ABSTRACT

Title of dissertation: A LOGIC-BASED FRAMEWORK FOR WEB ACCESS CONTROL POLICIES

Vladimir Kolovski, Doctor of Philosophy, 2008

Dissertation directed by: Professor James Hendler
Department of Computer Science

With the widespread use of web services, there is a need for adequate security and privacy support to protect the sensitive information these services could provide. As a result, there has been a great interest in access control policy languages which accommodate large, open, distributed and heterogeneous environments like the Web. XACML has emerged as a popular access control language, but because of its rich expressiveness and informal official semantics, it suffers from a) a lack of understanding of its formal properties, and b) a lack of automated, compile-time services that can detect errors in expressive, distributed and heterogeneous policies.

In this dissertation, I present a logic-based framework for XACML that addresses the above issues. One component of the framework is a Datalog-based mapping for XACML v3.0 that provides a theoretical foundation for the language: a concise and formal semantics and complexity results for full XACML and various fragments. Additionally, considering that most previous work on access control is based on some variant of Datalog, my mapping discovers close relationships between XACML and other logic based languages such as the Flexible Authorization Framework.

The second component of this framework provides a practical foundation for static analysis of expressive XACML policies. The analysis services detect semantic errors or differences between policies before they are deployed. To provide these services, I present a mapping from XACML to the Web Ontology Language (OWL), which is the standardized language for representing the semantics of information on the Web. In particular, I focus on the OWL-DL sub-language, which is a logic-based fragment of OWL. Finally, to demonstrate the practicality of using OWL-DL reasoners as policy analyzers, I have implemented an OWL-based XACML analyzer and performed extensive empirical evaluation using both real world and synthetic policy sets.

A LOGIC-BASED FRAMEWORK FOR WEB ACCESS CONTROL POLICIES

by

Vladimir Kolovski

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Dedication

To my family

Acknowledgments

I would like to thank here the many people that helped me during my graduate studies at University of Maryland. First of all, I would like to thank my advisor, James Hendler, who gave me the freedom to pursue my research topic and provided me with unwavering support during my graduate studies. Thank you Jim for your guidance and encouragement. I would also like to thank Jennifer Golbeck, who de facto became my co-advisor in the latter stages of my studies. A heartfelt thanks to you Jen for your extremely useful advice on all things dissertation-related. Finishing this thesis would not have been possible without the support of Prof. Hendler and Prof. Golbeck.

I would like to thank my thesis committee members: Ashok Agrawala, Daniel Weitzner, V.S. Subrahmanian and Bruce Jacob for providing valuable feedback on my dissertation. I would especially like to thank Prof. Subrahmanian, who provided me with important technical feedback after my thesis proposal that helped me narrow my research topic.

My colleagues at the MINDSWAP research group helped me in every stage of my research. Above all, I would like to thank Christian Halaschek-Wiener for being a close friend and a valuable source of technical and professional advice. I would also like to thank Bijan Parsia, who pushed me in the right direction during the rockiest year of my graduate studies. I received a lot of support from the other members or my research group, including Aditya Kalyanpur, Evren Sirin, Taowei (David) Wang, Bernardo Cuenca-Grau, Yarden Katz, Hamid Haidarian-Shahri, Bin Zhao, Naiwen Lin, Ugur Kuter and others I may have forgotten.

I would like to express my gratitude to Kendall Clark from Clark&Parsia and Andrew Schain from NASA HQ for their collaboration on developing realistic policy sets. Thank you for the numerous discussions on how to apply my research in a real-world setting.

On a more personal note, I would like to thank my close friends and housemates from LagunaHouse: Dov, Arkady, AT and Matus, who made my life in Prince George's county a lot easier and helped me immensely in finishing my thesis writing smoothly.

A heartfelt thank you to my parents Metodija and Katerina for providing a caring and stimulating intellectual environment at home. Thanks to my sister Jana for always loving me, no matter how much I annoyed her. Finally, I would like to thank my fiancee Marija for providing me with endless love and support during this whole process. Thank you Marija for making College Park feel like home.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the ever-increasing amount of data being made accessible on the Web, there is a corresponding increasing need for information publishers and data owners to control access to sensitive segments of their data. While access control specification and management has been an active field for more than 30 years[1], applying these security mechanisms in an open, distributed and heterogeneous environment like the World Wide Web (or the enterprise) presents novel challenges. For example, an access control language for such distributed environment should be powerful and flexible enough to express many different types of security policies (e.g., mandatory, discretionary and role-based access control policies) and support various datatype functions. In such an environment the resources that are being protected might be distributed, but also the policy specifications themselves could be also distributed, so a mechanism for references among policies and policy integration is needed.

There has been great interest coming from industry and academia in developing an expressive access control policy language for open, heterogeneous and distributed environments that can cover the above access control requirements [113, 15, 27, 87, 34, 72, 128]. To facilitate interoperability, there has been a strong push (mostly by enterprise

[1]One of the seminal papers on secure computer systems (by Bell and LaPadula) was published in 1973[28].

application providers like Oracle and SAP) to standardize one of the proposed security languages.

The eXtensible Access Control Markup Language (XACML [113]) has emerged as the language preferred by most leading enterprise application providers: it was standardized by OASIS in 2003. XACML was originally developed by Sun Microsystems and has gained momentum over the years: it is currently deployed or supported by more than 65 systems and projects [16]. The language has also attracted academic attention – there are more than 200 peer-reviewed papers published on XACML since 2003 [16].

XACML is a *declarative* access control language with a variety of features: it allows for distributed policies and resources, it supports attribute-based (as opposed to identity-based) access control, negative authorization (*Deny* policies), conflict resolution algorithms to integrate distributed policies and role-based access control (RBAC). Moreover, the language has more than 200 built-in functions that range from data-type comparison and conversion to boolean functions and even some with higher order flavor. The language is based on XML and specifies a processing model for checking policy compliance at run time. As an example, following is a XACML rule which returns a `Permit` only when the value of the *action-id* attribute is *read*.

```
<Rule RuleId="ReadRule" Effect="Permit">
  <Target>
    <Actions>
      <Action>
        <ActionMatch MatchId="...:string-equal">
          <AttributeValue DataType="#string">read</AttributeValue>
          <ActionAttributeDesignator DataType="...#string"
            AttributeId="...:action-id"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
</Rule>
```

1.1.1 Motivation: Lack of Formal Semantics

The specification of XACML v1.0 was published in 2003, and since then there has been continuing work on the standard. As of March 2008, XACML v3.0[113], which includes delegation policies, is close to standardization. During this time, the OASIS working group developing XACML has not yet provided a formal semantics for the language – the official semantics of XACML is given informally, in normative documents. Previous academic efforts [68] that formalized early versions of XACML notwithstanding, currently there is no formal semantics that covers XACML 3.0 and includes the profiles that are part of its latest version: delegation policies [114] and Web Service access control policies (WS-XACML)[19].

One consequence of the lack of a formal treatment is that the computational complexity properties of the language are still unknown. For example, consider *access request checking*, the main service the XACML processing model provides: given an access request R and a policy P, determine the access decision of P for R[2]. Given the rich set of features of XACML, its verbose specifications, and its lack of a formal treatment, it is still unknown if access request checking is tractable.

A formal, declarative semantics would also shed light on the relationship between XACML and previous work on languages for distributed access control policies. Even before work on XACML began, there were numerous policy language proposals coming

[2]XACML supports `Permit`, `Deny`, `Indeterminate` (for errors) and `NotApplicable` as access decision.

from both academia and industry [87, 32, 27, 72, 12]. While there have been some attempts to compare XACML to its closest competitors from industry (e.g., EPAL [15]), there has been no in-depth comparison with formal security languages such as the Flexible Authorization Framework [72], Delegation Logic [87] or SecPal [27]. Most of these academic policy languages have a formal foundation (usually logic-based) and tractable (polynomial) complexity. From an academic viewpoint, a logic-based semantics for XACML would help us understand where the language fits in (in terms of expressiveness and computational complexity) in the active research field of access control policy specification. If it can be shown that XACML is close to some of these languages, then from a practical perspective, these similarities could provide possible areas of improvement for XACML itself by adding features available in other languages.

1.1.2 Motivation: Analysis Services for Distributed Policies

Due to the expressiveness of XACML and its lack of ”compile time” support, it is non-trivial for a policy developer to understand the overall effect and consequences of the XACML policies he/she writes. Even arguably the most important feature in access control - checking that the policy will not result in leakage of permissions to an unintended or unauthorized principal, i.e., *safety* - is difficult (if not impossible) to do manually. For example, incomplete security policies might unintentionally give access to an intruder. How can a security administrator be certain that her policy covers all possible corner cases? Even if the administrator does discover a bug in the policy, and fixes it accordingly, the consequences of that fix (policy change) are difficult to analyze.

One approach for verifying correctness of policies is to perform testing [98], where a test case for a policy would consist of an access request and expected outcome (`Permit`, `Deny`, `NotApplicable` or `Indeterminate`). However, testing is not exhaustive and it is difficult to think of all possible scenarios that need to be tested.

To illustrate the limitations of testing, consider the following example. In this scenario, there are only two roles: *Manager* and *Developer*, one resource: *Report*, and two actions: *read*, *write*. The main (root) policy contains two policies which are combined using `First-Applicable` combining algorithm. `First-Applicable` is a XACML combining algorithm that, given a set of policies, returns the decision of the first policy that is successfully applied while ignoring the decisions of the subsequent policies. In the example below, if $P1$ returns a decision (`Permit` or `Deny`), this decision would be propagated to the parent policy $PS1$ without taking into account the decision of $PS2$.

The policy is presented in graphical form in Figure 1.1.

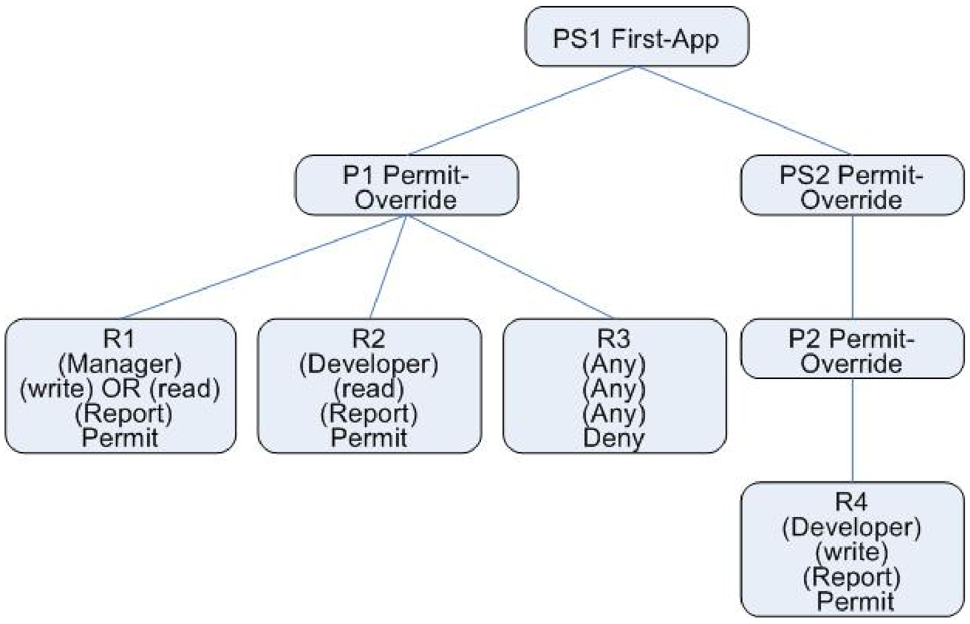

PS1 First-App
P1 Permit-Override
PS2 Permit-Override
R1 (Manager) (write) OR (read) (Report) Permit
R2 (Developer) (read) (Report) Permit
R3 (Any) (Any) (Any) Deny
P2 Permit-Override
R4 (Developer) (write) (Report) Permit

Figure 1.1: Example Policy

In this example, a security administrator could specify a test condition for this policy in the form of a XACML access request, "*Developer* requests *write* access to *report*", and the expected outcome, `Deny`. Testing policies in such manner is not exhaustive, since it is difficult to think of all possible conditions that need to be tested. In this particular example, the simple test would pass, since the `Deny` rule R_3 would fire, however the policy would be still vulnerable. Consider what happens if a user tries to both *read* and *write* to *File* as part of the same access request. In this case, rule R_2 will fire as well, thus the policy set would return a `Permit`. Essentially, this policy allows an invalid request (a *write+read* request) to piggyback on top of a valid one (*read* request).

Violations such as above will not necessarily be caught by simple testing. For this purpose, techniques such as *formal verification* of a policy against security properties have been investigated in literature [67, 74, 50, 131]. Formal verification explores all possible combinations of attributes in a policy in an attempt to break the security property, so it would catch the security error in the above example.

The above examples illustrated the benefits of formal verification of policies, and there has been some interest in providing such services for XACML [50, 66, 131]. However, the analysis approaches of previous work do not capture more expressive language features needed for the type of heterogeneous and distributed environments that XACML is intended to be deployed in. These features include:

- **Data-types** and **functions**. XACML supports all of the XML Schema Datatypes and in addition it defines four datatypes of its own: *ipAddress*, *x500Name*, *rfc-822Name* and *dnsName*. The language supports a powerful set of more than 200 functions that make use of these datatypes: from data-type matching functions (e.g., *string-equal*, *date-time-greater-than*) to boolean and set operations and functions with higher order flavor (i.e., functions that operate on other functions).

- **Heterogeneous** policies. There has also been interest in extending XACML with expressive policy descriptions and support for data integration, for the purpose of federated access control management. Consider what happens when access control is federated across multiple, independent organizations. These organizations might use different attribute schemes to describe subjects, actions and resources. For example, in an e-commerce scenario, an online book store might use a boolean attribute *adultAge* to represent adult customers, whereas a video rental store might use an integer attribute *age*. In a large enterprise with many different departments where each department potentially has a different attribute scheme, integrating and reasoning about such heterogeneous security information is a challenging problem.

