^n \to V$ is completely specified by a truth table containing n columns and m^n rows. However, not every truth table can necessarily be represented by a formula in a given logic $L = (V, \mathsf{Ops})$. L is said to be *functionally complete* if for every function $f : V^n \to V$, there is a formula $\phi \in \Phi(L)$ of arity n whose evaluation corresponds to the truth table.

A *selection operator* $S^j_{(a_1,\ldots,a_n)}(x_1, \ldots, x_n)$ is an n-ary operator defined as follows:

$$S^j_{(a_1,\ldots,a_n)}(x_1, \ldots, x_n) = \begin{cases} j & \text{if } (x_1, \ldots, x_n) = (a_1, \ldots, a_n), \\ 0 & \text{otherwise.} \end{cases}$$

We will write $\mathbf{a}$ to denote the tuple $(a_1, \ldots, a_n) \in V^n$ when no confusion can occur. Note that $S^0_{\mathbf{a}}$ is the same for all $\mathbf{a} \in V^n$, and $S^0_{\mathbf{a}}(\mathbf{x}) = 0$ for all $\mathbf{x} \in V^n$. Illustrative examples of unary and binary selection operators (for a 4-valued logic) with minimal truth value 0 are shown in Figure 2.8.

x	$S^2_0(x)$	$S^1_3(x)$
0	2	0
1	0	0
2	0	0
3	0	1

$S^2_{(1,1)}(x,y)$		y			
		0	1	2	3
x	0	0	0	0	0
	1	0	2	0	0
	2	0	0	0	0
	3	0	0	0	0

Figure 2.8: Examples of selection operators in a 4-valued logic

Selection operators play a central role in the development of canonically complete logics because an arbitrary function $f : V^n \to V$ can be expressed in terms of selection operators. Consider, for example, the function

$$f(x, y) = \begin{cases} 1 & \text{if } x = 0,\ y = 2, \\ 2 & \text{if } x = y = 1, \\ 3 & \text{if } x = 3,\ y = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then it is easy to confirm that

$$f(x, y) \equiv S^1_{(0,2)}(x, y) \curlyvee S^2_{(1,1)}(x, y) \curlyvee S^3_{(3,0)}(x, y),$$

since $S^1_{(0,2)}(x, y)$ returns 1 if $x = 0$ and $y = 2$, and 0 otherwise, and so forth for $S^2_{(1,1)}(x, y)$ and $S^3_{(3,0)}(x, y)$. (There is one selection operators for each "row" or "line" in $f(x, y)$.) Moreover, $S^c_{(a,b)}(x, y) \equiv S^c_a(x) \curlywedge S^c_b(y)$ for any $a, b, c, x, y \in V$. Thus,

$$f(x, y) \equiv \big(S^1_0(x) \curlywedge S^1_2(y)\big) \curlyvee \big(S^2_1(x) \curlywedge S^2_1(y)\big) \curlyvee \big(S^3_3(x) \curlywedge S^3_0(y)\big).$$

In other words, we can express f as the "disjunction" ($\curlyvee$) of "conjunctions" ($\curlywedge$) of unary selection operators.

More generally, given the truth table of function $f : V^n \to V$, we can write down an equivalent function in terms of selection operators. Specifically, let

$$A = \{\mathbf{a} \in V^n : f(\mathbf{a}) > 0\};$$

then, for all $\mathbf{x} \in V^n$,

$$g(\mathbf{x}) = \bigvee_{\mathbf{a} \in A} S_{\mathbf{a}}^{f(\mathbf{x})}(\mathbf{x}).$$

Simple inspection of $g(x)$ confirms that this logical formula expressed in terms of selection operators is equivalent the truth table of the function $f : V^n \to V$, that is

$$g(x) \equiv f(x).$$

Jobe established a number of results connecting the functional completeness of a logic with the unary selection operators, summarized in the following theorem.

Theorem 2.5.1 (Jobe [24, Theorems 1, 2; Lemma 1])**.** *A logic L is functionally complete if and only if each unary selection operator is equivalent to some formula in L.*

The proofs of Jobe's results are by induction and constructive. Informally, if each unary selection operator is equivalent to some formula in L, then we can construct a formula (in L) for any selection operator; and if we can construct a formula for any selection operator, then we can construct a formula for an arbitrary truth table (function).

Definition 2.5.1. *The* normal form *of formula ϕ in a canonically suitable logic is a formula ϕ' that has the same truth table as ϕ and has the following properties:*

- *the only binary operators it contains are $\curlyvee$ and $\curlywedge$;*
- *no binary operator is included in the scope of a unary operator;*
- *no instance of $\curlyvee$ occurs in the scope of the $\curlywedge$ operator.*

In other words, given a canonically suitable logic L containing unary operators $\sharp_1, \ldots, \sharp_\ell$, a formula in normal form has the form

$$\bigvee_{i=1}^{r} \bigwedge_{j=1}^{s} \sharp_{i,j} x_{i,j}$$

where $\sharp_{i,j}$ is a unary operator defined by composing the unary operators in $\sharp_1, \ldots, \sharp_\ell$. In the usual 2-valued propositional logic with a single unary operator (negation) this corresponds to disjunctive normal form.

Definition 2.5.2. *A canonically suitable logic is* canonically complete *if every unary selection operator can be expressed in normal form.*

2.5.2 Example multi-valued logics

Having introduced the definitions for canonical suitability, functional completeness and canonical completeness, we now discuss the relationships between these properties for multi-valued logics. It is clear that a functionally complete logic L is canonically suitable, as the existence of the formulae $\phi_{\max}$ and $\phi_{\min}$ is implied by functional completeness. Intuition may suggest that functional completeness also implies canonical completeness: since every possible function can be represented as a formulae in L, it should be possible that there are equivalent formulae that are in normal form. However, the existence of a normal form is not guaranteed by functional completeness, as the following results will show. It is known that there are canonically suitable 3-valued logics that are: (i) not functionally complete [24, 30]; (ii) functionally complete but not canonically complete [24, Theorem 4]; and (iii) canonically complete (and hence functionally complete) [24, Theorem 6].

Łukasiewicz [30] defined a 3-valued logic, which is not functionally complete. Słupecki [44] extended this logic, showing Łukasiewicz's logic could be made functionally complete through the addition of one unary operator. The decision set for Łukasiewicz's extended logic $\mathcal{L}$ is $\{1, 2, 3\}$ with total ordering $1 < 2 < 3$, and operators defined in Figure 2.9. However, Jobe [24, Theorem 4] showed that the extended Łukasiewicz logic $\mathcal{L}$ is not canonically complete, despite being canonically suitable and functionally complete.

x	$N(x)$	$T(x)$
1	3	2
2	2	2
3	1	2

(a) N and T

$\supset$	1	2	3
1	3	3	3
2	2	3	3
3	1	2	3

(b) $\supset$

Figure 2.9: Operators in Łukasiewicz's logic $\mathcal{L}$

As a result, Jobe defined a canonically complete 3-valued logic [24]. The decision set for Jobe's logic $\mathcal{J}$ is $\{1, 2, 3\}$ with total ordering $1 < 2 < 3$, and operators defined in Figure 2.10. It is easy to establish that

$$x \curlywedge y \equiv x \wedge_{\mathrm{j}} y \quad \text{and} \quad x \curlyvee y \equiv E_2(E_2(x) \wedge_{\mathrm{j}} E_2(y)).$$

Thus $\mathcal{J}$ is canonically suitable [24, Theorem 6]. Henceforth, we will write $x \vee_{\mathrm{j}} y$ to denote $E_2(E_2(x) \wedge_{\mathrm{j}} E_2(y))$. The normal form formulae for the unary selection operators are shown in Figure 2.10b. (Note that S_i^1 is the same for all i.) Thus $\mathcal{J}$ is functionally and canonically complete [24, Theorem 7].

x	$E_1(x)$	$E_2(x)$
1	2	3
2	1	2
3	3	1

$\wedge_\mathrm{j}$	1	2	3
1	1	1	1
2	1	2	2
3	1	2	3

(a) Operators

$S_i^1(x)$	$x \wedge_\mathrm{j} (E_1\, x) \wedge_\mathrm{j} (E_2\, x)$
$S_1^2(x)$	$(E_2 E_1\, x) \wedge_\mathrm{j} (E_1\, x)$
$S_2^2(x)$	$x \wedge_\mathrm{j} (E_2\, x)$
$S_3^2(x)$	$(E_1 E_2\, x) \wedge_\mathrm{j} (E_1 E_2 E_1\, x)$
$S_1^3(x)$	$(E_1 E_2\, x) \wedge_\mathrm{j} (E_2\, x)$
$S_2^3(x)$	$(E_2 E_1\, x) \wedge_\mathrm{j} (E_1 E_2 E_1\, x)$
$S_3^3(x)$	$x \wedge_\mathrm{j} (E_1\, x)$

(b) Normal forms for the unary selection operators

Figure 2.10: Jobe's logic $\mathcal{J}$

2.6 Summary and discussion

In this chapter we have laid the necessary theoretical foundations and introduced the core concepts that will be explored and discussed throughout this thesis. First, we defined the general syntax and semantics of tree-structured ABAC languages, and introduced the notions of obligations and indeterminacy, which are investigated extensively in Chapters 3 and 4.

Then, we introduced the XACML standard, which we will investigate in-depth in the following chapter. In Chapter 3 we provide a comprehensive review of XACML, covering the role of the indeterminate decision, redundancies between the combining algorithms and ultimately show that XACML is not functionally complete. We also introduced the tree-structured language PTaCL, an ABAC language intended to provide a generic framework for specifying target-based policy languages. In the duration of this thesis, we analyse and modify PTaCL significantly, suggesting a number of alterations to improve the overall functionality of the base PTaCL language.

Finally, we introduced multi-valued logics, and formally defined key concepts based on results of Jobe [24], for characterising properties of multi-valued logics, such as *canonical suitability, functional completeness, selection operators, normal form* and *canonical completeness*. These concepts are integral to the work presented in this thesis, and enable us to leverage properties of multi-valued logics to design tree-structured ABAC languages with desirable properties.

Chapter 3

Completeness of XACML

XACML [35, 36, 37] is the most commonly used authorization language for implementing attribute-based access control in the real world. Given its popularity, the XACML standard has been the focus of a significant amount of work in the literature [29, 34, 40]. A number of these highlight the various shortcomings of XACML, such as: the ambiguous behaviour of the "indeterminate" decision, and the poorly defined and counter-intuitive semantics [29, 34]. The main contribution of this chapter is to extend this work, showing that XACML is not functionally complete and that the set of extended indeterminate values is not a suitable method for handling errors in an unambiguous and uniform way.

In the following section, we summarise the historical development and role of the indeterminate decision in XACML, highlighting the undesirable behaviour that the indeterminate decision exhibits during policy evaluation in XACML versions 1.0 and 2.0. In an attempt to fix these issues, XACML 3.0 introduces the extended indeterminate decision set. However, we argue that this extended decision set is still inadequate, as it confuses the role of error reporting with authorization decisions, and suffers from poorly defined semantics. In Section 3.2 we prove there is significant duplication and redundancy in the XACML 3.0 rule- and policy-combining algorithms. In particular, we show only two XACML rule-combining algorithms are required to express all of the XACML rule-combining algorithms. We then conduct a detailed investigation into the expressiveness of these algorithms, demonstrating that XACML is not functionally complete in Section 3.3. We consider which binary operators can be constructed using the XACML operators in Section 3.4. Finally, in Section 3.5 we briefly discuss the advantages of replacing the XACML combining algorithms with the PTaCL operators, through the use of custom combining algorithms to make XACML functionally complete.

We have published the majority of the work presented in Sections 3.2 – 3.5 [15].

3.1 Indeterminacy in XACML

We briefly introduced the indeterminate decision in Section 2.2, however we now explore in more detail the evolution of the indeterminate decision through versions 1.0 to 3.0 of the XACML standard.

3.1.1 Versions 1.0 and 2.0

The XACML 1.0 standard [35] introduces a fourth authorization decision "indeterminate" (in addition to the standard allow, deny and not-applicable decisions found in many access control languages), and the use of this decision is continued in the XACML 2.0 standard [36]. The indeterminate decision, which we denote by ?, is used to indicate that errors have occurred during policy evaluation, meaning a decision could not be reached. There are four combining algorithms defined in XACML 1.0 that use the indeterminate value:

- deny-overrides;
- permit-overrides;
- first-applicable;
- only-one-applicable.

The first three combining algorithms listed are defined for rules and policies, while the latter is defined only for policies. XACML 2.0 defines two additional combining algorithms, ordered versions of deny- and permit-overrides, whose behaviour is identical to the unordered version with one exception: the order in which the collection of rules (policies) is evaluated must match the order listed in the policy (policy set).

In XACML versions 1.0 and 2.0 the combining algorithms for deny-overrides and permit-overrides behave differently, dependent on whether they are combining rules or policies. The rule-combining algorithms (RCA) and policy-combining algorithms (PCA) differ with respect to how they handle errors during evaluation. This is reflected in their use of the indeterminate decision. When evaluating the combination of rules, the RCA looks at the effect (decision) of the rule where an error occurred, and uses this in the evaluation of subsequent rules. We represent the decision from a rule that encounters an error which could have evaluated to 0 (1) by the notation $?_0$ ($?_1$). These decisions are commonly referred to as "Indeterminate Deny" and "Indeterminate Permit" respectively. We use the subscripts r and p to differentiate the RCA and PCA versions of `do` and `po`. The decision tables for deny-overrides, permit-overrides, first-applicable and only-only-applicable are depicted in Figure 3.1.

Looking at the effect of a rule which encountered an error during evaluation was the first attempt by XACML at using the "extended set of indeterminate

do_r	0	1	$\bot$	$?_0$	$?_1$
0	0	0	0	0	0
1	0	1	1	?	1
$\bot$	0	1	$\bot$	?	?
$?_0$	0	?	?	?	?
$?_1$	0	1	?	?	?

(a) Deny-overrides RCA

do_p	0	1	$\bot$	?
0	0	0	0	0
1	0	1	1	0
$\bot$	0	1	$\bot$	0
?	0	0	0	0

(b) Deny-overrides PCA

po_r	0	1	$\bot$	$?_0$	$?_1$
0	0	1	0	0	?
1	1	1	1	1	1
$\bot$	0	1	$\bot$	?	?
$?_0$	0	1	?	?	?
$?_1$	?	1	?	?	?

(c) Permit-overrides RCA

po_p	0	1	$\bot$	?
0	0	1	0	0
1	1	1	1	1
$\bot$	0	1	$\bot$	?
?	0	1	?	?

(d) Permit-overrides PCA

fa	0	1	$\bot$	?
0	0	0	0	0
1	1	1	1	1
$\bot$	0	1	$\bot$	?
?	?	?	?	?

(e) First-applicable

ooa	0	1	$\bot$	?
0	?	?	0	?
1	?	?	1	?
$\bot$	0	1	$\bot$	?
?	?	?	?	?

(f) Only-one-applicable

Figure 3.1: XACML 1.0 and 2.0 combining algorithms

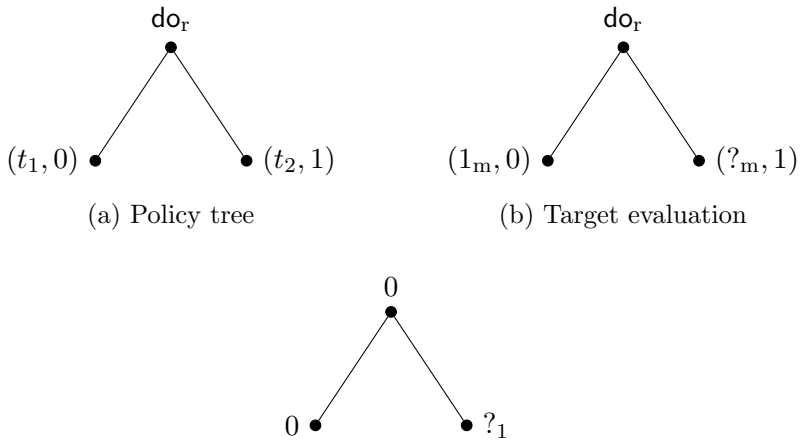
values", which was formally defined in the XACML 3.0 standard [37]. However, in XACML versions 1.0 and 2.0 it is done on an ad hoc basis, the extended indeterminate values $?_0$ and $?_1$ are only present during evaluation, they are never returned as a final decision nor are they visible to the user. As a result, information is lost by the reduction from $?_0$ and $?_1$ to the basic indeterminate decision ?.

Furthermore, the definitions of do_p and po_p are not consistent with each other. In do_p no indeterminate decisions are returned by policy evaluation, the algorithm is defined to take a "deny-by-default" approach if an indeterminate decision is encountered (final row and column of Figure 3.1b). On the other hand, po_p returns an indeterminate decision for the evaluations $? \; \mathsf{po}_\mathrm{p} \; \bot = \bot \; \mathsf{po}_\mathrm{p} \; ? = ? \; \mathsf{po}_\mathrm{p} \; ?$. There seems to be little sense in returning an indeterminate decision, when do_p, the "logical opposite" of po_p does not. Perhaps then, adopting the approach of do_p, po_p should return 1 instead of ? for these evaluations. However, this approach is undesirable, as the combination of a not-applicable policy with a policy that encounters an error returns a permit decision, which could be utilised by malicious actors. A malicious actor may then attempt to force an error in policy evaluation (by supplying

incorrectly formed attributes for example) in conjunction with a policy that they know will not be applicable, to gain a permit decision. Thus we suggest a more conservative approach, adopting the "deny-by-default" approach of do_{p}.

The idea of looking at the decision of a rule whose evaluation produced an error, and using this decision in the subsequent evaluation is not new [11, 12]. Similar methods are employed as a way to allow policy evaluation to "fail gracefully". There are scenarios where encountering an error in policy evaluation need not produce an error for the entire policy, it may still be possible to reach a conclusive decision. Consider, for example, the policy shown in Figure 3.2, which contains two rules $r_1 = (t_1, 0)$ and $r_2 = (t_2, 1)$, and suppose that

$$\tau_q(t_1) = 1_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_2) = ?_{\mathrm{m}}.$$



(a) Policy tree

(b) Target evaluation

(c) Policy evaluation

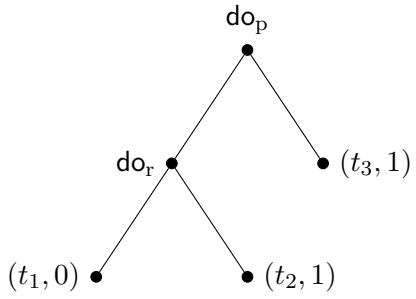
Figure 3.2: Example indeterminate deny-overrides policy

The first phase of request evaluation results in the tree shown in Figure 3.2b. The second phase of policy evaluation is shown in Figure 3.2c. As the rule r_2 encountered an error in evaluation and the contained effect is permit, the rule returns the indeterminate permit value $?_1$. When this is combined with 0 using do_{r}, the policy returns 0. Hence, a conclusive decision was reached, despite rule r_2 producing an error. Furthermore, the decision returned is the intuitively "correct" decision. If r_2 had evaluated without issue, it would return either 1 (if the rule was applicable) or $\perp$ (if the rule was not-applicable). In either case, we have $0\ \mathsf{do}_{\mathrm{r}}\ 1 = 0\ \mathsf{do}_{\mathrm{r}}\ \perp = 0$, the same outcome obtained from the evaluation which contained an error.

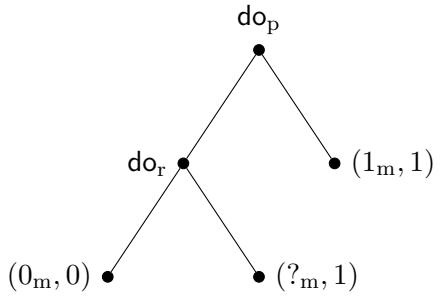
While the example presented in Figure 3.2 demonstrates the merit of the indeterminate decision, there are cases when an unintended decision is returned. Li *et al.* [29] constructed an example which highlights the issue in having different rule- and policy-combining algorithms defined for deny-overrides.

Consider the policy shown in Figure 3.3a and suppose that

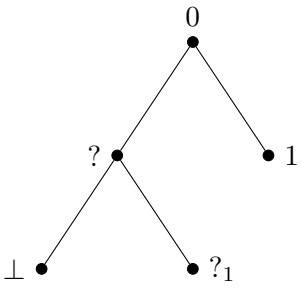
$$\tau_q(t_1) = 0_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_2) = ?_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_3) = 1_{\mathrm{m}}.$$



(a) Policy tree



(b) Target evaluation



(c) Policy evaluation

Figure 3.3: Example deny-overrides policy

The result from evaluating this policy is 0 (shown in Figure 3.3c), however this is undesirable since the only deny rule $(t_1, 0)$ is not applicable, and there is a policy $(t_3, 1)$ which permits the request. In addition, if $(t_2, 1)$ had evaluated without error, it would return either 1 (if the rule was applicable) or $\perp$ (if the rule was not applicable). In either case, the resulting policy evaluation would return 1. The introduction of the indeterminate value causes this undesirable behaviour, coupled with the fact that the rule- and policy-combining algorithms for deny-overrides differ.

3.1.2 Version 3.0

As a result of the work by Li *et al.* [29] and other work in the literature [34] highlighting the undesirable behaviour of the indeterminate decision ? and the differing rule- and policy-combining algorithms, changes were made in the XACML 3.0 standard [37]. An extended set of indeterminate values $\{?_0, ?_1, ?_{01}\}$ was introduced, where

- $?_0$ – represents an indeterminate decision from a policy or rule which could have evaluated to 0 but not 1;
- $?_1$ – represents an indeterminate decision from a policy or rule which could have evaluated to 1 but not 0;

- $?_{01}$ – represents an indeterminate decision from a policy or rule which could have evaluated to 1 or 0.

In addition to the introduction of the extended set of indeterminate values, the deny- and permit-overrides rule- and policy-combining algorithms from XACML versions 1.0 and 2.0 were deprecated to "legacy" versions. These versions were only included to provide backwards compatibility with existing systems, the use of these legacy algorithms is strongly discouraged. New versions of deny- and permit-overrides where introduced, which made use of the extended set of indeterminate values. The decision tables for the updated algorithms are shown in Figure 3.4.

do	0	1	$\perp$	$?_0$	$?_1$	$?_{01}$
0	0	0	0	0	0	0
1	0	1	1	$?_{01}$	1	$?_{01}$
$\perp$	0	1	$\perp$	$?_0$	$?_1$	$?_{01}$
$?_0$	0	$?_{01}$	$?_0$	$?_0$	$?_{01}$	$?_{01}$
$?_1$	0	1	$?_1$	$?_{01}$	$?_1$	$?_{01}$
$?_{01}$	0	$?_{01}$	$?_{01}$	$?_{01}$	$?_{01}$	$?_{01}$

(a) Deny-overrides 3.0

po	0	1	$\perp$	$?_0$	$?_1$	$?_{01}$
0	0	1	0	0	$?_{01}$	$?_{01}$
1	1	1	1	1	1	1
$\perp$	0	1	$\perp$	$?_0$	$?_1$	$?_{01}$
$?_0$	0	1	$?_0$	$?_0$	$?_{01}$	$?_{01}$
$?_1$	$?_{01}$	1	$?_1$	$?_{01}$	$?_1$	$?_{01}$
$?_{01}$	$?_{01}$	1	$?_{01}$	$?_{01}$	$?_{01}$	$?_{01}$

(b) Permit-overrides 3.0

Figure 3.4: XACML 3.0 algorithms

However, the indeterminate decision is used in XACML 3.0 for more than reporting errors. XACML uses the indeterminate value in two distinct ways:

1. as a decision returned (during normal evaluation) by the "only-one-applicable" policy-combining algorithm; and

2. as a decision returned when some (unexpected) error has occurred in policy evaluation.

In the second case, the indeterminate value is used to represent alternative outcomes of policy evaluation (had the error not occurred). We believe that the two situations described are quite distinct and require different policy semantics. However, the semantics of indeterminacy in XACML are confused because (i) the indeterminate value is used in two different ways, as described above, and (ii) there is no clear and uniform way of establishing the values returned by the combining algorithms when an indeterminate value is encountered.

Thus, in the remainder of this chapter we restrict the scope of the XACML combining algorithms to the decision set $\{0, \perp, 1\}$, omitting the extended indeterminate values. In Chapters 5 and 6 we explore the use of a fourth authorization decision representing a "conflict", which can be used in operators such as only-one-applicable and unanimity. In addition, we propose an alternative method to indeterminate decisions to handle errors in policy

evaluation, through the use of sets of decisions. Hence, we separate the two scenarios discussed above, providing well-defined semantics for each, opposed to the approach taken in the XACML standard.

3.2 Dependencies between the combining algorithms

In this section, we investigate what decision operators can be constructed using the XACML 3.0 combining algorithms. In doing so, we characterize the expressive power of XACML policies. Having restricted the scope of the XACML combining algorithms to the 3-valued decision set $\{0, \bot, 1\}$, we note that the decision tables for the ordered, unordered and legacy versions of do, po, dup and pud are identical for the decision set $\{0, \bot, 1\}$ (see Section 2.2.2). Hence, for investing the expressivity of the XACML combining algorithms, we may restrict our attention to do, po, dup, pud and fa (and ignore the ordered and legacy versions). We reiterate Figure 2.3 from Section 2.2.2 in Figure 3.5 for ease of reference, as the results presented in this section will rely on information contained in Figure 3.5.

do	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	$\bot$

po	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	$\bot$

dup	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	0

pud	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	1

fa	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	$\bot$

(a) Decision tables

Operator	Idempotent	$\cup$-operator	Commutative	Conclusive
do, po	Yes	Yes	Yes	No
dup, pud	No	No	Yes	No
fa	Yes	Yes	No	No

(b) Properties

Figure 3.5: XACML rule-combining algorithms do, po, dup, pud and fa

We begin our investigation into dependencies with the following remark:

Remark 3.2.1. *Any operator constructed using* fa*,* po *and* do *will be an idempotent $\cup$-operator, since $x \oplus \bot = \bot \oplus x = x$ for $\oplus \in \{\mathsf{fa}, \mathsf{po}, \mathsf{do}\}$. A corollary of this observation is that the operators* do*,* po *and* fa *are indistinguishable when one of the arguments is $\bot$.*

This seems somewhat counterintuitive, creates redundancy, and also sug-

gests that combinations of XACML operators will tend to behave in similar ways. In the rest of this section, we confirm these observations.

Interestingly, despite the fact that do and po are commutative operators, it is possible to construct non-commutative operators by combining these two operators. Specifically, we have the following, somewhat unexpected, result, which asserts that the fa algorithm is redundant.

Proposition 3.2.1. *For all rules r and r',*

$$r \;\mathsf{fa}\; r' \equiv r \;\mathsf{po}\; (r \;\mathsf{do}\; r').$$

Proof. The proof follows by inspection of the decision table in Figure 3.6. □

d_1	d_2	d_1 do d_2	d_1 po (d_1 do d_2)	d_1 fa d_2
0	0	0	0	0
0	1	0	0	0
0	$\perp$	0	0	0
1	0	0	1	1
1	1	1	1	1
1	$\perp$	1	1	1
$\perp$	0	0	0	0
$\perp$	1	1	1	1
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Figure 3.6: Encoding fa using do and po

Remark 3.2.2. *The ability to construct* fa *from* do *and* po *arises from the fact that* do *and* po *do not obey the identity $x \oplus (y \otimes z) = (x \oplus y) \otimes (x \oplus z)$, usually known as the* distributive law. *Specifically,*

$$0 \;\mathsf{po}\; (1 \;\mathsf{do}\; \perp) = 0 \;\mathsf{po}\; 1 = 1, \textit{ whereas}$$
$$(0 \;\mathsf{po}\; 1) \;\mathsf{do}\; (0 \;\mathsf{po}\; \perp) = 1 \;\mathsf{do}\; 0 = 0;$$

and

$$1 \;\mathsf{do}\; (0 \;\mathsf{po}\; \perp) = 1 \;\mathsf{do}\; 0 = 0, \textit{ whereas}$$
$$(1 \;\mathsf{do}\; 0) \;\mathsf{po}\; (1 \;\mathsf{do}\; \perp) = 0 \;\mathsf{po}\; 1 = 1.$$

We now show that dup and pud are also redundant. We first define the rules **1** and **0**, where, for all requests q,

$$\rho_q(\mathbf{1}) = 1 \quad \text{and} \quad \rho_q(\mathbf{0}) = 0.$$

Rule **1** may be realized in XACML by defining a rule such that the rule is applicable to every request and its effect is "permit"; rule **0** may be realized

in an analogous way. We can then define the unary operators deny-by-default (dbd) and permit-by-default (pbd), where

$$\mathsf{dbd}(r) \equiv (\mathbf{0}\ \mathsf{po}\ r) \quad \text{and} \quad \mathsf{pbd}(r) \equiv (\mathbf{1}\ \mathsf{do}\ r).$$

Note that dbd and pbd may be viewed as unary operators on the decision set $\{0, 1, \bot\}$, where $\mathsf{dbd}(x) = \mathsf{pbd}(x) = x$ if $x \in \{0, 1\}$; $\mathsf{dbd}(\bot) = 0$; and $\mathsf{pbd}(\bot) = 1$. We can use dbd and pbd to construct pud and dup.

Proposition 3.2.2. *For all rules r and r',*

$$r\ \mathsf{pud}\ r' \equiv \mathsf{pbd}(r\ \mathsf{do}\ r') \quad \textit{and} \quad r\ \mathsf{dup}\ r' \equiv \mathsf{dbd}(r\ \mathsf{po}\ r').$$

Proof. The proof for pud follows by inspection of the decision table in Figure 3.7a. A similar decision table for dup can be seen in Figure 3.7b. □

d_1	d_2	d_1 do d_2	pbd(d_1 do d_2)	d_1 pud d_2
0	0	0	0	0
0	1	1	0	0
0	$\bot$	1	0	0
1	0	0	0	0
1	1	1	1	1
1	$\bot$	1	1	1
$\bot$	0	0	0	0
$\bot$	1	1	1	1
$\bot$	$\bot$	$\bot$	1	1

(a) pud using do and pbd

d_1	d_2	d_1 po d_2	dbd(d_1 po d_2)	d_1 dup d_2
0	0	0	0	0
0	1	1	1	1
0	$\bot$	0	0	0
1	0	1	1	1
1	1	1	1	1
1	$\bot$	1	1	1
$\bot$	0	0	0	0
$\bot$	1	1	1	1
$\bot$	$\bot$	$\bot$	0	0

(b) dup using po and dbd

Figure 3.7: Operator encodings

We have shown there is a significant amount of duplication and redundancy between the 11 XACML rule-combining algorithms (defined in Table 2.1). In particular, we have shown it is sufficient, for the purposes of constructing new decision operators, to consider the decision operators do and po, together with

the constant rules **0** and **1**, and the unary operators dbd and pbd.

3.3 Incompleteness

Any XACML policy set is constructed by combining XACML policies using policy-combining algorithms. The decision obtained by evaluating an XACML policy set is determined by the action of the policy-combining algorithm on decisions. Given the way in which policy evaluation works in XACML, this is equivalent to asking what functions can we build using $\{\mathbf{0}, \mathbf{1}, \mathsf{dbd}, \mathsf{pbd}, \mathsf{do}, \mathsf{po}\}$. We have seen that do and po essentially act as logical AND and OR on the set $\{0, 1\}$; and we have seen that we can define two unary operators (dbd and pbd) for policies, which correspond to unary operators on $\{0, 1, \bot\}$.

There are two types of operators, likely to be useful in practice, that cannot be constructed using the XACML operators.

- A $\cap$-operator $\oplus$ has the property that $x \oplus \bot = \bot \oplus x = \bot$ for any $x \in \{0, 1, \bot\}$. In this context, we do not make a conclusive decision if at least one of the inputs is unknown. The operators do' and po' in Figure 3.8 are examples of this type of operator.