- **Delegation** policies. XACML allows users to specify delegation policies in addition to access policies. Delegation policies can specify which users have rights to add access policies: 'HR-Admins can create policies concerning the Payroll servers' , or can be used for delegation as well: 'Jack can approve expenses while Mary is on vacation'.

In previous work, there exists no formal analysis framework that can reason about XACML policies with the above features.

1.2 Proposed Solution

This dissertation presents a theoretical and practical foundation for analysis of expressive XACML policies. The theoretical component presents a formal, proof-theoretic semantics for XACML 3.0 based on natural deduction rules. The semantics covers the core of XACML along with its Delegation and Web Service Policies Profile, and represents a concise and unambiguous version of the informal semantics given in the language specification. To determine XACML's complexity properties, I provide a translation of various subsets of the language to variants of the rule-based language Datalog, thus establishing its polynomial data complexity and close relationship to other logic-based languages such as the Flexible Authorization Framework [73]. Additionally, I show that access request checking in XACML with arbitrary references between policies is NP-complete.

The practical component of this thesis presents a logic-based analysis framework that can reason about *administrative* and *heterogeneous* XACML policies. The analysis support is done at compile-time and enables discovery of semantic errors or differences between policies before they are deployed. One of the goals of this dissertation is to provide these analysis services while hiding the details of the logic formalism used and the internals of the analysis tool. To accomplish this, I allow users to specify security properties in XACML and present the verification results back in XACML. In addition to verification, the analysis framework also provides policy comparison (including checking for subsumption and compatibility/disjointness), and detecting redundant ("dead") policies.

As a basis for the practical analysis framework, I use Description Logics (DL), which are a family of formalisms that are decidable subsets of First-Order logic, and are the formal basis for the Web Ontology Language(OWL[3]) [45]. Because of the correspondence of policy analysis services to DL reasoning services (e.g., policy comparison can be reduced to concept subsumption, whereas formal verification can be reduced to concept satisfiability), my framework can leverage off-the-shelf DL reasoners optimized to provide the above-mentioned analysis services.

An important benefit of using a logic compatible with OWL is that I can leverage OWL being a W3C standard for representing and integrating information on the Web. Thus, the analysis framework has built-in support for representing expressive vocabulary domains for policies (as OWL ontologies), so it is able to reason about heterogeneous XACML policies distributed across the enterprise (or the Web).

To evaluate the practicality of my approach, I implemented the framework as a XACML analysis tool that reduces policy analysis services to DL reasoning tasks. I performed empirical evaluations using real-world XACML policies, testing the performance of my approach against other scalable XACML analyzers. Additionally, to fully evaluate the expressive features of the analysis framework, including support for information integration across heterogeneous policies, I used two real-world policy use cases: the NASA Federated Data Access use case[118] and an RBAC policy from the healthcare domain [62] (the policy is part of the Health Level 7 standard). Considering the size and complexity of both of these real-world policies, my empirical evaluation shows the approach

[3]OWL has three subsets: OWL-Lite, OWL-DL and OWL Full, the first two of which are grounded in description logics. From now on, I will use OWL to refer to the OWL-DL sublanguage.

is practical for large and expressive XACML policy sets.

1.3 Contributions

The contributions of this thesis are as follows:

- A formal, proof-theoretic semantics of XACML v3.0 that covers the core specification and the Administrative Policy Profile. The semantics is given using natural deduction rules.

- A mapping of XACML to Datalog that provides a model-theoretic semantics and computational complexity results for full XACML and various fragments. Additionally, an extensive comparison with other logic-based languages such as Flexible Authorization Framework [73] based on the Datalog mapping.

- A static analysis framework based on OWL-DL that can reason about expressive XACML policies. A comprehensive set of services are provided with this mapping: formal verification, policy comparison (change analysis) and redundancy checking.

- Demonstration that the analysis framework is applicable to other domains. This was accomplished by formalizing and analyzing the W3C standard language for web service policies (WS-Policy [129]) .

- An empirical evaluation of the scalability of the analysis framework. The evaluation demonstrates the scalability of the analysis framework using real-world policies: a XACML policy test suite consisting of policies used to evaluate other XACML analyzers, as well as datasets from two real-world use cases for expressive policies (NASA HQ data access and healthcare RBAC policies).

1.4 Organization

This thesis is organized as follows: Chapter 2 introduces background information and necessary preliminaries needed to understand the technical contributions of the dissertation. Chapter 3 surveys related work to this dissertation, namely: access control policy models and languages, and policy analysis and verification approaches. In Chapter 4 a formal, proof-theoretic semantics for XACML 3.0 is presented using natural deduction rules. Chapter 4 also provides a translation of various subsets of the language to locally stratified Datalog, thus establishing its polynomial data complexity and close relationship to other logic-based access control languages. Chapter 5 presents an analysis approach that can reason about both distributed and heterogeneous XACML policies. In particular, I show how with a mapping to description logics (DL), we can perform change analysis, formal verification, and coverage checking for XACML policies. An application of the formal framework to other domains is presented in Chapter 6. In particular, WS-Policy (a web services policy language) is formalized and analyzed in a similar fashion to XACML. In Chapter 7, I present my prototype implementation of a XACML-to-DL mapper and policy analyzer. I have performed an extensive empirical evaluation to determine if DL reasoners are suitable as policy analyzers; the results of the evaluation are also presented in Chapter 7. Finally, Chapter 8 presents concluding remarks as well as possible areas of future work.

Chapter 2

Preliminaries

In this chapter, I present background information on the language that is the focus of this dissertation: XACML, and on the logical formalisms used to analyze XACML, namely: Datalog and Description Logics. The purpose is to present the basic concepts, terminology and definitions that are used throughout this thesis.

2.1 XACML

The eXtensible Access Control Markup Language (XACML [113]) is a standardized, expressive and increasingly popular XML-based language for writing access control policies about distributed resources.

Before I discuss the syntax of XACML policies, I present a brief overview of the high level XACML policy model. Essentially, there are two high level components of a XACML-enabled system. A *PDP*, or Policy Decision Point, is the processing engine that evaluates access requests against XACML policies. A *PEP*, or Policy Enforcement Point, is the application-specific element that physically enforces access to a resource. The PEP generates the access requests to be sent to the PDP and enforces the access decisions made by the PDP. The focus of my dissertation is on how policy decisions are *made* (not enforced), so the discussion from now on will be mostly regarding the PDP.

At the root of all XACML policies is a `Policy` or a `PolicySet`. A `PolicySet` is a container that can hold other `Policies` or `PolicySets`, as well as references to policies found in remote locations. A `Policy` represents a single access control policy, expressed through a set of `Rules`. Each XACML policy document contains exactly one `Policy` or `PolicySet` root element. Following, an explanation of the basic elements of XACML (`Rules`, `Targets` and `Attributes`) is provided and then more advanced features of the language are discussed.

2.1.1 `Rules`, `Targets`, `Attributes` and `Requests`

`Rule` is the most basic policy element of XACML that actually makes an access decision. Essentially, a `Rule` is a function that takes an access request as input and yields a `Permit`, `Deny` or `Not-Applicable`. To determine if a `Rule` is applicable to an access request, the `Target` element is used.

The `Target` defines the set of requests to which the rule is intended to apply in the form of a logical expression on attributes in the request. `Target` is comprised of a conjunction of `DisjunctiveMatch` elements, where each `DisjunctiveMatch` contains a set of `ConjunctiveMatch` elements. Finally, each `ConjunctiveMatch` contains a list of attributes and values.

Attributes are the atomic unit in XACML. They represent characteristics of the subjects, resources, actions or the environment where the access request was made. For example, a user's role, their name, the file they want to access and the current date are all attribute values. *Access requests* in XACML are represented as a set of attribute-value pairs. Each attribute can belong to a category - the most common categories in XACML are `Subject`, `Resource`, `Action` and `Environment`[1].

An example of a rule that returns `Deny` for access requests that have value *read* for *action-type* attribute is given below:

```
<Rule RuleId="rule" Effect="Deny">
 <Target>
  <DisjunctiveMatch>
  <ConjunctiveMatch>
   <Match MatchId="function:string-equal">
    <AttributeValue DataType="#string">read</AttributeValue>
    <AttributeDesignator
      AttributeId="action-type"
      Category="...attribute-category:action"
      DataType="...#string"/>
   </Match>
  </ConjunctiveMatch>
  </DisjunctiveMatch>
 </Target>
</Rule>
```

2.1.2 Combining Algorithms

Because a `Policy` or `PolicySet` may contain multiple policies or `Rules`, each of which may evaluate to different access control decisions, XACML a mechanism to combine access decisions. This is accomplished using a collection of combining algorithms, where each algorithm represents a different way of combining multiple access decisions into a single one. Following is a list of the most common combining algorithms:

- `Permit-overrides`. If any rule evaluates to `Permit`, then the combined decision is also `Permit`.
- `Deny-overrides`. If any rule evaluates to `Deny`, then the combined decision is also `Deny`.

[1]The full names of these categories consist of a lenghty prefix, e.g., *urn:oasis:names:tc:xacml:-3.0:attribute-category:Action*. For clarity, I use shortened names in this dissertation.

- `First-applicable`. The effect of the first rule that applies is the decision of the policy. The rules must be evaluated in the order that they are listed.

- `Only-one-applicable`. If more than one rule is applicable, return `Indetermina-te`. Otherwise return the access decision of the applicable rule.

In this dissertation, I use the following notation: for a XACML policy element P, I refer to its `Target`, `Effect` (in cases of `Rules`), its ordered list of children policy elements, its parent policy element and combining algorithm using *P.target*, $P.effect$, *P.children*, *P.parent*, *P.comb* respectively. *P.pos* is used to refer to the position of P w.r.t its sibling policy elements.

2.1.3 Administrative XACML

So far, I have only been discussing XACML access policies, i.e., policies that specify the situations under which users are granted or denied access to a resource. However, XACML 3.0 also supports *administrative* (or delegation) policies, which essentially are policies that authorize access policies. For example, an administrative policy might state that members of group *Clinicians* are allowed to write access policies about *PatientReports*. This section described the basic elements of an administrative policy.

Policy Issuers and Delegates A policy in XACML can contain a `PolicyIssuer` element that describes the source of the policy. A special form of the `PolicyIssuer` element, called the *trusted* issuer, is used to specify that a policy is trusted by the Policy Decision Point (PDP). A missing `PolicyIssuer` element is shorthand for the trusted issuer. In previous versions of XACML with only access policies, the `PolicyIssuer` element is missing and all access policies are trusted (authorized) by default. If a policy's issuer is not trusted, then the policies has to be authorized by using available administrative policies.

The `Delegate` element of the administrative policy is used for matching against other policies. Essentially, if the `Delegate` of policy A matches the `PolicyIssuer` of policy B, that means that A can authorize policy B.

When an administrative policy authorizes an access policy, it can also specify under which conditions the access policy is authorized. This is called a constrained situation in [114], and is analogous to the `Target` attribute in access policies.

2.1.4 Hierarchical and Multiple Resource Profile

The policy evaluation performed by a XACML PDP is defined in terms of a single requested resource, with the authorization decision contained in a single `Result` element in the response. However, A Policy Enforcement Point, or PEP, may wish to submit a single request context for access to *multiple* resources, and may wish to obtain a single response context that contains a separate authorization decision (`Result`) for each requested resource. Such a request context might be used to avoid sending multiple decision request messages between a PEP and PDP, for example. Alternatively, a PEP may wish to submit a single request context for all the nodes in a hierarchy, and may wish to obtain a single authorization decision that indicates whether access is permitted to all of the requested nodes.

The Multiple Resource Profile provides a mechanism such that a PEP can request authorization decisions for multiple resources in a single request context. It is important to note that the Multiple Resource Profile does not affect the policy itself. It deals with the XACML Access Requests, introducing syntactic shorthand so that multiple requests contexts can be merged into one.

The Hierarchical Resource Profile[18] allows users to specify one policy that applies to an entire subtree of a hierarchy, rather than having to specify a separate policy for each node of the subtree. In this Profile, a resource organized as a hierarchy may be a with a single root (tree) or multiple roots (forest), however cycles are not allowed. The nodes in a hierarchical resource are treated as individual resources. An authorization decision that permits (or denies) access to an interior node does not imply that access to its descendant nodes is permitted (or denied).

2.2 Logic Preliminaries

2.2.1 Datalog

Datalog [111] is a logical language typically used to specify facts, rules and queries in deductive databases (i.e, databases that can infer information based on existing facts and rules). The basic building blocks of Datalog are the following:

- *term* can be either a *constant* or a *variable* (variables are denoted with a starting '?')

- *atom* – is of the form $P_i(t_1, \ldots, t_k)$ where P_i is a predicate symbol and t_i are terms.

Given an atom A, a *literal* stands for A or $\neg A$. $\mathcal{S}(P)$ denotes the set of all predicate symbols in the datalog program P.

- *fact* – assertion about a relevant piece of the world. Facts are expressed using literals that do not have any variable terms. For example, 'Vlad is enrolled at University of Maryland' can be expressed as $enrolled(Vlad, UniversityMaryland)$

- rule – logical sentence that allows us to infer facts from other facts. For example, 'If X is enrolled at a university, then X is a student' is a rule. The following Prolog-like syntax will be used to describe rules:

$$H \quad \text{:-} \quad B_1, \ldots, B_n$$

where H and B_i are literals. H represents the head of the Datalog rule and $B_1, \ldots, B_n$ the body literals. Given this, the rule example above can be written as:

$$student(?X) \quad \text{:-} \quad enrolled(?X, ?Y), university(?Y)$$

A predicate that occurs in the head of some rule is called an *IDB* (intensional database) predicate, while all other predicates are called extensional (*EDB*). The predicates occurring in the body can be either intensional or extensional.

Datalog can be considered as a special case of general logic programming, since it imposes the following conditions on logic programs:

- Safety condition: each variable that occurs in the head of a rule must also occur in the body of the same rule. This condition guarantees that the set of all facts that can be derived from a Datalog program is finite.

- Datalog does not allow nesting of terms terms (i.e., no function symbols). For example, terms such as $F(G(?x))$ are not allowed.

2.2.1.1 Semantics of Datalog

The Herbrand Base HB of a Datalog program P is the set of all facts (ground atoms) defined over the predicates in $\mathcal{S}(P)$. A Herbrand *interpretation* $\mathcal{M}$ is simply a subset of the Herbrand Base HB, which contains all of the ground facts that are true under $\mathcal{M}$. A Datalog rule of the form $H_0 : -B_1, \ldots, B_n$ is true under $\mathcal{M}$ iff for each substitution θ which replaces variables by constants, whenever $\theta(B_1), \ldots, \theta(B_n) \in \mathcal{M}$, then it also holds that $\theta(H_0) \in \mathcal{M}$.

A Herbrand interpretation where all of the rules and clauses in a program are true is called a Herbrand *model* for the program P. A model $\mathcal{M}$ is minimal if no subset of $\mathcal{M}$ is a model for P. Assuming P is finite and there is no negation of IDB predicates, then there exists a unique minimal Herbrand model, which represents the meaning of the program P.

2.2.1.2 Negation in Datalog

In positive Datalog, negated literals are not allowed in heads or bodies of rules. However, by adopting the Closed World Assumption (CWA) it is still possible to infer negative facts from a set of positive Datalog clauses. CWA states that if we cannot infer a fact from a set of Datalog rules, then we infer the negation of that fact.

Adding unrestricted negation in rule bodies produces more complicated semantics for the datalog program. Consider the following program consisting of one rule $P_a = \{student(John) \text{ :- } \neg professor(John)\}$. P_a has two minimal Herbrand models $\mathcal{M}_1 = \{student(John)\}$ and $\mathcal{M}_2 = \{professor(John)\}$. Having multiple minimal models produces more complicated semantics, since it is unclear which one of these models should be chosen during query answering (it also increases the computational complexity of the language).

To retain the unique minimal model property, there is a version of Datalog called *stratified* Datalog that allows a restricted form of negation. The intuition behind it is as follows: when evaluating a rule with one or more negative literals in the body, first evaluate the predicates corresponding to these negative literals. To evaluate these negative literals, we might need to evaluate additional rules that can have negative body literals as well, so we have to make sure that the program will allow such step-by-step evaluation without running into a cycle. Datalog programs that allow such evaluation are called *stratified*.

Definition 1 Stratified Datalog. A Datalog program is called stratified if there is a partition $P = P_1 \cup \ldots \cup P_N$ such that the following conditions hold for $i = 1, 2, \ldots, n$:

1. Each IDB predicate in P has all of its defining rules (i.e., rules where the predicate occurs in the head) in one partition of P,

2. Each partition P_i contains only rules where the negative literals correspond to predicates defined in partitions P_j where $j < i$.

Each partition P_i is called a *stratum* of P. The *level* of a predicate symbol is the index of the strata within which it is defined.

Query answering in positive and stratified Datalog has polynomial data complexity, whereas Datalog with unrestricted negation is data complete for co-NP [44] under the stable model semantics [54].

2.2.2 Description Logics

Description Logics (DL) are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood manner [22]. The name comes from the facts that, on the one hand, the application domains are described using concept *descriptions* and, on the other hand, they possess formal, *logic*-based semantics which can be given by a translation into first-order logic (FOL).

Each DL consists of the following building blocks: *atomic concepts*, *atomic roles* and *individuals*. Atomic concepts correspond to unary predicates in FOL (e.g., $Student(x)$), atomic roles correspond to binary predicates in FOL (e.g., $enrolledIn(x, y)$) and individuals represent constant terms in FOL.

Atomic concepts and roles are elementary descriptions of objects; complex ones can be built on top of them using DL *constructors*. For example, applying a concept disjunction constructor ($\sqcup$) on the atomic concepts $Male$ and $Female$, we retrieve the set of all individuals who are either Male or Female: $Male \sqcup Female$. In addition to disjunction, DLs typically provide the standard boolean operators as constructors: concept conjunction ($\sqcap$) and concept negation ($\neg$). Most DLs also provide a restricted quantification, in terms of universal and existential restrictions on roles. There are many other additional concept and role constructors; they will be discussed in Section 2.2.2.2.

In addition to constructors that allow us to form complex concepts and roles, a DL also provides means for expressing axioms (logical relations) involving concepts and roles. For example, we can specify concept inclusion of the form $Student \sqsubseteq Person$ stating that every student is a person, and role inclusion such as $isBrother \sqsubseteq isRelated$ stating that if two individuals are brothers, that implies that they are related.

DL knowledge bases (KB) typically consists of the following components:

- A *TBox* containing intensional knowledge (axioms and concepts) in the form of a terminology. The axioms in the TBox are concept inclusions of the form $C_1 \sqsubseteq C_2$ where C_1 and C_2 are concepts (not necessarily atomic).

- An *RBox* containing role inclusion axioms of the form $R_1 \sqsubseteq R_2$ where R_1 and R_2 are DL Roles.

- An *ABox* containing extensional knowledge about the individuals in the domain. Axioms in the ABox are of the form C(a), called concept (or type) assertions and R(a,b), called role assertions, where a,b are individual names, R is a role and C is a concept.

There are different types of TBoxes depending on the nature of the concepts occur in their axioms. The simplest TBox type consists of a restricted form of concept inclusion axioms called concept *definitions*: sentences of the form $A \sqsubseteq C$ or $A \equiv C$, where A is atomic. Restricting a TBox to concept definitions which are both *unique* (each atomic concept occurs only once on the LHS of an inclusion axiom) and *acyclic* (the RHS of an axiom cannot refer, directly or indirectly, to the concept in the LHS) yields a *definitorial* TBox. On the other hand, if a TBox contains axioms of the form $C \sqsubseteq D$ where C is non-atomic, then the axiom is called a *general concept inclusion* axiom (GCI) and the TBox is called a *general* TBox. The distinction between definitorial and general TBoxes is important since a definitorial TBox greatly reduces reasoning complexity.

A very common example of DL concept constructors often referred to in this thesis are DL number restrictions. The most expressive form is *qualified* number restrictions, which allow building of the concepts $\geq nR.C$ and $\leq nR.C$ from a role R, a natural number n and a concept C. For example, qualified number restrictions can be used to represent a father of exactly two sons:

$$Male \sqcap \leq 2 hasChild.Male \sqcap \geq 2 hasChild.Male$$

A more restricted form are unqualified number restrictions – these do not allow to specify a what kind of concept is used as role filler in the restriction. For example, unqualified number restrictions can be used to denote a father of exactly two children:

$$Male \sqcap \leq 2 hasChild \sqcap \geq 2 hasChild$$

Finally, a very important feature of DL for the purpose of this thesis is their datatype support, i.e., support for describing concepts using numbers, strings, regular expressions, IP addresses, etc. The main approach is to provide DLs with an interface to concrete domains, together with a set of built-in predicates which are associated with that interface. The interface is achieved by using a new type of roles, called *datatype* (or concrete) roles, which link abstract objects from the DL domain with datatype predicates from the concrete domain. Also, new concept constructors related to these datatype roles is added. For example, we can denote the set of all people who are more than 18 years old using a datatype role: $\exists age. \geq_{18}$. Since concrete domains are important for the purposes of this thesis, in the next section we formally present their definition and properties. The concrete domains presentation is based on [104].

2.2.2.1 Concrete Domains

Informally, a concrete domain provides a set of predicates with a predefined interpretation. If a decision procedure for checking satisfiability of finite conjunctions over concrete domain predicates exists, many DLs can be coupled with a concrete domain while retaining decidability. However, in [96] it was shown that a logic with GCIs and concrete domains is undecidable. In order to retain decidability, several restrictions were investigated (a survey is available at [97]). The Web Ontology Language, OWL [45] supports a basic form of concrete domains, referred to as OWL datatypes; thus, any OWL reasoner implementation also supports reasoning with (limited) concrete domains.

Definition 2 A concrete domain D is a pair ($\triangle_D$,Φ), where where $\triangle_D$ is a set called the domain of D, and Φ is a finite set of predicate names. Each $d \in \Phi$ is associated with an arity n and an extension $d^D \subset \triangle_D^n$. A concrete domain D is admissible if the following conditions holds:

- Φ is closed under negation. In other words, for each $d \in \Phi$, there exists a predicate $\neg d \in \Phi$ with $\neg d = \triangle_D^n \setminus d^D$

- Φ contains a unary predicate $\top_D$ interpreted as the universal concept ($\triangle_D$)

- satisfiability of finite conjunctions of the form $\bigwedge_{i=1}^{n} d_i(x_i)$ is decidable

The interpretation of concrete objects is usually separated from the interpretation of the other (abstract) DL objects in the logic to retain decidability.

Additionally, for web ontology languages such as OWL, the coupling has been such that only unary concrete predicates are allowed. However, the definition for concrete domains does not have such restriction. In fact, there are implementations of DL reasoners coupled with n-ary concrete domains (see RacerPro [58]) – this is important since n-ary concrete domains are extensively used in access control policy languages such as XACML.

Finally, without loss of generality we can consider only one concrete domain at a time, since an approach for combining two or more concrete domains into one has been presented in [23].

2.2.2.2 Syntax and Semantics of $\mathcal{SHIQ}(D)$

There has been a great amount of attention to developing reasoning algorithms for increasingly more expressive logics in the DL family. In this following section, I will formally present the syntax and semantics of the very expressive logic $\mathcal{SHIQ}(D)$, which is expressive enough for the purposes of this thesis.

Syntax of $\mathcal{SHIQ}(D)$

Definition 3 $\mathcal{SHIQ}(D)$ Roles Let N_R, N_R^c be the disjoint sets of abstract and concrete atomic roles. The set of $\mathcal{SHIQ}(D)$ abstract roles is the set $N_R \cup \{R^- \mid R \in N_R\}$, where R^- denotes the inverse of the atomic role R. To avoid considering roles such as R^{--}, we define the function Inv such that $Inv(R) = R^-$ and $Inv(R^-) = R$ for $R \in N_R$. There are no inverses on concrete roles, so the set of $\mathcal{SHIQ}(D)$ concrete roles is simply N_R^c. A role inclusion axiom is of the form $R \sqsubseteq S$ where $R, S \in N_R$, or of the form $u \sqsubseteq v$, where $u, v \in N_R^c$. A transitivity axiom is an expression of the form $Trans(R)$, where $R \in N_R$. Finally, an RBox $\mathcal{R}$ is a set of role inclusion and transitivity axioms.

Let $\sqsubseteq^*$ be the reflexive-transitive closure of $\sqsubseteq$. A role R is transitive if there is a role S s.t. $Trans(S) \in \mathcal{R}$ with $S \sqsubseteq^* R$ and $R \sqsubseteq^* S$. R is called a *simple* role if there is no role S s.t. $S \sqsubseteq^* R$ and S is transitive.

Definition 4 $\mathcal{SHIQ}(D)$ Concepts Let N_C, N_I stand for the set of concept and individual names, and $\mathcal{D} = (\triangle_D, \Phi_D)$ be a concrete domain, where $\triangle_D$ stands for the domain, and Φ_D for the set of predicate names in $\mathcal{D}$. The set of $\mathcal{SHIQ}(D)$ -concepts is defined inductively as the smallest set for which the following holds:

- every concept name $C \in N_C$ is a concept
- if C and D are concepts and R is a role, then $(C \sqcap D)$, $(C \sqcup D)$, $(\neg C)$, $(\forall R.C)$ and $(\exists R.C)$ are also concepts
- if C is a concept, R a simple role and n a natural number, then $(\leq\ nR.C)$ and

$(\geq\ nR.C)$ are also concepts (called at-most and at-least qualified number restrictions)

- If $P \in \Phi_D$ and $u \in N_R^c$ then $(\exists u.P)$, $(\forall u.P)$, $(\leq nu.P)$, $(\geq nu.P)$ are also concepts.

We write $\top$ and $\bot$ to abbreviate $C \sqcup \neg C$ and $C \sqcap \neg C$ respectively.

For concepts C, D, a *concept inclusion axiom* is an expression of the form $C \sqsubseteq D$. A TBox T is a finite set of concept inclusion axioms. An ABox A is a finite set of concept assertions of the form $C(a)$ (where C can be an arbitrary concept expression), role assertions of the form $R(a, b)$, concrete domain predicate assertions of the form $P(x_1, \ldots, x_n)$ and concrete role assertions $u(a, x)$ where $P \in \Phi_D, a \in N_I, x_i \in \triangle_D$ and $u \in R_d$.

Given all of the above, a $\mathcal{SHIQ}(D)$ KB K is a triple (T ,R ,A) consisting of a TBox T , RBox R and ABox A .

Semantics of $\mathcal{SHIQ}(D)$

The semantics of SHIQ(D) is defined using an *interpretation* $\mathcal{I}$, which is a pair $\mathcal{I} = (\triangle, .^{\mathcal{I}})$, where $\triangle$ is a non-empty set, called the *domain* of the interpretation, disjoint from the concrete domain $\triangle_D$ and $.^{\mathcal{I}}$ is the *interpretation function*. The interpretation function assigns to each atomic concept A a subset of $\triangle$, to each role R a subset of of $\triangle \times \triangle$ and to each individual a an element of $\triangle$. Additionally, the interpretation function assigns to each concrete atomic role $u \in N_R^c$ a subset of $\triangle \times \triangle_D$, to each predicate $P \in \Phi^{\mathcal{D}}$ a subset of $\triangle_D$ and to each $x \in D$ an element of $\triangle_D$. The interpretation function is extended to complex roles and concepts is given in Table 2.2.2.2 (note that R is an abstract role, S is a simple abstract role, u is a concrete role and # denotes cardinality). The model-theoretic semantics of $\mathcal{SHIQ}(D)$ axioms is shown in Table 2.2.2.2.