- The second type of operator has the property that a conclusive decision is returned whenever one is implied by at least one of the arguments. In this context, $\bot$ is interpreted as a value that could be either 0 or 1 but is not known at the time of evaluation. Thus $1 \,\mathsf{po}\, \bot = \bot \,\mathsf{po}\, 1 = 1$, since $1 \,\mathsf{po}\, x = x \,\mathsf{po}\, 1 = 1$ for any $x \in \{0, 1\}$; similarly $0 \,\mathsf{do}\, \bot = \bot \,\mathsf{do}\, 0 = 0$. The operators do'' and po'' in Figure 3.8 are examples of this type of operator. Note that do'' is equivalent to $\wedge_{\mathrm{p}}$ (the conjunction operator in PTaCL).

do'	0	1	$\bot$
0	0	0	$\bot$
1	0	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

do''	0	1	$\bot$
0	0	0	0
1	0	1	$\bot$
$\bot$	0	$\bot$	$\bot$

po'	0	1	$\bot$
0	0	1	$\bot$
1	1	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

po''	0	1	$\bot$
0	0	1	$\bot$
1	1	1	1
$\bot$	$\bot$	1	$\bot$

Figure 3.8: Commutative, idempotent operators that cannot be constructed using XACML operators

More formally, we have the following result.

Proposition 3.3.1. *It is not possible to construct $\cap$-operators using the XACML operators.*

Proof. The proof follows from the following observations: (i) all the binary XACML operators are $\cup$-operators; (ii) any combination of $\cup$-operators is itself a $\cup$-operator (since $x \oplus \perp = x$ for any $x \in \{0, 1, \perp\}$ and any $\cup$-operator $\oplus$); (iii) the two unary operators pbd and dbd remove $\perp$; and (iv) $x \oplus \perp \neq \perp$ and $\perp \oplus x \neq \perp$ for any $x \in \{0, 1\}$ for any $\cup$-operator $\oplus$. Thus it is impossible to construct an operator in which $x \oplus \perp = \perp$ for any $x \neq \perp$. □

Thus, we immediately have the following corollary.

Corollary 3.3.1. *XACML is not functionally or canonically complete.*

We have shown there is a significant amount of duplication and redundancy in the XACML rule-combining algorithms. Specifically, only do and po are required to express all 11 combining algorithms. We have also shown that XACML is not functionally or canonically complete. In particular, there are operators of practical relevance that cannot be constructed using the XACML operators, and XACML does not permit a normal form representation for policies. It is interesting to note that there is no way to negate policy decisions in XACML. Quite apart from the general incompleteness of XACML, the inability to negate decisions seems to be significant practical drawback to XACML, as negation is a useful unary policy operator in practice.

We note that the result presented in Corollary 3.3.1 applies to a simplified version of XACML, without indeterminate decisions or conditions, and excludes the use of custom combining algorithms. Throughout this chapter we have justified these assumptions: (i) the exclusion of the indeterminate decision is based on the logic that it is used for conflicting purposes within XACML, sometimes as a method for handling errors in policy evaluation, and sometimes as a core decision in combining algorithms; (ii) conditions in XACML are inadequately restricted, a condition can be any Boolean expression, including arbitrarily complex functions, which in practice means that it is possible to write a full program in the condition of a rule; and (iii) the use of custom combining algorithms needs to be carefully applied and used to avoid policy misconfigurations.

3.4 Constructable binary operators

We now consider which binary operators can be constructed using the XACML operators. We will write $+$ and $-$ to denote pbd and dbd, respectively, in order to simplify the notation. There are four possible idempotent, $\cup$-operators that can be constructed using the XACML operators: for all $x \in D, x \oplus x$, $x \oplus \perp$ and $\perp \oplus x$ are pre-determined; only $0 \oplus 1$ and $1 \oplus 0$ may vary. These operators are do, po, fa and what we might call "last-applicable" (la).[1] We

[1] Although fa (and hence la) is a redundant operator, we continue their use as a compact way of expressing operators (and later families of operators) instead of the lengthy expressions using do and po.

have $r_1 \,\mathsf{la}\, r_2 \equiv r_2 \,\mathsf{fa}\, r_1$, so we can construct each of these operators using the XACML operators. The commutative, idempotent $\cup$-operators are do and po.

We now consider operators having the general form $\diamond_1((\diamond_2 d_1) \oplus (\diamond_3 d_2))$ where $\diamond_1, \diamond_2, \diamond_3 \in \{-, +, \text{“”}\}$ (“” is used to denote the absence of a unary operator) and $\oplus \in \{\mathsf{do}, \mathsf{po}, \mathsf{fa}, \mathsf{la}\}$. If either $\diamond_2$ or $\diamond_3$ are $-$ or $+$, the application of $\diamond_1$ has no effect as the operator $(\diamond_2 d_1) \oplus (\diamond_3 d_2)$ will be conclusive (since $\diamond d \in \{0, 1\}$ and $x \oplus \bot = \bot \oplus x = x$ for any $\oplus \in \{\mathsf{do}, \mathsf{po}, \mathsf{fa}, \mathsf{la}\}$ and any $x \in \{0, 1\}$). This has the effect of limiting the number of possible operators of this form. The possible choices for $\diamond_1, \diamond_2, \diamond_3$ and $\oplus$ are tabulated in Table 3.1, which results in 44 quasi-idempotent operators.

$\diamond_1$	$\diamond_2$	$\diamond_3$	$\oplus$	Possible Ops
“”	$+$	3	4	12
“”	$-$	3	4	12
“”	“”	2	4	8
3	“”	“”	4	12

Table 3.1: Choices for $\diamond_1, \diamond_2, \diamond_3$ and $\oplus$

However, not all of these operators are unique, given the following equivalences between operators.

Proposition 3.4.1. *For any $x, y \in \{0, 1, \bot\}$,*

$$
\begin{aligned}
&(-x) \,\mathsf{po}\, y = x \,\mathsf{po}\, (-y) = -(x \,\mathsf{po}\, y) = (-x) \,\mathsf{po}\, (-y);\\
&(+x) \,\mathsf{po}\, y = (+x) \,\mathsf{po}\, (-y);\\
&x \,\mathsf{po}\, (+y) = (-x) \,\mathsf{po}\, (+y);\\
&(+x) \,\mathsf{do}\, y = x \,\mathsf{do}\, (+y) = +(x \,\mathsf{do}\, y) = (+x) \,\mathsf{do}\, (+y);\\
&(-x) \,\mathsf{do}\, y = (-x) \,\mathsf{do}\, (+y);\\
&x \,\mathsf{do}\, (-y) = (+x) \,\mathsf{do}\, (-y).
\end{aligned}
$$

These results follow by inspection of the relevant decision tables. The intuition behind the first three results is that $0 \,\mathsf{po}\, x = \bot \,\mathsf{po}\, x$ for all $x \in \{0, 1\}$; an analogous observation holds for the second three.

Then we can construct the following operators using do and different combinations of the unary operators $-$ and $+$:

$$
\begin{aligned}
x \,\mathsf{do}_0\, y &\stackrel{\text{def}}{=} x \,\mathsf{do}\, y & \qquad x \,\mathsf{do}_1\, y &\stackrel{\text{def}}{=} (-x) \,\mathsf{do}\, (-y)\\
x \,\mathsf{do}_2\, y &\stackrel{\text{def}}{=} (-x) \,\mathsf{do}\, y & \qquad x \,\mathsf{do}_3\, y &\stackrel{\text{def}}{=} x \,\mathsf{do}\, (-y)\\
x \,\mathsf{do}_4\, y &\stackrel{\text{def}}{=} -(x \,\mathsf{do}\, y) & \qquad x \,\mathsf{do}_5\, y &\stackrel{\text{def}}{=} +(x \,\mathsf{do}\, y)
\end{aligned}
$$

These operators comprise what we call the deny-overrides family of operators. These operators are all distinct and operate on $\{0, 1\}$ in exactly the same way

as do. Moreover,

$$0 \,\mathsf{do}_i \perp = \perp \mathsf{do}_i \, 0 = 0$$

for all i. They differ in their effect on elements in $\{1, \perp\}$, as shown in Figure 3.9. Notice that do_5 is equivalent to three other operators (by Proposition 3.4.1). Note that do_2 and do_3 are not commutative.

do_0	1	$\perp$
1	1	1
$\perp$	1	$\perp$

do_1	1	$\perp$
1	1	0
$\perp$	0	0

do_2	1	$\perp$
1	1	1
$\perp$	0	0

do_3	1	$\perp$
1	1	0
$\perp$	1	0

do_4	1	$\perp$
1	1	1
$\perp$	1	0

do_5	1	$\perp$
1	1	1
$\perp$	1	1

Figure 3.9: The family of deny-overrides operators

Analogously, we can define a family of six permit-overrides operators which act on $\{1, \perp\}$ in exactly the same way as the deny-overrides operators in Figure 3.9. Therefore, in total, we can construct six quasi-idempotent deny-overrides operators and six quasi-idempotent permit-overrides operators, of which one is idempotent and four are commutative.

In a similar manner we can identify the first-applicable and last-applicable families, consisting of operators obtained using fa and la respectively. We observe the following equivalences between operators.

Proposition 3.4.2. *For any $x, y \in \{0, 1, \perp\}$, $\diamond \in \{-, +\}$,*

$$\begin{aligned}
&(\diamond x) \,\mathsf{fa}\, y = (\diamond x) \,\mathsf{fa}\, (-y) = (\diamond x) \,\mathsf{fa}\, (+y);\\
&-(x \,\mathsf{fa}\, y) = x \,\mathsf{fa}\, (-y);\\
&+(x \,\mathsf{fa}\, y) = x \,\mathsf{fa}\, (+y);\\
&x \,\mathsf{la}\, (\diamond y) = (-x) \,\mathsf{la}\, (\diamond y) = (+x) \,\mathsf{la}\, (\diamond y);\\
&-(x \,\mathsf{la}\, y) = (-x) \,\mathsf{la}\, y;\\
&+(x \,\mathsf{la}\, y) = (+x) \,\mathsf{fa}\, y.
\end{aligned}$$

These results follow by inspection of the relevant decision tables. Unlike the deny-overrides and permit-overrides families, we obtain only five distinct operators for the first/last-applicable families. This is clear from the restrictions placed on the decision tables, and follows immediately from the equivalences in Proposition 3.4.1. Thus, in total, the 44 possible operators actually represent 22 distinct binary operators. The 44 operators and their duplicate forms are tabulated in Table 3.2.

Thus far, we only considered operators having the form

$$\diamond_1((\diamond_2 d_1) \oplus (\diamond_3 d_2)).$$

Op	Construction	Alternative forms
do_0	$x\ \mathsf{do}\ y$	
do_1	$(-x)\ \mathsf{do}\ (-y)$	
do_2	$(-x)\ \mathsf{do}\ y$	$(-x)\ \mathsf{do}\ (+y)$
do_3	$x\ \mathsf{do}\ (-y)$	$(+x)\ \mathsf{do}\ (-y)$
do_4	$-(x\ \mathsf{do}\ y)$	
do_5	$+(x\ \mathsf{do}\ y)$	$(+x)\ \mathsf{do}\ (+y), (+x)\ \mathsf{do}\ y, x\ \mathsf{do}\ (+y)$
po_0	$x\ \mathsf{po}\ y$	
po_1	$(+x)\ \mathsf{po}\ (+y)$	
po_2	$(+x)\ \mathsf{po}\ y$	$(+x)\ \mathsf{po}\ (-y)$
po_3	$x\ \mathsf{po}\ (+y)$	$(-x)\ \mathsf{po}\ (+y)$
po_4	$-(x\ \mathsf{po}\ y)$	
po_5	$+(x\ \mathsf{po}\ y)$	$(-x)\ \mathsf{po}\ (-y), (-x)\ \mathsf{do}\ y, x\ \mathsf{do}\ (-y)$
fa_0	$x\ \mathsf{fa}\ y$	
fa_1	$(-x)\ \mathsf{fa}\ y$	$(-x)\ \mathsf{fa}\ (-y), (-x)\ \mathsf{fa}\ (+y)$
fa_2	$(+x)\ \mathsf{fa}\ y$	$(+x)\ \mathsf{fa}\ (-y), (+x)\ \mathsf{fa}\ (+y)$
fa_3	$-(x\ \mathsf{fa}\ y)$	$x\ \mathsf{fa}\ (-y)$
fa_4	$+(x\ \mathsf{fa}\ y)$	$x\ \mathsf{fa}\ (+y)$
la_0	$x\ \mathsf{la}\ y$	
la_1	$x\ \mathsf{la}\ (-y)$	$(-x)\ \mathsf{la}\ (-y), (+x)\ \mathsf{la}\ (-y)$
la_2	$x\ \mathsf{la}\ (+y)$	$(-x)\ \mathsf{la}\ (+y), (+x)\ \mathsf{la}\ (+y)$
la_3	$-(x\ \mathsf{la}\ y)$	$(-x)\ \mathsf{la}\ y$
la_4	$+(x\ \mathsf{la}\ y)$	$(+x)\ \mathsf{la}\ y$

Table 3.2: Operator constructions and alternative forms

It is not obvious that these are the only forms that yield new, distinct binary operators. These forms only contain single instances of each decision variable d_1 and d_2, which raises the question of whether new operators can be constructed from forms which contain multiple instances of d_1 and d_2. Recall the definition of fa ($x\ \mathsf{fa}\ y \equiv x\ \mathsf{po}\ (x\ \mathsf{do}\ y)$), which is constructed using more than one instance of x. We now investigate whether the inclusion of multiple instances of d_1 and d_2 yields any further operators.

To answer this question, we developed a program with the aim of enumerating all constructible binary operators by brute force (see Appendix A.1). The program works by generating all operators that can be created by combining other operators. The program generates all binary operators which have the general form $\diamond\, x \oplus \triangle\, y$ where $\diamond, \triangle \in \{-, +, \text{“”}\}$ and $\oplus \in \{\mathsf{do}, \mathsf{po}\}$. Note we omit fa and la from the set of binary operators, as these operators (being expressible in terms of do and po) will be generated automatically as we recursively create operators. We initialize the array variables $x = [0, 0, 0, 1, 1, 1, \bot, \bot, \bot]$ and $y = [0, 1, \bot, 0, 1, \bot, 0, 1, \bot]$. We store the decision table of a binary operator in a similar array variable. We generate the $3 \times 2 \times 3 = 18$ decision tables for operators of the form $\diamond\, x \oplus \triangle\, y$, of which 6 are duplicates. We remove the duplicates, storing the decision tables for the remaining 12 operators in an array.

The process is repeated with each item in the array being reused as an input for x and y in the general form of a binary operator. This second iteration generates $12^2 \times 18 = 2592$ operators, of which 22 are distinct operators. We once again reuse these operators as inputs for x and y, yielding the the same 22 distinct operators. As no new operators are generated, the program terminates. The 22 operators discovered via exhaustive search correspond exactly to the operators we constructed above. The decision tables for these operators are listed in Figure 3.10.

To summarise, there are only 22 binary quasi-idempotent operators that can be constructed from the XACML operators (of the 192 that are possible). These operators fall into one of four families: (i) six do operators; (ii) six po operators; (iii) five fa operators; and (iv) five la operators.

3.5 PTaCL operators

Crampton and Morisset [12] showed that the three-valued logic expressed over the set $\{0, 1, \bot\}$ and defined by the operators $\wedge_{\mathrm{p}}, \neg$ and $\sim$ (see Figure 2.5) is functionally complete. Essentially, they proved that the PTaCL operators correspond to the operators of a logic that was known to be functionally complete.

Given that PTaCL is functionally complete, there appears to be a good case for using the PTaCL operators in a language like XACML. The unary operator $\sim$ is already implicitly defined in XACML (as dbd), thus we only need to consider adding $\wedge_{\mathrm{p}}$ and $\neg$ to the minimal set of XACML combining algorithms $\{\mathsf{do}, \mathsf{po}\}$. (Recall fa, pud and dup can be defined in terms of do and po.) It is easy to see that we cannot achieve functional completeness by adding just $\neg$ or just $\wedge_{\mathrm{p}}$ to the set of XACML operators. In the case of $\neg$, we would still be unable to construct $\cap$-operators, as there is still no operator that can change a conclusive decision into $\bot$. On the other hand, if we include $\wedge_{\mathrm{p}}$ but not $\neg$, we are unable to reverse the 0 and 1 decisions. In short, we must include both operators if we wish to make XACML functionally complete.

Given that PTaCL is functionally complete anyway, it seems pointless to provide po or do (nor any of the other 10 XACML operators). In particular, recall the following operators defined in Section 2.3.2:

$$
\begin{aligned}
d \vee_{\mathrm{p}} d' &\stackrel{\text{def}}{=} \neg((\neg d) \wedge_{\mathrm{p}} (\neg d'));\\
d \,\mathsf{po}\, d' &\stackrel{\text{def}}{=} (d \vee_{\mathrm{p}} (\sim d')) \wedge_{\mathrm{p}} ((\sim d) \vee_{\mathrm{p}} d');\\
d \,\mathsf{do}\, d' &\stackrel{\text{def}}{=} \neg((\neg d) \,\mathsf{po}\, (\neg d')).
\end{aligned}
$$

In other words, there appears to be a good case, at least from the perspective of functional completeness, for defining only three policy operators in an ABAC language such as XACML: negation, deny-by-default, and a form of deny-

do_0	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	$\bot$

do_1	0	1	$\bot$
0	0	0	0
1	0	1	0
$\bot$	0	0	0

do_2	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	0	0

do_3	0	1	$\bot$
0	0	0	0
1	0	1	0
$\bot$	0	1	0

do_4	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	0

do_5	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	1

(a) Deny-overrides family

po_0	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	$\bot$

po_1	0	1	$\bot$
0	0	1	1
1	1	1	1
$\bot$	1	1	1

po_2	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	1	1	1

po_3	0	1	$\bot$
0	0	1	1
1	1	1	1
$\bot$	0	1	1

po_4	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	0

po_5	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	1

(b) Permit-overrides family

fa_0	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	$\bot$

fa_1	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	0	0

fa_2	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	1	1	1

fa_3	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	0

fa_4	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	1

(c) First-applicable family

la_0	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	$\bot$

la_1	0	1	$\bot$
0	0	1	0
1	0	1	0
$\bot$	0	1	0

la_2	0	1	$\bot$
0	0	1	1
1	0	1	1
$\bot$	0	1	1

la_3	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	0

la_4	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	1

(d) Last-applicable family

Figure 3.10: Constructible Binary operators in XACML

overrides that only returns 1 when both arguments are 1.

It would be easy to write three custom XACML combining algorithms to implement the PTaCL operators. (The front-end of an XACML-based system could continue to expose specific algorithms (such as the usual deny-overrides and permit-overrides), if required by the application, but these can be compiled down into the three basic operators.) More complex policy-combining algorithms can be constructed, as required, from the three basic operators. We explore this in more detail in Chapter 6, demonstrating how we may leverage the well-defined parts of the XACML architecture and combine these with a functionally complete set of operators.

3.6 Summary and discussion

We have presented a number of shortcomings in the XACML standard in this chapter. First, we summarised the historical development and role of the indeterminate decision in XACML, originally intended as a way to handle errors in policy evaluation. However, the indeterminate decision is used for more than error handling, there are instances where it is used as a core policy decision where no errors have been encountered during evaluation. We believe this confuses the role of error handling and policy decisions, and is a symptom of poorly defined semantics.

Next, we highlighted various redundancies between the combining algorithms in XACML. In doing so, we characterise the expressive power of XACML, showing that XACML is not a functionally complete ABAC language. There are several policy operators including negation which cannot be expressed in XACML without the use of custom combining algorithms, which we believe is a significant drawback for the language.

In the interests of completeness, we demonstrated which binary policy operators can be constructed in XACML. Ultimately, after identifying various equivalences and through the use of a brute force algorithm, we showed only 22 quasi-idempotent operators can be constructed from the XACML operators (of a possible 192).

Finally, we argued for the replacement of the XACML combining algorithms with operators from an ABAC language that is known to be functionally complete, PTaCL. In doing so, we are able to leverage the XACML architecture and construct any policy operator using the functionally complete set $\{\wedge_{\mathrm{p}}, \neg, \sim\}$. However, it is still far from obvious how one would express an arbitrary policy as a policy defined using the PTaCL operators. We consider this issue in more detail in the following chapter.

Chapter 4

A Canonically Complete 3-valued PTaCL

One of the main difficulties with using a tree-structured language is writing the desired policy using the operators provided by the given language. In particular, if it is not possible to express a policy using a single target and decision, the policy author must engineer the desired policy by combining sub-policies using the set of operators specified in the given language. This is a non-trivial task, in general. Moreover, in XACML it may be impossible to write the desired policy due to its functional incompleteness. Thus, a policy author may be forced to write a policy that approximates the desired policy, which may lead to unintended or undesirable decisions for certain requests.

An alternative approach, supported by XACML, is to define custom combining algorithms. However, there is no guarantee that the addition of a new combining algorithm will make XACML functionally complete. Thus, more and more custom algorithms may be required over time. This, in turn, will make the design decisions faced by policy authors ever more complicated, thereby increasing the chances of errors and misconfigurations.

In other words, we believe it is preferable to define a small number of operators having unambiguous semantics and providing functional completeness. A functionally complete ABAC language, such as PTaCL, can be used to construct any conceivable policy using the operators provided by the language. However, policy authors still face the challenge of finding the correct way to combine sub-policies using those operators to construct the desired policy.

For example, PTaCL defines three policy operators $\neg, \sim$ and $\wedge_{\mathrm{p}}$. To express XACML's deny- and permit-overrides in PTaCL requires significant effort. In Section 2.3.2 we showed that:

$$d \,\mathsf{po}\, d' \equiv (d \vee_{\mathrm{p}} (\sim d')) \wedge_{\mathrm{p}} ((\sim d) \vee_{\mathrm{p}} d'), \text{ and}$$
$$d \,\mathsf{do}\, d' \equiv \neg((\neg d) \,\mathsf{po}\, (\neg d')).$$

The operators po and do are equivalent to the permit- and deny-overrides

policy-combining algorithms in XACML. As can be seen, the definitions of these operators in terms of the PTaCL operators are complex, and, more generally, it is a non-trivial task to derive such formulae.

We believe there will be, perhaps many, situations where the policy writer knows what decision should be returned for each authorization request, but is unable to construct the desired policy using the operators provided by the policy language. As a simple example, suppose we have policies defined by the tables in Figure 4.1. Here we are assuming there are two sub-policies P_1 and P_2, whose targets partition the set of all authorization requests. The rows represent values that are returned by evaluating P_1, while the columns represent values that are returned by evaluating P_2. Thus, if P_1 and P_2 evaluate to 0 and 1, respectively, the final result should be 1 if the policies are combined using $\oplus_1$ and $\perp$ if combined using $\oplus_2$.

$\oplus_1$	0	$\perp$	1
0	0	0	1
$\perp$	0	$\perp$	1
1	1	1	1

$\oplus_2$	0	$\perp$	1
0	0	0	$\perp$
$\perp$	0	$\perp$	$\perp$
1	$\perp$	$\perp$	1

Figure 4.1: Two combining operators for policies

Since PTaCL is functionally complete, it is possible, in principle, to construct the policies $P_1 \oplus_1 P_2$ and $P_1 \oplus_2 P_2$ by combining copies of P_1 and P_2 using the PTaCL operators $\{\neg, \sim, \wedge_{\mathrm{p}}\}$. It is also possible in XACML to define custom policy-combining algorithms to directly construct $\oplus_1$ and $\oplus_2$. However, it would be useful, both in theory and in terms of implementation, to develop an authorization language which is functionally complete, like PTaCL, and which enables a policy writer to write down any desired policy directly using the operators of the language. In propositional logic, for example, one can use the truth table for an arbitrary Boolean formula to write down a logically equivalent formula in disjunctive normal form. In short, in this chapter we wish to develop a policy authorization language that has a "normal form" in which *any* desired policy can be expressed.

In order to do this, we apply concepts introduced by Jobe [24] (see Section 2.5.1) in the study of multi-valued logics to the development of a policy language for attribute-based access control. The functional completeness of PTaCL implies that every unary selection operator has an equivalent formula in PTaCL. However, it is not clear that every unary selection operator has an equivalent formula in PTaCL *that is in normal form*. Faced with this issue, we can adopt one of two approaches:

- We could try to determine whether each selection operator does indeed have an equivalent PTaCL formula that is in normal form, thus establishing that PTaCL is canonically complete.

- Alternatively, we can ask whether the PTaCL operators could be replaced with the operators from a logic that is known to be functionally and canonically complete.

In the next section, we explore the first option, ultimately showing that PTaCL is not a canonically complete ABAC language. Following this, in Section 4.2, we investigate the second option, replacing the PTaCL operators with the operators from Jobe's logic $\mathcal{J}$, which is known to be canonically complete. We develop a variant of PTaCL, denoted by $\text{PTaCL}_{\mathsf{3}}^{\leq}$, and illustrate how this language addresses the problems faced in policy specification for languages like XACML and PTaCL discussed above. In Section 4.4 we introduce a method for handling indeterminacy in $\text{PTaCL}_{\mathsf{3}}^{\leq}$, which uses decision sets to allow policy evaluation to "fail gracefully" and present a comparison with XACML. Finally, we take inspiration from the XACML standard and formally define syntax and semantics for evaluating obligations in $\text{PTaCL}_{\mathsf{3}}^{\leq}$.

We have published the majority of the work presented in Sections 4.2 – 4.3 [15], and Sections 4.5 – 4.6 [14].

4.1 Completeness of PTaCL

We start by exploring the first option; determining whether or not PTaCL is canonically complete. Recall that PTaCL is defined on the decision set $\{0, \bot, 1\}$, and has three operators $\{\neg, \sim, \wedge_{\text{p}}\}$, defined in Figure 4.2. Canonical suitability for a 3-valued logic, depends on the ordering chosen on the set of truth values. Henceforth, we will assume a total ordering $0 < \bot < 1$. This assumption is justified by the behaviour of $\wedge_{\text{p}}$, which acts as a greatest lower bound operator with this ordering. For brevity, we denote the three valued decision set $\{0, \bot, 1\}$ with total ordering $0 < \bot < 1$ by $\mathsf{3}$. Hence, we may formally represent the "PTaCL logic" as the logic $L(\mathsf{3}, \{\neg, \sim, \wedge_{\text{p}}\})$, and explore whether this logic is canonically complete or not.

d	$\neg\, d$	$\sim d$
0	1	0
$\bot$	$\bot$	0
1	0	1

(a) $\neg$ and $\sim$

$\wedge_{\text{p}}$	0	$\bot$	1
0	0	0	0
$\bot$	0	$\bot$	$\bot$
1	0	$\bot$	1

(b) $\wedge_{\text{p}}$

Figure 4.2: Decision operators in PTaCL

It follows immediately from the functional completeness of $\{\neg, \sim, \wedge_{\text{p}}\}$ that $L(\mathsf{3}, \{\neg, \sim, \wedge_{\text{p}}\})$ is canonically suitable. Indeed, it is trivial to show that PTaCL is canonically suitable. Recall the following equivalence

$$x \vee_{\text{p}} y = \neg(\neg\, x \wedge_{\text{p}} \neg\, y).$$

Canonical suitability follows immediately, since

$$x \curlywedge y = x \wedge_{\mathrm{p}} y \quad \text{and} \quad x \curlyvee y = x \vee_{\mathrm{p}} y.$$

As 0 is the minimum truth value in the total ordering $\mathsf{3}$, the n-ary selection operator $S^j_{\mathbf{a}}$ for $\mathsf{3}$ is defined by the following function:

$$S^j_{\mathbf{a}}(\mathbf{x}) = \begin{cases} j & \text{if } \mathbf{x} = \mathbf{a}, \\ 0 & \text{otherwise.} \end{cases}$$

Functional completeness also implies all unary selection operators can be expressed as formulae in the logic $L(\mathsf{3}, \{\neg, \sim, \wedge_{\mathrm{p}}\})$. However, we have the following result, which proves that PTaCL is not canonically complete.

Proposition 4.1.1. *$L(\mathsf{3}, \{\neg, \sim, \wedge_{\mathrm{p}}\})$ is not canonically complete.*

Proof. It is impossible to represent all unary selection operators in normal form. The statement follows from the following observations: (i) PTaCL defines two unary operators $\neg$ and $\sim$; (ii) the only binary operators that may be used in normal form are $\wedge_{\mathrm{p}}$ ($\curlywedge$) and $\vee_{\mathrm{p}}$ ($\curlyvee$); (iii) for any operator $\oplus \in \{\wedge_{\mathrm{p}}, \vee_{\mathrm{p}}\}$ we have $0 \oplus 1 = 1 \oplus 0 \neq \bot$; and (iv) for any operator $\diamond \in \{\neg, \sim\}$ we have $\diamond 0 \neq \bot$ and $\diamond 1 \neq \bot$. Thus it is impossible to construct a unary operator of the form $S^{\bot}_d$ for any $d \neq \bot$. □

4.2 PTaCL with Jobe's logic

Having established that PTaCL is not canonically complete, we now investigate the latter option, that is, replacing the operators in PTaCL with operators from a canonically complete logic. In Section 2.5.2 we introduced Jobe's logic $\mathcal{J}$, a canonically complete 3-valued logic. The decision set for Jobe's logic $\mathcal{J}$ is $\{1, 2, 3\}$ with total ordering $1 < 2 < 3$, and operators defined in Figure 4.3.

x	$E_1(x)$	$E_2(x)$
1	2	3
2	1	2
3	3	1

(a) E_1 and E_2

$\wedge_{\mathrm{j}}$	1	2	3
1	1	1	1
2	1	2	2
3	1	2	3

(b) $\wedge_{\mathrm{j}}$

Figure 4.3: Operators in Jobe's logic $\mathcal{J}$

Using the operators in Jobe's logic $\mathcal{J}$ as a replacement for the operators in PTaCL is an intuitively reasonable choice for a number of reasons. Jobe's logic $\mathcal{J}$ and PTaCL both operate over a 3-valued decision set, and both assume a total ordering on the set of decisions. In fact, two of the operators defined in

$\mathcal{J}$ and PTaCL are equivalent under the bijection $1 \to 0, 2 \to \bot, 3 \to 1$:

$$x \wedge_{\mathrm{j}} y \equiv x \wedge_{\mathrm{p}} y \quad \text{and} \quad E_2(x) \equiv \neg\, x.$$

Likewise, the decisions sets and ordering are equivalent under this bijection. The only difference is in terms of the other unary operators, E_1 and $\sim$. These intuitions lead us to believe that Jobe's logic $\mathcal{J}$ provides a suitable basis for a canonically complete language for specifying ABAC policies. It it worth noting that PTaCL was defined directly from Jobe's logic $\mathcal{J}$, however the operator $\sim$ was chosen instead of E_1 as it had an intuitive use in ABAC policies; as a deny-by-default operator for policies, and as a must-not-be-present operator for targets. However, as we showed in Proposition 4.1.1, by choosing $\sim$, the canonical completeness of PTaCL was sacrificed.

Thus, we now define a variant of PTaCL, denoted by $\mathrm{PTaCL}_{\mathsf{3}}^{<}$, which uses Jobe's logic as a basis. In regards to the notation, $\mathsf{3}$ represents the decision set $\{0, \bot, 1\}$ and $<$ represents the total ordering. We use this notation throughout this thesis, and it is intended to easily distinguish variants PTaCL based on their (i) decision set; and (ii) ordering on the decision set. Later, in Chapter 6, we define a variant of PTaCL in which the decision set is partially ordered, and denote this by the notation $\mathrm{PTaCL}^{\leqslant}$.

We define the set of operators for $\mathrm{PTaCL}_{\mathsf{3}}^{<}$ to be $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$, where $\ddagger$ is equivalent to E_1 from Jobe's logic $\mathcal{J}$. The evaluation tables for $\neg, \ddagger$ and $\wedge_{\mathrm{p}}$ are shown in Figure 4.4a. The $\mathrm{PTaCL}_{\mathsf{3}}^{<}$ language, defines atomic policies and policies in exactly the same way as PTaCL. That is, an atomic policy has the form (t, d), where t is a target and $d \in \{0, 1\}$. The semantics of $\mathrm{PTaCL}_{\mathsf{3}}^{<}$ policies are defined in Figure 4.4b.

d	$\neg\, d$	$\ddagger\, d$
0	1	$\bot$
$\bot$	$\bot$	0
1	0	1

$\wedge_{\mathrm{p}}$	0	$\bot$	1
0	0	0	0
$\bot$	0	$\bot$	$\bot$
1	0	$\bot$	1

(a) $\neg, \ddagger$ and $\wedge_{\mathrm{p}}$

$$\begin{aligned}
\rho_q(d) &= d;\\
\rho_q(\neg\, p) &= \neg\, \rho_q(p);\\
\rho_q(\ddagger\, p) &= \ddagger\, \rho_q(p);\\
\rho_q(p \wedge_{\mathrm{p}} p') &= \rho_q(p) \wedge_{\mathrm{p}} \rho_q(p');\\
\rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}},\\ \bot & \text{otherwise.} \end{cases}
\end{aligned}$$

(b) Policy semantics

Figure 4.4: Decision operators and policy semantics in $\mathrm{PTaCL}_{\mathsf{3}}^{<}$

Of course, given the function completeness of PTaCL, it is possible to define the operator $\ddagger$ in terms of the PTaCL operators $\{\neg, \sim, \wedge_{\mathrm{p}}\}$. Indeed, we identify the following equivalence

$$\ddagger\, x \equiv (x \vee_{\mathrm{p}} \bot) \wedge_{\mathrm{p}} (\sim(x \vee_{\mathrm{p}} \neg\, x)).$$

We could then substitute the above equivalence for E_1 ($\ddagger$) in Figure 2.10b, to obtain representations for the unary selection operators in PTaCL, expressed as operators from the set $\{\neg, \sim, \wedge_{\mathrm{p}}\}$. However, the unary selection operators would not be in a *normal form*. Recall, the definition of the normal form of a formula ϕ requires that *no binary operator is included in the scope of a unary operator*. This is violated by the term $\sim(x \vee_{\mathrm{p}} \neg x)$). Hence, despite the ability to define $\ddagger$ in PTaCL, PTaCL is not canonically complete (confirming our previous result and justifying our replacement of $\sim$ with $\ddagger$, and the formation of $\mathrm{PTaCL}_3^<$).