$\top^I$	$=$	$\triangle$
$\bot^I$	$=$	$\emptyset$
$(\neg C)^I$	$=$	$\triangle \setminus C^I$
$(C \sqcap D)^I$	$=$	$C^I \wedge D^I$
$(C \sqcup D)^I$	$=$	$C^I \vee D^I$
$(\forall R.C)^I$	$=$	$\{x \mid \forall y : (x,y) \in R^I \rightarrow y \in C^I\}$
$(\exists R.C)^I$	$=$	$\{x \mid \exists y : (x,y) \in R^I \wedge y \in C^I\}$
$(\leq nR.C)^I$	$=$	$\{x \mid \#\{y \mid (x,y) \in R^I \wedge y \in C^I\} \leq n\}$
$(\geq nR.C)^I$	$=$	$\{x \mid \#\{y \mid (x,y) \in R^I \wedge y \in C^I\} \geq n\}$
$(Inv(R))^I$	$=$	$\{(a,b) \mid (b,a) \in R^I\}$
$(\forall u.P)^I$	$=$	$\{a \mid \forall x \in \triangle_D : (a,x) \in u^I \rightarrow x \in P^I\}$
$(\exists u.P)^I$	$=$	$\{a \mid \exists x \in \triangle_D : (a,x) \in u^I \wedge x \in P^I\}$
$(\leq nu.P)^I$	$=$	$\{a \mid \#\{x \in \triangle_D \mid (a,x) \in u^I \wedge x \in P^I\} \leq n\}$
$(\geq nu.P)^I$	$=$	$\{a \mid \#\{x \in \triangle_D \mid (a,x) \in u^I \wedge x \in P^I\} \geq n\}$

Table 2.1: Interpretations of $\mathcal{SHIQ}(D)$ Concepts and Roles

$C \sqsubseteq D$	$=$	$C^I \subseteq D^I$
$C \equiv D$	$=$	$C^I = D^I$
$R \sqsubseteq S$	$=$	$R^I \subseteq S^I$
$u \sqsubseteq v$	$=$	$u^I \subseteq v^I$
$Trans(R)$	$=$	$(R^I)^+ \subseteq R^I$
$C(a)$	$=$	$a^I \in C^I$
$R(a,b)$	$=$	$(a^I, b^I) \in C^I$
$u(a,x)$	$=$	$(a^I, x^I) \in u^I$
$a = b$	$=$	$a^I = b^I$
$a \neq b$	$=$	$a^I \neq b^I$

Table 2.2: Interpretations of $\mathcal{SHIQ}(D)$ Axioms

The interpretation $\mathcal{I}$ is a model of the RBox R (respectively of the TBox T and ABox A) if it satisfies all the axioms in R (respectively T and A). $\mathcal{I}$ is a model of $\mathsf{K} = (\mathsf{T}, \mathsf{R}, \mathsf{A})$, denoted by $\mathcal{I} \models \mathsf{K}$, iff $\mathcal{I}$ is a model of T , R and A .

A KB K is inconsistent if there is no possible model for it, i.e., there is no interpretation $\mathcal{I}$ that satisfies the semantics of all of the axioms in T ,R and A .

2.2.2.3 DL Reasoning Services

There are a few basic reasoning services in DL, which allow users to deduce implicit knowledge from explicitly represented knowledge:

- *Consistency Checking* – The process of ensuring that the knowledge base has a model (i.e., does not contain contradictory facts).
- *Concept Satisfiability* – Given a concept C, checking if C is satisfiable w.r.t KB K is the task of determining if there exists an interpretation $\mathcal{I}$ of K s.t. the interpretation of C (C^I) is non empty.
- *Subsumption* – Given concepts C and D, C is subsumed by D w.r.t K , denoted $\mathsf{K} \models C \sqsubseteq D$, if in all interpretations $\mathcal{I}$ of K , $C^I \subseteq D^I$.
- *Instance checking* – determines instance relationships: an individual i is an instance of concept C if in all interpretations $\mathcal{I}$ of K , $a^I \in C^I$.

It is important to note that all of the above services can be reduced to consistency checking. For example, if we want to check if a concept C is satisfiable w.r.t KB K , when we generate a new individual a s.t. $a^I \in C^I$, and check the consistency of the KB

$\mathsf{K} \cup \{C(a)\}$. If K is consistent, that means the interpretation of C is non-empty and C is satisfiable.

Reasoning in DL is usually done using systems based on tableau algorithms (see [24] for a survey). Tableau algorithms are already used for consistency checking in various optimized DL reasoners that are freely or commercially available, such as RACER [124], FACT++ [65] and Pellet [109].

2.3 Semantic Web

The Semantic Web is intended to be an extension of the current World Wide Web in which information on the Web is represented in a machine processable format with a well defined meaning (semantics). Representing the knowledge on the Web in such a manner provides a variety of benefits including ease of knowledge exchange and integration and machine-automated reasoning. A set of standardized knowledge representation languages (published as W3C recommendations) form the foundation of the Semantic Web and are structured as a layered stack. Following is a brief overview of the standardized Semantic Web languages:

- The Resource Description Framework (RDF [35]) is a fairly simple language to describe resources on the web and relations between them. The RDF model is based on the idea of making statements about resources in the form of triples: subject-predicate-object expressions. RDF is based on the same architectural principles that made the Web successful: it uses Universal Resource Identifiers (URI) to identify and link resources on the web. Unlike traditional URLs, however, RDF URIs can refer to any identifiable thing, including things that may not be directly retrievable on the web.

- RDF Schema [39] is a vocabulary for describing properties and classes of RDF resources. It is a more expressive modeling language than RDF, with the capability for expressing subclasses and subproperties, along with domain and range constraints on properties.

- OWL [45] adds even more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. ”exactly one”), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. OWL comes in three different flavors with increasing expressivity: OWL Lite, OWL DL and OWL Full. OWL DL and OWL Lite build on the research tradition of Description Logics (they are firmly grounded in DL), so they are both decidable with well studied decision procedures.

Since OWL is based on the successful architecture of the Web, it is designed to be open, scalable and distributed which makes it suitable for a web policy representation language. Its key properties as an ontology language include the use of the URI (Universal Resource Identifier) as the unique identifier for named OWL classes, properties and individuals, and the ability to freely link and/or import ontology models using URIs directly. In addition, OWL assumes open-world semantics, which makes it different from closed-world database-schema languages in that information which is not explicitly asserted in the OWL KB is assumed to be unknown or missing instead of non-existent or false.

Chapter 3

Related Work

In this chapter, an overview of related work in access control languages, access models and verification approaches is presented. First, in Section 3.1 a discussion of the most common access control models (DAC, MAC and RBAC) is presented, followed by an overview of research on access policy languages (Section 3.2). In Section 3.3, I discuss related work in formal verification of access control languages (including XACML), using both state-based and logic-based techniques.

3.1 Access Control Models

For the purpose of this thesis, a general understanding of basic access control models is necessary, so this section contains an overview. The three most widely recognized access control models are the following:

Mandatory Access Control (MAC) In MAC [8], the access policy is enforced independently of users' preferences – in other words, even though a user might be an owner of a resource, he is not allowed to specify an access policy for that resource. MAC is used in systems that process highly sensitive information (e.g., military). One of the most widely used MAC systems is the *Multi-Level Security* (MLS) policy, i.e., the Bell-LaPadula security model [28]. The Bell-LaPadula model represents a set of simple access control rules which use security *labels* on objects and *clearances* for subjects. These labels form a lattice; for example, a well known set of labels is $unclassified \leq confidential \leq secret \leq top\text{-}secret$. Given these labels, there are only two access rules in the model:

- The *Simple Security Property* states that a subject at a given security level may not read an object at a higher security level (no **read-up**).
- The **-property* (star-property) states that a subject at a given security level is not allowed to write to any object at a lower security level (no **write-down**). This property is used to prevent users or programs from declassifying sensitive information.

Because of its inflexible nature (only two fixed access rules), MAC has not been deemed expressive enough for commercial purposes.

Discretionary Access Control (DAC) In DAC [9], the access policy about an object is determined solely by the object's owner. An example of this is file ownership in the Unix file system: every file in the system has an owner, and the file's initial owner can specify its access privileges for other users. Additionally, in DAC users can also delegate control over objects to other users. Discretionary Access Control has been widely used to enforce access policies in operating systems. The most common types of DAC implementations are discussed below.

An *Access control matrix* is a a two-dimensional matrix representing users on the rows and objects on the columns. Each entry in the matrix represents the access type held by that user to that object. Since access control matrices are usually sparsely populated, there exist other, more storage-efficient mechanisms such as *access control lists* (ACL) and *capability-based* matrices. ACLs represent each column of the matrix as a list, hence each object is associated with the set of authorized users. An ACL-like mechanism is used in the UNIX file system, where each file is associated with access rights for its owner, group, and everyone else in the system. On the other hand, capability-based systems store the access control matrix by rows. In such systems, each subject is associated with a list of objects he is authorized to access.

While more flexible than MAC, Discretionary Access Control suffers from some drawbacks as well. In particular, managing systems with a large number of subjects of objects can be very time-consuming. For example, removing a user (i.e., subject) from the system involves traversing and removing the subject from the ACL of each object.

Role Based Access Control RBAC [10] is a newer alternative to MAC and DAC, expressive enough to cover both. In RBAC, *roles* are created for various job functions. Each role can have a number of *permissions* assigned to it; these permissions refer to certain operations needed for the particular function. Users are then assigned particular roles, and through these assignments acquire the permissions needed to perform the job functions. RBAC differs from ACLs in that it assigns permissions to specific operations with meaning in the organization, rather than to low level data objects.

There has been a lot of interest in developing RBAC models in the past years [116, 107, 106, 125, 100], however the consensus is that $RBAC_0$ [116] has emerged as the core RBAC model (it is also being standardized by NIST [10]). In $RBAC_0$, the key components are sets of users (U), roles (R) and permissions (P). The policy is then specified by a user assignment relation, associating users to roles they hold, and a permission assignment relation linking roles to sets of permissions. Additionally, roles have to be activated within sessions. A user can activate multiple roles in a single session; she can also act in multiple sessions at the same time.

There have been a number of extensions to $RBAC_0$, the most important being *role hierarchies* ($RBAC_1$) and *constraints* ($RBAC_2$). Role hierarchies allow a role to inherit all of the access privileges of another role. For example, one can state that role *Manager* is senior to *Intern*, so a user activating the *Manager* role will inherit all of the permissions of *Intern*.

$RBAC_2$ supports constraints, which essentially impose restrictions on acceptable configurations of the different RBAC components. A common motivation for constraints is the example of mutually disjoint roles, such as purchasing manager and accounts manager. In most organizations, the same individual will not be permitted to be a member of both roles, because this creates a possibility for committing fraud. This is the well-known security principle of *separation of duties*, and is available as a constraint in $RBAC_2$. Additionally, it is possible to limit the number of roles which an individual user can activate (using a *cardinality* constraint).

Since XACML subsumes $RBAC_2$ in terms of expressiveness [17], all of the XAML analysis services presented in this dissertation apply to RBAC implementations as well.

3.2 Access Control Languages

This section contains an overview of related work in logic-based security languages as well policy languages based on Semantic Web techniques. The goal is to compare my logic-based, semantic web-enabled policy framework with previous work in the area.

3.2.1 Logic-Based Languages

The languages discussed in this section all benefit from having unambiguous semantics and well understood computational properties. Most of them are based on Datalog, so evaluating whether a request satisfies the policy is done in PTIME. Some of the proposals [72] even materialize the unique model of the underlying Datalog program to avoid doing any reasoning at runtime.

3.2.1.1 Centralized Policies

Centralized access policies refers to security systems with only one PDP (Policy Decision Point). Note that this still allows the policies to be distributed – however, during deployment they have to be retrieved and evaluated at a single PDP. There are many logic-based policy frameworks that fit this description [30, 71, 73, 32]. This section presents two representative approaches: the work by Woo and Lam [128], which is one of the earliest attempts at a logic-based authorization framework, and the Flexible Authorization Framework (FAF) [71] which represents one of the most influential policy frameworks.

Authorization using Default Logic (Woo and Lam) One of the earliest attempts at a general, logic-based framework for expressing authorizations was made by Woo and Lam [128], who proposed the use of default logic to model authorization and control rules. Default logic is a very expressive framework, allowing the authors to cover both open and closed policy bases with their languages, as well as the Bell-LaPadula model.

To overcome the complexity drawbacks of default logic (it's undecidable in the general case), the authors used a subset of the logic called *extended logic* programs (ELP) as the basis for their framework. ELPs are essentially a class of stratified logic programs with both classical negation and negation as failure, but with a unique minimal model that can be computed in quadratic time. The authors, however, did not address the question of how this restriction will affect the expressiveness of their policies. Additionally, they did not provide any analysis services such as formal verification of policies.

Flexible Authorization Framework (FAF) Jajodia et al.[71] proposed a logical language for specification of authorizations that allows users to specify, together with the authorizations, the policy according to which access control decisions are to be made. Policies are expressed by means of rules which enforce derivation of authorizations, conflict resolution, access control, and integrity constraint checking.

The architecture of FAF consists of the following components:

- A *history* table whose rows describe the access requests processed.
- An *authorization* table whose rows are composed of authorization triples `(o,s,+a)` and `(o,s,-a)`. Informally, `(o,s,+a)` is a positive authorization triple – it means that subject s can perform action a on object o; `(o,s,-a)` forbids the action.
- A *propagation* policy that specifies how to derive authorizations from the explicit authorization table above. In particular, FAF supports the following propagation policies: no propagation, no overriding, most specific overrides and path overrides.
- A *conflict resolution and decision* policy that specifies how to override conflicts

when one or two conflicting authorizations apply to a given authorization triple. Examples of conflict resolution policies supported by FAF include: *no-conflict* (conflicts are considered errors), *denials-take-precedence*, *permissions-take-precedence* and *nothing-takes-precedence* (conflicts remain unsolved). The decision policy also determines the system's final response to every access request. For example, the decision policy can force a deny decision whenever conflicts occur, or can force a decision to fill in the gaps in the absence of any access decision for a particular access request.

- A set of *integrity constraints* that impose restrictions on the content of the individual components in FAF.

The authorization table in FAF is viewed as a database. The propagation, conflict resolution and decision policies are expressed using stratified logic programs, guaranteeing that the overall policy system has a unique stable model. As a result, FAF corresponds to a quadratic time data complexity fragment of logic programming. To improve scalability of the framework, the authors proposed a materialization technique that allows for incremental updates of the unique stable model at run-time.

One of the contributions of my work is a detailed comparison between XACML and FAF, provided in Chapter 5.

3.2.1.2 Distributed Policies

By distributed access policies I refer to policies that exist in a distributed environment such as the Web and security systems with multiple PDPs. Such decentralized environments have the following distinctive characteristics:

- Identity of all possible access requesters cannot be known before hand, so subjects present digital credentials (which may be distributed themselves) in order to gain access.
- In addition to the access policy on the server, the access subjects (clients) might have policies for their own data as well. In such case, there exists a need for *trust negotiation* to determine if client's and server's policies are compatible.
- *Delegation* is essential to distributed PDPs. Depending on the object *o* being accessed, a particular security point might not be qualified to make an access decision about *o*; in such cases, the access request is delegated to a security point with the system that has authority over objects of type *o*.

In the following, a survey of logic-based approaches to distributed policy authorization is presented.

Delegation Logic [88] combines the following features: it is based on logic programs, can express delegation depth explicitly and supports a wide variety of complex delegation principles (e.g., k-out-of-n threshold). In addition, Delegation Logic provides a concept of proof-of-compliance that is based on model-theoretic semantics. The framework is based on a restricted class of logic programming called Ordinary Logic Programs (OLP [56]), and a transformation is presented from OLP to positive Datalog programs. However, the transformation is exponential in the number of logical variables used in the policy, and no evaluation is presented, so it is unclear whether the approach scales.

Additionally, Delegation Logic can be extended with non-monotonicity, negation and prioritized conflict handling; however, severe restrictions on the usage of the delegation constructs are placed to ensure tractability [85].

PeerAccess [127] is a trust negotiation framework for reasoning about authorization in open distributed systems. It supports a declarative description of the behavior of peers that selectively push and/or pull information from certain other peers. PeerAccess local knowledge bases encode the basic knowledge of each peer, its policies governing the release of each possible piece of information to other peers.PeerAccess proofs of authorization are verifiable and nonrepudiable, and their construction relies only on the local information possessed by peers and their parameterized behavior with respect to query answering, information push/pull and information release policies.

In addition to PeerAccess and Delegation Logic, there are other Datalog-based languages that support delegation; SecPal [27], Binder [46], SD3 [75], RT [90] and Cassandra [101] all use Datalog as basis for syntax and semantics. (Cassandra and RTC are based on Datalog with constraints for higher flexibility.)

In contrast to the above approaches, Proof-carrying Authorization (PCA) and related distributed proof systems [26] are an authorization framework based on a higher-order logic where different domains in the system use different, less expressive, application-specific logics. The higher-order logic (AF logic) used to check the proofs is undecidable, though this problem is avoided by forcing clients to generate proofs on their own, using only a decidable subset of AF logic. Consequently, the authorizing servers task of proof-checking is reduced to a tractable type-checking problem - however this leads to large rate of increase of sizes of the client proof.

3.2.1.3 Dynamic/Temporal Access Policy Languages

Most of previous work that was discussed dealt with *static authorization*, i.e., access policies where the authorizations do not change over time and do not have any temporal dependencies. This section surveys approaches that deal with dynamic policies. First, I discuss a seminal approach that supports evolving subjects, resources and authorizations, and then discuss a more recent policy language extension of RBAC that supports temporally dependent authorizations.

The framework of Harrison, Ruzzo and Ullman [61] is one of the earliest approaches that allows for changing number of subjects, roles, resources, and authorizations. The HRU model is very expressive; it could model most of the protection systems in use at that time when it was proposed. However, because of the expressiveness, there is no algorithm to decide if a given subject can *eventually* obtain an access privilege to a given object (it is undecidable).

Bertino et al [29] presented a temporal extension of the RBAC model called TRBAC. TRBAC supports periodic role enabling and disabling—possibly with individual exceptions for particular users—and temporal dependencies among such actions, expressed by means of role triggers. Role trigger actions may be either immediately executed, or deferred by an explicitly specified amount of time. Enabling and disabling actions may be given a priority, which is used to solve conflicting actions. A formal semantics for the specification language was provided, and a polynomial safeness check was introduced to reject ambiguous or inconsistent specifications. The authors also presented an implementation of TRBAC on top of a conventional DBMS.

3.2.2 Semantic Web-Based languages

Recently there has been a great amount of attention to how Semantic Web technologies can be used in policy systems. In particular, there have been a number of proposals that show how to ground or express policies in a Semantic Web framework [126, 77, 78, 122].

Rei [77] is a policy specification language based on a combination of OWL-Lite, logic-like variables and rules. It allows users to develop declarative policies over domain specific ontologies in RDF and OWL. Rei allows policies to be specified as constraints over allowable and obligated actions on resources in the environment. A distinguishing feature of Rei is that it includes specifications for speech acts for remote policy management and policy analysis specifications like what-if analysis and use-case management.

The successor of Rei is Rein [78], which is a policy framework grounded in semantic web technologies that allows for different policy languages and supports heterogeneous policy systems. Rein provides an ontology for describing policy domains in a decentralized manner and provides a reasoning engine built on top of CWM, an N3 rules reasoner. Using Rein and CWM, the authors showed how it is possible to develop domain and policy language specific security systems. Rein has been successfully used as a policy management system in the Policy Aware Web project [126], which in turn provides an architecture for scalable, discretionary, rule-based access control in open and distributed environments.

PeerTrust [52] deals with discretionary access control on the web using semantic web technologies. It provides a mechanism for gaining access to secure information/services on the web by using semantic annotations, policies and automated trust negotiation. In PeerTrust, trust is established incrementally through an iterative process which involves gradually disclosing credentials and requests for credentials. PeerTrust's policy language for expressing access control policies is based on definite Horn clauses. A distinguishing feature of PeerTrust is that it expects both parties to exchange credentials in order to trust each other and assumes that policies are private, which is appropriate for critical resources such as military applications and e-commerce sites.

Finally, KaOS Policy and Domain Services [122] use ontology concepts encoded in OWL to build policies. These policies constrain allowable actions performed by actors which may be clients or agents. The KAoS Policy Service distinguishes between authorizations and obligations. The applicability of the policy is defined by a class of situations which definition can contain components specifying required history, state and currently undertaken action.

3.2.3 Discussion

One of the goals of this section was to demonstrate that recently there has been a great amount of interest in logic-based policy languages, and that most of these languages are based on some variant of logic programming (most often, Datalog). Given that, it is surprising that a standardized and widely distributed language such as XACML has not been formally compared to these languages. The main impediment for this was the lack of formal, logic-based semantics for the language. This problem is addressed in Chapter 5, where I show that the core of XACML can be embedded in stratified Datalog.

Additionally, note that most of the policy languages discussed in this section address the challenges that a policy system should overcome to be usable in a massively open and distributed setting. Thus, they mostly discuss the architectural, privacy and scalability aspects of a policy framework. Much less attention is being paid to security analysis issues such as formal verification of policies against security properties, change analysis of different policies and coverage checking which are important to ensure the access policy has no bugs. Most of the above approaches base their semantics in Datalog, in order to maintain balance between policy expressiveness and computational complexity. However, as shown in [31], in Datalog-based models it is very hard (actually undecidable) to provide change analysis, i.e., for two policies expressed as Datalog programs to determine if they would always return the same access decision for any potential access request. To provide such analysis services, instead of logic programming I use a different family of logics, called Description Logics – more information is presented in Chapter 6.

3.3 Policy Analysis and Verification

The policy languages surveyed in the previous section usually offer two basic services: 1) checking the *consistency* of an access policy set, i.e., determining if there are any conflicting policies in the system, and 2) for a given request R and a policy P determining the access decision of P for R. However, recently, there has been a great amount of interest into other types of analysis services for policies [92, 91, 14, 83, 50, 66, 131, 117, 50, 36], the most common being verification of a policy against given safety properties. For example, as a part of a company-wide access policy, one could state 'Junior developers should never be allowed to sign expense reports' or 'At any given time, a user cannot activate more than one of the following three roles {JuniorDeveloper, SeniorDeveloper, AccountsClerk}'. Then, company security officers would use automated tools to *verify* that these constraints will not be violated against *all* possible access requests. In the event violations are discovered and the policy is updated, *change analysis* can be performed to ensure no new bugs were unintentionally introduced – for example, using queries of the form 'Show me all requests involving Expense Reports that used to map to Deny but now are mapping to Permit'. The above mentioned services of verification and change analysis have been proposed as the building block of a useful policy analysis tool ([50]).

This section surveys previous work on such analysis services for access control policies. First, I provide an overview of security policy analysis (using state-based or logic-based approaches), then in Section 3.3.1 I discuss approaches that embed existing policy languages into a logic, thus providing analysis services previously not available. Finally, in Section 3.3.2 I survey such logical embeddings of XACML itself.

Elisa Bertino et al [31] proposed a formal framework for reasoning about different access control models. Their framework is logic-based and can capture DAC, MAC and RBAC models. Each instance of the proposed framework corresponds to a Datalog program, interpreted according to the stable model semantics. To demonstrate its expressiveness, the authors mapped the Bell and LaPadula model [28] and NIST RBAC [116] to the framework. They also proposed some parameters (along with decidability results) along which access control models can be compared. For example, they showed that checking for structural subsumption/equivalence between different access control models is decidable, however access request equivalence is not. The difference with our work is the language we analyze (XACML) and the services we provide (e.g., we show that access request equivalence is decidable for XACML).

Chomicki and Lobo [38] introduced a declarative policy description language called $\mathcal{PDL}$ in which policies are described as sets of event-condition-action (ECA) rules. They provided a framework for detecting and resolving conflicts between the ECA rules and any action constraints. This is performed using a policy monitor, which, in order to resolve conflicts chooses or ignores certain events, essentially preventing the ECA rule from activating and causing the conflict. The semantics of the ECA rules and conflict detection and resolution are defined using logic programs. Unlike XACML, where conflict resolution is built in the language, in their framework whenever a conflict occurs the policy engine has to generate a minimal set of actions to be removed in order to remain consistent.

Dougherty et al. [48] presented a model for formal analysis of access-control policies in dynamic environments, taking into account the possible interactions of the policies with their environments. For this model, they proposed two analysis services: a) *goal reachability*, which checks if there is some accessible state in the dynamic access model which satisfies some boolean expression over the policy facts and b) *contextual policy containment*, which essentially checks if one policy is more permissive than another. These services are provided using a combination of relational and temporal reasoning. Policies that change their environment are not addressed in my thesis; on the other hand, we perform static analysis services on a richer policy language ([48] does not support concrete domains and ontology-based policy models).

Lithium [59] is a language for reasoning about digital rights and is based on a fragment of first order logic. It is different from Datalog-based approaches since it allows full negation in the conclusion as well as in the premises of policy rules. To show its expressiveness, the authors gathered a large collection of policies from different types of libraries and mapped them to Lithium. They also showed how large fragments of XrML [60] and ODRL [112] can be translated in the language. Two core analysis services are provided:

- Given a set of policies, a policy environment and an access request, does it follow that the access request is permitted by the policy set?
- Consistency checking of a policy set. Unlike [71], the language does not support conflict resolution mechanisms, so whenever both a permit and deny is returned by a policy, it is treated as an error.

In order to remain decidable, Lithium restricts recursion and cannot easily express delegation. Unlike XACML, Lithium does not provide any conflict resolution mechanisms, and it does not support change analysis and coverage checking of policies.

There has also been research on security analysis for access control [92, 91] that uses the notions of *states* of policy systems and *transitions* (for example, adding a role, or changing a permission) that alter those states. Then, usually a set of queries is proposed that investigates the possible consequences of certain changes in the policy. Simple safety checking is an example of a query; it checks if there exists a reachable state in which a (presumably untrusted) principal has access to a resource. Although early results showed that this type of safety analysis can easily lead to undecidability [61], there has been recent work that demonstrates a class of access control models and queries for which safety is decidable and efficient algorithms exist [92]. Unfortunately, there are limitations to the expressiveness of the models that are analyzed: the states are described using positive Datalog programs, so there is no support for classical negation. No classical negation in turn implies no support for negative authorizations and common constraints such as mutually exclusive roles.

As another state-based approach, Schaad et al. [117] examineed the problem of verifying a policy that is subject to change coming from another policy. Using the Alloy [70] specification language and its model-checking facilities, they showed how to specify an RBAC96-style model, ARBAC97-style extensions and a set of separation of duty properties. There were no implementation or evaluation results given, so it is difficult to compare with our approach.

In [131] the authors presented a model-checking algorithm which can be used to evaluate access control policies, and a tool which implements it. Their tool provides *reachability* analysis: not only checks whether the policies give legitimate users enough permissions to reach their goals, but also whether the policies prevent intruders from reaching their malicious goals. Policies of the access control system and goals of agents are described in the language *RW* [57]. RW and the analysis framework presented in this thesis provide a complementary set of services: theirs is reachability analysis in presence of rule interactions, cooperating agents and multi-step actions, while in this thesis the services provided are change analysis, coverage checking and formal verification for a static policy system.

There are also proposals for analyzing policies based on description [132] or modal logics [99]. Both of these provide a formalization of RBAC, and show how tableau-based decision methods can be used for consistency checking of policies, evaluating access requests and verifying policies against security properties. Zhao et al [132] presented a formalization of RBAC based on the description logic $\mathcal{ALCQ}$. They also showed how RBAC policy constraints (separation of duty, role hierarchies) can be captured in this logic. Massacci [99] formalized RBAC using multi modal logic and presented a decision method based on analytic tableaux. Because tableau-based algorithms are used , services similar to ours are provided: logical consequence, model generation and consistency checking of policies. The analysis framework in my thesis is for a more expressive language (e.g., neither of the approaches supports conflict resolution algorithms or concrete domains).