4.3 The value of canonical completeness

We now illustrate why having a normal form may make it simpler to construct ABAC policies. Specifically, we represent the operator $\oplus_2$ from Figure 4.1 in normal form using the $\mathrm{PTaCL}_3^<$ operators. To reiterate, it is impossible to represent $\oplus_2$ as any combination of XACML operators (without defining custom combining algorithms) and it is difficult to see how to express $\oplus_2$ using the PTaCL operators (although it is theoretically possible to do so).

Using the truth table in Figure 4.1 and by definition of the selection operators and $\curlyvee$, we have $x \oplus_2 y$ is equivalent to

$$
\begin{aligned}
&S^0_{(0,0)}(x,y) \curlyvee S^0_{(0,\bot)}(x,y) \curlyvee S^\bot_{(0,1)}(x,y) \curlyvee \\
&S^0_{(\bot,0)}(x,y) \curlyvee S^\bot_{(\bot,\bot)}(x,y) \curlyvee S^\bot_{(\bot,1)}(x,y) \curlyvee \\
&S^\bot_{(1,0)}(x,y) \curlyvee S^\bot_{(1,\bot)}(x,y) \curlyvee S^1_{(1,1)}(x,y).
\end{aligned}
$$

Moreover, $S^c_{(a,b)}(x,y) = S^c_a(x) \curlywedge S^c_b(y)$. Hence, $x \oplus_2 y$ is equivalent to

$$
\begin{aligned}
&\Big(S^0_0(x) \curlywedge S^0_0(y)\Big) \curlyvee \Big(S^0_0(x) \curlywedge S^0_\bot(y)\Big) \curlyvee \Big(S^\bot_0(x) \curlywedge S^\bot_1(y)\Big) \curlyvee \\
&\Big(S^0_\bot(x) \curlywedge S^0_0(y)\Big) \curlyvee \Big(S^\bot_\bot(x) \curlywedge S^\bot_\bot(y)\Big) \curlyvee \Big(S^\bot_\bot(x) \curlywedge S^\bot_1(y)\Big) \curlyvee \\
&\Big(S^\bot_1(x) \curlywedge S^\bot_0(y)\Big) \curlyvee \Big(S^\bot_1(x) \curlywedge S^\bot_\bot(y)\Big) \curlyvee \Big(S^1_1(x) \curlywedge S^1_1(y)\Big).
\end{aligned}
$$

Each unary selection operator S^b_a can be represented in terms of the operators $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$. We have derived expressions in normal form for the unary selection operators S^b_a, which are shown in Figure 4.5. Note that $S^0_a(x) = 0$ for all $a, x \in \{0, \bot, 1\}$.

Hence, we may replace each instance of S^b_a with the equivalent expression in terms of $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$, resulting in a formula in normal form for $\oplus_2$. (We omit the full expression of $\oplus_2$ in terms of $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$ here, as it is lengthy and can be produced via simple substitution.)

Of course, one would not usually construct the normal form by hand, as we have done above. Indeed, it is easy to develop an algorithm that would con-

$S_i^0(x)$	$x \wedge_{\mathrm{p}} (\ddagger x) \wedge_{\mathrm{p}} (\neg x)$
$S_0^{\perp}(x)$	$(\ddagger x) \wedge_{\mathrm{p}} (\neg \ddagger x)$
$S_{\perp}^{\perp}(x)$	$x \wedge_{\mathrm{p}} (\neg x)$
$S_1^{\perp}(x)$	$(\ddagger \neg x) \wedge_{\mathrm{p}} (\neg \ddagger \neg x)$
$S_0^1(x)$	$(\neg x) \wedge_{\mathrm{p}} (\ddagger \neg x)$
$S_{\perp}^1(x)$	$(\neg \ddagger x) \wedge_{\mathrm{p}} (\neg \ddagger \neg x)$
$S_1^{\perp}(x)$	$x \wedge_{\mathrm{p}} (\ddagger x)$

Figure 4.5: Normal forms for the unary selection operators

struct the normal form of a policy from its decision table. Moreover, since our language is a tree-structured language, having exactly the same operational semantics as XACML and PTaCL, we can implement the $\text{PTaCL}_3^{\leq}$ operators as custom XACML combining algorithms and then specify XACML policies using these operators. Thus we can readily obtain a functionally and canonically complete policy language, whose policies can be embedded in the rich framework for ABAC provided by XACML (in terms of its languages for representing targets and requests) and its enforcement architecture (in terms of the policy enforcement, policy decision and policy administration points).[1] We provide details and an analysis of this algorithm, and demonstrate how the XACML architecture may be leveraged in Chapter 6.

Using the structure and evaluation strategy for PTaCL policies and the operators $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$ makes it possible to define arbitrary policies and to represent them in normal form. We believe that this provides a number of advantages, in addition to those mentioned above, which we now briefly discuss. First, it is known that policy misconfigurations can be costly, both in terms of data leakage (when actions that should not be possible are authorized by the policy) and in terms of administration (when actions that should be possible are not authorized and the policy needs to be updated) [5]. We believe that the use of a canonically complete policy language is likely to make policy specification easier to understand for policy authors, thereby reducing the number of errors and policy misconfigurations. We investigate this further in Chapter 7.

Second, policies in normal form may be more efficient to evaluate. Given a formula in a 3-valued logic expressed in normal form, any literal that evaluates to 0 causes the entire clause to evaluate to 0, while any clause evaluating to 1 means the entire formula evaluates to 1. We may also be able to apply some of the equivalences described by Jobe to minimize the size of a formula in normal form, thereby further reducing the effort required to evaluate it.

[1] This is contrast to proposals in the literature, which require the use of non-standard components, such as multi-terminal binary decision diagrams [40] or non-deterministic finite automata [29].

4.4 Indeterminacy in PTaCL$_3^<$

Since PTaCL and PTaCL$_3^<$ differ only in the choice of the unary operators $\sim$ and $\ddagger$, it is trivial to extend the method for handling indeterminacy in PTaCL to PTaCL$_3^<$. PTaCL$_3^<$ handle errors in target evaluation (and thus indeterminacy) using sets of possible decisions [11, 12, 29]. Informally, when target evaluation fails, denoted by $\tau_q(t) = ?_{\mathrm{m}}$, PTaCL$_3^<$ assumes that either $\tau_q(t) = 1_{\mathrm{m}}$ or $\tau_q(t) = 0_{\mathrm{m}}$ could have been returned, and returns the union of the (sets of) decisions that would have been returned in both cases. The formal semantics for policy evaluation in PTaCL$_3^<$ in the presence of indeterminacy are defined in Figure 4.6.

$$
\begin{aligned}
\rho_q(d) &= \{d\}\,; \\
\rho_q(\neg\, p) &= \{\neg\, d : d \in \rho_q(p)\}\,; \\
\rho_q(\ddagger\, p) &= \{\ddagger\, d : d \in \rho_q(p)\}\,; \\
\rho_q(p_1 \wedge_{\mathrm{p}} p_2) &= \{d_1 \wedge_{\mathrm{p}} d_2 : d_i \in \rho_q(p_i)\}\,; \\
\rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \{\bot\} & \text{if } \tau_q(t) = 0_{\mathrm{m}}, \\ \{\bot\} \cup \rho_q(p) & \text{if } \tau_q(t) = ?_{\mathrm{m}}. \end{cases}
\end{aligned}
$$

Figure 4.6: Semantics for PTaCL$_3^<$ with indeterminacy

The semantics for the operators $\{\neg, \ddagger, \wedge_{\mathrm{p}}\}$ operate on sets, rather than single decisions, in the natural way. The unary operators $\neg$ and $\ddagger$ are applied to each decision in the set, while $\wedge_{\mathrm{p}}$ takes the Cartesian product of each decision set from p_1 and p_2. A straightforward induction on the number of operators in a policy establishes that the decision set returned by these extended semantics will be a singleton if no target evaluation errors occur; moreover, that decision will be the same as that returned by the standard semantics. We formalise this result below.

Lemma 4.4.1. *Let p be a policy which contains targets $t_1, \ldots, t_n$ and let q be a request. If $\tau_q(t_i) \neq \bot_{\mathrm{m}}$ for all i, then $\rho_q(p) = \{x\}$ for some $x \in \{0, 1, \bot\}$.*

Proof. By induction on the structure of policies.

Base case: an atomic policy $p = d$ for some $d \in \{0, 1\}$. Then $\rho_q(p) = \{d\}$ for all q.

Induction hypothesis: assume that the policies p, p_1 and p_2 return a singleton decision set.

Inductive step: we will now show that the addition of a policy operator $\neg$ or $\ddagger$ does not introduce a non-singleton decision set for a policy p. By assumption $\rho_q(p) = \{x\}$ for some $x \in \{0, 1, \bot\}$. Then for all q,

$$\rho_q(\neg\, p) = \{\neg\, d : d \in \rho_q(p)\} = \{\neg\, x\}$$

which is a singleton decision set. Likewise, for all q,

$$\rho_q(\ddagger\, p) = \{\ddagger\, d : d \in \rho_q(p)\} = \{\ddagger\, x\}$$

which is a singleton decision set. We now show that the combination of two policies p_1 and p_2 using $\wedge_{\mathrm{p}}$ does not introduce a non-singleton decision set. By assumption $\rho_q(p_1) = \{x\}$ for some $x \in \{0, 1, \bot\}$ and $\rho_q(p_2) = \{y\}$ for some $y \in \{0, 1, \bot\}$. Then for all q,

$$\rho_q(p_1 \wedge_{\mathrm{p}} p_2) = \{d_1 \wedge_{\mathrm{p}} d_2 : d_i \in \rho_q(p_i)\} = \{x \wedge_{\mathrm{p}} y\} = \{z\}$$

for some $z \in \{0, 1, \bot\}$, which is a singleton decision set. Finally, we show that the addition of a target t_k with policy p does not introduce a non-singleton decision set. By assumption $\tau_q(t_k) \neq \bot_{\mathrm{m}}$ and $\rho_q(p) = \{x\}$ for some $x \in \{0, 1, \bot\}$. Then for all q

$$\rho_q(t_k, p) = \begin{cases} \{x\} & \text{if } \tau_q(t_k) = 1_{\mathrm{m}}, \\ \{\bot\} & \text{if } \tau_q(t_k) = 0_{\mathrm{m}}. \end{cases}$$

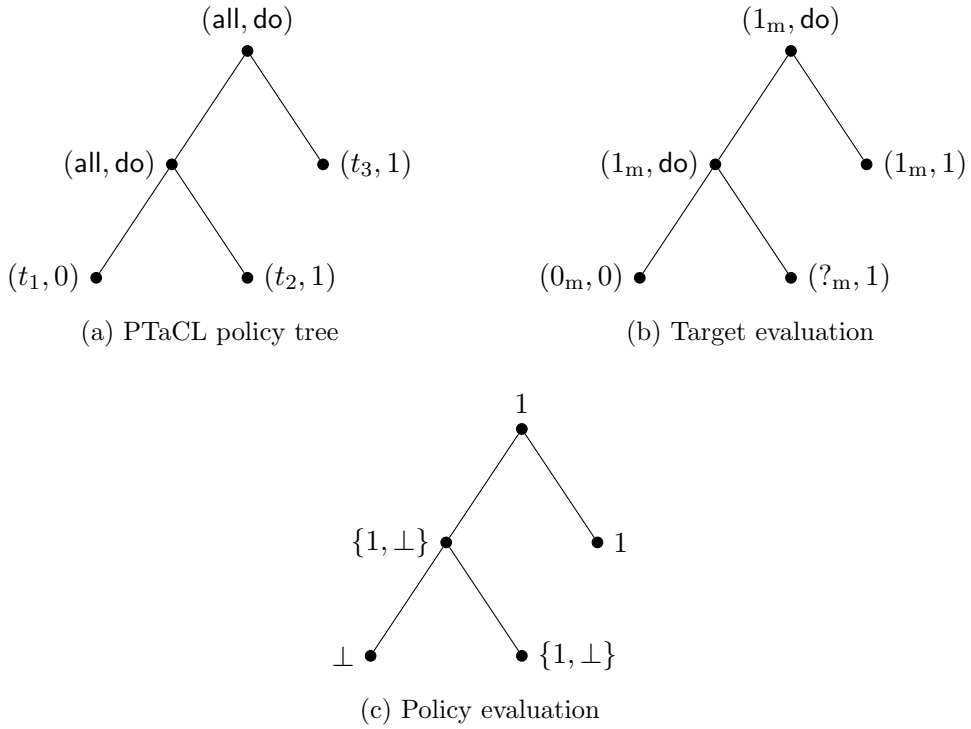
□

Handling indeterminacy through the use of decision sets permits policy evaluation to "fail gracefully" [11, 12]. We now demonstrate, through the use of an example [29], how a failure in target evaluation need not effect the evaluation of the entire policy. Consider the policy shown in Figure 4.7a and suppose that

$$\tau_q(t_1) = 0_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_2) =?_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_3) = 1_{\mathrm{m}}.$$

The first phase of request evaluation results in the tree shown in Figure 4.7b. The second phase of policy evaluation is shown in Figure 4.7c. As the target t_2 encountered an error, the policy $p_2 = (t_2, 1)$ returns a set of decisions rather than a singleton. PTaCL$^{\leq}_{3}$ policy evaluation assumes that either $\tau_q(t_2) = 1_{\mathrm{m}}$ or $\tau_q(t_2) = 0_{\mathrm{m}}$ could have been returned, resulting in the decisions 1 and $\bot$ respectively. As this set is combined further up the policy tree, $\bot$ do $\{1, \bot\}$ returns the set $\{1, \bot\}$, since $\bot$ do $1 = 1$ and $\bot$ do $\bot = \bot$. At the top level of the policy, we have 1 do $\{1, \bot\}$, which returns a single conclusive decision 1, as 1 do $1 = 1$ do $\bot = 1$. Thus, despite encountering errors in target evaluation, the policy was able to return a conclusive decision.

Furthermore, the decision returned is the "correct" decision, irrespective of the failure encountered during target evaluation. If t_2 had returned 0_{m}, then the overall policy would return 1, the same is true if t_2 had returned 1_{m}. In fact, the policies show in Figure 4.7 and Figure 3.3 (which highlighted the undesirable behaviour of indeterminacy in XACML versions 1.0 and 2.0)



(a) PTaCL policy tree

(b) Target evaluation

(c) Policy evaluation

Figure 4.7: Evaluating a PTaCL policy with indeterminacy

are equivalent. However, $\text{PTaCL}_3^{\leq}$ returns the "correct" decision, that is, 1, unlike XACML. Hence, we argue that the method for handling indeterminacy in $\text{PTaCL}_3^{\leq}$ addresses the shortcomings exhibited in XACML; by handling indeterminacy in a precise and desirable manner.

4.5 Obligations in $\text{PTaCL}_3^{\leq}$

We now define a method for incorporating obligations in $\text{PTaCL}_3^{\leq}$. In this section, we are not concerned with the specific types of obligations, how they will be provided by the policy information point, or how they will be enforced by the policy enforcement point. Instead, we focus on how they will be combined by the policy decision point (following the approach taken by the XACML standard). Thus, we simply assume the existence of some "abstract" set of obligations O.

The method we define for computing obligations in $\text{PTaCL}_3^{\leq}$ is inspired by the XACML standard, and one of our results shows that the obligations returned by a $\text{PTaCL}_3^{\leq}$ policy will be the same as those returned by an equivalent XACML policy. While recognizing that there may be other ways of computing obligations, we make the assumption that the behaviour specified by the XACML standard is that expected by the practitioners who designed it and is, therefore, a reasonable proxy for the required behaviour of an obligations-combining strategy.

4.5.1 Defining obligations

In XACML, each policy or policy set may be associated with one or more obligations. An obligation is associated with an effect (a decision in $\text{PTaCL}_3^{\leq}$), which may be Permit or Deny (denoted by 1 and 0, respectively, in $\text{PTaCL}_3^{\leq}$). Thus, the obligation associated with Permit is applied when the effect of a policy is Permit for a particular request. Informally, then, the result of evaluating a request in XACML is a pair comprising a decision and an obligation. Thus, we extend $\text{PTaCL}_3^{\leq}$ syntax in the following ways.

- The $\text{PTaCL}_3^{\leq}$ policy d, where $d \in \{0, 1\}$, may only return d, so it suffices to extend the syntax for such policies to (d, o) (where $o \in O$).
- The unary policy operators $\neg$ and $\ddagger$ are used only to switch policy decisions, so we will assume that obligations are not associated with these operators. When evaluating policies with the operators $\neg$ and $\ddagger$ the obligations from child nodes are passed up with no change.
- All other policies (generated using $\wedge_{\text{p}}$ or targets) may return 0 or 1, so we extend the syntax for a policy p to (p, o_0, o_1), where $o_i \in O$ is the obligation that should be returned if the evaluation of p returns decision $i \in \{0, 1\}$.

Henceforth, we will write $\sigma_q(p)$ to denote the obligations returned by the evaluation of policy p for request q. We may not wish to specify obligations for every policy and every decision, so we assume the existence of a "null" obligation, denoted by ϵ.

4.5.2 Computing obligations

In general terms, when a policy language includes obligations, the policy decision point will return a decision and a set of obligations as a result of evaluating an access request. In terms of our notation, then, request evaluation will return the pair $(\rho_q(p), \sigma_q(p))$: $\rho_q(p)$ is an element of D, as we have seen, and is determined by applying the relevant binary operator to the decisions returned by the child policies; $\sigma_q(p)$ is a subset of O and, informally, is determined by taking the union of the sets of obligations associated with particular child policies (together with any relevant obligation for the parent policy). This method leverages the tree-structured, bottom-up evaluation strategy of PTaCL (and XACML) to return obligations from the nodes in the policy tree that influence the final decision returned by policy evaluation.

More formally, $\text{PTaCL}_3^{\leq}$ obligation semantics are shown in Figure 4.8a. The interesting case is policy conjunction, where we only take the obligations from child policies that return a decision equal to that of the parent policy. Thus we take obligations from both child policies if they return the same decision (as well as the relevant obligation from the parent policy), and if

child policy p_i returns 0 and the other does not then we return $\{o_0\} \cup \sigma_q(p_i)$. (In all other cases, the decision returned is $\bot$ and the obligation set is empty.) We interpret $\{\epsilon\}$ as the empty set $\emptyset$.

$$\sigma_q(0, o) = \sigma_q(1, o) = \{o\}$$
$$\sigma_q(\neg\, p) = \sigma_q(\ddagger\, p) = \sigma_q(p)$$
$$\sigma_q(p_1 \wedge_{\mathrm{p}} p_2, o_0, o_1) = \begin{cases} \{o_0\} \cup \sigma_q(p_1) & \text{if } \rho_q(p_1) = 0 \text{ and } \rho_q(p_2) \neq 0 \\ \{o_0\} \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) \neq 0 \text{ and } \rho_q(p_2) = 0 \\ \{o_0\} \cup \sigma_q(p_1) \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = 0 \text{ and } \rho_q(p_2) = 0 \\ \{o_1\} \cup \sigma_q(p_1) \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = 1 \text{ and } \rho_q(p_2) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$
$$\sigma_q(t, p, o_0, o_1) = \begin{cases} \{o_0\} \cup \sigma_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}} \text{ and } \rho_q(p) = 0 \\ \{o_1\} \cup \sigma_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}} \text{ and } \rho_q(p) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

(a) Obligation semantics in $\mathrm{PTaCL}^{<}_{3}$

	0	$\bot$	1
0	$(0, \{O_1, O_2\})$	$(0, \{O_1\})$	$(0, \{O_1\})$
$\bot$	$(0, \{O_2\})$	$(\bot, \emptyset)$	$(\bot, \emptyset)$
1	$(0, \{O_2\})$	$(\bot, \emptyset)$	$(1, \{O_1, O_2\})$

(b) A look-up table for $\wedge_{\mathrm{p}}$ with decision-obligation pairs

Figure 4.8: Obligation semantics and look-up table

By an abuse of notation, we can build an evaluation table for $\wedge_{\mathrm{p}}$, as shown in Figure 4.8b (with the understanding that the relevant obligation needs to be included from the parent policy, the set of obligations associated with the decisions indexing the rows is O_1 and the set of obligations indexing the columns is O_2).

4.5.3 Computing obligations for derived policy operators

Given that (i) we can define arbitrary policy operators in terms of $\neg$, $\ddagger$ and $\wedge_{\mathrm{p}}$ and (ii) we have defined how obligations are computed for these operators, we can extend our method of computing obligations to arbitrary policy operators. For example, we can define the obligations that should be returned by $\vee_{\mathrm{p}}$, do and po, as shown in Figure 4.9. (As in Figure 4.8b for $\wedge_{\mathrm{p}}$, we assume that the relevant obligation from the parent policy will be included during policy evaluation; O_1 and O_2 are the obligations associated with the evaluation of p_1 and p_2, respectively.) Note that $\sim$ is not explicitly defined in $\mathrm{PTaCL}^{<}_{3}$, however we use it here as syntactic sugar. Indeed, it is easy to show that

$$\sim d \equiv d \wedge_{\mathrm{p}} \ddagger\, d.$$

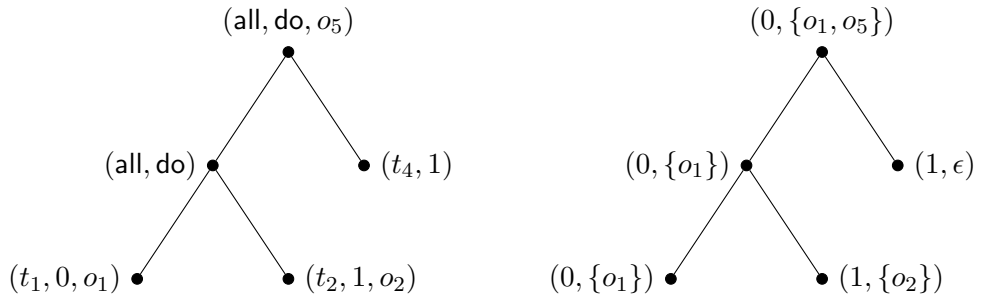
p_1	p_2	$p_1 \vee_\mathrm{p} p_2$	$p_1 \vee_\mathrm{p} \sim p_2$	$\sim p_1 \vee_\mathrm{p} p_2$	p_1 po p_2	p_1 do p_2
0	0	$(0, \{O_1, O_2\})$	$(0, \{O_1, O_2\})$	$(0, \{O_1, O_2\})$	$(0, \{O_1, O_2\})$	$(0, \{O_1, O_2\}))$
0	1	$(1, \{O_2\})$	$(1, \{O_2\})$	$(1, \{O_2\})$	$(1, \{O_2\})$	$(0, \{O_1\})$
0	$\bot$	$(\bot, \emptyset)$	$(0, \{O_1\})$	$(\bot, \emptyset)$	$(0, \{O_1\})$	$(0, \{O_1\})$
1	0	$(1, \{O_1\})$	$(1, \{O_1\})$	$(1, \{O_1\})$	$(1, \{O_1\})$	$(0, \{O_2\})$
1	1	$(1, \{O_1, O_2\})$	$(1, \{O_1, O_2\})$	$(1, \{O_1, O_2\})$	$(1, \{O_1, O_2\})$	$(1, \{O_1, O_2\})$
1	$\bot$	$(1, \{O_1\})$	$(1, \{O_1\})$	$(1, \{O_1\})$	$(1, \{O_1\})$	$(1, \{O_1\})$
$\bot$	0	$(\bot, \emptyset)$	$(\bot, \emptyset)$	$(0, \{O_2\})$	$(0, \{O_2\})$	$(0, \{O_2\})$
$\bot$	1	$(1\,\{O_2\})$	$(1, \{O_2\})$	$(1, \{O_2\})$	$(1, \{O_2\})$	$(1, \{O_2\})$
$\bot$	$\bot$	$(\bot, \emptyset)$	$(\bot, \emptyset)$	$(\bot, \emptyset)$	$(\bot, \emptyset)$	$(\bot, \emptyset)$

Figure 4.9: Decisions and obligations for the $\mathrm{PTaCL}_3^{\leq}$ $\vee_\mathrm{p}$, po and do operators

Consider the policy shown in Figure 4.10, which extends our example from Figure 4.7a with the addition of obligations. Note that the policy p_3 does not contain any parent obligations. Parent policies are not required to define the parent obligations o_0 and o_1; however our semantics support them if they are (see Figure 4.8a). Likewise, child nodes are not required to specify any obligations, such is the case for p_4.

$$
\begin{aligned}
p_1 &= (t_1, 0, o_1) \\
p_2 &= (t_2, 1, o_2) \\
p_3 &= (\mathsf{all}, p_1 \,\mathsf{do}\, p_2) \\
p_4 &= (t_4, 1) \\
p_5 &= (\mathsf{all}, p_3 \,\mathsf{do}\, p_4, o_5)
\end{aligned}
$$

(a) Policy



(b) Policy tree

(c) Evaluation

Figure 4.10: Example policy and policy tree with obligations

We assume that $\tau_q(t_i) = 1_\mathrm{m}$ for all i and, where obligations are not shown, they are assumed to be ϵ. To evaluate this policy tree with obligations, we use the do semantics for decisions and obligations taken from Figure 4.9. The result of evaluating the policy with respect to q is $(0, \{o_1, o_5\})$, as illustrated in Figure 4.10c. In particular, $(0, \{o_1\})$ do $(1, \{o_2\}) = (0, \{o_1\})$. The root policy has an obligation o_5, which is always returned (irrespective of the decision), so we return the set of obligations $\{o_1, o_5\}$ along with the 0 decision.

Our approach to obligations thus provides considerably greater flexibility than XACML, which only specifies how obligations should be computed for the pre-defined rule- and policy combining algorithms. Moreover, it is easy to show that the obligations computed by PTaCL for the po and do operators are identical to those computed by XACML.

The set of obligations returned by XACML processing is, informally, the set of obligations associated with those sub-policies that return the same decision as the root policy. With that in mind, consider the obligations that would be returned for a policy set that comprises two policies p_1 and p_2, associated with obligations o_1 and o_2 respectively, and uses the permit-overrides policy-combining algorithm. The set of obligations that would be returned for this policy set is tabulated in Figure 4.11. By inspection, we can see that the obligations returned for po in Figure 4.9 are identical to those for the permit-overrides policy-combining algorithm in Figure 4.11. An analogous result holds for computing obligations for do and the deny-overrides policy-combining algorithm.

p_1	p_2	permit-overrides(p_1, p_2)	deny-overrides(p_1, p_2)
0	0	$(0, \{o_1, o_2\})$	$(0, \{o_1, o_2\})$
0	1	$(1, \{o_2\})$	$(0, \{o_1\})$
0	$\perp$	$(0, \{o_1\})$	$(0, \{o_1\})$
1	0	$(1, \{o_1\})$	$(0, \{o_2\})$
1	1	$(1, \{o_1, o_2\})$	$(1, \{o_1, o_2\})$
1	$\perp$	$(1, \{o_1\})$	$(1, \{o_1\})$
$\perp$	0	$(0, \{o_2\})$	$(0, \{o_2\})$
$\perp$	1	$(1, \{o_2\})$	$(1, \{o_2\})$
$\perp$	$\perp$	$(\perp, \emptyset)$	$(\perp, \emptyset)$

Figure 4.11: Decisions and obligations for the XACML permit- and deny-overrides combining algorithms

In other words, PTaCL is (i) consistent with XACML in terms of the obligations returned for standard operators, and (ii) provides an extensible mechanism for computing obligations for arbitrary policy operators.

4.5.4 Indeterminacy and obligations

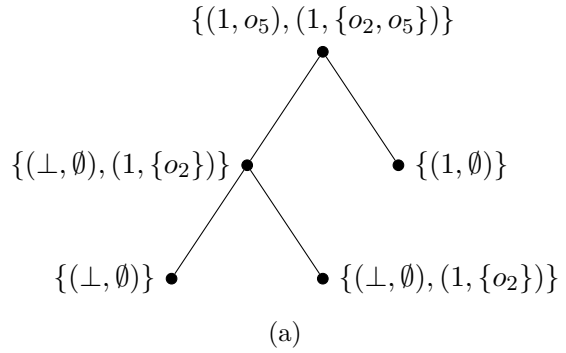
When target evaluation fails, $\text{PTaCL}_3^{\leq}$ returns a decision set as opposed to a single decision. We can extend this method when obligations are included in $\text{PTaCL}_3^{\leq}$. Without indeterminacy, request evaluation returns a decision-obligation pair; with indeterminacy, therefore, it returns a set of decision-obligation pairs. Consider the leaf policy (t, d, o). If $\tau_q(t) =?_{\text{m}}$, $\text{PTaCL}_3^{\leq}$ assumes that either $\tau_q(t) = 1_{\text{m}}$ or $\tau_q(t) = 0_{\text{m}}$ could have been returned, and thus returns the union of the sets of decision-obligation pairs that would have

been returned in both cases. More formally, we have

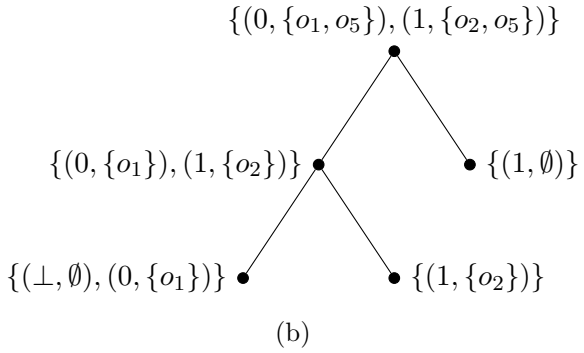
$$\sigma_q(t, d, o) = \{\bot, \emptyset\} \cup \{d, o\} = \{(\bot, \emptyset), (d, o)\}$$

We now investigate how this method may be applied to more complex policies. Consider the policy shown in Figure 4.10b and suppose that

$$\tau_q(t_1) = 0_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_2) = ?_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_4) = 1_{\mathrm{m}}.$$



(a)



(b)

Figure 4.12: PTaCL$_3^<$ policy evaluation with indeterminacy and obligations

The resulting policy evaluation is shown in Figure 4.12a and returns $\{(1, o_5), (1, \{o_2, o_5\})\}$. We see on this occasion that policy evaluation returns the same decision but different obligation sets. The reason for this discrepancy arises from the two alternative paths of the policy tree which affect the final decision:

- $\{1, o_5\}$ – policies p_1, p_2 and p_3 are not-applicable, hence the decision 1 comes *only* from p_4 and o_5 comes from the parent policy;
- $\{(1, \{o_2, o_5\})\}$ – policy p_2 is applicable, so the pair $(1, \{o_2\})$ is carried up the tree, and combined at the top with the right branch and parent obligation.

Now suppose that, for the same policy, we have

$$\tau_q(t_1) = ?_{\mathrm{m}} \quad \text{and} \quad \tau_q(t_i) = 1_{\mathrm{m}} \quad \text{for all other } i.$$

The resulting policy evaluation is shown in Figure 4.12b. In this case, different decisions and different obligation sets are obtained.