In [14], the authors proposed a set of services under the name of policy *ratification*. In particular, they presented algorithms for analysis tasks such as dominance, coverage and consistency check that can be performed independently of policy model and language and require little domain-specific knowledge. They presented algorithms from constraint, linear, and logic programming disciplines to help perform ratification tasks. Also, an algorithm is provided to efficiently assign priorities to the policies based on relative policy preferences indicated by policy administrators. Finally, they present how these algorithms have been integrated with a working policy system to provide feedback to a policy administrator regarding potential interactions of policies. The techniques the authors use for ratification are similar to the algorithms used for datatype reasoning in this thesis. However, our work can be considered an extension of theirs since it covers a more expressive policy language (XACML).

3.3.1 Embedding Policy Languages in Logic

As mentioned in the previous section, the authors of Lithium showed that large fragments of ODRL [112] and XrML [60] can be translated to their FOL-based language. In addition to giving ODRL formal semantics, the authors considered the practical problem of determining whether a set of ODRL statements imply a permission or prohibition. Using their semantics, they formally defined the problem and showed that it is NP-hard. They also showed that by removing a component of ODRL whose meaning seems to be somewhat unclear, a tractable fragment of the language results. They proved that the fragment is tractable by creating a polynomial-time algorithm to determine whether a set of ODRL statements imply a permission (or prohibition). They presented a similar contribution for XrML in [60]: propose a formal semantics, show that deciding access requests in the language is NP-hard, then show an expressive fragment of the language for which deciding access requests is polynomial. The relationship between XrML and ODRL on one hand and XACML on the other is unclear; however we discussed the differences between Lithium (the language used to formalize XrML and ODRL) and our formal framework in the previous section.

In [89] the authors presented a first-order logic (FOL) semantics for the Simple Distributed Security Infrastructure (SDSI [115]). The authors proved that the FOL semantics is equivalent to the string rewriting semantics used by SDSI designers, for all queries associated with the rewriting semantics. Using their semantics, they discovered a few problems in the proof procedures for SDSI. Finally, they compared SDSI with $\mathrm{RT}_1^{\mathrm{C}}$, a datalog-based language that is part of the RT policy framework [90]. The authors did not discuss formal verification or change analysis for their language.

3.3.2 XACML Analysis and Verification

In this section I will discuss approaches that formalize and analyze fragments of XACML [66, 131, 117, 50, 121, 36, 40] and are most related to my dissertation.

Hughes et al. [66] proposed a framework for automated verification of XACML policies based on relational First-Order Logic. They introduced a formal model for systematically specifying access to resources, and showed that XACML policies can be translated to a simple form which partitions the input domain to four classes: permit, deny, error, and notapplicable. The authors showed how to automatically verify policies using an existing automated analysis tool, Alloy [70]. Because using the first-order constructs of Alloy to model XACML policies is prohibitively expensive (in terms of performance), the authors used only the propositional constructs. The limitation of their approach is that they do not fully support data-types, policy vocabularies and delegation policies. I have provided a thorough empirical comparison of my prototype analyzer against their XACML analysis tool; results are discussed in Chapter 8.

Bryans et al. [36] formalized XACML policies using a process algebra known as Communicating Sequential Processes (CSP [63]). This allows them to use model checkers such as FDR for formally verifying properties of policies and for comparing access control policies. In addition, the authors showed how limited workflows can also be mapped to CSP. The workflow is sequential in nature and in that sense their approach is more expressive than first-order logic approaches. The authors provide no information on any empirical results or prototype implementation, so I have not performed an comparison with my XACML reasoner.

In [50], the authors expressed XACML policies using Multi-Terminal Binary Decision Diagrams (MTBDDs). MTBDDs [51] are a more general version of Binary Decision Diagrams, that maps bit vectors over a set of variables to a finite set of results. In [50], variables in the decision diagram are used to represent attribute/value pairs (such as role=Student, action=View, etc.) and the policy results (Deny, Permit, Indeterminate) are mapped to diagram terminals. The approach from their paper is implemented in Margrave, a tool for analyzing XACML policies. Margrave provides verification and comprehensive change-impact analysis support based on the semantic differences between the MTBDDs representing the policies. Compared to Margrave, our analysis framework covers a richer subset of XACML (administrative policies, datatypes, ontology-based policy models). In addition, for the subset that both Margrave and our tool support, I provide a detailed performance comparison of the tools in Chapter 8.

Finally, in [42] the authors extended the work by Fisler et al. [50] by adding more expressiveness to the language being analyzed and supporting additional analysis services. Their tool, called EXAM (comprehensive framework for analysis of access control policies), in addition to supporting core XACML defined in [50] also supports datatype domains. One of the components in the framework is a policy similarity analyzer [93] which is used to filter out policies with low similarity score. While EXAM provides datatype support, it does not explicitly address ontology-based policy models or delegation policies.

.

Chapter 4

Operational Semantics of XACML

In this chapter, a formal semantics for XACML v3.0 is presented. The semantics is provided through a concise set of rules that capture the meaning of the XACML constructs as presented in the official specification. This semantics is used to prove the correctness of the Datalog and Description Logic mappings presented in Chapters 5 and 6.

4.1 Syntax

To avoid the verbose XML representation of XACML, a lisp-like syntax is used, similarly to [121]. A typewriter font denotes names of syntax elements as they occur in the XACML specification. Thus, `Policy` refers to the XACML syntactic element that contains `Rules`, whereas a policy can refer to a set of `Policy` or `PolicySet` elements. Also, terminal nodes in the syntax grammar below are denoted by a lower case starting letter.

The syntax for XACML `Policies`, `PolicySets` and `Rules` is shown in Table 4.1. Table 4.2 presents the syntax of `Targets` and matching functions.

Table 4.3 shows the syntax of `Condition` elements in `Rules`. Note that arbitrary nesting of functions is allowed in the `Condition` element. I only show a few of the supported functions in the table; a full listing can be found in the XACML 3.0 Specification [113]. Finally, Table 4.4 contains the syntax for XACML access requests.

S	::=	(`PolicySet` Comb T S^* id)
	|	(`Policy` Comb T R^* id)
R	::=	(`Rule` Cond T Effect)
Comb	::=	`permit-Overrides`
	|	`deny-Overrides`
	|	`first-Applicable`
	|	`only-One-Applicable`
Effect	::=	`Permit` | `Deny`

Table 4.1: Syntax of Policy Elements.

T	::=	(`Target` DM^*)
DM	::=	(`DisjunctiveMatch` CM^+)
CM	::=	(`ConjunctiveMatch` M^+)
M	::=	(`Match` AV AD MatchFcn)
	|	(`Match` AV AS MatchFcn)
AD	::=	(`AttributeDesignator` cat attr-ID Type $\text{issuer}^?$ $\text{mustBePresent}^?$)
AS	::=	(`AttributeSelector` contextPath Type $\text{present}^?$)
AV	::=	(`AttributeValue` value Type)
MatchFcn	::=	type-equal | type-greater-than | type-greater-than-or-equal| type-less-than | type-less-than-or-equal | type-regexp-match
Type	::=	string | boolean | integer | ...

Table 4.2: Syntax of `Targets` and Matching Functions.

Example 4.1.0.1 This example illustrates how a XACML `Rule` element in normative XML syntax is translated to my abbreviated lisp-like representation.

```
<Rule RuleId= "example:SimpleRule1" Effect="Permit">
<Target>
 <DisjunctiveMatch>
  <ConjunctiveMatch>
    <Match MatchId="...:rfc822Name-match">
      <AttributeValue DataType="...#string">
        med.example.com
      </AttributeValue>
      <AttributeDesignator
        Category="...subject-category:access-subject"
        AttributeId="...:subject:subject-id"
        DataType="...:rfc822Name"/>
    </Match>
   </ConjunctiveMatch>
```

Cond	::=	(`Condition` Expr*)
Expr	::=	`Apply` | AS | AV
	|	`Function` | `VariableReference` | AD
`Apply`	::=	(Fcn-ID Expr*)
`Function`	::=	(Fcn-ID Expr*)
`Fcn-ID`	::=	any-of | all-of |regexp-match . . .

Table 4.3: Syntax of `Condition` element.

RQ	::=	(`Request` ATS*)
ATS	::=	(`Attributes` (AT content)* cat)
AT	::=	(`Attribute` AV* attr-ID issuer)
ARQ	::=	(`Request` Delegated Del-info Delegate ATS*)
Delegate	::=	((AT content)* delegate)
Del-info	::=	((AT content)* del-info)
Delegated	::=	((AT content)* delegated)

Table 4.4: Syntax of access requests. ARQ represents an administrative request (a special case of access requests).

```
  </DisjunctiveMatch>
 </Target>
</Rule>
```

The example `Rule` is abbreviated as:

```
(Rule SimpleRule1 ()
  (Target
   (DisjunctiveMatch
     (ConjunctiveMatch
      (Match
       (AttributeValue med.example.com string)
       (AttributeDesignator access-subject subject-id rfc822Name ()())
       rfc822Name-match ))))
  Permit)
```

4.2 Proof-theoretic Semantics

In this section, a proof-theoretic semantics of XACML is presented using natural deduction rules. I use the following notation: for a syntactic element P, X_P refers to a child element of P that is of type X. For example, for a `Policy` element P, T_P refers to its `Target`.

4.2.1 Matching Functions

XACML is an attribute-based language, so in its most basic form it matches attribute values from a request with policy `Targets`. This section (specifically Tables 4.5 and 4.6) presents the inference rules that determine if a `Request` matches a `Target`.

Table 4.5 contains the inference rules that determine how an `AttributeDesignator` or `AttributeSelector` element in a policy selects an attribute value from a `Request`. In the case of an `AttributeDesignator`, the value will be selected only if the attribute id, category and datatype all match (Rules 1 and 2). For `AttributeSelectors`, an `AttributeValue` is selected by evaluating the xpath function in the selector against the `Request` (Rule 3). Rules 4 and 5 are for cases when no `AttributeValue` is selected, yet `mustBePresent` is set to true; in those cases, `Indeterminate` is returned.

Table 4.6 contains the inference rules that determine how a `Request` is matched against a `Target`. Rule 1 matches a selected `AttributeValue` from the `Request` against the comparison function in the `Match` element. If the comparison function is true, the `Request` RQ matches the `Match` element M. The rest of the rules in Table 4.6 show how this match can be propagated through `ConjunctiveMatch` and `DisjunctiveMatch` to `Target` elements.

In Table 4.7, the semantics of matching a `Request` against a `Rule` are presented. Notice that in Rule 1, the rule firing depends on the evaluation of its `Condition` element. Note that in XACML v3.0, a total of 237 different functions are allowed in the `Condition`. The semantics of these functions has already been covered in [68], so I omit their discussion here.

If a syntax error occurs during matching, in the specification `Indeterminate` is returned as result. Since this chapter focuses on the semantic properties of XACML, it is assumed that the policy and request in question are syntactically correct. Observe that even with a syntactically correct policy set, an `Indeterminate` could be returned – e.g., if the *mustBePresent* attribute of an `AttributeDesignator` is true and there is no attribute in the request that matches it (Rule 3 in 4.5).

RULE 1
$$\frac{\begin{array}{c}\text{attr-id}_{AD} = \text{attr-id}_{AT}\\ \text{type}_{AV_{AT}} = \text{type}_{AD}\\ \forall \text{issuer}_{AD}, \text{issuer}_{AT} : \text{issuer}_{AD} = \text{issuer}_{AT}\end{array}}{\text{AD}, \text{AT} \models \text{AV}_{AT}}$$

RULE 2
$$\frac{\begin{array}{c}\exists \text{ATS} \in \text{RQ}, \text{AT} \in \text{ATS} :\\ \text{cat}_{AD} = \text{cat}_{ATS} \qquad \text{AD}, \text{AT} \models \text{AV}_{AT}\end{array}}{\text{AD}, \text{RQ} \models \text{AV}_{AT}}$$

RULE 3
$$\frac{\text{Value} = \text{xpath-select}(\text{RQ}, \text{contextPath}_{AS})}{\text{AS}, \text{RQ} \models \text{Value}}$$

RULE 4
$$\frac{\begin{array}{c}\forall \text{ATS} \in \text{RQ}, \forall \text{AT} \in \text{ATS} : \text{AD}, \text{RQ} \not\models \text{AV}_{AT}\\ \text{mustBePresent}_{AD} = \texttt{True}\end{array}}{\text{AD}, \text{RQ} \models \texttt{Indeterminate}}$$

RULE 5
$$\frac{\text{AS}, \text{RQ} \not\models \text{Value} \qquad \text{mustBePresent}_{AS} = \texttt{True}}{\text{AS}, \text{RQ} \models \texttt{Indeterminate}}$$

Table 4.5: Matching `AttributeDesignator` and `AttributeSelector` with attributes in `Request`.