To summarise, when evaluating decisions and obligations in the presence of indeterminacy, there are scenarios where the policy decision point will return: (i) the same decision and obligation set, (ii) the same decision and different obligation sets, and (iii) different decisions and different obligation sets. The behaviour of the policy enforcement point would need to be defined for each situation. For example, given the set $\{(1, \{o_5\})(1, \{o_2, o_5\})\}$, a conservative approach may be implemented by taking the union of all the obligation sets and requiring that all obligations are enforced. Another possibility is to adopt the use of a resolution function, as described by Crampton and Huth [11]. The approach taken is application and domain specific, and dependent on the overall purpose, scope and management goal of an access control policy.[2] Hence, we do not explore different approaches further in this thesis, leaving it as a topic for future research.

4.6 Obligations in XACML and other related work

In XACML, each rule, policy and policy set may be associated with one or more obligations. An obligation is defined by the FulfillOn attribute (whose value is either "Permit" or "Deny") and an action (such as "create audit entry"). Policy evaluation returns a decision and a set of obligations to the policy enforcement point, which is required to enforce the decision and execute any obligations. The XACML 2.0 and 3.0 standards [36, 37] define how this set of obligations is computed:

> *"When such a policy or policy set is evaluated, an obligation SHALL be passed up to the next level of evaluation (the enclosing or referencing policy, policy set or authorization decision) only if the effect of the policy or policy set being evaluated matches the value of the FulfillOn attribute of the obligation..."*
>
> *"...no obligations SHALL be returned to the PEP if...the decision resulting from evaluating the policy or policy set does not match the decision resulting from evaluating an enclosing policy set."*
>
> *"...If the PDP's evaluation is viewed as a tree of policy sets and policies, each of which returns "Permit" or "Deny", then the set of obligations returned by the PDP to the PEP will include only the obligations associated with those paths where the effect at each level of evaluation is the same as the effect being returned by the PDP."*

[2] This is similar to the choice of which bias to use for a policy enforcement point in XACML [37].

Like much of the XACML standard, this statement lacks formality and prior work has indicated that the way in which policy-combining algorithms and the way of computing obligations produces some counterintuitive results [29].

The XACML 2.0 standard [36] defines how obligations are returned when target evaluation fails:

> *"...no obligations SHALL be returned to the PEP if the policies or policy sets from which they are drawn are not evaluated, or if their evaluated result is "Indeterminate" or "NotApplicable"..."*

Thus, obligations from any policy that evaluates to "Indeterminate" are lost in the evaluation process. The XACML 3.0 standard has improved the way in which decisions are computed in the presence of indeterminacy [37], but has not changed how obligations are computed.

The specification and computation of obligations in $\text{PTaCL}_3^{\leq}$ has some similarities to, and some notable differences, from XACML, which we summarize in Table 4.1. We also include a comparison with the work on obligations by Li *et al.* [29], discussed below.

Feature	XACML	PCL	$\text{PTaCL}_3^{\leq}$
Obligations associated with different policy decisions	Yes	Yes	Yes
Selective "inheritance" of obligations from child policies by parent	No	Yes	No
Computation of obligations rigorously defined	No	Yes	Yes
Obligations associated with rules	Yes	No	Yes
Obligations returned when policy evaluation is indeterminate	No	No	Yes
Obligations defined for all policy combining algorithms	No	No	Yes

Table 4.1: Comparison of XACML, PCL and $\text{PTaCL}_3^{\leq}$

Arguably the two greatest improvements offered by the approach we propose are (i) the ability to return obligations when policy evaluation is indeterminate, and (ii) the ability to compute the set of obligations for any policy, irrespective of the operators used. In the first case, it seems natural to allow obligations to be returned even for indeterminate policies (and these could be considered as "default obligations") and provides more fine-grained control over which requests are subject to indeterminacy. In the second case, we believe it is important to specify the computation of obligations as completely and unambiguously as possible, thus minimizing the likelihood that an implementation will be incorrect.

Other work exists that define methods for handling obligations in XACML. Alqatawna *et al.* [3] introduce a way of using obligations to implement

a discretionary overriding mechanism in XACML. They do this by using two algorithms, an effects-combining algorithm which is similar to standard policy-combining algorithms and an obligations-combining algorithm, the implementation of the latter being left to the discretion of the policy author. We believe it will be more useful, in general, to provide, as we have done, standardized mechanisms for combining obligations that are natural extensions of the existing decision-combining algorithms.

Li *et al.* [29] defined semantics for handling obligations in XACML, largely following the definition in the XACML standard where obligations are returned only from paths which "contribute" to the final decision returned by the PDP. Li *et al.* do define an algorithm for computing the set of obligations for an arbitrary policy operator, although this algorithm requires the operator to have certain properties. In contrast, our approach to obligations is completely general: any policy operator can be defined using $\text{PTaCL}_3^{\leq}$ and a decision and a set of obligations can be computed. Like our definition of obligations in $\text{PTaCL}_3^{\leq}$, if the outcome of policy evaluation is not-applicable then the set of obligations is defined to be empty. Like XACML, if the outcome of policy evaluation is indeterminate, then the set of obligations is defined to be indeterminate. We would argue that it is more useful to return as much information as possible to the PEP, which can then decide what obligations, if any, should be enforced. The work by Li *et al.* differs when combining obligations from two sub-policies that return the same decision, by allowing for three different methods to be specified in the policy combining language: both, first and either, leaving the choice to the policy author. There is some merit gained by allowing this choice to be made by the policy author in terms of the fine-grained control over obligation handling it provides. However, in practice, ABAC policies may be large and complex, thus specifying this choice each time sub-policies are combined will be time-consuming, and will require a case-by-case analysis to ensure the correct obligations are computed.

Subsequently, Li *et al.* [28] developed an architecture extending the XACML architecture in order to handle access control policies with different types of obligations. The focus of their work is how to enforce the obligations once they have been returned to the PEP, while we focus on which obligations should be returned in the first place. A combination of Li *et al.*'s architecture and our method for returning obligations may be an interesting and beneficial solution to some of the issues in XACML. Finally, we note that there exists work on dependencies between obligations and the effect these might have on the ability to fulfil obligations [22, 23, 31]. These considerations are outside the scope of this thesis, but may prove fruitful areas for future research.

4.7 Summary and discussion

We discussed many of the shortcomings of the XACML standard in Chapter 3: it is functionally incomplete; there are numerous redundancies and dependencies between the combining algorithms; and it has an ambiguous method for handling indeterminacy. We believe that PTaCL$_3^<$ addresses all of these shortcomings.

Firstly, we have shown that PTaCL$_3^<$ is functionally complete. This means that any arbitrary policy is expressible in PTaCL$_3^<$, without needing to define custom combining algorithms (like XACML). Moreover, we proved that PTaCL$_3^<$ is canonically complete, and demonstrated how to convert arbitrary operators to an equivalent normal form comprising the PTaCL$_3^<$ operators. In doing so, we overcame one of the main difficulties encountered in other tree-structured languages, that is, writing the desired policy using the operators provided in the given language.

Secondly, we defined precise semantics for handling errors in target evaluation (indeterminacy) in PTaCL$_3^<$. Through a comparative example, we showed that PTaCL$_3^<$ can return a conclusive decision despite encountering errors; and that this decision is the intuitively "correct" decision. This directly contrasts the undesirable behaviour exhibited in the method for handling indeterminacy XACML 2.0 standard, seen in Section 3.1.1.

Finally, we introduced syntax and semantics for evaluating obligations in PTaCL$_3^<$, motivated by the XACML method for returning obligations. We showed that our method is consistent with XACML, and more extensible; as it provides a means for computing obligations for arbitrary policy operators. Furthermore, we extended the semantics for indeterminacy to incorporate obligations, by returning a set of decision-obligation pairs.

There are many opportunities for future work in the administration of policy enforcement points. In particular, the following challenges require investigation: (i) resolution functions and methods for handling decision-obligation pairs; and (ii) effective enforcement and fulfilment of obligations once they have been returned to the policy enforcement point.

Thus far, we have restricted our attention to ABAC languages defined over the decision set $\{0, \bot, 1\}$, and assumed a total ordering on this decision set of $0 < \bot < 1$. (We omitted the indeterminate decision from XACML for reasons discussed in Chapter 3). There are many ABAC languages that make use of a fourth authorization decision, which we explore in more detail in the following chapter.

Chapter 5

Canonical Completeness in Lattice-based Multi-valued Logics

Previously, in Chapter 4, during the development of $\text{PTaCL}_3^{\leq}$, we assumed a total order on the set of decisions ($0 < \bot < 1$). This ordering does not really reflect the intuition behind the use of $0, 1$ and $\bot$ in ABAC languages. In the context of access control, 0 and 1 are incomparable conclusive decisions, while $\bot$ represents the inability to reach a conclusive decision. This provides us with motivation to explore other more intuitive orderings on the set of decisions, such as partial orderings and lattices. Furthermore, we believe there is value in having an ABAC language for which policy evaluation can return a fourth value $\top$, representing a 'conflict' or 'excess' amount of information. Indeed, there are many languages such as PBel [10], BelLog [45] and Rumpole [32] which utilize this fourth decision. We discuss the value of a fourth authorization decision in more detail in Chapter 6. In this chapter, we focus on formally defining lattice-based multi-valued logics and extending the concept of canonical completeness to such logics.

The main contribution of this chapter is the construction of a canonically complete latticed-based multi-valued logic. In the next section, we introduce the necessary prerequisite formal definitions of partially ordered sets, lattices and bilattices. We then review Belnap logic [7] in Section 5.2, the most well known lattice-ordered logic in the literature. In Section 5.3 we extend the definitions of canonical suitability, selection operators, normal form and canonical completeness to lattice-based logics, enabling us to evaluate these properties for lattice-based logics. We prove that Belnap logic, and any ABAC authorization language which uses Belnap logic as the underlying logic, is not canonically complete in Section 5.4. In Section 5.5 we identify connections between the symmetric group and unary operators on the set of authorization decisions, enabling us to construct a canonically complete 4-valued lattice-

based logic. Finally, in Section 5.6, we extend the work of Jobe [24], showing a construction of a total-ordered, canonically complete m-valued logic.

We have published the majority of the work presented in Sections 5.2 – 5.6 [17].

5.1 Partially ordered sets and lattices

This section introduces the necessary definitions and terminology for lattices, which are a special type of partially ordered sets. We begin with the definition of a partially ordered set.

Definition 5.1.1. *A pair $(X, \leqslant)$, where $\leqslant$ is a binary relation over a set of values X, is a* partially ordered set *or* poset *if for all $x, y, z \in X$:*

- $x \leqslant x$;
- $x \leqslant y$ *and* $y \leqslant x$ *implies* $x = y$; *and*
- $x \leqslant y$ *and* $y \leqslant z$ *implies* $x \leqslant z$.

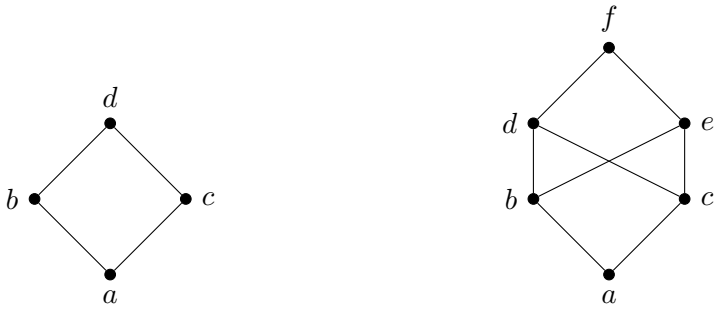
In other words, $\leqslant$ is a binary relation on X that is reflexive, anti-symmetric and transitive. We refer to $\leqslant$ as a *partial order*.

Definition 5.1.2. *Let $(X, \leqslant)$ be a partially ordered set and $x, y \in X$. We say y* covers *x, or x is* covered by *y, denoted $x \lessdot y$, if $x < y$ and for all $z \in X$, $x \leqslant z < y$ implies $x = z$.*

A partially ordered set $(X, \leqslant)$ is often represented by a *Hasse diagram*, which is the graph of the *covering relation* on X. That is, each element of X is represented by a node (or vertex), and an edge exists between x and y if $x \lessdot y$. An edge is not drawn from a node to itself, nor is an edge drawn between x and y if there exists z such that $x < z < y$. Typical convention states that if $x < y$, then the node labelled x will be lower than the node labelled y in the Hasse diagram. Examples of two partially ordered sets are shown in Figure 5.1. Henceforth, when the ordering $\leqslant$ is clear from context, we will write X to mean the partially ordered set $(X, \leqslant)$.

Definition 5.1.3. *Let X be a partially ordered set, and let $Y \subseteq X$.*

- *An element $u \in X$ is an* upper bound *of Y if $y \leqslant u$ for all $y \in Y$.*
- *We say that u' is a* least upper bound *or* supremum *of Y if $u' \leqslant u$ for all upper bounds u of Y. We denote the supremum of Y by* $\sup Y$.
- *An element $v \in X$ is a* lower bound *of Y if $v \leqslant y$ for all $y \in Y$.*
- *We say that v' is a* greater lower bound *or* infimum *of Y if $v \leqslant v'$ for all lower bounds v of Y. We denote the infimum of Y by* $\inf Y$.



(a) (b)

Figure 5.1: Hasse diagrams

Note that the least upper bound and greater lower bounds of Y (if they exist) are unique.

Definition 5.1.4. *A partially ordered set X is a lattice if, and only if, for all $x, y \in X$ there exists a least upper bound of x and y, denoted* $\sup\{x, y\}$*, and a greatest lower bound of x and y, denoted by* $\inf\{x, y\}$*.*

The least upper bound of x and y is written as $x \vee y$ (the "join" of x and y) and the greatest lower bound is written as $x \wedge y$ (the "meet" of x and y). The partially ordered set represented in Figure 5.1a is a lattice. However, the partially ordered set represented in Figure 5.1b is not a lattice, since $\{b, c\}$ does not have a least upper bound.

Definition 5.1.5. *Let X be a lattice. If for all $Y \subseteq X$,* $\sup Y$ *and* $\inf Y$ *exist in X, then X is a* complete *lattice.*

If $(X, \leqslant)$ is a finite lattice, as we will assume henceforth, then $(X, \leqslant)$ has a maximum element (that is, a unique maximal element) and a minimum element. In other words, all finite lattices are complete.

In the final part of this section we define *bilattices*, proposed by Ginsberg [20, 21] as a generalization of the Belnap [7] lattice (discussed in more detail in Section 5.2). Ginsberg's work on bilattices were extended by Fitting [18, 19], who introduced *pre-bilattices*.

Definition 5.1.6. *A* pre-bilattice *is a structure $\mathcal{B} = \{B, \leqslant_t, \leqslant_k\}$, where*

- *B is a set containing at least four elements; and*
- *$(B, \leqslant_t)$ and $(B, \leqslant_k)$ are complete lattices.*

The lattice $(B, \leqslant_t)$ is commonly known as the *truth lattice* or *t-lattice*, and the order associated with it is called the *truth ordering*. The meet and join of this lattice are denoted by $\wedge$ and $\vee$ respectively. The lattice $(B, \leqslant_k)$ is commonly known as the *knowledge lattice* or *k-lattice*, and the order associated with it is called the *knowledge ordering*. The meet and join of this lattice are denoted by $\otimes$ and $\oplus$ respectively.

Typically in pre-bilattices there is some connection between the two orders. At least two ways of characterizing such connections between orders have been explored in the literature. The first, introduced by Fitting [18], imposes monotonic properties between the lattice operations.

Definition 5.1.7. *A pre-bilattice* $\mathcal{B} = \{B, \leqslant_t, \leqslant_k\}$ *is* interlaced *if each of the four lattice operations* $\wedge, \vee, \otimes$ *and* $\oplus$ *are monotonic with respect to* $\leqslant_t$ *and* $\leqslant_k$*. That is, when the following equalities hold:*

- $x \leqslant_t y$ *implies that* $x \otimes z \leqslant_t y \otimes z$ *and* $x \oplus z \leqslant_t y \oplus z$*; and*
- $x \leqslant_k y$ *implies that* $x \wedge z \leqslant_k y \wedge z$ *and* $x \vee z \leqslant_k y \vee z$*.*

The second characterization, introduced by Ginsberg [21], imposes distributive properties between the lattice operations.

Definition 5.1.8. *A pre-bilattice* $\mathcal{B} = \{B, \leqslant_t, \leqslant_k\}$ *is* distributive *when all twelve distributive laws hold concerning the four lattice operations* $\wedge, \vee, \otimes$ *and* $\oplus$*. That is, the following identities hold:*

$$x \circ (y \bullet z) = (x \circ y) \bullet (x \circ z) \quad \text{for every } \circ, \bullet \in \{\wedge, \vee, \otimes, \oplus\} \text{ with } \circ \neq \bullet.$$

Ginsberg [20] formally defined *bilattices* which expand the algebraic structure of pre-bilattices with a unary operator.

Definition 5.1.9. *A* bilattice *is a structure* $\mathcal{B} = \{B, \leqslant_t, \leqslant_k, \neg\}$ *such that* $\{B, \leqslant_t, \leqslant_k\}$ *is a pre-bilattice and negation* $\neg$ *is a unary operation satisfying that for every* $x, y \in B$*:*

- *if* $a \leqslant_t b$*, then* $\neg b \leqslant_t \neg a$*;*
- *if* $a \leqslant_k b$*, then* $\neg a \leqslant_k \neg b$*; and*
- $a = \neg\neg a$*.*

The interlacing and distributive properties extend to bilattices in the obvious way: we say that a bilattice is interlaced (distributive) when its pre-bilattice is interlaced (distributive).

5.2 Belnap logic

Belnap logic [7] was developed with the intention of defining ways to handle inconsistent and incomplete information in a formal manner. It uses the truth values 0, 1, $\bot$, and $\top$, representing "false", "true", "lack of information" and "too much information", respectively. Henceforth, we will denote the four valued decision set $\{\bot, 0, 1, \top\}$ by $\mathsf{4}$.

The truth values 0, 1, $\bot$ and $\top$ have an intuitive interpretation in the context of access control: 0 and 1 are interpreted as the standard "deny"

and "allow" decisions, $\bot$ is interpreted as "not-applicable" and $\top$ represents a conflict of decisions. There are certain situations where it is useful to have four decisions available, and some ABAC languages, such as PBel [10], BelLog [45] and Rumpole [32], use Belnap's four truth values and logic as the underlying logic for their language.

The set of truth values in Belnap logic admits two orderings: a truth ordering $\leqslant_t$ and a knowledge ordering $\leqslant_k$. In the truth ordering, 0 is the minimum element and 1 is the maximum element, while $\bot$ and $\top$ are incomparable indeterminate values. In the knowledge ordering, $\bot$ is the minimum element, $\top$ is the maximum element while 0 and 1 are incomparable. Both $(\mathsf{4}, \leqslant_t)$ and $(\mathsf{4}, \leqslant_k)$ are lattices, together forming the pre-bilattice $(\mathsf{4}, \leqslant_t, \leqslant_k)$, illustrated as a Hasse diagram in Figure 5.2.

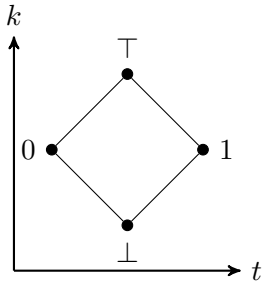


Figure 5.2: The Belnap Hasse diagram

We write the meet and join in $(\mathsf{4}, \leqslant_t)$ as $\wedge_\mathrm{b}$ and $\vee_\mathrm{b}$, respectively; and the meet and join in $(\mathsf{4}, \leqslant_k)$ as $\otimes_\mathrm{b}$ and $\oplus_\mathrm{b}$, respectively. (We use the subscript b to differentiate the Belnap operators from the PTaCL operators $\wedge_\mathrm{p}$ and $\vee_\mathrm{p}$.) In addition, Belnap logic defines a binary operator $\supset_\mathrm{b}$ and unary operator $\neg$. The former is an extension of implication to the Belnap space, while the latter has the effect of switching the values 0 and 1, leaving $\bot$ and $\top$ fixed; in other words, it acts like "classical" negation. The truth tables for the operators $\wedge_\mathrm{b}$, $\vee_\mathrm{b}$, $\otimes_\mathrm{b}$, $\oplus_\mathrm{b}$, $\supset_\mathrm{b}$ and $\neg$ are shown in Figure 5.3.

It is immediately clear that the structure $(\mathsf{4}, \leqslant_t, \leqslant_k, \neg)$ is a bilattice, which is often denoted as **Four** in the literature. Furthermore, the twelve distributive laws hold for the operators $\wedge_\mathrm{b}, \vee_\mathrm{b}, \otimes_\mathrm{b}$ and $\oplus_\mathrm{b}$, as do De Morgan's laws, and all of these operators are monotone with respect to both the $\leqslant_t$ and $\leqslant_k$ orderings. Specifically, for all $x, y, z \in \mathsf{4}$:

- $x \leqslant_t y$ implies that $x \otimes_\mathrm{b} z \leqslant_t y \otimes_\mathrm{b} z$ and $x \vee_\mathrm{b} z \leqslant_t y \vee_\mathrm{b} z$; and
- $x \leqslant_k y$ implies that $x \wedge_\mathrm{b} z \leqslant_k y \wedge_\mathrm{b} z$ and $x \vee_\mathrm{b} z \leqslant_k y \vee_\mathrm{b} z$.

Hence, the Belnap bilattice $(\mathsf{4}, \leqslant_t, \leqslant_k, \neg)$ is distributive and interlaced.

In Belnap logic, we may interpret values in $\mathsf{4}$ as operators of arity 0 (that is, constants). Then we can formally represent Belnap logic as the logic $L(\mathsf{4}, \{\neg, \wedge_\mathrm{b}, \vee_\mathrm{b}, \otimes_\mathrm{b}, \oplus_\mathrm{b}, \supset_\mathrm{b}, \bot, 0, 1, \top\})$. It is known that $L(\mathsf{4}, \{\neg, \wedge_\mathrm{b}, \vee_\mathrm{b}, \otimes_\mathrm{b}, \oplus_\mathrm{b}, \supset_\mathrm{b}, \bot, 0, 1, \top\})$ is functionally complete [4, Theorem

$\wedge_b$	0	$\bot$	$\top$	1
0	0	0	0	0
$\bot$	0	$\bot$	0	$\bot$
$\top$	0	0	$\top$	$\top$
1	0	$\bot$	$\top$	1

(a) $\wedge_b$

$\vee_b$	0	$\bot$	$\top$	1
0	0	$\bot$	$\top$	1
$\bot$	$\bot$	$\bot$	1	1
$\top$	$\top$	1	$\top$	1
1	1	1	1	1

(b) $\vee_b$

$\otimes_b$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$	0
1	$\bot$	$\bot$	1	1
$\top$	$\bot$	0	1	$\top$

(c) $\otimes_b$

$\oplus_b$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

(d) $\oplus_b$

$\supset_b$	0	$\bot$	$\top$	1
0	1	1	1	1
$\bot$	1	1	1	1
$\top$	0	$\bot$	$\top$	1
1	0	$\bot$	$\top$	1

(e) $\supset_b$

d	$\neg d$
0	1
$\bot$	$\bot$
$\top$	$\top$
1	0

(f) $\neg$

Figure 5.3: Operators in Belnap logic

12] and that $\{\neg, \oplus_b, \supset_b, \bot\}$ is a minimal functionally complete set of operators [4, Proposition 17]. We will see however, that Belnap logic is not canonically complete.

5.3 Canonical completeness for lattice-based logics

Jobe's definition of canonical suitability for multi-valued logics assumes a total ordering on the set of truth values. Given that Belnap logic [7], is a 4-valued logic in which the set of truth values forms a lattice, we seek to extend the definitions of canonical suitability, selection operators and canonical completeness to lattice-based logics. This will enable us to establish the canonical completeness (or incompleteness) of lattice-based logics. It is important to note that the definitions below are similar to those for total-ordered logics (presented in Section 2.5.1), the only difference lies in the definition of canonical suitability and selection operators. We reiterate the remaining definitions here for the interests of clarity and continuity, and recast the examples in the context of a lattice-based logic.

Let $L = (V, \mathsf{Ops})$ be a logic associated with a lattice $(V, \leqslant)$ of truth values and a set of logical operators Ops. We omit V and Ops when no ambiguity can occur. We write $\Phi(L)$ to denote the set of formulae that can be written

in the logic L.

We say L is *canonically suitable* if and only if there exist formulae $\phi_{\max}$ and $\phi_{\min}$ of arity 2 in $\Phi(L)$ such that $\phi_{\max}(x, y)$ returns $\sup\{x, y\}$ and $\phi_{\min}(x, y)$ returns $\inf\{x, y\}$. If a logic is canonically suitable, we will write $\phi_{\max}(x, y)$ and $\phi_{\min}(x, y)$ using infix binary operators as $x \curlyvee y$ and $x \curlywedge y$, respectively.

The existence of $\sup\{x, y\}$ and $\inf\{x, y\}$ is guaranteed in a lattice; this is not true in general for partially ordered sets. And for a totally ordered (finite) set, $\sup\{x, y\} = \max\{x, y\}$ and $\inf\{x, y\} = \min\{x, y\}$, so our definitions are compatible with those of Jobe's for totally ordered sets of truth values.

A function $f : V^n \to V$ is completely specified by a truth table containing n columns and m^n rows. However, not every truth table can be represented by a formula in a given logic $L = (V, \mathsf{Ops})$. L is said to be *functionally complete* if for every function $f : V^n \to V$, there is a formula $\phi \in \Phi(L)$ of arity n whose evaluation corresponds to the truth table.

Let $\underline{x}$ denote the minimum value in $(V, \leqslant)$. (Such a value must exist in a finite lattice.) We will write $\mathbf{a}$ to denote the tuple $(a_1, \ldots, a_n) \in V^n$ when no confusion can occur. Then, for $\mathbf{a} \in V^n$, the n-ary *selection operator* $S^j_{\mathbf{a}}$ is defined as follows:

$$S^j_{\mathbf{a}}(\mathbf{x}) = \begin{cases} j & \text{if } \mathbf{x} = \mathbf{a}, \\ \underline{x} & \text{otherwise.} \end{cases}$$

Note $S^{\underline{x}}_{\mathbf{a}}(x) = \underline{x}$ for all $\mathbf{a}, \mathbf{x} \in V^n$. Illustrative examples of unary and binary selection operators for the lattice $(\mathsf{4}, \leqslant_k)$ (the Belnap knowledge ordering) with minimal value $\bot$ are shown in Figure 5.4.

x	$S^0_0(x)$	$S^\top_1(x)$
$\bot$	$\bot$	$\bot$
0	0	$\bot$
1	$\bot$	$\top$
$\top$	$\bot$	$\bot$

$S^0_{(1,\top)}(x,y)$		y = $\bot$	0	1	$\top$
	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
	0	$\bot$	$\bot$	$\bot$	$\bot$
x	1	$\bot$	$\bot$	$\bot$	0
	$\top$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 5.4: Examples of selection operators in Belnap logic

The construction of arbitrary functions using selection operators, Jobe's results, and definitions for normal form and canonical completeness for lattice-based logics are identical to total-ordered logics. Nevertheless, we reiterate the constructions, results and definitions here in interests of clarity.

Selection operators play a central role in the development of canonically complete logics because an arbitrary function $f : V^n \to V$ can be expressed

in terms of selection operators. Consider, for example, the function

$$f(x,y) = \begin{cases} \top & \text{if } x = 0,\ y = 1, \\ 0 & \text{if } x = y = 0, \\ 1 & \text{if } x = 1,\ y = \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Then it is easy to confirm that

$$f(x,y) \equiv S^{\top}_{(0,1)}(x,y) \curlyvee S^{0}_{(0,0)}(x,y) \curlyvee S^{1}_{(1,\bot)}(x,y).$$

Moreover, $S^{c}_{(a,b)}(x,y) \equiv S^{c}_{a}(x) \curlywedge S^{c}_{b}(y)$ for any $a, b, c, x, y \in V$. Thus,

$$f(x,y) \equiv \big(S^{\top}_{0}(x) \curlywedge S^{\top}_{1}(y)\big) \curlyvee \big(S^{0}_{0}(x) \curlywedge S^{0}_{0}(y)\big) \curlyvee \big(S^{1}_{1}(x) \curlywedge S^{1}_{\bot}(y)\big).$$

In other words, we can express f as the "disjunction" ($\curlyvee$) of "conjunctions" ($\curlywedge$) of unary selection operators.

More generally, given the truth table of function $f : V^n \to V$, we can write down an equivalent function in terms of selection operators. Specifically, let

$$A = \{\mathbf{a} \in V^n : f(\mathbf{a}) > \bot\};$$

then, for all $\mathbf{x} \in V^n$,

$$g(\mathbf{x}) = \bigvee_{\mathbf{a} \in A} S^{f(\mathbf{x})}_{\mathbf{a}}(\mathbf{x}).$$

Simple inspection of $g(x)$ confirms that this logical formula expressed in terms of selection operators is equivalent the truth table of the function $f : V^n \to V$, that is

$$g(x) \equiv f(x).$$

Theorem 5.3.1 (Jobe [24, Theorems 1, 2; Lemma 1])**.** *A logic L is functionally complete if and only if each unary selection operator is equivalent to some formula in L.*

Definition 5.3.1. *The* normal form *of formula ϕ in a canonically suitable logic is a formula ϕ' that has the same truth table as ϕ and has the following properties:*

- *the only binary operators it contains are $\curlyvee$ and $\curlywedge$;*
- *no binary operator is included in the scope of a unary operator;*
- *no instance of $\curlyvee$ occurs in the scope of the $\curlywedge$ operator.*

In other words, given a canonically suitable logic L containing unary ope-

rators $\sharp_1, \ldots, \sharp_\ell$, a formula in normal form has the form

$$\bigvee_{i=1}^{r} \bigwedge_{j=1}^{s} \sharp_{i,j} x_{i,j}$$

where $\sharp_{i,j}$ is a unary operator defined by composing the unary operators in $\sharp_1, \ldots, \sharp_\ell$. In the usual 2-valued propositional logic with a single unary operator (negation) this corresponds to disjunctive normal form.

Definition 5.3.2. *A canonically suitable logic is* canonically complete *if every unary selection operator can be expressed in normal form.*

5.4 Canonical completeness of Belnap logic

Having extended the definitions of canonical suitability, selection operators and canonical completeness to lattice-based logics, we now investigate how these concepts can be applied to Belnap logic [7]. The meet and join operators of the two lattices $(\mathsf{4}, \leqslant_t)$ and $(\mathsf{4}, \leqslant_k)$ defined in Belnap logic are different. Canonical suitability for a 4-valued logic, defined as it is in terms of the ordering on the set of truth values, will thus depend on the ordering we choose on $\mathsf{4}$. Consequently, the $\curlywedge$ and $\curlyvee$ operators, along with the selection operators, will differ depending on the lattice that we choose.

In Chapter 6, we will argue in more detail for the use of a lattice-based ordering on $\mathsf{4}$ to support a tree-structured ABAC language. For now, we state that we will use knowledge-ordered lattice $(\mathsf{4}, \leqslant_k)$. The intuition is that the minimum value in this lattice is $\bot$ (rather than 0 in $(\mathsf{4}, \leqslant_t)$) and that this value should be the default value for a policy (being returned when the policy is not applicable to a request). In the interests of brevity, we will henceforth write $\mathsf{4}_t$ and $\mathsf{4}_k$, rather than $(\mathsf{4}, \leqslant_t)$ and $(\mathsf{4}, \leqslant_k)$ respectively. Hence, we now explore whether the logic $L(\mathsf{4}_k, \{\neg, \oplus_\mathrm{b}, \supset_\mathrm{b}, \bot\})$ is canonically complete or not. Note that we choose to use the minimal functionally complete set of Belnap operators [4]. It is standard for ABAC authorization languages to specify as few operators as possible, to limit the number of operators that are required in an implementation [10, 12].

It follows from the functional completeness of $\{\neg, \oplus_\mathrm{b}, \supset_\mathrm{b}, \bot\}$ that $L(\mathsf{4}_k, \{\neg, \oplus_\mathrm{b}, \supset_\mathrm{b}, \bot\})$ is canonically suitable. Indeed, it is trivial to show that the knowledge-ordered Belnap logic is canonically suitable, since

$$x \curlywedge y = x \otimes_\mathrm{b} y \quad \text{and} \quad x \curlyvee y = x \oplus_\mathrm{b} y.$$

As $\bot$ is the minimum truth value in the lattice $\mathsf{4}_k$, the n-ary selection

operator $S^j_{\mathbf{a}}$ for $\mathsf{4}_k$ is defined by the following function:

$$S^j_{\mathbf{a}}(\mathbf{x}) = \begin{cases} j & \text{if } \mathbf{x} = \mathbf{a}, \\ \bot & \text{otherwise.} \end{cases}$$

Functional completeness also implies all unary selection operators can be expressed as formulae in the logic $L(\mathsf{4}_k, \{\neg, \oplus_{\mathrm{b}}, \supset_{\mathrm{b}}, \bot\})$. However, we have the following result, from which it follows that this logic is not canonically complete.