RULE 1

$$\frac{\begin{array}{c}\exists \mathrm{AD} \in \mathrm{M}: \\ \mathrm{AD}, \mathrm{RQ} \models \mathrm{AV}_{AT} \\ \mathrm{fcn}_M(\mathrm{AV}_M, \mathrm{AV}_{AT}) = \texttt{True}\end{array}}{\mathrm{M}, \mathrm{RQ} \models \texttt{True}}$$

RULE 2

$$\frac{\exists \mathrm{AD} \in \mathrm{M} : \mathrm{AD}, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{M}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 3

$$\frac{\begin{array}{c}\exists \mathrm{AS} \in \mathrm{M}: \\ \mathrm{AS}, \mathrm{RQ} \models \mathrm{AV}_{AT} \\ \mathrm{fcn}_M(\mathrm{AV}_M, \mathrm{AV}_{AT}) = \texttt{True}\end{array}}{\mathrm{M}, \mathrm{RQ} \models \texttt{True}}$$

RULE 4

$$\frac{\exists \mathrm{AS} \in \mathrm{M} : \mathrm{AS}, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{M}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 5

$$\frac{\forall \mathrm{M}_{CM} : \mathrm{M}_{CM}, \mathrm{RQ} \models \texttt{True}}{\mathrm{CM}, \mathrm{RQ} \models \texttt{True}}$$

RULE 6

$$\frac{\forall \mathrm{M}_{CM} : \mathrm{M}_{CM}, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{CM}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 7

$$\frac{\exists \mathrm{CM}_{DM} : \mathrm{CM}_{DM}, \mathrm{RQ} \models \texttt{True}}{\mathrm{DM}, \mathrm{RQ} \models \texttt{True}}$$

RULE 8

$$\frac{\forall \mathrm{CM}_{DM} : \mathrm{CM}_{DM}, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{DM}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 9

$$\frac{\forall \mathrm{DM}_{T} : \mathrm{DM}_{T}, \mathrm{RQ} \models \texttt{True}}{\mathrm{T}, \mathrm{RQ} \models \texttt{True}}$$

RULE 10

$$\frac{\exists \mathrm{DM}_{T} : \mathrm{DM}_{T}, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{T}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

Table 4.6: Matching a `Request`(RQ) with a `Target`(T).

RULE 1

$$\frac{\mathrm{T}_R, \mathrm{RQ} \models \texttt{True} \qquad \mathrm{Cond}_R, \mathrm{RQ} \models \texttt{True}}{\mathrm{R}, \mathrm{RQ} \models \mathrm{Effect}_R}$$

RULE 2

$$\frac{\mathrm{T}_R, \mathrm{RQ} \models \texttt{Indeterminate}}{\mathrm{R}, \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 3

$$\frac{\mathrm{R}, \mathrm{RQ} \not\models \texttt{Effect}_R \qquad \mathrm{R}, \mathrm{RQ} \not\models \texttt{Indeterminate}}{\mathrm{R}, \mathrm{RQ} \models \texttt{NotApplicable}}$$

Table 4.7: Evaluating a `Rule` against a Request.

4.2.2 Rules, Policies, PolicySets

This section contains rules that capture the semantics of the four basic rule- and policy-combining algorithms: permit-overrides, deny-overrides, first-applicable and only-one-applicable. The rule- and policy-combining algorithms are very similar; the only difference occurs when an `Indeterminate` is returned while evaluating a child element. In a `PermitOverrides` rule-combining algorithm, an `Indeterminate` decision overrides a `Deny`, whereas in the equivalent policy-combining algorithm `Deny` decision would override `Indeterminate`.

Table 4.8 contains the inference rules for `Permit-Overrides` and `Deny-Overrides` rule-combining algorithms. Because the rules for Permit- and Deny-Overrides are symmetrical, I have presented only one set of them (using a shorthand notation ϵ to stand for `Permit/Deny`). Notice that for `Permit-Overrides` (resp. `Deny-Overrides`) if a `Permit` (resp. `Deny`) `Rule` returns `Indeterminate`, then assuming that there are no `Deny` (resp. `Permit`) rules that fired, the parent `Policy` would return `Indeterminate` as well. This subtle behavior was missed in previous formalizations of XACML [121].

Tables 4.9 and 4.10 present the semantics for `First-Applicable` and `OnlyOne-Applicable` rule-combining algorithms, respectively. While evaluating `First-Applicable`, the `Rules` are ordered according to their position in the `Policy` element. Given such ordering, as soon as a decision is returned from a `Rule`, that decision is returned and all subsequent `Rules` are ignored.

In Table 4.11, the semantics for `Permit-Overrides` and `Deny-Overrides` are shown. Note that unlike their rule-combining counterparts, `Permit-Overrides` and

Rule 1

$$\frac{\exists i : \mathrm{R}_i, \mathrm{RQ} \models \epsilon \qquad \mathrm{T}, \mathrm{RQ} \models \mathtt{True}}{(\mathtt{Policy}\ \epsilon\texttt{-Overrides}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \epsilon}$$

Rule 2

$$\frac{\begin{array}{c}\exists i : \mathrm{R}_i, \mathrm{RQ} \models \mathtt{Indeterminate} \wedge \mathrm{Effect}_{R_i} = \epsilon \\ \mathrm{T}, \mathrm{RQ} \models \mathtt{True} \\ \forall j : \mathrm{R}_j, \mathrm{RQ} \not\models \epsilon\end{array}}{(\mathtt{Policy}\ \epsilon\texttt{-Overrides}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \mathtt{Indeterminate}}$$

Rule 3

$$\frac{\begin{array}{c}\forall i : \mathrm{R}_i, \mathrm{RQ} \not\models \mathtt{Indeterminate} \vee \mathrm{Effect}_{R_i} = \bar{\epsilon} \\ \exists i : \mathrm{R}_i, \mathrm{RQ} \models \bar{\epsilon} \qquad \mathrm{T}, \mathrm{RQ} \models \mathtt{True} \\ \forall j : \mathrm{R}_j, \mathrm{RQ} \not\models \epsilon\end{array}}{(\mathtt{Policy}\ \epsilon\texttt{-Overrides}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \bar{\epsilon}}$$

Table 4.8: ϵ-Overrides Rule Combining Algorithm. ϵ stands for `Permit` or `Deny`, and $\bar{\epsilon}$ for the opposite.

Rule 1

$$\frac{\begin{array}{c}\mathtt{Effect} \in \{\mathtt{Permit}, \mathtt{Deny}, \mathtt{Indeterminate}\} \qquad \mathrm{T}, \mathrm{RQ} \models \mathrm{True} \\ \exists \mathrm{R}_i \text{ s.t. } \mathrm{R}_i, \mathrm{RQ} \models \mathtt{Effect} \\ \forall R_j \text{s.t. j < i :} \\ \mathrm{R}_j, \mathrm{RQ} \models \mathtt{NotApplicable}\end{array}}{(\mathtt{Policy}\ \texttt{First-Applicable}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \mathtt{Effect}}$$

Table 4.9: First-Applicable Rule Combining Algorithms.

RULE 1

$$\frac{\exists i : \mathrm{R}_i, \mathrm{RQ} \models \texttt{Indeterminate} \qquad \mathrm{T}, \mathrm{RQ} \models \texttt{True}}{(\texttt{Policy Only-One-Applicable}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 2

$$\frac{\begin{array}{c}\exists i, j : (\mathrm{R}_i, \mathrm{RQ} \models \texttt{Deny} \vee R_i, \mathrm{RQ} \models \texttt{Permit}) \wedge \\ (\mathrm{R}_j, \mathrm{RQ} \models \texttt{Deny} \vee R_j, \mathrm{RQ} \models \texttt{Permit}) \wedge \\ i \neq j \\ \mathrm{T}, \mathrm{RQ} \models \texttt{True}\end{array}}{(\texttt{Policy Only-One-Applicable}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 3

$$\frac{\begin{array}{c}\exists i : \mathrm{R}_i, \mathrm{RQ} \models \texttt{Effect} \\ \forall \mathrm{j} \text{ s.t. } j \neq i : \mathrm{R}_j, \mathrm{RQ} \models \texttt{NotApplicable} \\ \mathrm{T}, \mathrm{RQ} \models \texttt{True}\end{array}}{(\texttt{Policy Only-One-Applicable}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \texttt{Effect}}$$

Table 4.10: Only-One-Applicable Rule Overriding Algorithm.

`Deny-Overrides` are not symmetrical. The `Only-One-Applicable` and `First-Applicable` policy-combining algorithms are same as for `Rules`, so they are not shown here.

Finally, Table 4.12 contains the generic rules that hold regardless of the combining algorithm in question. These rules are used to infer `Indeterminate` or `NotApplicable` and are always applied with lowest priority, only in cases when no access decision was made using other inference rules.

4.2.3 Multiple Resource and Hierarchical Profile

While an important part of XACML, the Multiple Resource [20] and Hierarchical Profile [18] profiles do not actually add any expressive power to the language. Instead, they introduce abbreviated syntax that can be used to express access requests covering multiple resource in a concise manner. The semantics of such requests is given by 'un-

RULE 1

$$\frac{\exists i : \mathrm{P}_i, \mathrm{RQ} \models \texttt{Permit} \qquad \mathrm{T}, \mathrm{RQ} \models \texttt{True}}{(\texttt{PolicySet}\ \texttt{Permit-Overrides}\ \mathrm{T}\ \mathrm{P}_1, \ldots, \mathrm{P}_n), \mathrm{RQ} \models \texttt{Permit}}$$

RULE 2

$$\frac{\exists i : \mathrm{P}_i, \mathrm{RQ} \models \texttt{Deny} \qquad \mathrm{T}, \mathrm{RQ} \models \texttt{True} \\ \forall j : \mathrm{P}_j, \mathrm{RQ} \not\models \texttt{Permit}}{(\texttt{PolicySet}\ \texttt{Permit-Overrides}\ \mathrm{T}\ \mathrm{P}_1, \ldots, \mathrm{P}_n), \mathrm{RQ} \models \texttt{Deny}}$$

RULE 3

$$\frac{\exists i : \mathrm{P}_i, \mathrm{RQ} \models \texttt{Deny} \vee \mathrm{P}_i, \mathrm{RQ} \models \texttt{Indeterminate} \qquad \mathrm{T}, \mathrm{RQ} \models \texttt{True}}{(\texttt{PolicySet}\ \texttt{Deny-Overrides}\ \mathrm{T}\ \mathrm{P}_1, \ldots, \mathrm{P}_n), \mathrm{RQ} \models \texttt{Deny}}$$

RULE 4

$$\frac{\exists i : \mathrm{P}_i, \mathrm{RQ} \models \texttt{Permit} \qquad \mathrm{T}, \mathrm{RQ} \models \texttt{True} \\ \forall j : \mathrm{P}_j, \mathrm{RQ} \not\models \texttt{Deny} \wedge \mathrm{P}_j, \mathrm{RQ} \not\models \texttt{Indeterminate}}{(\texttt{PolicySet}\ \texttt{Deny-Overrides}\ \mathrm{T}\ \mathrm{P}_1, \ldots, \mathrm{P}_n), \mathrm{RQ} \models \texttt{Permit}}$$

Table 4.11: Permit-Overrides and Deny-Overrides Policy Combining Algorithms.

RULE 1

$$\frac{\mathrm{Comb} \neq \texttt{Deny-Overrides} \\ \exists i : \mathrm{P}_i, \mathrm{RQ} \models \texttt{Indeterminate} \\ \mathrm{T}, \mathrm{RQ} \models \texttt{True} \\ \forall j : \mathrm{P}_j, \mathrm{RQ} \models \texttt{Indeterminate} \vee \mathrm{P}_j, \mathrm{RQ} \models \texttt{NotApplicable}}{(\texttt{PolicySet}\ \mathrm{Comb}\ \mathrm{T}\ \mathrm{P}_1, \ldots, \mathrm{P}_n), \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 2

$$\frac{\exists i : \mathrm{R}_i, \mathrm{RQ} \models \texttt{Indeterminate} \\ \mathrm{T}, \mathrm{RQ} \models \texttt{True} \\ \forall j : \mathrm{R}_j, \mathrm{RQ} \not\models \texttt{Permit} \wedge \mathrm{R}_j, \mathrm{RQ} \not\models \texttt{Deny}}{(\texttt{Policy}\ \mathrm{Comb}\ \mathrm{T}\ \mathrm{R}_1, \ldots, \mathrm{R}_n), \mathrm{RQ} \models \texttt{Indeterminate}}$$

RULE 3

$$\frac{\mathrm{P}, \mathrm{RQ} \not\models \texttt{Permit} \qquad \mathrm{P}, \mathrm{RQ} \not\models \texttt{Deny} \\ \mathrm{P}, \mathrm{RQ} \not\models \texttt{Indeterminate}}{\mathrm{P}, \mathrm{RQ} \models \texttt{NotApplicable}}$$

Table 4.12: Generic `Indeterminate` rule for policies and rules.

wrapping' them into individual access requests, and then evaluating each request separately. Without any loss of generality, the discussion is focused on *individual* access requests.

4.2.4 Administrative Profile

So far, I have only been discussing access policies, i.e., policies that specify the situations under which users are granted or denied access. An administrative policy, on the other hand, specifies who (and under what conditions) is authorized to write access policies. For example, an administrative policy might state that members of group Administrators are allowed to write access policies about Files. Following, I describe the basic processing model of administrative XACML.

When a new access request R is to be checked against a XACML policy, it is first applied against all access policies in the set. If some policy applies to R and yields an access decision, then the access decision needs to be authorized by a *trusted* policy using a process defined in [114] as *reduction*.

Reduction is performed by applying the access request against any administrative policy P_s which is sibling of P, generating administrative requests ARQ. Then, using the administrative requests and other policies reduction *edges* are generated between the policies. The semantics of reduction is shown in Table 4.13. Finally, a search is performed through the reduction graph starting from the original access policy P, following reduction edges until a trusted policy is reached. The access decision of P is authorized only if the graph search reaches a trusted policy. The propagation rules (pertaining to the graph search) are shown in Table 4.14.

Rule 3
$$\frac{ARQ = (\texttt{Request}\ \text{Delegated Del-Info Delegate ATS-Admin})}{\text{ARQ}, \text{P} \models_{red\epsilon} (\texttt{Request}\ \text{ATS-Admin}\ \epsilon\ \text{Issuer}_P\ \emptyset)}$$

Rule 1
$$\frac{\begin{array}{c}\forall \text{P}_i, \text{P}_j, \mathit{s.t.parent}(\text{P}_i) = \mathit{parent}(\text{P}_j) \\ \text{RQ}, \text{P}_i \models_{red\epsilon} \text{AR} \qquad \text{AR}, \text{P}_j \models \epsilon\end{array}}{\epsilon\text{P}(\text{P}_i, \text{P}_j)}$$

Rule 2
$$\frac{\begin{array}{c}\forall \text{P}_i, \text{P}_j, \mathit{s.t.parent}(\text{P}_i) = \mathit{parent}(\text{P}_j) \\ \text{RQ}, \text{P}_i \models_{red\epsilon} \text{AR} \\ \text{AR}, \text{P}_j \models \texttt{Indeterminate}\end{array}}{\epsilon\text{I}(\text{P}_i, \text{P}_j)}$$

Table 4.13: Reduction Rules. ϵ stands for a `Permit` or `Deny`.