Proposition 5.4.1. *$L(\mathsf{4}_k, \{\neg, \wedge_{\mathrm{b}}, \vee_{\mathrm{b}}, \otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}, \supset_{\mathrm{b}}, \bot, 0, 1, \top\})$ is not canonically complete.*

Proof. It is impossible to represent all unary selection operators in normal form. The statement follows from the following observations: (i) Belnap logic defines one unary operator $\neg$; (ii) the only binary operators that may be used in normal form are $\oplus_{\mathrm{b}}$ ($\curlywedge$) and $\otimes_{\mathrm{b}}$ ($\curlyvee$); (iii) for any operator $\oplus \in \{\otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}\}$ we have $\bot \oplus \bot = \bot$; and (iv) we have $\neg \bot = \bot$. Thus it is impossible to construct a unary operator of the form $S^d_{\bot}$ for any $d \neq \bot$. □

Corollary 5.4.1. *Any logic $L = (\mathsf{4}_k, \mathsf{Ops})$ where $\mathsf{Ops} \subseteq \{\neg, \wedge_{\mathrm{b}}, \vee_{\mathrm{b}}, \otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}, \supset_{\mathrm{b}}, \bot, 0, 1, \top\}$ is not canonically complete.*

Proof. Since the logic $L(\mathsf{4}_k, \{\neg, \wedge_{\mathrm{b}}, \vee_{\mathrm{b}}, \otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}, \supset_{\mathrm{b}}, \bot, 0, 1, \top\})$ is not canonically complete, any logic which uses a subset of these operators is trivially not canonically complete. □

As a result of Corollary 5.4.1, we can immediately establish that any ABAC authorization language that uses Belnap logic as the underlying logic is not canonically complete. In other words, languages such as PBel [10], BelLog [45] and Rumpole [32] are not canonically complete. It is worth noting that results analogous to those presented above for the knowledge ordering can be obtained for the truth ordering.

5.5 A canonically complete 4-valued logic

Belnap logic is the underlying logic in many authorization languages [10, 32, 45] found in the literature. However, we have proven that Belnap logic is not canonically complete, thus none of these languages are able to utilise the advantages that arise from using a canonically complete logic. Hence, we now investigate how to construct a canonically complete 4-valued logic in which the set of truth values forms a lattice.

In the proof of Proposition 5.4.1, we were unable to construct all unary selection operators using operators from the set $\{\neg, \otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}\}$, because there is no operator in which $\bot \oplus \bot \neq \bot$. This suggests that we will require at least

one additional unary operator $-$, say, such that $-\bot \neq \bot$. Accordingly, we start with the unary operator $-$, sometimes called "conflation" [18], such that

$$-\bot = \top,\ -\top = \bot,\ -0 = 0,\ \text{and}\ -1 = 1.$$

Conflation is analogous to negation $\neg$, but inverts knowledge values rather than truth values. In addition to $-$, we include the operator $\otimes_{\mathrm{b}}$ in our set of operators, since this is the join operator for $\mathsf{4}_k$.

Proposition 5.5.1. *$L(\mathsf{4}_k, \{-, \otimes_{\mathrm{b}}\})$ is canonically suitable.*

Proof. The following equivalence holds [4]:

$$d \oplus_{\mathrm{b}} d' \equiv -(-d \otimes_{\mathrm{b}} -d').$$

□

The decision table establishing this equivalence is given in Figure 5.5. Hence, we conclude that the set of operators $\{-, \otimes_{\mathrm{b}}\}$ is canonically suitable, since $\curlywedge$ corresponds to $\otimes_{\mathrm{b}}$ and $\curlyvee$ corresponds to $\oplus_{\mathrm{b}}$.

d	d'	$-d$	$-d'$	$-d \otimes_{\mathrm{b}} -d'$	$-(-d \otimes_{\mathrm{b}} -d')$	$d \oplus_{\mathrm{b}} d'$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	0	$\top$	0	0	0	0
$\bot$	1	$\top$	1	1	1	1
$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
0	$\bot$	0	$\top$	0	0	0
0	0	0	0	0	0	0
0	1	0	1	$\bot$	$\top$	$\top$
0	$\top$	0	$\bot$	$\bot$	$\top$	$\top$
1	$\bot$	1	$\top$	1	1	1
1	0	1	0	$\bot$	$\top$	$\top$
1	1	1	1	1	1	1
1	$\top$	1	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\top$	0	$\bot$	0	$\bot$	$\top$	$\top$
$\top$	1	$\bot$	1	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

Figure 5.5: Encoding $\oplus_{\mathrm{b}}$ using $-$ and $\otimes_{\mathrm{b}}$

Proposition 5.5.2. *$L(\mathsf{4}_k, \{-, \otimes_{\mathrm{b}}\})$ is not functionally complete.*

Proof. The proof follows from the following observations: (i) for the operators $-$ and $\otimes_{\mathrm{b}}$, $-(0) = 0$ and $0 \otimes_{\mathrm{b}} 0 = 0$; and (ii) for any operator $\circ$ which is a combination of $-$ and $\otimes_{\mathrm{b}}$, we have $0 \circ 0 = 0$. Thus it is impossible to construct an operator in which $0 \circ 0 \neq 0$. □

To summarize: $L(\mathsf{4}_k, \{\neg, \wedge_{\mathrm{b}}, \vee_{\mathrm{b}}, \otimes_{\mathrm{b}}, \oplus_{\mathrm{b}}, \supset_{\mathrm{b}}, \bot, 0, 1, \top\})$ is not canonically complete and $L(\mathsf{4}_k, \{-, \otimes_{\mathrm{b}}\})$ is not functionally complete. We now investigate what additional operators should be defined to construct a set of operators which is canonically complete (and hence functionally complete).

Given that we cannot use any operators besides $\curlyvee$ and $\curlywedge$ in normal form, we focus on defining additional unary operators on $\mathsf{4}_k$. An important observation at this point is that any permutation (that is, a bijection) $\pi : \mathsf{4} \to \mathsf{4}$ defines a unary operator on $\mathsf{4}$. Accordingly, we now explore the connections between the group of permutations on $\mathsf{4}$ and unary operators on $\mathsf{4}$.

5.5.1 The symmetric group and unary operators

The *symmetric group* $(S_X, \circ)$ on a finite set of $|X|$ symbols is the group whose elements are all permutations of the elements in X, and whose group operation $\circ$ is function composition. In other words, given two permutations π_1 and π_2, $\pi_1 \circ \pi_2$ is a permutation such that

$$(\pi_1 \circ \pi_2)(x) \stackrel{\text{def}}{=} \pi_1(\pi_2(x)).$$

We write π^k to denote the permutation obtained by composing π with itself k times.

A *transposition* is a permutation which exchanges two elements and keeps all others fixed. Given two elements a and b in X, the permutation

$$\pi(x) = \begin{cases} b & \text{if } x = a, \\ a & \text{if } x = b, \\ x & \text{otherwise,} \end{cases}$$

is a transposition, which we denote by $(a\, b)$. A *cycle* of *length* $k \geqslant 2$ is a permutation π for which there exists an element x in X such that $x, \pi(x), \pi^2(x), \ldots, \pi^k(x) = x$ are the only elements changed by π. Given a, b and c in X, for example, the permutation

$$\pi(x) = \begin{cases} b & \text{if } x = a, \\ c & \text{if } x = b, \\ a & \text{if } x = c, \\ x & \text{otherwise,} \end{cases}$$

is a cycle of length 3, which we denote by $(a\, b\, c)$. (Cycles of length two are transpositions.) The symmetric group S_X is *generated* by its cycles. That is, every permutation may be represented as the composition of some combination of cycles.

In fact, stronger results are known. We first introduce some notation.

Let $X = \{x_1, \ldots, x_n\}$ and let S_n denote the symmetric group on the set of elements $\{1, \ldots, n\}$. Then $(S_X, \circ)$ is trivially isomorphic to $(S_n, \circ)$ (via the mapping $x_i \mapsto i$).

Theorem 5.5.1. *For $n \geqslant 2$, S_n is generated by the transpositions $(1\,2), (1\,3), \ldots, (1\,n)$.*

Theorem 5.5.2. *For $1 \leqslant a < b \leqslant n$, the transposition $(a\,b)$ and the cycle $(1\,2\,\ldots\,n)$ generate S_n if and only if the greatest common divisor of $b - a$ and n equals 1.*

In other words, it is possible to find a generating set comprising only transpositions, and it is possible to find a generating set containing only two elements.

5.5.2 New unary operators

We now define three unary operators $\sim_0, \sim_1$ and $\sim_\top$, which swap the value of $\bot$ and the truth value in the operator's subscript. The truth tables for these operators are shown in Figure 5.6. Note that $\sim_\top$ is identical to the conflation operator $-$. However, in the interests of continuity and consistency we will use the $\sim_\top$ notation in the remainder of this section.

d	$\sim_0 d$	$\sim_1 d$	$\sim_\top d$
$\bot$	0	1	$\top$
0	$\bot$	0	0
1	1	$\bot$	1
$\top$	$\top$	$\top$	$\bot$

Figure 5.6: $\sim_0, \sim_1$ and $\sim_\top$

Notice that $\sim_0$, $\sim_1$ and $\sim_\top$ permute the elements of $\mathsf{4}$ and correspond to the transpositions $(\bot\,0)$, $(\bot\,1)$ and $(\bot\,\top)$, respectively. Thus we have the following elementary result.

Proposition 5.5.3. *Any permutation on $\mathsf{4}$ can be expressed using only operators from the set $\{\sim_0, \sim_1, \sim_\top\}$.*

Proof. The operators $\sim_0, \sim_1$ and $\sim_\top$ are the transpositions $(\bot\,0), (\bot\,1)$ and $(\bot\,\top)$ respectively. By Theorem 5.5.1, these operators generate all the permutations in S_4. □

Lemma 5.5.1. *It is possible to express any function $\phi : \mathsf{4} \to \mathsf{4}$ as a formula in $L(\mathsf{4}_k, \{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\})$.*

Proof. For convenience, we represent the function $\phi : \mathsf{4} \to \mathsf{4}$ as the tuple

$$\big(\phi(\bot), \phi(0), \phi(1), \phi(\top)\big) = (a, b, c, d).$$

Then, given $x, y, z \in \mathsf{4}$, we define the function

$$\phi_x^y(z) = \begin{cases} x & \text{if } z = y, \\ \bot & \text{otherwise.} \end{cases}$$

Thus, for example, $\phi_a^\bot = (a, \bot, \bot, \bot)$. Then it is easy to see that for all $x \in \mathsf{4}$

$$\phi(x) = \phi_a^\bot(x) \oplus_\mathrm{b} \phi_b^0(x) \oplus_\mathrm{b} \phi_c^1(x) \oplus_\mathrm{b} \phi_d^\top(x).$$

That is $\phi = \phi_a^\bot \oplus_\mathrm{b} \phi_b^0 \oplus_\mathrm{b} \phi_c^1 \oplus_\mathrm{b} \phi_d^\top$.

Thus, it remains to show that we can represent $\phi_a^\bot$, ϕ_b^0, ϕ_c^1 and $\phi_d^\top$ as formulae using the operators in $\{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\}$. First consider the permutations $\phi_{a,0}$, $\phi_{a,1}$ and $\phi_{a,\top}$, represented by the tuples $(a, \bot, b_1, c_1)$, $(a, b_2, \bot, c_2)$ and $(a, b_3, c_3, \bot)$, respectively.[1] Since $\phi_{a,0}$, $\phi_{a,1}$ and $\phi_{a,\top}$ are permutations, we know they can be written as some combination of the unary operators. Moreover,

$$\phi_a^\bot \equiv \phi_{a,0} \otimes_\mathrm{b} \phi_{a,1} \otimes_\mathrm{b} \phi_{a,\top}$$

Clearly, we can construct ϕ_b^0, ϕ_c^1 and $\phi_d^\top$ in a similar fashion. The result now follows. □

To give the reader some visual aide for the proof of Lemma 5.5.1, we depict the decision tables showing the construction of $\phi_a^\bot$ and ϕ in Figure 5.7.

x	$\phi_{a,0}$	$\phi_{a,1}$	$\phi_{a,\top}$	$\phi_{a,0} \otimes_\mathrm{b} \phi_{a,1} \otimes_\mathrm{b} \phi_{a,\top}$
$\bot$	a	a	a	a
0	$\bot$	b_2	b_3	$\bot$
1	b_1	$\bot$	c_3	$\bot$
$\top$	c_1	c_2	$\bot$	$\bot$

(a) $\phi_a^\bot$

x	$\phi_a^\bot$	ϕ_b^0	ϕ_c^1	$\phi_d^\top$	$\phi_a^\bot \oplus_\mathrm{b} \phi_b^0 \oplus_\mathrm{b} \phi_c^1 \oplus_\mathrm{b} \phi_d^\bot$
$\bot$	a	$\bot$	$\bot$	$\bot$	a
0	$\bot$	b	$\bot$	$\bot$	b
1	$\bot$	$\bot$	c	$\bot$	c
$\top$	$\bot$	$\bot$	$\bot$	d	d

(b) ϕ

Figure 5.7: Expressing $\phi : \mathsf{4} \to \mathsf{4}$ using operators in $\{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\}$

Theorem 5.5.3. *$L(\mathsf{4}_k, \{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\})$ is functionally and canonically complete.*

Proof. By Lemma 5.5.1, it is possible to express any function $\phi : \mathsf{4} \to \mathsf{4}$ as a formula using operators from the set $\{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\}$. In particular,

[1] Note that the specific values of b_i and c_i are not important: it suffices that each of $\phi_{a,0}$, $\phi_{a,1}$ and $\phi_{a,\top}$ are permutations; once b_i is chosen such that $b_i \notin \{a, \bot\}$, then c_i is fixed.

all unary selection operators can be expressed in this way. Hence by Theorem 5.3.1, the set of operators $\{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}, \oplus_\mathrm{b}\}$ is functionally complete.

Moreover, all formulae constructed in the proof of Lemma 5.5.1 contain only the binary operators $\oplus_\mathrm{b}(\curlyvee)$ and $\otimes_\mathrm{b}(\curlywedge)$, and unary operators defined as compositions of $\sim_0, \sim_1$ and $\sim_\top$. Thus, by definition, the unary selection operators are in normal form. □

Corollary 5.5.1. *$L(\mathsf{4}_k, \{\sim_0, \sim_1, \sim_\top, \otimes_\mathrm{b}\})$ is functionally and canonically complete.*

Proof. The conflation operator $-$ and $\sim_\top$ are identical. Hence

$$d \oplus_\mathrm{b} d' \equiv -(-d \otimes_\mathrm{b} -d') \equiv \sim_\top(\sim_\top d \otimes_\mathrm{b} \sim_\top d').$$

Therefore, the set of operators is canonically suitable, and, by Theorem 5.5.3, it is functionally and canonically complete (since we can construct $\oplus_\mathrm{b}$). □

Corollary 5.5.2. *Let $\diamond$ be the unary operator corresponding to the permutation given by the cycle $(\bot\, 0\, 1\, \top)$. Then $L(\mathsf{4}_k, \{\sim_\top, \diamond, \otimes_\mathrm{b}\})$ is functionally and canonically complete.*

Proof. By Theorem 5.5.2, $\sim_\top$ and $\diamond$ generate all permutations in S_4. The remainder of the proof follows immediately from Lemma 5.5.1 and Theorem 5.5.3, via replacement of $\{\sim_0, \sim_1, \sim_\top\}$ with $\{\sim_\top, \diamond\}$. □

It is important to note that we could choose any transposition $(a\, b)$, such that $\mathsf{gcd}(b - a, n) = 1$. We specifically selected the transposition $(\bot\, \top)$, as this has the effect of reversing the minimum and maximum knowledge values. Another choice for this transposition is one which swaps 0 and 1, specifically the transposition $(0\, 1)$. This transposition is the truth negation operator $\neg$, which in the context of access control is also a useful operator, since it swaps allow and deny decisions.

5.6 Canonically complete m-valued logics

Having shown the construction for a canonically complete 4-valued logic, in which the set of logical values forms a lattice, we briefly return to totally ordered logics. We now construct a totally ordered, canonically complete m-valued logic (thus extending the work of Jobe [24], who only showed how to construct a canonically complete 3-valued logic).

Let V be a totally ordered set of m truth values, $\{1, \ldots, m\}$, with $1 < \cdots < m$. We define two unary operators $\dagger$ and $\diamond$, which are the transposition $(1\, m)$ and the cycle $(1\, 2\, \ldots\, m)$, respectively. In addition, we define one binary operator $\wedge_t$, where $x \wedge_t y = \min\{x, y\}$ for all $x, y \in \{1, \ldots, m\}$.

Proposition 5.6.1. *Any permutation on V can be expressed using only operators from the set $\{\dagger, \diamond\}$.*

Proof. The operator $\dagger$ is the transposition $(1\, m)$ and the operator $\diamond$ is the cycle $(1\, 2\, \ldots\, m)$. By Theorem 5.5.2, these operators generate all the permutations in S_V. □

Proposition 5.6.2. *$L(V, \{\dagger, \diamond, \wedge_t\})$ is canonically suitable.*

Proof. Clearly $x \curlywedge y \equiv x \wedge_t y$, it remains to show the operator $\curlyvee$ can be expressed in L. By Proposition 5.6.1 we can express any permutation of V in terms of $\dagger$ and $\diamond$. In particular, we can express the permutation f, where $f(i) = m - i + 1$, which swaps the values 1 and m, 2 and $m - 1$, and so on. We denote the unary operator which realizes this permutation by $\updownarrow$. Then $x \curlyvee y \equiv x \vee_t y \equiv \updownarrow(\updownarrow x \wedge_t \updownarrow y)$. □

Theorem 5.6.1. *$L(V, \{\dagger, \diamond, \wedge_t\})$ is functionally and canonically complete.*

We omit the proof, as it proceeds in an analogous manner to those for Lemma 5.5.1 and Theorem 5.5.3. It is interesting to note that we have constructed a canonically complete m-valued logic which uses only two unary operators. This is somewhat unexpected; intuition might suggest that $m - 1$ unary operators would be required for a canonically complete m-valued logic.

5.7 Summary and discussion

The primary goal of this chapter was to extend our understanding of canonical completeness to lattice-based multi-valued logics. In doing so, we are able to determine properties of well-known lattice-based logics such as Belnap logic, upon which a number of ABAC languages are based.

We first showed how Jobe's theoretical foundations may be extended to encompass lattice-based logics, enabling us to reason about the canonical suitability and completeness of such logics. Then, we demonstrated that Belnap logic is not canonically complete, and hence any ABAC language based the Belnap set of operators is not canonically complete.

Given that Belnap logic is not canonically complete, we then explored how we may define a 4-valued canonically complete lattice-based logic. To do so, we identified connections between the symmetric group and unary operators on the set of authorization decisions. This enabled us to leverage theorems from symmetric group theory to precisely define unary operators which create a 4-valued canonically complete logic. Our approach may be trivially generalised to m values, and we showed how the same construction can be used to constructed an m-valued totally-ordered logic.

In the following chapter, we show the practical use of a 4-valued canonically complete lattice-based logic, and formally define the language $\mathrm{PTaCL}^{\leqslant}_{4}$ which makes use of four unique authorization decisions.

Chapter 6

A Canonically Complete 4-valued PTaCL

In Chapter 5, we proved that Belnap logic is not canonically complete. As a direct result, any ABAC authorization language based on Belnap logic is also not canonically complete. In other words, languages such as PBel [10], BelLog [45] and Rumpole [32] are unable to

- express policies in a normal form;
- trivially support the specification of any desired policy; and
- automatically convert policies into machine-enforceable form.

While $\text{PTaCL}_{\mathsf{3}}^{<}$ (Chapter 4) is a canonically complete ABAC authorization language, we have already discussed how the total ordering on the set of decisions ($0 < \bot < 1$) does not accurately reflect the intuition behind the use of these decisions in access control. In this chapter, we present a canonically complete ABAC authorization language in which the set of decisions forms a lattice, and argue why this is a more intuitive structure to use for decisions in an ABAC language. We use the canonically complete logic $L(\mathsf{4}_k, \{\sim_\top, \diamond, \otimes_\mathrm{b}\})$ as a foundation, and develop a 4-valued, latticed-ordered variant of PTaCL, denoted by $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$.

In the following section, we formally define the syntax and semantics for $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$, demonstrating how to construct policies, and present a different method to expressing policies as trees. In Section 6.2 we discuss why XACML's way of handling indeterminacy is flawed, and present an alternative way of handling indeterminacy in $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$. Then, in Section 6.3 we explore how the syntax and semantics of $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$ may be extended to incorporate obligations, an important aspect of many access control mechanisms. We then demonstrate how we may leverage the well-defined parts of XACML, and combine these with the canonically complete language $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$ to produce a canonically complete ABAC language with a standardized, well-defined architecture in Section 6.4. Finally, we develop an algorithm which takes an arbitrary policy

expressed as a decision table, and outputs the equivalent normal form for the policy expressed using the PTaCL$_4^{\leqslant}$ operators in Section 6.4.1. This algorithm enables us to automatically convert policies into machine-enforceable form.

We have published the majority of the work presented in Sections 6.1 and 6.4 [17].

6.1 PTaCL$_4^{\leqslant}$

In Chapter 4, we showed how the operators in PTaCL can be replaced with an alternative set of operators, taken from Jobe's logic $\mathcal{J}$, to obtain the canonically complete 3-valued ABAC language PTaCL$_3^{\leq}$. We now adopt a similar approach, replacing the decision set and operators in PTaCL with the decision set and operators from $L(\mathsf{4}_k, \{\sim_\top, \diamond, \otimes_\mathrm{b}\})$.

6.1.1 Decision set

A fourth decision value (in addition to $0, 1$ and $\bot$) is used in many ABAC languages. The XACML 1.0 standard includes a fourth authorization decision "indeterminate" [35]. This is used to indicate errors have occurred during policy evaluation, meaning that a decision could not be reached. The XACML 3.0 standard extends the definition of the indeterminate decision to indicate decisions that might have been reached, had evaluation been possible [37]. Its use is somewhat ad hoc and confusing since it can be used to indicate (a) an error in policy evaluation, or (b) a decision that arises for a particular operator during normal policy evaluation. More generally, a conflict decision is used in PBel [10] (and languages such as Rumpole [32] and BelLog [45]) to indicate that two sub-policies return different conclusive decisions.

We will use this fourth value to denote that (normal) policy evaluation has led to conflicting decisions (and we do not wish to use deny-overrides or similar operators to resolve the conflict at this point in the evaluation). Thus, the decision set for PTaCL$_4^{\leqslant}$ is $\{0, 1, \bot, \top\}$, which is identical to the Belnap decision set $\mathsf{4}$. We explain how we handle indeterminacy arising from errors in policy evaluation in Section 6.2. Two specific operators, "only-one-applicable" (ooa) and "unanimity" (un) could make use of $\top$: the ooa operator returns the value of the applicable sub-policy if there is only one such policy, and $\top$ otherwise; whereas the un operator returns $\top$ if the sub-policies return different decisions, and the common decision otherwise. The decision tables for these operators are shown in Figure 6.1.

In the context of access control, 0 and 1 are incomparable conclusive decisions, and $\bot$ and $\top$ are decisions that reflect the inability to reach a conclusive decision either because a policy or its sub-policies are inapplicable ($\bot$) or because a policy's sub-policies return conclusive decisions that are incompatible in some sense ($\top$). Moreover, we can subsequently resolve $\bot$ and $\top$ into one of

ooa	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	$\top$	$\top$	$\top$
1	1	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

un	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$
0	$\top$	0	$\top$	$\top$
1	$\top$	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figure 6.1: Operators using $\top$

two (incomparable) conclusive decisions using unary operators such as "deny-by-default" and "permit-by-default". Due to this intuition, we chose to use the knowledge-based ordering on 4 from Belnap logic as the ordering for decisions in $\text{PTaCL}^{\leqslant}_{4}$. (The truth-based ordering on 4 does not correspond nearly so well to the above intuitions.)

6.1.2 Operators and policies

We define the set of operators for $\text{PTaCL}^{\leqslant}_{4}$ to be $\{\sim_{\top}, \diamond, \otimes_{\text{b}}\}$, which we established is canonically complete in Corollary 5.5.2. Recall that $\sim_{\top}$ is equivalent to conflation $-$; we will use the simpler notation $-$ in the remainder of this chapter. An atomic policy has the form (t, d), where t is a target and $d \in \{0, 1\}$. (There is no reason for an atomic policy to return $\top$ – which signifies a conflict has taken place – in an atomic policy.) Then we have the policy semantics defined in Figure 6.2b.

d	$-d$	$\diamond d$
$\bot$	$\top$	0
0	0	1
1	1	$\top$
$\top$	$\bot$	$\bot$

$\otimes_{\text{b}}$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$	0
1	$\bot$	$\bot$	1	1
$\top$	$\bot$	0	1	$\top$

(a) $-, \diamond$ and $\otimes_{\text{b}}$

$$
\begin{aligned}
\rho_q(d) &= d; \\
\rho_q(-p) &= -\rho_q(p); \\
\rho_q(\diamond p) &= \diamond \rho_q(p); \\
\rho_q(p \otimes_{\text{b}} p') &= \rho_q(p) \otimes_{\text{b}} \rho_q(p'); \\
\rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\text{m}}, \\ \bot & \text{otherwise.} \end{cases}
\end{aligned}
$$

(b) Policy semantics

Figure 6.2: Decision operators and policy semantics in $\text{PTaCL}^{\leqslant}_{4}$

Representing ooa in other languages such as PTaCL and PBel is possible due to their functional completeness, however it is a non-trivial task to do so using the operators defined in each language. We now show how to represent the operator only-one-applicable (ooa) in normal form. Using the truth table in Figure 6.1 and by definition of the selection operators and Υ, we have

x ooa y is equivalent to

$$
\begin{aligned}
&S^{\bot}_{(\bot,\bot)}(x,y) \curlyvee S^{0}_{(\bot,0)}(x,y) \curlyvee S^{1}_{(\bot,1)}(x,y) \curlyvee S^{\top}_{(\bot,\top)}(x,y) \curlyvee \\
&S^{0}_{(0,\bot)}(x,y) \curlyvee S^{\top}_{(0,0)}(x,y) \curlyvee S^{\top}_{(0,1)}(x,y) \curlyvee S^{\top}_{(0,\top)}(x,y) \curlyvee \\
&S^{1}_{(1,\bot)}(x,y) \curlyvee S^{\top}_{(1,0)}(x,y) \curlyvee S^{\top}_{(1,1)}(x,y) \curlyvee S^{\top}_{(1,\top)}(x,y) \curlyvee \\
&S^{\top}_{(\top,\bot)}(x,y) \curlyvee S^{\top}_{(\top,0)}(x,y) \curlyvee S^{\top}_{(\top,1)}(x,y) \curlyvee S^{\top}_{(\top,\top)}(x,y).
\end{aligned}
$$

Moreover, $S^{c}_{(a,b)}(x,y) = S^{c}_{a}(x) \curlywedge S^{c}_{b}(y)$. Hence, x ooa y is equivalent to

$$
\begin{aligned}
&\Big(S^{\bot}_{\bot}(x) \curlywedge S^{\bot}_{\bot}(y)\Big) \curlyvee \Big(S^{0}_{\bot}(x) \curlywedge S^{0}_{0}(y)\Big) \curlyvee \Big(S^{1}_{\bot}(x) \curlywedge S^{1}_{1}(y)\Big) \curlyvee \Big(S^{\top}_{\bot}(x) \curlywedge S^{\top}_{\top}(y)\Big) \curlyvee \\
&\Big(S^{0}_{0}(x) \curlywedge S^{0}_{\bot}(y)\Big) \curlyvee \Big(S^{\top}_{0}(x) \curlywedge S^{\top}_{0}(y)\Big) \curlyvee \Big(S^{\top}_{0}(x) \curlywedge S^{\top}_{1}(y)\Big) \curlyvee \Big(S^{\top}_{0}(x) \curlywedge S^{\top}_{\top}(y)\Big) \curlyvee \\
&\Big(S^{1}_{1}(x) \curlywedge S^{1}_{\bot}(y)\Big) \curlyvee \Big(S^{\top}_{1}(x) \curlywedge S^{\top}_{0}(y)\Big) \curlyvee \Big(S^{\top}_{1}(x) \curlywedge S^{\top}_{1}(y)\Big) \curlyvee \Big(S^{\top}_{1}(x) \curlywedge S^{\top}_{\top}(y)\Big) \curlyvee \\
&\Big(S^{\top}_{\top}(x) \curlywedge S^{\top}_{\bot}(y)\Big) \curlyvee \Big(S^{\top}_{\top}(x) \curlywedge S^{\top}_{0}(y)\Big) \curlyvee \Big(S^{\top}_{\top}(x) \curlywedge S^{\top}_{1}(y)\Big) \curlyvee \Big(S^{\top}_{\top}(x) \curlywedge S^{\top}_{\top}(y)\Big).
\end{aligned}
$$

Each unary selection operator S^{b}_{a} is a function $\phi : \mathsf{4} \to \mathsf{4}$, which can be represented in terms of the operators $\{-, \diamond, \otimes_{\mathrm{b}}\}$. Indeed, we have derived expressions in normal form for the unary selection operators S^{b}_{a}, which are shown in Figure 6.3. (In Lemma 5.5.1 we only showed that such expressions exist.) Note that $S^{\bot}_{a}(x) = \bot$ for all $a, x \in \{0, 1, \bot, \top\}$.

$S^{\bot}_{a}(x)$	$x \otimes_{\mathrm{b}} (\diamond - x) \otimes_{\mathrm{b}} (\diamond - \diamond - x)$
$S^{0}_{\bot}(x)$	$(\diamond x) \otimes_{\mathrm{b}} (- \diamond - \diamond - x) \otimes_{\mathrm{b}} (- \diamond - \diamond x)$
$S^{0}_{0}(x)$	$x \otimes_{\mathrm{b}} (- x) \otimes_{\mathrm{b}} (- \diamond - x)$
$S^{0}_{1}(x)$	$(\diamond - \diamond - x) \otimes_{\mathrm{b}} (- \diamond - \diamond - x) \otimes_{\mathrm{b}} (\diamond x)$
$S^{0}_{\top}(x)$	$(\diamond - x) \otimes_{\mathrm{b}} (- \diamond - x) \otimes_{\mathrm{b}} (\diamond \diamond - \diamond x)$
$S^{1}_{\bot}(x)$	$(\diamond - \diamond x) \otimes_{\mathrm{b}} (- \diamond - \diamond x) \otimes_{\mathrm{b}} (\diamond \diamond x)$
$S^{1}_{0}(x)$	$(\diamond x) \otimes_{\mathrm{b}} (- \diamond x) \otimes_{\mathrm{b}} (\diamond - x)$
$S^{1}_{1}(x)$	$x \otimes_{\mathrm{b}} (- x) \otimes_{\mathrm{b}} (\diamond \diamond - \diamond x)$
$S^{1}_{\top}(x)$	$(\diamond - \diamond - x) \otimes_{\mathrm{b}} (- \diamond - \diamond - x) \otimes_{\mathrm{b}} (\diamond \diamond - x)$
$S^{\top}_{\bot}(x)$	$(- \diamond - x) \otimes_{\mathrm{b}} (- \diamond - \diamond - x) \otimes_{\mathrm{b}} (- x)$
$S^{\top}_{0}(x)$	$(\diamond - \diamond - x) \otimes_{\mathrm{b}} (\diamond - \diamond x) \otimes_{\mathrm{b}} (\diamond \diamond x)$
$S^{\top}_{1}(x)$	$(\diamond x) \otimes_{\mathrm{b}} (\diamond - x) \otimes_{\mathrm{b}} (\diamond - \diamond \diamond - \diamond x)$
$S^{\top}_{\top}(x)$	$x \otimes_{\mathrm{b}} (- \diamond x) \otimes_{\mathrm{b}} (- \diamond - \diamond x)$

Figure 6.3: Normal forms for the unary selection operators

Hence, we may replace each instance of S^{b}_{a} with the equivalent expression in terms of $\{-, \diamond, \otimes_{\mathrm{b}}\}$, resulting in a formula in normal form for ooa. (We omit the full expression of ooa in terms of $\{-, \diamond, \otimes_{\mathrm{b}}\}$ here, as it is lengthy and can be produced via simple substitution.)

6.1.3 An alternative method for policy specification

Functional completeness implies we can write any binary operator (such as XACML's deny-overrides policy-combining algorithm) as a formula in $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\mathrm{b}}\})$, and hence we can use any operator we wish in $\mathrm{PTaCL}^{\leqslant}_{\mathsf{4}}$ policies. However, canonical completeness and the decision set $\mathsf{4}_k$ allows for a completely different approach to constructing ABAC policies. Suppose for example, a policy administrator has identified three sub-policies p_1, p_2 and p_3 and wishes to define an overall policy p in terms of the decisions obtained by evaluating these sub-policies. Then the policy administrator can tabulate the desired decision for all relevant combinations of decisions for the sub-policies, as shown in the table below. The default decision is to return $\bot$, indicating that p is "silent" for other combinations.

p_1	p_2	p_3	p
$\bot$	0	0	0
0	0	0	0
1	0	0	$\top$
1	1	0	1
1	1	1	1

Then, treating p as a function of its sub-policies, we have

$$p \equiv\ S^0_{(\bot,0,0)} \curlyvee S^0_{(0,0,0)} \curlyvee S^{\top}_{(1,0,0)} \curlyvee S^1_{(1,1,0)} \curlyvee S^1_{(1,1,1)}.$$

(We omit the arguments (p_1, p_2, p_3) from the notation of each selection operator, it is obvious from context.) The construction of p as a "disjunction" ($\curlyvee$) of selection operators ensures that the correct value is returned for each combination of values (in the same way in which disjunctive normal form may be used to represent the rows in a truth table in traditional 2-valued propositional logic). Note that if p_1, p_2 and p_3 evaluate to a tuple of values other than one of the rows in the table, each of the selection operators will return $\bot$ and thus p will evaluate to $\bot$. Each operator of the form $S^d_{(a,b,c)}$ can be represented as the "conjunction" ($\curlywedge$) of unary selection operators (specifically $S^d_a \curlywedge S^d_b \curlywedge S^d_c$). We may then substitute each S^b_a with the equivalent expressions in terms of $\{-, \diamond, \otimes_{\mathrm{b}}\}$ from Figure 6.3, in an analogous way to the construction of ooa.

Thus, we have developed an alternative method for specifying policies, that is simple and intuitive for policy authors. By representing a policy as a decision table, it is obvious to the policy author how the policy will behave under each different evaluation of the sub-policies, something that is not clear in conventional tree-structured policies and languages (we reinforce this point in Chapter 7). Furthermore, we can convert policies expressed as tables into expressions that comprise entirely of the operators defined in $\mathrm{PTaCL}^{\leqslant}_{\mathsf{4}}$, which are in normal form and machine-enforceable. In Chapter 7 we extend this idea

further, developing an ABAC authorization language centred around decision tables.

Of course, one would not usually construct the normal form representation of policies by hand, as we have done above. Indeed, we have developed an algorithm which takes an arbitrary policy expressed as a decision table as input, and outputs the equivalent normal form expressed in terms of the operators $\{-, \diamond, \otimes_{\mathrm{b}}\}$. We present and analyse this algorithm in Section 6.4.1.

6.2 Indeterminacy in PTaCL$_4^{\leqslant}$

As we have discussed throughout this thesis, XACML uses the indeterminate value in two distinct ways:

1. as a decision returned (during normal evaluation) by the "only-one-applicable" policy-combining algorithm; and

2. as a decision returned when some (unexpected) error has occurred in policy evaluation.

In the second case, the indeterminate value is used to represent alternative outcomes of policy evaluation (had the error not occurred). We believe that the two situations described above are quite distinct and require different policy semantics. However, the semantics of indeterminacy in XACML are confused because (i) the indeterminate value is used in two different ways, as described above, and (ii) there is no clear and uniform way of establishing the values returned by the combining algorithms when an indeterminate value is encountered.

We have seen how $\top$ may be used to represent decisions for operators such as ooa and un. We handle errors in target evaluation (and thus indeterminacy) using sets of possible decisions [11, 12, 29]. (This approach was adopted in a rather ad hoc fashion in XACML 3.0, using an extended version of the indeterminate decision.) Informally, when target evaluation fails, denoted by $\tau_q(t) = ?_{\mathrm{m}}$, PTaCL assumes that either $\tau_q(t) = 1_{\mathrm{m}}$ or $\tau_q(t) = 0_{\mathrm{m}}$ could have been returned, and returns the union of the (sets of) decisions that would have been returned in both cases.

We may easily extend this same technique to PTaCL$_4^{\leqslant}$. The formal semantics for policy evaluation in PTaCL$_4^{\leqslant}$ in the presence of indeterminacy are defined in Figure 6.4.

The semantics for the operators $\{-, \diamond, \otimes_{\mathrm{b}}\}$ operate on sets, rather than single decisions, in the natural way. A straightforward induction on the number of operators in a policy establishes that the decision set returned by these extended semantics will be a singleton if no target evaluation errors occur; moreover, that decision will be the same as that returned by the standard semantics (see Lemma 4.4.1).

$$
\begin{aligned}
\rho_q(d) &= \{d\}\,; \\
\rho_q(-p) &= \{-d : d \in \rho_q(p)\}\,; \\
\rho_q(\diamond\, p) &= \{\diamond\, d : d \in \rho_q(p)\}\,; \\
\rho_q(p_1 \otimes_{\mathrm{b}} p_2) &= \{d_1 \otimes_{\mathrm{b}} d_2 : d_i \in \rho_q(p_i)\}\,; \\
\rho_q(t, p) &= \begin{cases} \rho_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}}, \\ \{\bot\} & \text{if } \tau_q(t) = 0_{\mathrm{m}}, \\ \{\bot\} \cup \rho_q(p) & \text{if } \tau_q(t) = ?_{\mathrm{m}}. \end{cases}
\end{aligned}
$$

Figure 6.4: Semantics for $\mathrm{PTaCL}_4^{\leqslant}$ with indeterminacy

6.3 Obligations in $\mathrm{PTaCL}_4^{\leqslant}$

In Section 4.5, we formally defined syntax for obligations in $\mathrm{PTaCL}_3^{\leqslant}$, semantics for computing obligations, and demonstrated the ease with which obligations can be computed for arbitrary policy operators. We now extend the semantics of $\mathrm{PTaCL}_4^{\leqslant}$ to incorporate obligations, taking an identical approach to the one taken in Section 4.5. The only extra considerations we make are: how the inclusion of a fourth decision, $\top$, affects obligations, and how the policy operator $\otimes_{\mathrm{b}}$ should compute obligations. Once again, we assume the existence of some "abstract" set of obligations O. Then, we extend $\mathrm{PTaCL}_4^{\leqslant}$ syntax in the following ways.

- The $\mathrm{PTaCL}_4^{\leqslant}$ policy d, where $d \in \{0, 1\}$, may only return d, so it suffices to extend the syntax for such policies to (d, o) (where $o \in O$). (Recall that atomic policies can not return $\top$.)

- The unary policy operators $-$ and $\diamond$ are used only to switch policy decisions, so we will assume that obligations are not associated with these operators. When evaluating policies with the operators $-$ and $\diamond$ the obligations from child nodes are passed up with no change.

- All other policies (generated using $\otimes_{\mathrm{b}}$ or targets) may return $0, 1, \bot$ or $\top$, so we extend the syntax for a policy p to $(p, o_0, o_1, o_\top)$, where $o_i \in O$ is the obligation that should be returned if the evaluation of p returns decision $i \in \{0, 1, \top\}$.

The $\mathrm{PTaCL}_4^{\leqslant}$ obligation semantics are shown in Figure 6.5. Similar to $\mathrm{PTaCL}_3^{\leqslant}$, the interesting case is the operator $\otimes_{\mathrm{b}}$, which acts as a greatest lower bound operator. Keeping with the approach of the XACML standard and $\mathrm{PTaCL}_3^{\leqslant}$, we take only the obligations from child policies that return a decision equal to that of the parent policy. Thus we take obligations from both child policies if they return the same decision (as well as the relevant obligation from the parent policy). The addition of $\top$ introduces new combinations of

decisions and obligations. More formally, if a child policy p_i returns $d \in \{0, 1\}$ and the other policy returns $\top$, then we return $\{o_d\} \cup \sigma_q(p_i)$. If both child policies return $\top$, then we return only the conflict obligation $o_\top$ from the parent policy (and no obligations from the child policies). In all other cases, the decision returned is $\bot$ and the obligation set is empty. We interpret $\{\epsilon\}$ as the empty set $\emptyset$.

$$\sigma_q(0, o) = \sigma_q(1, o) = \{o\}$$
$$\sigma_q(-p) = \sigma_q(\diamond p) = \sigma_q(p)$$
$$\sigma_q(p_1 \otimes_{\mathrm{b}} p_2, o_0, o_1, o_\top) = \begin{cases} \{o_0\} \cup \sigma_q(p_1) & \text{if } \rho_q(p_1) = 0 \text{ and } \rho_q(p_2) = \top \\ \{o_0\} \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = \top \text{ and } \rho_q(p_2) = 0 \\ \{o_0\} \cup \sigma_q(p_1) \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = 0 \text{ and } \rho_q(p_2) = 0 \\ \{o_1\} \cup \sigma_q(p_1) & \text{if } \rho_q(p_1) = 1 \text{ and } \rho_q(p_2) = \top \\ \{o_1\} \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = \top \text{ and } \rho_q(p_2) = 1 \\ \{o_1\} \cup \sigma_q(p_1) \cup \sigma_q(p_2) & \text{if } \rho_q(p_1) = 1 \text{ and } \rho_q(p_2) = 1 \\ \{o_\top\} & \text{if } \rho_q(p_1) = \top \text{ and } \rho_q(p_2) = \top \\ \emptyset & \text{otherwise} \end{cases}$$
$$\sigma_q(t, p, o_0, o_1, o_\top) = \begin{cases} \{o_0\} \cup \sigma_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}} \text{ and } \rho_q(p) = 0 \\ \{o_1\} \cup \sigma_q(p) & \text{if } \tau_q(t) = 1_{\mathrm{m}} \text{ and } \rho_q(p) = 1 \\ \{o_\top\} & \text{if } \tau_q(t) = 1_{\mathrm{m}} \text{ and } \rho_q(p) = \top \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 6.5: Obligation semantics in $\mathrm{PTaCL}^{\leqslant}_{4}$

The remainder of the techniques introduced in Section 4.5 can be directly applied to $\mathrm{PTaCL}^{\leqslant}_{4}$ in combination with the semantics from Figure 6.5. That includes techniques for: (i) building evaluation tables for operators, (ii) computing operators for derived policy operators, and (iii) computing obligations in the presence of indeterminacy. Hence we do not reiterate these here for $\mathrm{PTaCL}^{\leqslant}_{4}$.

6.4 Leveraging the XACML architecture

XACML is a well-known, standardized language, and many of the components and features of XACML are well-defined. However, we demonstrated in Chapter 3 that XACML is not functionally complete, meaning there are certain policies that cannot be expressed in XACML (without the use of custom combining algorithms). Furthermore, it has been shown that the rule- and policy-combining algorithms defined in the XACML standard suffer from some shortcomings [29], notably inconsistencies between the rule- and policy-combining algorithms. Lastly, we showed that there are significant redundancies between the rule- and policy-combining algorithms. PTaCL, on which

$\text{PTaCL}_4^{\leqslant}$ is based, is a tree-structured ABAC language that is explicitly designed to use the same general policy structure and evaluation methods as XACML. However, PTaCL differs substantially from XACML in terms of policy combination operators and semantics.

Thus, we suggest that $\text{PTaCL}_4^{\leqslant}$ operators could replace the rule- and policy-combining algorithms of XACML, while those parts of the language and architecture that seem to function well may be retained. Specifically, we use the XACML architecture to: (i) specify requests; (ii) specify targets; (iii) decide whether a policy target is applicable to a given request; and (iv) use the policy decision point to evaluate policies. In addition, we would retain the enforcement architecture of XACML, in terms of the policy decision, policy enforcement and policy administration points, and the relationships between them (see Section 2.2.3).

We believe it would be relatively easy to modify the XACML PDP to

- handle four decisions, extending the current set of values ("allow", "deny" and "not-applicable") to include "conflict";
- implement the policy operators $\{-, \diamond, \otimes_{\text{b}}\}$ as custom combining algorithms; and
- work with decision sets, in order to handle indeterminacy in a uniform manner.

In Figures 6.6 and 6.7 we illustrate how $\text{PTaCL}_4^{\leqslant}$ extensions could be incorporated in XACML by encoding the $\text{PTaCL}_4^{\leqslant}$ decision set and $\otimes_{\text{b}}$ operator using the syntax of the XACML standard.

```
<xs:element name=''Decision'' type=''xacml:DecisionType''/>
<xs:simpleType name=''DecisionType''>
  <xs:restriction base=''xs:string''>
     <xs:enumeration value=''Permit''/>
     <xs:enumeration value=''Deny''/>
     <xs:enumeration value=''Conflict''/>
     <xs:enumeration value=''NotApplicable''/>
  </xs:restriction>
</xs:simpleType>
```

Figure 6.6: The $\text{PTaCL}_4^{\leqslant}$ decision set in XACML syntax

The main difference to end-users would be in the simplicity of policy authoring. Using standard XACML, policy authors must decide which rule- and policy-combining algorithms should be used to develop a policy or policy set that is equivalent to the desired policy. This can be error-prone, and it may not even be possible to express the desired policy using only the XACML combining algorithms (due to the functional incompleteness of XACML). Using XACML with the policy-combining mechanisms of $\text{PTaCL}_4^{\leqslant}$, we can present an entirely different interface for policy authoring to the end-user. The policy author would first specify the atomic policies (XACML rules), then combine

```
Decision ptaclCombiningAlgorithm(Node[] children)
{
  Boolean atLeastOneDeny  = false;
  Boolean atLeastOnePermit  = false;
  for( i=0 ; i < lengthOf(children) ; i++ )
  {
     Decision decision = children[i].evaluate();
     if (decision == NotApplicable)
     {  return NotApplicable;  }
     if (decision == Permit)
     {
       atLeastOnePermit = true;
       continue;
     }
     if (decision == Deny)
     {
       atLeastOneDeny = true;
       continue;
     }
     if (decision == Conflict)
     {  continue;  }
  }
  if (atLeastOneDeny && atLeastOnePermit)
  {  return NotApplicable;  }
  if (atLeastOneDeny)
  {  return Deny;  }
  if (atLeastOnePermit)
  {  return Permit;  }
  return Conflict;
}
```

Figure 6.7: The $\text{PTaCL}_4^{\leqslant}$ operator $\otimes_{\text{b}}$ encoded as an XACML combining algorithm

atomic policies using decision tables to obtain more complex policies (as illustrated in Section 6.1.2). Those policies can be further combined by specifying additional decision tables. At each stage a back-end policy compiler can be used to convert those policies into policy sets (using $\text{PTaCL}_4^{\leqslant}$ operators) that can be evaluated by the XACML PDP.

6.4.1 Automatic policy generation

We demonstrated in Section 6.1.3 how to convert policies expressed as decision tables into expressions that use only the operators defined in $\text{PTaCL}_4^{\leqslant}$, and mentioned an algorithm for automating this process. We now present said algorithm.

The output of our automatic policy generation algorithm only contains operators from the set $\{-, \diamond, \otimes_{\text{b}}\}$. Taking the approach discussed above, that is, replacing the XACML combining algorithms with custom combining algorithms that implement $\{-, \diamond, \otimes_{\text{b}}\}$, our algorithm then outputs "machine-enforceable" policies that can be parsed by the (custom) policy enforcement point in XACML. Our implementation of the algorithm, comprising just 185 lines of Python code (see Appendix A.2), shows the ease with which construction of policies can be both automated and simplified, utilizing the numerous advantages of a canonically complete language that have have been discussed throughout this thesis.

We present the pseudo code for our algorithm in Algorithm 1. The input of our algorithm is an $r \times c$ decision table, stored as a 2-dimensional array, where r is the number of rows and c is the number of columns. Lines 4 to 6 convert each row in the decision table into their equivalent representation as unary selection operators. The for-loop runs over r rows, which splits each row into $c-1$ variables, thus the time complexity is $\mathcal{O}(r(c-1))$. Lines 7 to 11 convert each unary selection operator into an equivalent expression in terms of the $\text{PTaCL}_4^{\leqslant}$ operators $\{-, \diamond, \otimes_{\text{b}}\}$. A nested for-loop is required, to convert each entry of the 2-dimensional array (which contain individual unary selection operators) into their equivalent expression. The time complexity for this nested for-loop is $\mathcal{O}(r(c-1))$. Finally, lines 12 to 15 concatenate each row of the decision table with the $\curlyvee$ operator, and return the final expression in normal form. A single for-loop is used, resulting in a time complexity of $\mathcal{O}(r)$.

To summarise, the total time complexity of our algorithm is $\mathcal{O}(rc)$. In the worst case scenario, $\mathcal{O}(4^c)$ rows will be required, however in practice this is very unlikely as rows which return $\bot$ may be omitted by design (as $\bot$ is the default policy decision).

Algorithm 1 Convert a decision table to normal form

```
1: function ConvertPolToNormalForm(PolArr)
2:     Input: A 2-dimensional policy array PolArr, with r rows and c columns
3:     Output: Equivalent normal form in terms of −, ⋄ and ⊗b
4:     for each row in PolArr do
5:         PolArr(row) ← ConvertStringToUnarySelop(PolArr(row))  ▷ converts a string 010⊥|1 to S_0^1 ↓ S_1^1 ...
6:     end for
7:     for each row in PolArr do
8:         for each col in PolArr do
9:             PolArr(row,col) ← ConvertUnarySelopToPTaCLOps(PolArr(row,col))  ▷ converts S_0^1 to (x ↓ (⋄ x))
10:        end for
11:    end for
12:    for each row in PolArr do
13:        OutputNF ← ConcatenatePolArr(PolArr(row)) ▷ Joins each row ((x ↓ (⋄ x)) ↓ (−x)) ... ) with ⋎
14:    end for
15:    return OutputNF
16: end function
```

We can also calculate the space complexity required by our algorithm. Initially, we require space complexity $\mathcal{O}(rc)$ to store the input 2-dimensional policy array. Lines 4 to 6 require space complexity $\mathcal{O}(r(c-1))$ to store each unary selection operator; the final column is lost from the 2-dimensional policy array as this is the superscript in each unary selection operator. Likewise, lines 7 to 11 require space complexity $\mathcal{O}(r(c-1))$ to store each equivalent expression of the unary selection operators. Lines 12 to 15 require space complexity $\mathcal{O}(r)$ to store the concatenated string, which is the final policy output in normal form. In summary, the total space complexity of our algorithm is $\mathcal{O}(rc)$.

It is worth noting the ease in which our algorithm can be adapted to handle any canonically complete language. Indeed, the only part of our algorithm

that needs changing is line 9, that is, the conversion of the unary selection operators into equivalent expressions in terms of the $\text{PTaCL}_4^{\leqslant}$ operators. The equivalent expressions for the unary selection operators may be replaced with those from another language. For example, we may use the normal forms for the unary selection operators for $\text{PTaCL}_3^{<}$ shown in Figure 4.5. Hence, our algorithm can be used to automatically generate policies in $\text{PTaCL}_3^{<}$, and any other conceivable canonically complete authorization language.

6.5 Summary and discussion

In this chapter, we have shown how the canonically complete set of operators $\{-, \diamond, \otimes_{\text{b}}\}$ can be used as the foundation of an ABAC language, and present the advantages of doing so. In particular, we are no longer forced to use a totally ordered set of three decisions to obtain canonical completeness (as is the case in $\text{PTaCL}_3^{<}$). Moreover, the overall design of PTaCL, and hence $\text{PTaCL}_4^{\leqslant}$, is compatible with the overall structure of XACML policies. We discussed how the XACML decision set and combining algorithms can be modified to support $\text{PTaCL}_4^{\leqslant}$. Doing so enables us to retain the rich framework provided by XACML for ABAC, alongside the benefits of a canonically complete ABAC language.

Thus, we are able to propose an enhanced XACML framework within which any desired policy may be expressed. Moreover, the canonical completeness of $\text{PTaCL}_4^{\leqslant}$ means that the desired policy may be represented in simple terms by a policy author (in the form of a decision table) and automatically compiled into a PDP-readable equivalent policy. We demonstrated an algorithm for this conversion, and commented on the ease in which it can be adapted for *any* canonically complete ABAC language (for instance, $\text{PTaCL}_3^{<}$).

Our work paves the way for a considerable amount of future work. In particular, we intend to develop a modified XACML PDP that implements the $\text{PTaCL}_4^{\leqslant}$ operators. We also hope to develop a policy authoring interface in which users can simply state what decisions a policy should return for a particular combination of decisions from sub-policies. This would enable us to evaluate the usability of such an interface and compare the accuracy with which policy authors can generate policies using standard XACML combining algorithms compared with the methods that $\text{PTaCL}_4^{\leqslant}$ can support.

On the more technical side, we would like to revisit the notion of *monotonicity* [12] in targets and how this affects policy evaluation in ABAC languages. The definition of monotonicity is dependant on the ordering chosen for the decision set and existing work on monotonicity assumes the use of a total-ordered 3-valued set (comprising 0, $\bot$ and 1). So it will be interesting to consider how the use of a 4-valued lattice-ordered decision set affects monotonicity. Motivation can be taken from work by Crampton and Morisset [13], which explores monotonically complete languages.

In the following chapter we further develop the idea of using decision tables to specify policies (opposed to specifying them as trees), and build a new ABAC language with the core foundations based on decision tables (rather than modifying an existing tree-structured language like PTaCL to support decision tables). We will also explore methods for reducing the size of decision tables (policy compression), since these tables can grow quickly in size as more sub-policies are specified. The compression techniques we develop may also be applied to decision tables found in $\text{PTaCL}_3^{\leqslant}$ and $\text{PTaCL}_4^{\leqslant}$.

Chapter 7

Attribute Expressions and Policy Tables

Traditionally in attribute-based access control, subjects and objects are associated with attributes, requests are collections of attributes associated with the subjects and objects, and these attributes determine whether a request is authorized or not. We may imagine representing a policy as a table in which columns are indexed by attributes, rows represent the presence or otherwise of the respective attribute in a request, with the final column in the table indicating the authorization decision associated with a particular collection of attributes.

A simple example is shown in Table 7.1, where 1 indicates the presence of the attribute in the request and 0 indicates the attribute is absent; the dash indicates that the presence or otherwise of a particular attribute is irrelevant to the decision. Thus the first row of the table indicates that the deny decision (0) should be returned if attributes a_1 and a_2 are present in a request. If no row exists for a particular combination, then we assume that the decision is $\perp$ ("not-applicable"); that is, the policy is "silent" for such a request and does not return a conclusive decision. Thus, for example, the decision is $\perp$ if attribute a_1 is not present in the request.

a_1	a_2	a_3	d
1	1	–	0
1	0	1	1

Table 7.1: A simple policy table

While it is certainly convenient and intuitive to represent authorization policies in tabular form, this representation is not compatible with many languages that have been discussed through this thesis. XACML [37], PTaCL [12] and PBel [10], for example, are tree-structured languages, where policies are, essentially, terms in a logic-based formalism. These terms may be represented by trees, in which leaf nodes are attribute-decision pairs and interior nodes

are attribute-operator pairs.

Figure 7.1 illustrates the policy $\big((a_1, \mathsf{do}), ((a_2, 0), (a_3, 1))\big)$. (Recall, the operator do represents the XACML "deny-overrides" operator.) A request is evaluated by first pruning the nodes that are not matched by the attributes in the request. Then the decisions in the remaining leaf nodes are combined using the policy combining operator(s). If, for example, all attributes are present in a request (so no pruning is performed), then the resulting decision is $0 \ \mathsf{do} \ 1 = 0$, corresponding to the first line in Table 7.1.

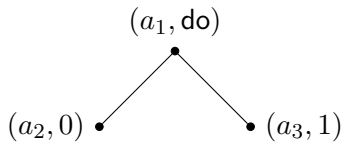


Figure 7.1: A simple tree-structured policy

In fact, the tree in Figure 7.1 represents an equivalent policy to the one tabulated in Table 7.1, although this is not immediately apparent. It is this gap – between (i) how one is likely to conceive of a policy, and (ii) how one must construct the policy using existing languages for attribute-based access control – that provides the motivation for the work in this chapter.

A further shortcoming of existing work on languages for attribute-based access control is the way in which requests and attributes are matched. Suppose we have an attribute name-value pair (n, v) and a request that contains multiple name-value pairs, including (n, v) and (n, v'), where $v' \neq v$. Then one might argue the request matches the attribute (since it contains (n, v)); on the other hand, one might argue it doesn't match the attribute (since it also contains (n, v')). XACML always assumes the former interpretation, which may be inappropriate if, for example, the policy author wishes to insist that the request contains exactly one name-value pair for the named attribute. Although PTaCL has a slightly more complex match semantics for requests and attributes, it ignores several possible match semantics that might be relevant in practice.

In this chapter, we use our previous work on canonical completeness to develop a new way of defining authorization policies using policy tables. In doing so, we support all possible match semantics for an attribute and request, thereby facilitating much greater control over policy specification. In the next section we formally define our Attribute Expression Policy Language (AEPL). Then, we demonstrate various methods for reducing the size of policy tables in AEPL in Section 7.2. We compare AEPL with XACML and PTaCL in Section 7.3, and show that AEPL is more expressive and intuitive than both of these languages. Finally, we show how such policy tables can be used to enhance existing access control paradigms, such as access control lists and role-based access control, by making them "attribute-aware" in Section 7.4.

We will use the logic $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\mathrm{b}}\})$ as the underlying logic when we develop our new policy authorization language.

We have published the majority of the work presented in this chapter [16]. We note that Section 7.1 formalises various concepts presented throughout this thesis, such as policy tables, and introduces methods for target evaluation that act as syntactical sugar. The material in Sections 7.2 - 7.4 is based upon fundamentally new concepts.

7.1 The AEPL language

An authorization policy may be represented as a function $P : Q \to D$, where Q is the set of requests and D is the set of decisions. In other words, $P(q)$ represents the result of evaluating policy P for request q, thereby determining whether q is authorized by policy P.

In attribute-based access control, a *request* is assumed to be a set of name-value pairs, where a name is an attribute and a value is taken from some domain over which the binary relations $=$, $\neq$, $<$, $\leqslant$, $>$ and $\geqslant$ are defined. We assume it is possible to determine whether a pair of values belongs to a given relation efficiently. In particular, we assume henceforth that all attribute values are strings of bounded length defined over some alphabet Σ.

It is usually impossible to specify an attribute-based policy P by specifying a decision for every possible request, given the size of the domain of P. Thus, it is usual to specify P in terms of the relationship between a policy and the attribute name-value pairs that constitute a request. This relationship is typically expressed in terms of "targets", which are predicates specified in terms of attributes values and whose truth values are determined by comparing the attribute values specified in the target with those present in the request.

7.1.1 Attribute expressions

Our attribute-expression policy language, AEPL, is based on the idea of an *attribute expression*. Informally, the input to P is a tuple of logical values, and those values are determined by "matching" a request to a set of attribute expressions. More formally, we define an *attribute expression* to be a tuple $(n, v, \sim, \oplus)$, where n is an attribute name, v is an attribute value or regular expression, $\sim$ is an associative, commutative, binary relation, and $\oplus$ is a binary operator.

Given a binary relation R defined over some domain V, we write $\overline{R}$ to denote the complement of R: that is $(a, b) \in \overline{R}$ iff $(a, b) \notin R$. We will usually write R as an infix relation $\sim$ and $\overline{R}$ as $\nsim$. Typical examples of R and $\overline{R}$ include $=$ and $\neq$, $<$ and $\not<$ (that is, $\geqslant$).

In many cases, v will be an attribute value and $\sim$ will be a comparison operator, such as equality or greater-than. However, the use of regular expres-

sions means that more complex attribute expressions may be defined. Given a regular expression e (defined over Σ), let $\mathcal{L}(e)$ denote the set of strings that match e. Then we define $\sim_e$ to be the set of pairs $\{(e, w) : w \in \mathcal{L}(e)\}$. Moreover, for any regular expression e, there exists a regular expression $\overline{e}$, the *complement* of e, such that $w \notin \mathcal{L}(e)$ if and only if $w \in \mathcal{L}(\overline{e})$.

In the interests of clarity of exposition, we will assume henceforth that $\sim$ is always $=$ (corresponding to exact string matching). Note that this does not affect the generality of our approach: we only require that it is efficient to determine whether a pair belongs to the relation $\sim$.

The operator $\oplus$ determines the result of evaluating a request in which some name-value pairs match the attribute expression and some don't. (This contrasts with the approach taken in XACML and PTaCL.) We discuss this in more detail in Section 7.1.2.

We define three binary operators in Figure 7.2: $\vee$ and $\wedge$ are defined on the set $\{0, 1, \bot\}$; and ! is defined on $\{0, 1, \bot, \top\}$. Since $\oplus \in \{\wedge, \vee, !\}$ is associative and commutative, the expression

$$(\ldots((x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_{k-1}) \oplus x_k)$$

may be written without ambiguity as $\bigoplus_{i=1}^{k} x_i$.

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	1
0	0	0	0
1	1	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	0	1
0	0	0	1
1	1	1	1

!	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figure 7.2: Binary operators for attribute expressions

7.1.2 Evaluating requests

A *request* is a set of name-value pairs of the form (n, v). The *evaluation* of a request $q = \{(n_1, v_1), \ldots, (n_\ell, v_\ell)\}$ with respect to an attribute expression $\alpha = (n, v, \sim, \oplus)$ is denoted by $\mathsf{eval}(q, \alpha)$. Informally, $\mathsf{eval}(q, \alpha)$ is determined by combining the results of evaluating the elements of the request (n_i, v_i) using $\oplus$. More formally, we define:

$$\mathsf{eval}(\emptyset, (n, v, \sim, \oplus)) = \bot_{\mathrm{m}};$$

$$\mathsf{eval}(\{(n', v')\}, (n, v, \sim, \oplus)) = \begin{cases} 1_{\mathrm{m}} & \text{if } n = n' \text{ and } v \sim v', \\ 0_{\mathrm{m}} & \text{if } n = n' \text{ and } v \nsim v', \\ \bot_{\mathrm{m}} & \text{otherwise.} \end{cases}$$

We say a name-value pair (n', v') *matches* an attribute expression α if $\mathsf{eval}(\{(n', v')\}, \alpha) = 1_{\mathrm{m}}$; and we say (n', v') *does not match* α if

$\mathsf{eval}(\{(n', v')\}, \alpha) = 0_{\mathrm{m}}$. Throughout this section, we use the subscript m (for "match") to denote explicitly logical values that arise from the evaluation of attribute expressions (request matches). In Section 7.1.3, we use the subscript d to denote logical values that arise from policy evaluation (authorization decisions). When no ambiguity can occur we will omit the subscripts.

A request $q = \{(n_1, v_1), \ldots, (n_\ell, v_\ell)\}$ may contain two name-value pairs, one of which matches $\alpha = (n, v, \sim, \oplus)$ and one which doesn't. The choice of operator $\oplus$ determines how the results of the matches will be combined. Formally, we have

$$\mathsf{eval}(q, (n, v, \sim, \oplus)) = \bigoplus_{i=1}^{k} \mathsf{eval}(\{(n_i, v_i)\}, (n, v, \sim, \oplus)).$$

Thus, we have the following possibilities.

- $\mathsf{eval}(q, (n, v, \sim, \vee)) = 1_{\mathrm{m}}$ if there exists i such that $\mathsf{eval}(\{(n_i, v_i)\}, (n, v, \sim, \vee)) = 1_{\mathrm{m}}$; in other words, if the request contains at least one name-value pair that matches the attribute expression.
- $\mathsf{eval}(q, (n, v, \sim, \wedge)) = 0_{\mathrm{m}}$ if there exists i such that $\mathsf{eval}(\{(n_i, v_i)\}, (n, v, \sim, \wedge)) = 0_{\mathrm{m}}$; in other words, if the request contains at least one name-value pair that does not match the attribute expression.
- $\mathsf{eval}(q, (n, v, \sim, !)) = \top_{\mathrm{m}}$, indicating conflict, if there exist i and j such that $\mathsf{eval}(\{(n_i, v_i)\}, (n, v, \sim, !)) = 0_{\mathrm{m}}$ and $\mathsf{eval}(\{(n_j, v_j)\}, (n, v, \sim, !)) = 1_{\mathrm{m}}$.

It is worth noting that neither XACML nor PTaCL provide this level of control over how a request is evaluated with respect to singular targets (the concept analogous to an attribute expression). Roughly speaking, target evaluation in both languages only returns 0_{m} or 1_{m} and (effectively) always assumes the use of $\vee$ when a request contains attribute values that both match and don't match a target. We discuss later how one could use multiple targets together to distinguish conflicts in PTaCL target evaluation.

7.1.3 AEPL policies

A *policy* in AEPL is a pair $P = (A(P), F(P))$, where

- $A(P) = \{\alpha_1, \ldots, \alpha_\ell\}$ is a set of attribute expressions,
- $D_i \subseteq \{\bot_{\mathrm{m}}, 0_{\mathrm{m}}, 1_{\mathrm{m}}, \top_{\mathrm{m}}\}$ is the range of values that $\mathsf{eval}(q, \alpha_i)$ can take, and
- $F(P) : D_1 \times \cdots \times D_\ell \to \{\bot_{\mathrm{d}}, 0_{\mathrm{d}}, 1_{\mathrm{d}}, \top_{\mathrm{d}}\}$ is a function.

Then we define:

$$P(q) = F(\mathsf{eval}(q, \alpha_1), \ldots, \mathsf{eval}(q, \alpha_\ell)).$$

We will write A and F for $A(P)$ and $F(P)$, respectively, when P is clear from context. We will also write $\mathsf{eval}(q, A)$ for $\mathsf{eval}(q, \alpha_1), \ldots, \mathsf{eval}(q, \alpha_\ell)$ where no confusion can occur.

We may visualize F as a table having $\ell + 1$ columns. The first ℓ columns are indexed by the attribute expressions in A. The entries in the ith column are the possible values that $\mathsf{eval}(q, \alpha_i)$ can take. The final entry in the row with entries $d_1, \ldots, d_\ell$ is $F(d_1, \ldots, d_\ell)$. In other words, policies are defined in the form suggested in the introduction and illustrated in Table 7.1. Thus we specify an AEPL policy in two steps: define the relevant attribute expressions; and then define the policy table. Note that a policy is defined directly in terms of the match relationships that exist between the elements of a request and the policy's attribute expressions.

In the remainder of the chapter, we use an example of a simple policy containing two attribute expressions. Let $P_{\text{ex}} = (A_{\text{ex}}, F_{\text{ex}})$ be a policy, where

$$A_{\text{ex}} = \{\alpha_1, \alpha_2\} = \{(n_1, v_1, =, \wedge), (n_2, v_2, =, \wedge)\}$$

and F_{ex} is defined in Table 7.2.

$x_1 = \mathsf{eval}(q, \alpha_1)$	$x_2 = \mathsf{eval}(q, \alpha_2)$	$F_{\text{ex}}(x_1, x_2)$
$\bot_{\text{m}}$	$\bot_{\text{m}}$	$\bot_{\text{d}}$
$\bot_{\text{m}}$	0_{m}	$\bot_{\text{d}}$
$\bot_{\text{m}}$	1_{m}	1_{d}
0_{m}	$\bot_{\text{m}}$	0_{d}
0_{m}	0_{m}	0_{d}
0_{m}	1_{m}	0_{d}
1_{m}	$\bot_{\text{m}}$	1_{d}
1_{m}	0_{m}	0_{d}
1_{m}	1_{m}	1_{d}

Table 7.2: Policy function defined as a table

The decisions in the final column of Table 7.2 are determined by the policy author, for each combination of attribute expression matches. This allows for precise specification of how the policy P_{ex} should behave under each different outcome of attribute expression evaluation, and differs significantly from the evaluation of targets in XACML and PTaCL. We discuss this in more detail in Section 7.3.

7.1.4 Policies in normal form

In Chapters 4 and 6, we extended work by Jobe [24] to develop a method for converting arbitrary tables representing functions of the form $F : D^n \to D$, where D is the set of values in a multi-valued logic, into an equivalent logical formula using selection operators. We now show how this method can be applied to AEPL policies to generate standardized policy representations that

can be evaluated automatically. We use the canonically complete 4-valued logic $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\mathrm{b}}\})$ as the underlying logic in AEPL.

We write $\mathcal{D}$ to denote $D_1 \times \cdots \times D_\ell$. Given $(x_1, \ldots, x_\ell) \in \mathcal{D}$, we will write $\mathbf{x}$ where no ambiguity can occur. For each row in the table representing F, we may construct an equivalent logical formula comprising a "disjunction" of selection operators. Specifically, if $F(\mathbf{a}) = d$ for $\mathbf{a} \in \mathcal{D}$, then, we may write this as $S^d_{\mathbf{a}}(\mathbf{x})$. Let $\mathcal{D}^+ = \{\mathbf{x} \in \mathcal{D} : F(\mathbf{x}) \neq \bot\}$. Then

$$F(\mathbf{x}) = \bigvee_{\mathbf{a} \in \mathcal{D}^+} S^d_{\mathbf{a}}(\mathbf{x}) \qquad \text{and} \qquad S^d_{(a_1,\ldots,a_\ell)}(\mathbf{x}) \equiv \bigwedge_{i=1}^{\ell} S^d_{a_i}.$$

Hence F may be represented as a disjunction of conjunctions of unary selection operators.

Consider F_{ex}, and let $x_1 = \mathsf{eval}(q, \alpha_1)$ and $x_2 = \mathsf{eval}(q, \alpha_2)$. Then, we may express the policy $P_{\mathrm{ex}}(q) = F_{\mathrm{ex}}(x_1, x_2)$ as a disjunction of selection operators:

$$\begin{aligned} F_{\mathrm{ex}}(x_1, x_2) \equiv\ & S^{\bot}_{(\bot,\bot)}(x_1, x_2) \curlyvee S^{\bot}_{(\bot,0)}(x_1, x_2) \curlyvee S^{1}_{(\bot,1)}(x_1, x_2) \\ & \curlyvee S^{0}_{(0,\bot)}(x_1, x_2) \curlyvee S^{0}_{(0,0)}(x_1, x_2) \curlyvee S^{0}_{(0,1)}(x_1, x_2) \\ & \curlyvee S^{1}_{(1,\bot)}(x_1, x_2) \curlyvee S^{0}_{(1,0)}(x_1, x_2) \curlyvee S^{1}_{(1,1)}(x_1, x_2). \end{aligned}$$

This, in turn, may be represented as a disjunction of conjunctions of unary selection operators (as described above).

Earlier in Chapter 6, we derived expressions for the unary selection operators in terms of the operators $\{-, \diamond, \otimes_{\mathrm{b}}\}$ (see Figure 6.3). Hence, we can derive a formula in normal form for the policy $P_{\mathrm{ex}} = (A_{\mathrm{ex}}, F_{\mathrm{ex}})$. Of course, one would not usually construct the normal form by hand, as we have done above. Indeed, we may reuse the algorithm we developed for $\mathrm{PTaCL}^{\leqslant}_{\mathsf{4}}$ policies (see Section 6.4.1), which takes an arbitrary policy expressed as a table as input, and outputs the equivalent normal form expressed in terms of the operators $\{-, \diamond, \otimes_{\mathrm{b}}\}$.

7.1.5 AEPL policy trees

An AEPL *policy* is a pair (A, F). While this method of policy specification provides an intuitive and flexible method for defining policies, it will not scale to situations where many attribute expressions need to be specified and evaluated. In this case, it makes sense to use the policy-combining operators found in XACML and other tree-structured authorization languages to combine the results of evaluating multiple policies, each using a small number of attribute expressions. Thus, the set of attribute expressions in each policy will act in the same way as a target in a language like XACML.

In this section we revise the syntax and semantics for policy evaluation of the tree-structured ABAC language $\mathrm{PTaCL}^{\leqslant}_{\mathsf{4}}$. We use the operators $-, \diamond$ and

$\otimes_{\mathrm{b}}$ from $\mathrm{PTaCL}_4^{\leqslant}$ (defined in Chapter 6) and demonstrate how policies of the form (A, F) can be used in tree-structured policies, and how we may replace targets with attribute expressions.

We define an *attribute expression based target*, or simply an *AE-target*, to be a pair (A, T), where $A = \{\alpha_1, \ldots, \alpha_\ell\}$ is a set of attribute expressions and $T \subseteq D_1 \times \cdots \times D_\ell$, where D_i is the set of values that $\mathsf{eval}(q, \alpha_i)$ can take. If q is a request and T is an AE-target such that $\mathsf{eval}(q, A) \in T$ then q is said to *match* T.

Given an AEPL policy $P = (A, F)$, then P, $\diamond P$ and $-P$ are AEPL policy trees, where

$$(\diamond P)(q) \stackrel{\text{def}}{=} \diamond P(q) \qquad \text{and} \qquad (-P)(q) \stackrel{\text{def}}{=} -P(q).$$

If P_1 and P_2 are AEPL policy trees and T is an AE-target, then $(T, P_1 \otimes_{\mathrm{b}} P_2)$ is a AEPL policy tree, where

$$(T, P_1 \otimes_{\mathrm{b}} P_2)(q) \stackrel{\text{def}}{=} \begin{cases} P_1(q) \otimes_{\mathrm{b}} P_2(q) & \text{if } \mathsf{eval}(q, A) \in T, \\ \bot & \text{otherwise.} \end{cases}$$

The ability to use AEPL policies (A, F) as leaf nodes (atomic policies) in AEPL policy trees provides us with a number of advantages (over $\mathrm{PTaCL}_4^{\leqslant}$). We get the additional expressive power of policy specification for leaf nodes, together with the full power and functional completeness of $\mathrm{PTaCL}_4^{\leqslant}$. By facilitating high-level operators in addition to specifying policies via a policy table and attribute expressions, we provide a hybrid means of constructing policies, which can be both bottom-up and top-down. This provides a great deal of flexibility and expressivity for policy authors. We can support low level policies specified by functions, and merge the policies using high-level policy operators. In addition, we have greater control of the applicability of policies due to the use of targets based on attribute expression (over "traditional" targets in $\mathrm{PTaCL}_4^{\leqslant}$). We discuss $\mathrm{PTaCL}_4^{\leqslant}$ targets and their limitations in more depth in Section 7.3.2.

7.2 Policy compression

While representing a policy P as a pair (A, F) is more concise and an easier task than specifying a decision for every possible request, the policy tables will be large when many attribute expressions are involved. To tackle this problem, we now investigate methods for policy compression, with the aim of reducing the size of these policy tables.

7.2.1 Removing redundancies

We begin with the following two remarks about methods for merging and omitting rows from policy tables.

Remark 7.2.1. *Suppose $F(a, x_2) = d$ for all $x_2 \in D_2$, as illustrated in the policy table fragment below.*

x_1	x_2	$F(x_1, x_2)$
a	$\bot$	d
a	0	d
a	1	d
a	$\top$	d

Then it is easy to show that

$$S^d_{(a,\bot)}(x_1, x_2) \curlyvee S^d_{(a,0)}(x_1, x_2) \curlyvee S^d_{(a,1)}(x_1, x_2) \curlyvee S^d_{(a,\top)}(x_1, x_2)$$

is equivalent to $S^d_a(x_1)$. (This equivalence is formally established in Table 7.3.) And this may be represented in tabular form as a single row, shown below, where we use $-$ to signify that x_2 can take any value.

x_1	x_2	$F(x_1, x_2)$
a	$-$	d

x_1	x_2	$S^d_{(a,\bot)}(x_1, x_2)$	$S^d_{(a,0)}(x_1, x_2)$	$S^d_{(a,1)}(x_1, x_2)$	$S^d_{(a,\top)}(x_1, x_2)$	$S^d_a(x_1)$
a	$\bot$	d	$\bot$	$\bot$	$\bot$	d
a	0	$\bot$	d	$\bot$	$\bot$	d
a	1	$\bot$	$\bot$	d	$\bot$	d
a	$\top$	$\bot$	$\bot$	$\bot$	d	d

Table 7.3: Equivalence of selection operators

Remark 7.2.2. *We may omit any rows from the policy table in which the final column contains the value $\bot$. Recall*

$$S^d_{\mathbf{a}}(\mathbf{x}) = \begin{cases} d & \text{if } \mathbf{x} = \mathbf{a}, \\ \bot & \text{otherwise.} \end{cases}$$

Moreover, $(\bot \curlyvee x) = (x \curlyvee \bot) = x$ for all $x \in \{0, 1, \bot, \top\}$. Thus

$$S^{d_1}_{\mathbf{a_1}}(\mathbf{x}) \curlyvee S^{d_2}_{\mathbf{a_2}}(\mathbf{x}) \curlyvee \ldots \curlyvee S^{d_n}_{\mathbf{a_n}}(\mathbf{x}) = \bot,$$

except when $\mathbf{x} \in \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$.

Thus, we may assume the policy returns $\bot$ if the table does not contain an

entry for a particular tuple $\mathbf{x}$. In this case, we say the policy is *not applicable* for any request q such that $\mathsf{eval}(q, A) = \mathbf{x}$.

Returning to our example policy P_{ex}, we apply the results from the remarks above to reduce the size of the policy table which defines the function F_{ex}. First, note that

$$F_{\text{ex}}(0, \bot) = F_{\text{ex}}(0, 0) = F_{\text{ex}}(0, 1) = 0.$$

Thus, by Remark 7.2.1, we may merge these three rows into a single row, represented by $F_{\text{ex}}(0, -) = 0$. In addition, by Remark 7.2.2, we may omit the rows $F_{\text{ex}}(\bot, \bot)$ and $F_{\text{ex}}(\bot, 0)$ since they contain $\bot$ in the final column. Hence, we have the reduced policy table shown in Figure 7.4.

$x_1 = \mathsf{eval}(q, \alpha_1)$	$x_2 = \mathsf{eval}(q, \alpha_2)$	$F_{\text{ex}}(x_1, x_2)$
$\bot_{\text{m}}$	1_{m}	1
0_{m}	$-$	0
1_{m}	$\bot_{\text{m}}$	1
1_{m}	0_{m}	0
1_{m}	1_{m}	1

Table 7.4: Reduced policy table

Expressing the policy $P_{\text{ex}}(q) = F_{\text{ex}}(x_1, x_2)$ as a disjunction of selection operators, we have

$$F_{\text{ex}}(x_1, x_2) \equiv S^1_{(\bot,1)}(x_1, x_2) \curlyvee S^0_0(x_1) \curlyvee S^1_{(1,\bot)}(x_1, x_2) \curlyvee S^0_{(1,0)}(x_1, x_2) \curlyvee S^1_{(1,1)}(x_1, x_2).$$

It is worth noting that we may apply Remark 7.2.1 directly during policy specification. If, for example, a policy author decides during the construction of a policy table that if $x_1 = 0$ then the value of x_2 is irrelevant, the policy should return 0 (the case in our example). In other words, we can, if desired, directly encode a deny-overrides or permit-overrides in the policy table when certain attribute expressions are matched or not matched. And we can allow the policy author to use $-$ as syntactic sugar for a "decision" in the policy table, thereby saving the policy author from entering multiple rows (as seen in Table 7.2).

7.2.2 Policies as Boolean functions

We now demonstrate that it is possible to reduce certain policies to Boolean functions. Specifically, if $F(\mathbf{x}) \in \{0, 1\}$ for all $\mathbf{x} \in \mathcal{D}$, then we can eliminate the values $\bot$ and $\top$ from the policy table. In particular, we may replace an attribute expression $\alpha = (n, v, \sim, \oplus)$ with two simpler attribute expressions $\alpha_1 = (n, v, \sim)$ and $\alpha_2 = (n, v, \nsim)$. We then encode the semantics of $\oplus$ directly in a policy table only containing 0s and 1s.

Consider the example in Figure 7.3. There are four values in the decision set $D = \{\bot, 0, 1, \top\}$, and there are four unique combinations of 0 and 1, represented by the four rows in Figure 7.3b. Each of these values arises because of matches or the absence of matches, thus allowing us to encode the semantics of $\oplus$ directly in a policy table only containing 0s and 1s.

$(n, v, \sim, !)$	F
$\bot$	0
0	0
1	1
$\top$	0

(a) Policy containing $\bot$ and $\top$

$(n, v, \sim)$	$(n, v, \nsim)$	F
0	0	0
0	1	0
1	0	1
1	1	0

(b) Policy using only 0 and 1

Figure 7.3: Converting a simple policy into a Boolean function

There are a number of advantages in expressing an attribute expression $\alpha = (n, v, \sim, \oplus)$ as a combination of two attribute expressions $(n, v, \sim)$ and $(n, v, \nsim)$ and encoding the semantics of $\oplus$ directly. In particular, the resulting policy table contains only binary values. Hence, we may employ existing techniques for Boolean function minimization [2].

7.3 Comparison with XACML and PTaCL

Having defined the AEPL policy authorization language, we now provide a brief summary of XACML and compare it with AEPL. We discuss the limitations of targets in XACML and PTaCL, before showing how XACML rules and policies may be represented in AEPL. We conclude by showing how an XACML policy set may be represented using a single AEPL policy.

7.3.1 XACML targets

An XACML *target*, like an attribute expression, is expressed in terms of attribute name-value pairs. It is used to determine whether a rule, a policy or a policy set is applicable to a request.

A target is defined in terms of `AllOf` and `AnyOf` elements. The `AllOf` element is used to group name-value pairs. Such an element is "matched" by a request if the request matches each of the name-value pairs. The `AnyOf` element is used to group `AllOf` elements. Such an element is matched if any one of the `AllOf` elements is matched. Evaluation of an XACML target returns one of two values ("matched" or "not-matched"), unlike the evaluation of an attribute expression.

Moreover, evaluation of an XACML target disregards whether a request contains a name-value pair that doesn't match a target if it also contains a name-value pair that does match. In other words, XACML provides less

control over target evaluation than our approach for the evaluation of attribute expressions.

7.3.2 PTaCL targets

A PTaCL target is defined to be a tuple (n, v, f), where n is an attribute name, v is an attribute value and f is a binary predicate. The key difference between PTaCL targets and attribute expressions, comes in the choice of the binary operator $\oplus$ present in attribute expressions. Attribute expressions allow this operator to be selected from the set $\{\wedge, \vee, !\}$, dependent on the way in which conflicting attribute values should be handled. Targets in PTaCL implicitly assume that the operator $\vee$ is always used. This makes it impossible to distinguish scenarios where there are two name value pairs (n', v') and (n'', v'') such that $n = n', v = v'$ and $n = n'', v \neq v''$, through the use of a single target.

However, we are able to use a combination of targets to structure a target that can effectively detect conflicts between two name value pairs where $n = n', v = v'$ and $n = n'', v \neq v''$. Let t_1, t_2 and t_3 be targets, where

$$t_1 = (n, v, =),$$
$$t_2 = (n, v, \neq),$$
$$t_3 = t_1 \wedge_{\mathrm{p}} t_2.$$

Then t_3 will return 1_{m} if a request contains two name value pairs (n', v') and (n'', v'') such that $n = n', v = v'$ and $n = n'', v \neq v''$, and thus can identify a conflict of attribute values. While PTaCL can indirectly support the detection of conflicts in attribute values in targets, we believe that our approach is AEPL is more desirable for practical use. By using an explicit fourth decision $\top$ to represent conflict, it becomes obvious to policy authors what is happening in target evaluation, opposed to the use of three targets as described above.

7.3.3 XACML rules

An XACML rule is specified by a target t and a decision d (known as the "effect" of the rule in XACML), which may be either "allow" or "deny". The evaluation of a rule for a given request returns d if the request matches the target and "not-applicable" otherwise. Thus, an XACML rule may be encoded as a particularly simple policy table. Specifically:

- each `AllOf` element is encoded as a row in the table, in which the last entry is always d; and
- the `AnyOf` element is encoded by the different rows in the table.

Clearly, our policy tables can encode more general structures than XACML rules. Informally, a policy table would have to be encoded using two XACML

rules, one for the rows for which the decision is 1 and one for the rows for which the decision is 0. Even so, such an encoding could not, for example, return $\top$. In other words, AEPL provides a richer framework than the target-decision paradigm for specifying the "leaf" policies in tree-structured languages such as XACML or PTaCL.

7.3.4 XACML policies

We now illustrate how attribute expressions can be used to encode an entire XACML policy set directly. Consider the tree-structured policy P illustrated in Figure 7.4, where do and po represent the XACML deny-overrides and permit-overrides combining algorithms respectively. This policy tree represents a XACML policy set (t_1, do) which contains one policy (t_3, po) and one rule $(t_2, 0)$; and the policy (t_3, po) in turn contains two rules $(t_4, 1)$ and $(t_5, 0)$[1]. We assume for simplicity that each target is a single name-attribute pair. In Section 7.4.1 we explain how we can extend our method of specifying a policy as a pair (A, F) to more complex scenarios.

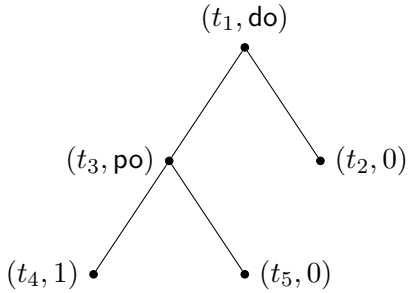


Figure 7.4: A simple tree-structured policy

Then we can represent this policy as the following policy table, which is created by considering every possible outcome of matching requests to targets. We write x_i to denote $\mathsf{eval}(q, t_i)$, and $-$ to signify the value of $\mathsf{eval}(q, t_i)$ is irrelevant.

x_1	x_2	x_3	x_4	x_5	$P(q)$
0	$-$	$-$	$-$	$-$	$\bot$
1	1	$-$	$-$	$-$	0
1	0	0	$-$	$-$	$\bot$
1	0	1	1	$-$	1
1	0	1	0	1	0
1	0	1	0	0	$\bot$

Hence, P may be represented by the pair (A, F), where $A = \{t_1, t_2, t_3, t_4, t_5\}$ and $F : \{0,1\}^5 \to \{\bot, 0, 1\}$ is defined in the table above.

[1]This is a simplification in terms of the structure of XACML policy sets, policies and rules but approximate enough for the sake of exposition. For explicit definitions the reader is referred to the XACML standard [37].

Then

$$F(x_1, \ldots, x_5) \equiv S^0_{(1,1)}(x_1, x_2) \curlyvee S^1_{(1,0,1,1)}(x_1, x_2, x_3, x_4) \curlyvee S^0_{(1,0,1,0,1)}(x_1, x_2, x_3, x_4, x_5).$$

Note that we need not include the selection operators $S^{\perp}_0(x_1)$, $S^{\perp}_{(1,0,0)}(x_1, x_2, x_3)$ and $S^{\perp}_{(1,0,1,0,0)}(x_1, x_2, x_3, x_4, x_5)$ representing the first, third and sixth rows, since the policy evaluates to $\perp$ in these rows.

The representation of the policy tree in Figure 7.4 can be reduced to a simple policy table, which in turn is reduced into a formula comprising just three selection operators. By expressing this policy tree as a policy table, it is much easier for a policy author to understand how this policy will behave under each different result of target evaluation. Furthermore, we have a simple formula that can be automatically converted into a machine-enforceable policy and evaluated by a PDP.

7.4 Applications

We now demonstrate how complex policies can be built, enabling distributed policy specification and evaluation (much as in XACML and PTaCL). Furthermore, we explore how policy tables can be used to enhance existing access control paradigms, such as role-based access control and access control lists. Informally, in the first case, we show that a set of attribute expressions in an AEPL policy table (each of which evaluates to an element in $\{0, 1, \perp, \top\}$) may be replaced with a set of policies. And in the second case, we show that the policy decisions in an AEPL table can be replaced with a set of role identifiers or similar.

7.4.1 Complex policies as tables

By specifying policies as a pair (A, F), we implicitly restrict the depth of policies (or policy trees) to one. However, this may not be the way in which some policies are structured in the real world (indeed, most policy trees in XACML have depth greater than one). We now develop a method for constructing more complex policies from simple AEPL policies, using the structure of simple policies as a template. A complex policy $\mathcal{P}$ is a pair $(\{P_1, \ldots, P_\ell\}, F)$, where $P_i = (A_i, F_i)$ is a simple AEPL policy. We define

$$\mathcal{P}(q) = F(P_1(q), \ldots, P_\ell(q)).$$

Now, instead of the columns of the policy table representing the function F being indexed by attribute expressions, they are indexed by policies. Each row represents a possible combination of the values that may arise from the evaluation of the respective policies $P_1, \ldots, P_n$. Of course, each of these po-

licies is itself defined by a set of attribute expressions and function (A_i, F_i), which will need to be specified and evaluated first. The policy $\mathcal{P}$, much like previous policies, can be automatically converted into a machine-enforceable form via the use of selection operators. Hence, we have developed a way in which to combine arbitrary policies into machine-enforceable form. This approach can be easily scaled, providing the means to construct policies of any desired depth (much in the same manner as XACML policies).

One of the main advantages in using this method for building up complex policies, is the distributed nature in which it can be applied. For instance, in a large organization, each department could construct their own complex policy, which can be converted into a tree. Each department's policy can then become a node in a bigger tree, and be combined with other policies through the use of another policy table. A simple example of this is shown in Table 7.5, demonstrating how four individual policies P_1, P_2, P_3 and P_4 produced by each department can be combined in a policy table to create the overall organizations policy P_{org}.

P_1	P_2	P_3	P_4	P_{org}
d_1	d_2	d_3	d_4	d_{org}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 7.5: Combining policies in another table

This policy table can be specified by someone who understands the complete policy structure of the organization and can place adequate restrictions on the interactions of policies between departments. Constructing policies in this manner allows each department to design their own specific policy, without the need to worry about how their policy interacts with other department's policies. The combination of policies is then moderated by a person who understands the organization wide policy strategy. We believe this both simplifies specification of complex corporate policies, and reduces the likely number of misconfigurations and errors.

In addition to the distributed nature in which policies can be specified using the approach, policy evaluation may also be distributed. In the real world it is common practice for multiple PDPs to be deployed, and this architecture may be leveraged by our method of specifying complex policies. For instance, imagine the scenario where a central PDP is in charge of evaluating the organization's policy P_{org}. This central PDP may then delegate the evaluation of policies $P_1, \ldots, P_4$ to other PDPs, and combine the resulting decisions that are reported back by each PDP. There are many reasons why distributing the evaluation of policies in this way could be advantageous: (i) the load on the central PDP is reduced, (ii) free or available PDPs are fully utilised, (iii) the evaluation time for policies is reduced, and (iv) in some instances requests

Age	Role
≥ 3	Child
≥ 11	Juvenile
≥ 16	Adolescent
≥ 18	Adult

Table 7.6: Age to role assignment

may even be evaluated locally.

7.4.2 ABAC policies for RBAC

In role-based access control (RBAC) [42], we tend to assume that users are authorized for roles on the basis of identity. With the emergence of attribute-based access control, we have an alternative option: authorizing users on the basis of their attributes. Al-Kahtani and Sandhu [1] created a model for attribute-based user-role assignment, in which an enterprise defines a set of rules that are triggered to automatically assign roles to users. The motivation for a mechanism to do this, is to reduce the number of manual user-to-role assignments that are required, which can become troublesome in large environments such as utility companies and popular online websites [1].

We now demonstrate how we can automatically assign roles to users using policies in AEPL. Previously, authorization policies were represented by a function $P : Q \to D$, where Q is the set of requests and D is the set of decisions. Now, we represent a role assignment authorization policy by a function $P : Q \to R$, where Q is the set of requests and R is the set of roles. The function P is used to determine how users are assigned to roles based on their attributes. Consider the simple example for an attribute-role table which assigns roles based on the attribute "age" for the purpose of filtering age-restricted content, shown in Table 7.6 [1].

Let P be a policy which comprises of four attribute expressions $\alpha_1, \alpha_2, \alpha_3$ and α_4, where $\alpha_1 = (\text{age}, 3, \geq), \alpha_2 = (\text{age}, 11, \geq), \alpha_3 = (\text{age}, 16, \geq)$ and $\alpha_4 = (\text{age}, 18, \geq)$[2], with policy function F, defined in Table 7.7. (Note that we abbreviate $\mathsf{eval}(q, (\text{age}, x, \geq))$ as age $\geq x$ in Table 7.7 in the interests of space.)

age ≥ 3	age ≥ 11	age ≥ 16	age ≥ 18	Role
1	0	0	0	Child
1	1	0	0	Juvenile
1	1	1	0	Adolescent
1	1	1	1	Adult

Table 7.7: Policy table

[2]The choice of operator $\oplus$ is irrelevant in this example, since requests will not contain two pairs (n, v') and (n, v'') such that n is age and $v' \neq v''$, hence we omit it.

Hence, we have represented the age-to-role assignment as a policy $P = (A, F)$, which may in turn be represented as a combination of selection operators and converted into a machine-enforceable policy. It is easy to imagine how this methodology could be extended and applied in a setting where it is useful to automatically assign roles to users on the basis of attributes rather than identity. Our approach is scalable, simple for policy authors to understand and can be applied with various other techniques discussed throughout this chapter such as ways to compress policies and using policies as leaf nodes in tree-structured languages, to produce a machine-enforceable policy which assigns roles to users.

7.4.3 Access control lists

In an access control system using access control lists based on identifiers, a user is associated with one or more identifiers: a unique user identifier (UID) and zero or more group identifiers (GIDs). Each object is associated with an access control list (ACL). Each ACL may be modelled as a list of access control entries (ACEs), where an ACE comprises an identifier and a set of authorized actions. Finally, a request contains a UID and a set of GIDs, an object identifier (OID) and a requested action. The UID and GIDs in the request will be compared with those in the ACEs of the object's ACL and a decision will be reached based on the actions that are authorized by the ACEs and those that have been requested.

We may extend this idea of identity-based ACLs to attribute-based ACLs. Each ACE contains a group identifier, as before, which represents an attribute-based policy. Then, we represent a group membership policy as a function $P : Q \to G$, were Q is the set of requests and G is the set of group identifiers. The policy P specifies the attributes that a user must have to be regarded as a member of that group. We may represent this policy using a set of attribute expressions A and a function F defined as a policy table, in an identical manner to that illustrated in Section 7.4.2. Hence, we can use AEPL to support attribute-based access control in a ACL-based system.

7.5 Summary and discussion

In this chapter, we have made important contributions to the development of ABAC authorization languages. We defined attribute expressions, which provide us with more fine-grained control over how requests are evaluated with respect to attributes, compared to XACML and PTaCL. Thus we are able to distinguish between scenarios where requests contain matching and non-matching pairs of name-value pairs, and provide a choice of three semantics for resolving the different name-value pairs.

Then, we specified policies as tables, in which the columns are indexed

by attribute expressions. By defining policies in this manner, we provided a simple, intuitive method for specifying policies, in which it is obvious how a policy will behave under each different evaluation of attribute expressions (unlike standard tree-structured languages such as XACML and PTaCL). Furthermore, we demonstrated how we may leverage a canonically complete logic to compile policies expressed as tables into machine-enforceable policies. We also explored various methods for policy compression, thus reducing the size of policy tables (and these methods may also be applied to decision tables in $\text{PTaCL}_3^{\leq}$ and $\text{PTaCL}_4^{\leqslant}$).

We then compared XACML and PTaCL with AEPL, showing how attribute expressions provide more control over target evaluation than the traditional targets in XACML and PTaCL. In addition, we demonstrated how an XACML policy represented as a policy tree may be converted into a policy table. We argued that tabular representations of XACML policies make it easier for policy authors to understand how policies will behave under each different result of target evaluation. We concluded by showing some of the applications of AEPL: building and evaluating enterprise-wide policies, and using policy tables in RBAC and ACLs for role and group assignments respectively.

Chapter 8

Conclusions and Future Work

Broadly speaking, the main contributions of this thesis enhance our understanding of languages for attribute-based access control, and formalise the underlying connections with multi-valued logics. In particular, we address the three general difficulties in specifying and designing ABAC languages described in the introduction: expressivity, policy specification and the matching of requests with attributes via targets.

The development and specification of attribute-based access control languages such as XACML [37], PTaCL [12] and PBel [10] will continue to increase to meet the demand for open, distributed, interconnected and dynamic systems. While XACML is a standardized language, and the "de facto" ABAC language for practical implementations, in Chapter 3 we demonstrated numerous flaws in XACML. Specifically, we proved that XACML is not functionally complete, which means there are a number of operators which are useful in practice (such as policy negation) that cannot be constructed using the XACML rule- and policy-combining algorithms.

We recognise that XACML supports the specification of custom combining algorithms, however we believe that use of custom combining algorithms needs to be carefully considered. There is no guarantee that the addition of new combining algorithms will make XACML functionally complete. Thus, to meet the requirements of policy authors, more and more custom combining algorithms may be required over time. This has a twofold negative affect: the implementation of XACML combining algorithms may become more cluttered and many algorithms could become redundant, and the decisions faced by policy authors will become more complicated, increasing the likelihood of errors and misconfigurations. We believe that languages that specify a small number of operators, that are known to be functionally complete, are of more interest and will be easier for policy authors to use [10, 12].

This led naturally to the material in Chapter 4, where we focussed our attention on PTaCL, a language that is known to be functionally complete [12]. However, despite being a functionally complete language, we argued that the way in which PTaCL policies must be written because of the underlying struc-

ture of the language is not particularly helpful to policy authors. Indeed, it is in general a non-trivial task to construct an arbitrary policy in PTaCL, and the policies are often incredibly complex (for example, recall the definition of XACML's do and po in PTaCL, shown in Section 2.3.2).

To tackle the problem of expressing arbitrary policies in PTaCL, we applied the theoretical foundations of Jobe [24] for multi-valued logics to PTaCL, establishing that PTaCL is not canonically complete. However, through a simple replacement of a single unary operator, we developed a canonically complete variant of PTaCL called $\text{PTaCL}_3^{\leq}$. We demonstrated a method for converting any policy expressed as a table into a normal form, using only the operators defined in $\text{PTaCL}_3^{\leq}$. (The method is similar to converting a truth table for an arbitrary formula to a logically equivalent formula in disjunctive normal form in propositional logic.) We believe that specifying policies in this manner is both intuitive and easier for policy authors, and we developed an algorithm for automatically converting policy tables into a normal form that is machine-enforceable. In addition, we extended the syntax and semantics of $\text{PTaCL}_3^{\leq}$ to incorporate obligations, showing that our method is both consistent with XACML and more extensible.

During the development of the canonically complete $\text{PTaCL}_3^{\leq}$ we assumed a total-ordering on the set of authorization decisions, $0 < \bot < 1$. We acknowledge that this assumption is not consistent with the intuitive interpretation of access control decisions, in which 0 and 1 are incomparable, conclusive decisions, while $\bot$ represents the inability to reach a conclusive decision. Furthermore, there are many ABAC languages that utilise a fourth authorization decision [10, 32, 45], to represent a conflict in decisions. XACML also uses a fourth decision, but does so in an ad hoc manner for the only-one-applicable combining algorithm, by reusing the indeterminate error decision. However, in this instance, the indeterminate decision is returned when more than one policy is applicable, which by itself is not an error.

Recognising the value of a fourth authorization decision, we extended Jobe's theoretical foundations to lattice-based logics. We proved that Belnap logic [7] is not canonically complete, and thus any ABAC language which is based on Belnap logic is also not canonically complete. Hence languages such as PBel [10], BelLog [45] and Rumpole [32] are not canonically complete, and suffer from the same difficulty encountered in PTaCL, that is, the challenge in constructing arbitrary policies using the operators specified in the given language.

In Chapter 5 we developed a canonically complete 4-valued lattice-based logic $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\mathrm{b}}\})$, without having to explicitly construct the unary selection operators in normal form (unlike Jobe [24]). By identifying the connection between the generators of the symmetric group and the unary operators of logics, we have developed a simple and generic method for identifying a set of unary

operators that will guarantee the functional and canonical completeness of m-valued lattice-based logics. We also showed that there is a set of operators containing only three connectives which is functionally complete for Belnap logic, in contrast with the set of size four identified by Arieli and Avron [4].

Naturally, in Chapter 6 we defined a canonically complete 4-valued ABAC language, $\text{PTaCL}^{\leqslant}_{\mathsf{4}}$, based on the logic $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\text{b}}\})$ and presented the advantages in doing so. We introduced a novel method for policy specification, in which sub-policies index columns in a table, and the policy author can tabulate the desired decision for all relevant combinations of decisions for sub-policies. Specifying policies in this manner is both simple and intuitive, policy authors can easily see how different combinations of sub-policies interact and specify behaviour for each case, something that is difficult to do in tree-structured languages such as XACML and PTaCL.

We also discussed how the XACML decision set and combining algorithms may be modified to support $\text{PTaCL}^{\leqslant}_{\mathsf{4}}$. Doing so enables us to retain the rich framework provided by XACML for ABAC (in terms of its languages for representing targets and requests) and its enforcement architecture (in terms of the policy enforcement, policy decision and policy administration points). Thus, we are able to propose an enhanced XACML framework within which any desired policy may be expressed. Moreover, the canonically completeness of $\text{PTaCL}^{\leqslant}_{\mathsf{4}}$ means that the desired policy may be represented in simple terms by a policy author (in the form of a decision table) and automatically compiled into a PDP-readable equivalent policy.

In Section 6.4.1 we provided details of the algorithm for converting policy tables into a PDP-readable equivalent policy, and showed the time- and space-complexities are order $\mathcal{O}(rc)$, where r is the number of rows and c is the number of columns. In the worst case scenario, $\mathcal{O}(4^c)$ rows will be required for expressing policies, however in practice this is unlikely as rows which return $\bot$ may be omitted by design (as $\bot$ is the default policy decision). We commented on the ease in which our algorithm may be adapted for any canonically complete ABAC language, thereby facilitating the automatic construction of any arbitrary policy in any canonically complete language.

We defined a novel ABAC language AEPL in Chapter 7, which represents the accumulation of contributions made throughout this thesis, in terms of the expressive power, and ease of policy specification in ABAC languages. First, we defined attribute expressions, which provide greater control over how requests are evaluated, compared to XACML and PTaCL. Attribute expressions remove the unnecessary two layers of abstraction present in current ABAC languages, which exist in the form of targets and policies, by combining the two. We then specified policies as tables, in which the columns are indexed by attribute expressions. In doing so, we leverage all the previously discussed advantages of expressing policies in this manner, in terms of reducing the

number of policy misconfigurations and errors, and the ability to automatically compile policies.

Then, we compared XACML and PTaCL with AEPL, showing that AEPL provides more control over target evaluation than XACML and PTaCL. Furthermore, we showed how an XACML policy can be converted into a policy table. By representing XACML policies in tabular form, it becomes easier for a policy author to understand how policies will behave under each different result of target evaluation. Finally, we demonstrated the various applications of AEPL. We showed how complex policies can be constructed as tables, thus enabling a distributed method for building and evaluating enterprise-wide policies. We also showed how policy tables can be used in RBAC [1, 42] and ACLs for role and group assignments respectively, making these paradigms "attribute-aware".

In this thesis we have addressed several issues in languages for attribute-based access control by examining the underlying connections with multi-valued logics. While existing ABAC languages leverage properties of multi-valued logics to achieve functional completeness, they do not utilise any of the other properties of multi-valued logics. Through the application of Jobe's [24] concept of canonical completeness to ABAC languages, we have developed an extensible framework for specifying ABAC languages that are functionally complete, and permit a normal form. We believe that the results presented throughout this thesis have been encouraging, and address a number of concerns in the development and implementation of ABAC languages.

8.1 Future work

A general criticism that could be made of the work in this thesis is the lack of implementation of the theoretical material. Therefore, one priority for future work is the development of a custom XACML PDP, and development of software that implements some of the ideas presented in this thesis. We discuss this in more detail below. Another general criticism that may be levelled at this thesis is the absence of any usability studies to test the hypothesis that writing policies in a tabular manner is easier than writing tree-structured policies. In addition, Chapters 3 to 7 provide a number of opportunities for future research.

8.1.1 Development of a custom XACML PDP

As noted throughout Chapters 4, 6 and 7, an essential extension to our work is the development of a custom XACML policy decision point, that implements a set of custom combining algorithms (and later a custom decision set). We believe it is simple to modify the XACML syntax to support the operators from either $\text{PTaCL}_3^{\leqslant}$ or $\text{PTaCL}_4^{\leqslant}$. Indeed, we presented the encoding of $\otimes_{\text{b}}$ and the

decision set $\{0, 1, \bot, \top\}$ in XACML in Section 6.4. We also hope to modify an XACML PDP to support the evaluation of attribute expressions, allowing us to evaluate AEPL policies in the rich framework provided by XACML for attribute-based access control.

8.1.2 Software development

We plan to develop policy authoring software that provides a graphical user interface for policy authors, allowing them to specify attribute expressions or regular policies (found in $\text{PTaCL}_3^{\leq}$ or $\text{PTaCL}_4^{\preccurlyeq}$), and construct a table for the desired policy. This table will then be automatically converted into a machine-enforceable policy, using a version of the algorithm presented in Section 6.4.1. We also plan to support the methods for reducing the size of policy tables discussed in Section 7.2, allowing policy authors to directly encode deny- and permit-overrides in the policy table. Naturally, we hope to use a modified XACML PDP to evaluate the machine-enforceable policies produced by the policy authoring software.

There is also motivation to develop a tool which converts a policy expressed as a tree-structured policy into an equivalent policy table, much like the example in Section 7.3.4. This well help facilitate the smooth transition from XACML tree-structured policies to policies defined as tables.

8.1.3 Usability study

Throughout this thesis, we have assumed that expressing policies as tables is a more intuitive and simple method for constructing ABAC policies (compared to constructing policies in a bottom-up tree-structured manner in languages like XACML and PTaCL). While it is clear that understanding how policies will behave under each different result of target evaluation is easier in tabular form compared to policy trees (see, for example, Section 7.3.4), we still need to test the hypothesis for the construction of policies through a usability study.

Our vision of this study requires the participants to construct a policy presented in natural language, first as a tree-structured XACML policy, and then as a policy table in AEPL. We may then compare various elements such as (i) the ease with which the testers could construct each policy, (ii) whether the XACML and AEPL policy are equivalent, (iii) the "correctness" of each policy (how close they are to the described policy in natural language), and (iv) other metrics such as the time taken to construct each policy.

8.1.4 Other

In Chapters 4 and 6 we were not concerned with the specific type or scope of obligations, nor how they are handled by the system after policy evaluation. This is however an important aspect of obligations within authorization sys-

tems, and there has been extensive research into this area [22, 23, 28]. Future work could extend $\text{PTaCL}_{\mathsf{3}}^{\leq}$ and $\text{PTaCL}_{\mathsf{4}}^{\leqslant}$ to handle different types of obligations and model the behaviour that occurs after a decision-obligation pair has been returned to the PEP. Furthermore, when an error is encountered during evaluation, a set of decision-obligation pairs is returned. Thus, a method for handling sets of decision-obligations pairs will need to be defined for the PEP. Inspiration may be taken from work by Crampton and Huth [11], who use the idea of a resolution function when errors are encountered during policy evaluation.

We would like to revisit the idea of *monotonicity* [12] in targets and how this affects policy evaluation in ABAC languages. By definition, monotonicity is dependant on the ordering chosen for the decision set, and existing work by Crampton and Morisset [13] studies the effect of different orderings on monotonicity. We plan to extend this work and investigate how monotonicity of targets is affected by the use of a 4-valued lattice-ordered decision set, such as the ones discussed through this thesis.

Finally, Jobe identifies a number of equivalences (between formulae in the logic $\mathcal{J}$) that may be applied to reduce the size of a normal form formula in $\mathcal{J}$. We plan to investigate the relevance of these equivalences to authorization policies in $\text{PTaCL}_{\mathsf{3}}^{\leq}$ in future work. We would also like to investigate whether these equivalences may be extended to the logic $L(\mathsf{4}_k, \{-, \diamond, \otimes_{\mathrm{b}}\})$ and, more generally, to canonically complete m-valued logics.

References

[1] Mohammad A. Al-Kahtani and Ravi S. Sandhu. A model for attribute-based user-role assignment. In *18th Annual Computer Security Applications Conference (ACSAC 2002), 9-13 December 2002, Las Vegas, NV, USA*, pages 353–362. IEEE Computer Society, 2002.

[2] Eric Allender, Lisa Hellerstein, Paul McCabe, Toniann Pitassi, and Michael E. Saks. Minimizing DNF formulas and AC0 circuits given a truth table. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006), 16-20 July 2006, Prague, Czech Republic*, pages 237–251. IEEE Computer Society, 2006.

[3] Ja'far Alqatawna, Erik Rissanen, and Babak Sadighi Firozabadi. Overriding of access control in XACML. In *8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2007), 13-15 June 2007, Bologna, Italy*, pages 87–95. IEEE Computer Society, 2007.

[4] Ofer Arieli and Arnon Avron. The value of the four values. *Artif. Intell.*, 102(1):97–141, 1998.

[5] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Inf. Syst. Secur.*, 14(1):2, 2011.

[6] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical Report MITRE 2547, Volume I, DTIC Document, 1973.

[7] Nuel D Belnap Jr. A useful four-valued logic. In *Modern uses of multiple-valued logic*, pages 5–37. Springer, 1977.

[8] Piero Bonatti, Nahid Shahmehri, Claudiu Duma, Daniel Olmedilla, Wolfgang Nejdl, Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, Paolo Coraggio, et al. Rule-based policy specification: State of the art and future work. Technical report, Working Group I2, EU NoE REWERSE, August 2004.

[9] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[10] Glenn Bruns and Michael Huth. Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.*, 14(1):9, 2011.

[11] Jason Crampton and Michael Huth. An authorization framework resilient to policy evaluation failures. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 472–487. Springer, 2010.

[12] Jason Crampton and Charles Morisset. PTaCL: A language for attribute-based access control in open systems. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 390–409. Springer, 2012.

[13] Jason Crampton and Charles Morisset. Monotonicity and completeness in attribute-based access control. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, volume 8743 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.

[14] Jason Crampton and Conrad Williams. Obligations in PTaCL. In Sara Foresti, editor, *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings*, volume 9331 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2015.

[15] Jason Crampton and Conrad Williams. On completeness in languages for attribute-based access control. In X. Sean Wang, Lujo Bauer, and Florian Kerschbaum, editors, *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, SACMAT 2016, Shanghai, China, June 5-8, 2016*, pages 149–160. ACM, 2016.

[16] Jason Crampton and Conrad Williams. Attribute expressions, policy tables and attribute-based access control. In Elisa Bertino, Ravi Sandhu, and Edgar Weippl, editors, *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2017, Indianapolis, USA, June 21-23, 2017*, pages 79–90. ACM, 2017.

[17] Jason Crampton and Conrad Williams. Canonical completeness in lattice-based languages for attribute-based access control. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 47–58. ACM, 2017.

[18] Melvin Fitting. Bilattices and the semantics of logic programming. *J. Log. Program.*, 11(1&2):91–116, 1991.

[19] Melvin Fitting. Kleene's logic, generalized. *J. Log. Comput.*, 1(6):797–810, 1991.

[20] Matthew L. Ginsberg. Multi-valued logics. In Tom Kehler, editor, *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science.*, pages 243–249. Morgan Kaufmann, 1986.

[21] Matthew L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[22] Manuel Hilty, David A. Basin, and Alexander Pretschner. On obligations. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, volume 3679 of *Lecture Notes in Computer Science*, pages 98–117. Springer, 2005.

[23] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 134–143. ACM, 2006.

[24] William H. Jobe. Functional completeness and canonical forms in many-valued logics. *J. Symb. Log.*, 27(4):409–422, 1962.

[25] Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, NJ, 1950.

[26] Stephen C. Kleene, Nicolaas G. de Bruijn, Johannes de Groot, and Adriaan C. Zaanen. *Introduction to Metamathematics*, volume 483. van Nostrand New York, 1952.

[27] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[28] Ninghui Li, Haining Chen, and Elisa Bertino. On practical specification and enforcement of obligations. In Elisa Bertino and Ravi S. Sandhu, editors, *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 71–82. ACM, 2012.

[29] Ninghui Li, Qihua Wang, Wahbeh H. Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: Theory meets practice. In *SACMAT 2009, 14th ACM Symposium on Access Control Models and Technologies, Proceedings*, pages 135–144, 2009.

[30] Jan Łukasiewicz. Philosophische Bemerkungen zu mehrwertigen Systemen des Aussagekalküls. *Comtes rendus des séances de la Société des Sciences et des Lettres de Varsovie*, Classe III(vol. 23):55–57, 1930.

[31] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A rigorous framework for specification, analysis and enforcement of access control policies. *CoRR*, abs/1612.09339, 2016.

[32] Srdjan Marinovic, Naranker Dulay, and Morris Sloman. Rumpole: An introspective break-glass access control language. *ACM Trans. Inf. Syst. Secur.*, 17(1):2:1–2:32, 2014.

[33] Charles Morisset and Nicola Zannone. Reduction of access control decisions. In Sylvia L. Osborn, Mahesh V. Tripunitara, and Ian Molloy, editors, *19th ACM Symposium on Access Control Models and Technologies, SACMAT '14, London, ON, Canada - June 25 - 27, 2014*, pages 53–62. ACM, 2014.

[34] Qun Ni, Elisa Bertino, and Jorge Lobo. D-algebra for composing access control policy decisions. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pages 298–309. ACM, 2009.

[35] OASIS. *eXtensible Access Control Markup Language (XACML) Version 1.0*, 2003. (Simon Godik and Tim Moses, editors).

[36] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005. (Tim Moses, editor).

[37] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*, 2012. (Erik Rissanen, editor).

[38] Graham Priest. The logic of paradox. *J. Philosophical Logic*, 8(1):219–241, 1979.

[39] Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, and Flemming Nielson. The logic of XACML. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, volume 7253 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2011.

[40] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. An algebra for fine-grained integration of XACML policies. In *SACMAT 2009, 14th ACM Symposium on Access Control Models and Technologies, Stresa, Italy, June 3-5, 2009, Proceedings*, pages 63–72, 2009.

[41] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[42] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[43] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[44] Jerzy Słupecki. Der volle dreiwertige Aussagen-kalkül. *Comtes rendus des séances de la Société des Sciences et des Lettres de Varsovie*, Classe III(vol. 29):9–11, 1936.

[45] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David A. Basin. Decentralized composite access control. In *POST*, volume 8414 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2014.

[46] Duminda Wijesekera and Sushil Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, 2003.

Appendix A

Code Listings

A.1 XACML operator brute-force combinations

Listing A.1 shows the Python code of the algorithm used to brute force the construction of all possible XACML binary operators, which have the general form $\diamond x \oplus \triangle y$ where $\diamond, \triangle \in \{-, +, \text{“”}\}$ and $\oplus \in \{\mathsf{do}, \mathsf{po}\}$.

```
print '0 a a b 1 1 d c c'

Binary = [0,1,2]

# 1 and bot col agree
a = b = c = d = 0
C = [0, a, a, b, 1, 1, d, c, c]
D = []
count = 0
for i in range(2):
  C[1] = Binary[i]
  C[2] = Binary[i]
  for j in range(2):
    C[3] = Binary[j]
    for k in range(2):
      C[7] = Binary[k]
      C[8] = Binary[k]
      for l in range(2):
        C[6] = Binary[l]
        Temp = []
        Temp.extend(C)
        D.append(Temp)
        count = count + 1
        for s in range(0,3):
          print C[s],
        print
        for r in range(3,6):
          print C[r],
        print
        for t in range(6,9):
          print C[t],
        print
        print
  print

print count

```

```
# 1 and bot row agree
a = b = c = d = 0
C = [0, a, b, c, 1, d, c, 1, d]
E = []
count = 0
for i in range(2):
  C[1] = Binary[i]
  for j in range(2):
    C[2] = Binary[j]
    for k in range(2):
      C[3] = Binary[k]
      C[6] = Binary[k]
      for l in range(2):
        C[5] = Binary[l]
        C[8] = Binary[l]
        count = count + 1
        Temp = []
        Temp.extend(C)
        E.append(Temp)
        for s in range(0,3):
          print C[s],
        print
        for r in range(3,6):
          print C[r],
        print
        for t in range(6,9):
          print C[t],
        print
        print
  print

print count

# 0 and bot col agree
a = b = c = d = 0
C = [0, a, 0, b, 1, b, c, d, c]
F = []
count = 0
for i in range(2):
  C[1] = Binary[i]
  for j in range(2):
    C[3] = Binary[j]
    C[5] = Binary[j]
    for k in range(2):
      C[6] = Binary[k]
      C[8] = Binary[k]
      for l in range(2):
        C[7] = Binary[l]
        count = count + 1
        Temp = []
        Temp.extend(C)
        F.append(Temp)
        for s in range(0,3):
          print C[s],
        print
        for r in range(3,6):
          print C[r],
        print
        for t in range(6,9):
          print C[t],
        print
```

```
        print
  print

print count

# 0 and bot row agree
a = b = c = d = 0
C = [0, a, b, c, 1, d, 0, a, b]
G = []
count = 0
for i in range(2):
  C[1] = Binary[i]
  C[7] = Binary[i]
  for j in range(2):
    C[2] = Binary[j]
    C[8] = Binary[j]
    for k in range(2):
      C[3] = Binary[k]
      for l in range(2):
        C[5] = Binary[l]
        count = count + 1
        Temp = []
        Temp.extend(C)
        G.append(Temp)
        for s in range(0,3):
          print C[s],
        print
        for r in range(3,6):
          print C[r],
        print
        for t in range(6,9):
          print C[t],
        print
        print
  print

print count

#Commutative Quasi Idem
a = b = c = d = 0
C = [0, a, b, a, 1, c, b, c, d]
H = []
count = 0
for i in range(2):
  C[1] = Binary[i]
  C[3] = Binary[i]
  for j in range(2):
    C[2] = Binary[j]
    C[6] = Binary[j]
    for k in range(2):
      C[5] = Binary[k]
      C[7] = Binary[k]
      for l in range(3):
        C[8] = Binary[l]
        count = count + 1
        Temp = []
        Temp.extend(C)
        H.append(Temp)
        for s in range(0,3):
          print C[s],
        print
```

```
        for r in range(3,6):
          print C[r],
        print
        for t in range(6,9):
          print C[t],
        print
        print
  print

print count

#a = [1, 2, 3, 4, 5]
#b = [9, 8, 7, 6, 5]
#print set(a) & set(b)
print
print

Superset = H + G + F + E + D
print Superset
print len(Superset)

# D E F G H
print
AllDuplicates = [x for x in D if x in E] + [x for x in D if x in F] + [x
     for x in D if x in G] + [x for x in D if x in H] + [x for x in E if
     x in F] + [x for x in E if x in G] + [x for x in E if x in H] + [x
    for x in F if x in G] + [x for x in F if x in H] + [x for x in G if
    x in H]
print AllDuplicates
print len(AllDuplicates)

UniqueDuplicates = []
for i in AllDuplicates:
  if i not in UniqueDuplicates:
    UniqueDuplicates.append(i)
print UniqueDuplicates
print len(UniqueDuplicates)
# s is the list of duplicates
# Superset is the complete list



DuplicatesRemoved = []
for i in Superset:
  if i not in UniqueDuplicates:
    DuplicatesRemoved.append(i)

print DuplicatesRemoved
print len(DuplicatesRemoved)
```

Listing A.1: Python code to generate policies

A.2 Automatic policy generation

Listing A.2 shows the Python code of the algorithm used to automatically convert arbitrary $\text{PTaCL}_4^{\leqslant}$ polices expressed as decision tables into an equivalent normal form expressed in terms of the operators $\{-, \diamond, \otimes_{\text{b}}\}$.

```
import numpy
import time

start = time.time()

############## Functions ################

def UnarySelectionOpsToPolicyOps( selop, iterate):
  if selop == 'Sb_b':
    temp = 'd_' + str(iterate) + ' \minop (\udia\ucon d_' + str(iterate)
     +') \minop (\udia\ucon\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'S0_b':
    temp = 'd_' + str(iterate) + ' \minop (\udia\ucon d_' + str(iterate)
     +') \minop (\udia\ucon\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'S1_b':
    temp = 'd_' + str(iterate) + ' \minop (\udia\ucon d_' + str(iterate)
     +') \minop (\udia\ucon\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'St_b':
    temp = 'd_' + str(iterate) + ' \minop (\udia\ucon d_' + str(iterate)
     +') \minop (\udia\ucon\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'Sb_0':
    temp = '(\udia d_' + str(iterate) + ') \minop (\ucon\udia\ucon\udia\
    ucon d_' + str(iterate) +') \minop (\ucon\udia\ucon\udia d_' + str(
    iterate) +')'
    return temp
  elif selop == 'S0_0':
    temp = 'd_' + str(iterate) + ' \minop (\ucon d_' + str(iterate) +')
    \minop (\ucon\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'S1_0':
    temp = '(\udia\ucon\udia\ucon d_' + str(iterate) + ') \minop (\ucon\
    udia\ucon\udia\ucon d_' + str(iterate) +') \minop (\udia d_' + str(
    iterate) +')'
    return temp
  elif selop == 'St_0':
    temp = '(\udia\ucon d_' + str(iterate) + ') \minop (\ucon\udia\ucon
    d_' + str(iterate) +') \minop (\udia\udia\ucon\udia d_' + str(
    iterate) +')'
    return temp
  elif selop == 'Sb_1':
    temp = '(\udia\ucon\udia d_' + str(iterate) + ') \minop (\ucon\udia\
    ucon\udia d_' + str(iterate) +') \minop (\udia\udia d_' + str(
    iterate) +')'
    return temp
  elif selop == 'S0_1':
    temp = '(\udia d_' + str(iterate) + ') \minop (\ucon\udia d_' + str(
    iterate) +') \minop (\udia\ucon d_' + str(iterate) +')'
    return temp
  elif selop == 'S1_1':
```

```
40      temp = 'd_' + str(iterate) + ' \minop (\ucon d_' + str(iterate) +')
        \minop (\udia\udia\ucon\udia d_' + str(iterate) +')'
41      return temp
42    elif selop == 'St_1':
43      temp = '(\udia\ucon\udia\ucon d_' + str(iterate) + ') \minop (\ucon\
        udia\ucon\udia\ucon d_' + str(iterate) +') \minop (\udia\udia\ucon
        d_' + str(iterate) +')'
44      return temp
45    elif selop == 'Sb_t':
46      temp = '(\ucon\udia\ucon d_' + str(iterate) + ') \minop (\ucon\udia\
        ucon\udia\ucon d_' + str(iterate) +') \minop (\ucon d_' + str(
        iterate) +')'
47      return temp
48    elif selop == 'S0_t':
49      temp = '(\udia\ucon\udia\ucon d_' + str(iterate) + ') \minop (\udia\
        ucon\udia d_' + str(iterate) +') \minop (\udia\udia d_' + str(
        iterate) +')'
50      return temp
51    elif selop == 'S1_t':
52      temp = '(\udia d_' + str(iterate) + ') \minop (\udia\ucon d_' + str(
        iterate) +') \minop (\udia\ucon\udia\udia\ucon\udia d_' + str(
        iterate) +')'
53      return temp
54    elif selop == 'St_t':
55      temp = 'd_' + str(iterate) + ' \minop (\ucon\udia d_' + str(iterate)
         +') \minop (\ucon\udia\ucon\udia d_' + str(iterate) +')'
56      return temp
57
58
59  # In this function, we take a nary selection operator Sabc....n_X
60  # and convert it into unary selection operators. We iterate over the
        length
61  # of the selop -3 to remove _X from the construction.
62  # tempString starts at i+1 to ignore S, and we add _X via "_" + selop
        [-1]
63  def convertNarySelectionOptoUnary2( selop ):
64    temp = []
65    SelectionOpLength = len(selop)
66    for i in range(SelectionOpLength - 3):
67      tempString ='S' + selop[i+1] + "_" + selop[-1]
68      temp.append(tempString)
69    return temp
70
71  # This function converts the numerical rows into selection opertators
72  def convertPolicyTableToSelctionOps( policy ):
73    tempString = ''
74    rows = len(policy)
75    cols = len(policy[0])
76    temp = []
77    for i in range(rows-1):
78      for j in range(cols-1):
79        tempString = tempString + str(policy[i+1][j])
80      tempSelop = 'S' + tempString + '_' + str(policy[i+1][j+1])
81      temp.append(tempSelop)
82      tempString = ''
83    return temp
84
85
86  ############### Core Code ################
87
88  # This is the policy decision table we will convert into normal form
```

```
89 print
90 PolicyArray = [['p_1','p_2','p_3','P'],['t','0','1','0'],['b','0','1','t
       '],['0','1','b','1'],['1','0','t','b'],['t','b','t','t']]
91 # Saving the Policy decision table into a txt file ready for input in
       latex
92 numpy.savetxt("Policy_table_4_values.txt", PolicyArray, fmt='%s',
       delimiter=' & ', newline=' \\\\\n')
93
94
95 rows = len(PolicyArray)
96 cols = len(PolicyArray[0])
97
98 # Outputting the policy decision table
99 print 'We are converting the following Policy Table into Selection
       Operators'
100 for i in range(rows):
101   for j in range(cols):
102     print PolicyArray[i][j],
103   print
104
105 print
106
107
108 # Converting the policy decision table into n-ary selection operators
       combined with \maxop
109 PolicyArrayAsSelectionOps = convertPolicyTableToSelctionOps(PolicyArray)
110 print 'This policy array is specified by the following combination of
       selection operators:'
111 tempStringSelop = ''
112 for i in range(len(PolicyArrayAsSelectionOps)):
113   if i == len(PolicyArrayAsSelectionOps) -1 :
114     tempStringSelop = tempStringSelop + PolicyArrayAsSelectionOps[i]
115   else:
116     tempStringSelop = tempStringSelop + PolicyArrayAsSelectionOps[i] + '
       \maxop '
117 print tempStringSelop
118 print
119
120 # Converting each n-ary selection operator into their composition of
       unary selection operators
121 UnarySelectionOpsArray = []
122 for i in range(len(PolicyArrayAsSelectionOps)):
123   temp = convertNarySelectionOptoUnary2(PolicyArrayAsSelectionOps[i])
124   UnarySelectionOpsArray.append(temp)
125
126 temp1 = ''
127 temp2 = ''
128 for i in range(len(UnarySelectionOpsArray)):
129   for j in range(len(UnarySelectionOpsArray[0])):
130     if j == len(UnarySelectionOpsArray) -2 :
131       temp1 = temp1 + UnarySelectionOpsArray[i][j]
132     else:
133       temp1 = temp1 + UnarySelectionOpsArray[i][j] + ' \minop '
134   if i == len(UnarySelectionOpsArray) -1:
135     temp2 = temp2 + '\Big(' + temp1 + '\Big)'
136   else:
137     temp2 = temp2 + '\Big(' + temp1 + '\Big) \maxop '
138   temp1 = ''
139 print 'This can be expanded to an expression consisting of unary
       selection operators:'
140 print temp2
```

```
141 print
142
143 UnRows = len(UnarySelectionOpsArray)
144 UnCols = len(UnarySelectionOpsArray[0])
145
146
147
148 for i in range(UnRows):
149   for j in range(UnCols):
150     UnarySelectionOpsArray[i][j] =  UnarySelectionOpsToPolicyOps(
      UnarySelectionOpsArray[i][j],j+1)
151
152 #print UnarySelectionOpsArray
153 # Converting each unary selection operator to their equivalent
      expression comprising of E_1,E_2 and \minop
154 #for i in range(UnRows):
155 # UnarySelectionOpsArray[i] = ArrayOfSelectionOpsToPolicyOps(
      UnarySelectionOpsArray[i],i)
156
157 print 'The result'
158 print UnarySelectionOpsArray
159
160 temp1 = ''
161 temp2 = ''
162 for i in range(len(UnarySelectionOpsArray)):
163   for j in range(len(UnarySelectionOpsArray[0])):
164     if j == len(UnarySelectionOpsArray) -1 :
165       temp1 = temp1 + UnarySelectionOpsArray[i][j]
166     else:
167       temp1 = temp1 + UnarySelectionOpsArray[i][j] + ' \minop '
168   if i == len(UnarySelectionOpsArray) -1:
169     temp2 = temp2 + '\Big(' + temp1 + '\Big)'
170   else:
171     temp2 = temp2 + '\Big(' + temp1 + '\Big) \maxop \\\\'
172   temp1 = ''
173 print 'Converting the unary selection operators to expressions
      comprising of E_1, E_2 and \minop:'
174 print temp2
175
176 # Saving the unary selection operators output in a text file ready for
      input into latex
177 with open('Unary_selection_operators_4_values.txt', 'w') as f:
178     f.write(temp2)
179
180
181 print 'Blank Line'
182
183 end = time.time()
184 runtime = end - start
185 print 'Time to execute:' + str(runtime)
```

Listing A.2: Python code to generate policies