Rule 1
$$\frac{\begin{array}{c}\text{RQ}, \text{P} \models \epsilon \\ \exists \mathit{path} = (\text{P}, \ldots, \text{P}_n)\ \text{s.t.}\ \forall 1 \leq i < n : \\ \epsilon\text{P}(\text{P}_i, \text{P}_{i+1}) \wedge \text{issuer}_{\text{P}_n} = \mathit{trusted}\end{array}}{\text{RQ}, \text{P} \models_{auth} \epsilon}$$

Rule 2
$$\frac{\begin{array}{c}\text{RQ}, \text{P} \models \epsilon \vee \text{RQ}, \text{P} \models \texttt{Indeterminate} \\ \exists \mathit{path} = (\text{P}, \ldots, \text{P}_n)\ \text{s.t.}\ \forall 1 < i < n : \\ (\epsilon\text{P}(\text{P}_i, \text{P}_{i+1}) \vee \epsilon\text{I}(\text{P}_i, \text{P}_{i+1})) \wedge \\ \text{issuer}_{\text{P}_n} = \mathit{trusted}\end{array}}{\text{RQ}, \text{P} \models_{auth} \texttt{Indeterminate}}$$

Table 4.14: Propagation Rules

4.3 Discussion

This chapter presented a proof theoretic semantics for XACML v3.0 using natural deduction rules. These deduction rules closely follow the official XACML specification – in cases of ambiguities, I consulted the public XACML mailing list[1]. To verify the correctness of my interpretation of XACML, I also used Sun's reference implementation[2] of a XACML policy engine.

[1]OASIS XACML mailing list is available at `http://lists.oasis-open.org/archives/xacml/`
[2]Project information (including source code) available at `http://sunxacml.sourceforge.net`

The operational semantics is a contribution in itself since it presents a concise, unambiguous version of the official XACML specification (which, including the Administrative Profile specification, is over 150 pages) and it extends and improves previous work by covering more XACML features and fixing errors from previous semantics proposals (e.g., treatment of `Permit-Overrides` rule-combining algorithm as discussed in Section 4.2.2). For the purpose of this dissertation, this operational semantics is crucial since it is used to prove the correctness of the Datalog and Description Logic mappings presented in Chapters 5 and 6.

Chapter 5

Datalog-Based Theoretical Foundation of XACML

The first version of XACML was published in 2003, and since then there has been ongoing work on new versions – currently, XACML 3.0 [113] is close to standardization. Despite the amount of interest in XACML, the language does not have an official formal semantics that could clarify the official (informal) language specification and provide a comparison to other formal access control languages.

There has been previous work on providing formal treatments of XACML [66, 131, 117, 50, 36], mostly for the purpose of providing analysis services such as formal verification. Additionally, there has been work on providing a formal semantics for an early version of XACML [68], as well as an investigation of the compositional properties of the language [121]. While previous work does provide insight into some formal properties of the language, the following questions have still remained open:

- What is the complexity of access request checking in XACML (given an access request *R* and a policy *P*, determining the access decision of *P* for *R*) ?
- If access request checking is intractable for full XACML, then what are the subsets that make it polynomial? Which language features lead to intractability?
- How does XACML compare to other logic-based access control languages such as the Flexible Authorization Framework (FAF)? Can we extend XACML with features from FAF without compromising the worst case complexity of the language?

To answer the above questions, in this chapter I present a semantics of XACML based on a variant of Datalog. This formalization covers the latest version (3.0) of the language [113], including its Administrative Policy Profile [114]. Using this semantics, I provide complexity bounds for full XACML and various fragments. Additionally, using this Datalog semantics I discover features of XACML that are underspecified and ambiguous in the official specification, such as cyclic references of `PolicySets`. Finally, using the Datalog mapping I provide a comparison of XACML to other rule-based policy frameworks such as FAF and show how XACML can be extended with features from FAF while preserving the desirable computational properties of Datalog (polynomial data complexity and unique model property).

5.1 Mapping XACML to Datalog

This section provides a polynomial time reduction of XACML to the logic programming language Datalog such that access request checking in XACML is reduced to entailment in Datalog. Using this mapping, I show which fragments of XACML have provably polynomial data complexity and provide a comparison to other Datalog-based policy languages.

In this section, I will use a Prolog-like syntax to represent Datalog rules:

$$\mathtt{H}_0\mathtt{:-B}_0 \wedge \ldots \mathtt{B}_n, c$$

where B_i represents a literal, and c a boolean constraint. Datalog is a well-studied logic programming language [37], with a clear and concise semantics and polynomial data complexity. There is a variant of Datalog that allows limited negation of predicates in bodies of rules, called *stratified* Datalog – this variant maintains the desirable computational properties (i.e., unique minimal model and polynomial data complexity) [43]. More information about stratified Datalog can be found in Chapter 2.

The mapping takes a policy set PS and request RQ as input and generates a Datalog program $\mathcal{P}$. It is organized as follows:

- A set of extensional (EDB) predicates is generated to represent the relations between the policies in PS and their child elements. In this step, EDB predicates such as `hasRule` and `hasTarget` are instantiated for each `Policy` element.

- A series of rules and intensional (IDB) predicates are added to $\mathcal{P}$ to represent the semantics of matching access `Requests` against `Targets`, and the propagation of access decisions made by `Rules` to `Policies` and `PolicySets`. These first two steps are independent of the request; they are done before the policy is deployed.

- During runtime, each `Request` is compiled to a set of extensional predicates and these are added to $\mathcal{P}$. The access decision is retrieved after computing the unique minimal model of $\mathcal{P}$.

5.1.1 Mapping XACML policy structure to Datalog

I assume that each policy element – not only `Rules`, `Policies` and `PolicySets` but also `AttributeDesignators`, `AttributeSelectors`, and all other `Target` syntax elements – has a unique ID, so it can be distinguished it from the other elements. For an element P, P_{id} is used to refer to P's identifier.

To capture the parent/child relationship between policy elements in XACML, facts (binary predicates) are added to the Datalog program. For example, if a `Policy` P contains a `Rule` element R, then `hasRule`(P_{id}, R_{id}) is added to the Datalog program. To capture these relationships, I start from the root policy element and traverse the whole policy. Given a policy element S which contains of a list of children elements of type X^i:

$$\text{S} ::= (\texttt{Element} \quad X^1 \ldots X^n)$$

the following facts are generated:

- If X^i is a terminal node, then the fact $\texttt{hasX}^i(S_{id}, X^i_{val})$ is added where S_{id} is the identifier of S and X^i_{val} is the value of the child element.
- If X^i is non-terminal, then the fact $\texttt{hasX}^i(S_{id}, X^i_{id})$ is added where X^i_{id} is the identifier of the child element X^i.

Similarly to above, for a `Request` R and for each attribute AT in the request, unique IDs are generated. For a Request R and each of its `Attribute` elements AT, the following predicate symbols are added: `hasAT`$(R\text{-}id, AT\text{-}id)$, `hasCat`$(AT\text{-}id, cat)$, `hasAttrID`$(AT\text{-}id, attr\text{-}id)$, `hasIssuer`$(AT\text{-}id, issuer)$ and `hasValue`$(AT\text{-}id, AV)$.

Example 5.1.1.1 Consider the following `Rule` element.

```
(Rule  ()
       (Target
         (DisjunctiveMatch
         (ConjunctiveMatch
         (Match
         (AttributeValue 25 integer)
         (AttributeDesignator subject-cat age integer () false)
```

```
      integer-equal))))
    Permit))
```

Assume that r-id, t-id, dm-id, cm-id, m-id, ad-id are the unique ID's generated for the `Rule`, `Target`, `DisjunctiveMatch`, `ConjunctiveMatch`, `Match` and `Attribute-Designator` elements above, respectively.

The following facts would be added for the `Rule`: `hasEffect`(r-id, Permit) and `hasT`(r-id, t-id), linking the rule with its `Target` element. To capture the relationship between Target and its children, the following facts are added: `hasDM`(t-id, dm-id), `hasCM`(dm-id, cm-id) and `hasM`(cm-id, m-id). `hasValue`(m-id, 25), `hasValueType`(m-id, integer) are added to denote the relationship between the `Match` element above and its `Attribute-Value`. Finally, the following relations are added to $\mathcal{P}$ to initialize the `Attribute-Designator`: `hasAD`(m-id, ad-id), `hasCat`(ad-id, subject-cat), `has-attr-id`(ad-id, age), `hasType`(ad-id, integer), `mustBePresent`(ad-id, false), `hasMatchFcn`(m-id, integer-equal).

5.1.2 Mapping `Rules`, `Policies` and `PolicySets`

To capture access decisions, I use four intensional (head) predicates (`permit`, `deny`, `indet`, `na`) for each policy element type. For example, for `Rules` the following predicates are introduced: `permitR`(?P, ?RQ), `denyR`(?P, ?RQ), `indetR`(?P, ?RQ) and `naR`(?P, ?RQ). The semantics of these predicates is such that inferring ϵP(R_{id}, RQ_{id}) for a rule R_{id} and request RQ_{id} corresponds to $\mathrm{R}_{id}, \mathrm{RQ}_{id} \models \epsilon$ in the natural semantics[1].

I present a translation of the natural deduction rules presented in Chapter 4 to Datalog. The mapping consists of three Datalog programs:

[1] ϵ is shorthand for any one member of { `permit`, `deny`, `indet`, `na`}

- A program $\mathcal{P}_R$ capturing the rules needed to match a `Request` against a `Rule`.
- A Datalog program $\mathcal{P}_P$ that contains the rules needed to propagate an access decision from `Rules` to `Policies`.
- A program $\mathcal{P}_{PS}$ that contains the rules needed to propagate access decisions through `Policies` and `PolicySets`.

The final result of this translation will be a Datalog program $\mathcal{P} = \mathcal{P}_R \cup \mathcal{P}_P \cup \mathcal{P}_{PS}$ s.t. for any policy element P, request RQ and access decision ϵ,

$$\mathcal{P} \models \epsilon\mathrm{P}(\mathrm{P}_{id}, \mathrm{RQ}_{id}) \leftrightarrow \mathrm{P}_{id}, \mathrm{RQ}_{id} \models \epsilon$$

5.1.2.1 Matching Requests to Rules

For the purposes of matching requests to `Targets`, the following intensional predicates are added: `matchAD`, `matchM`, `matchCM`, `matchDM`, `matchT` (`indet` predicates are added as well). These elements have similar meaning to the effect predicates used for `Rules` and `Policies`; the only difference being that `matchAD` in addition to matching it also *selects* an attribute value from the request, so it is a ternary predicate: `matchAD`(?AD,?RQ, ?V). All other intensional predicates in this section are binary, having only the matching element and the request as arguments.

The Datalog rules that are used to match a request against an AttributeDesignator and a Target are shown in Table 5.1 and 5.2 respectively. Note that `fcn`$(?V, ?V_M)$ is the only constraint function that occurs, and is treated as a boolean constraint, i.e., both arguments will be ground by the time the `fcn` is being evaluated. Finally, the rules used to determine the access decision for a `Rule` are shown in Table 5.3. The `Condition` element is omitted here since its discussion warrants a separate section (Section 5.1.4).

$$
\begin{array}{lllll}
\texttt{matchAD}(?AD, ?RQ, ?V) & :- & \texttt{hasAttribute}(?RQ, ?AT) & \wedge & \texttt{hasValue}(?AT, ?V) \\
 & & \texttt{hasAttrID}(?AD, ?id) & \wedge & \texttt{hasAttrID}(?AT, ?id)\wedge \\
 & & \texttt{hasType}(?AT, ?type) & \wedge & \texttt{hasType}(?AD, ?type)\wedge \\
 & & \texttt{hasIssuer}(?AT, ?issuer) & \wedge & \texttt{hasIssuer}(?AD, ?issuer)\wedge \\
 & & \texttt{hasCat}(?AT, ?cat) & \wedge & \texttt{hasCat}(?AD, ?cat)\wedge \\
\texttt{indetAD}(?AD, ?RQ) & :- & \neg\texttt{matchAD}(?AD, ?RQ, ?V) & \wedge & \texttt{mustBePresent}(?AD, \mathrm{true}).
\end{array}
$$

Table 5.1: Matching a request (?RQ) against an `AttributeDesignator` (?AD) in Datalog.

To handle `AttributeSelectors`, I pre-process the policies in the following way: whenever an `AttributeSelector` *AS* encountered used, it is replaced it with a set of `AttributeValues` that are returned when *AS* is applied against the request. I use a predicate `selected`(?AS, ?RQ, ?V), which is instantiated with values (?V) selected when applying the `AttributeSelector` against the `Request` ?RQ. Because this operation is request-dependent, it might be considered an overhead at runtime. However, during the normal operation of a XACML engine these evaluations will have to be performed regardless, so I believe that such preprocessing does not add any substantial overhead. Rules 3 and 4 in Table 5.2 capture the semantics of matching `AttributeSelectors`.

The following lemma shows the Datalog mapping of `Rules` is consistent with respect to the XACML semantics presented earlier in this chapter.

Lemma 1 For a XACML `Rule` *RQ*, a request *R*, and ϵ one of {`indeterminate`, `not-applicable`, `permit`, `deny`}:

$$\mathcal{P} \models \epsilon\mathrm{P}(\mathrm{R}_{id}, \mathrm{RQ}_{id}) \leftrightarrow \mathrm{R}_{id}, \mathrm{RQ}_{id} \models \epsilon.$$

matchM($?M, ?RQ$)	:–	matchAD($?AD, ?RQ, ?V$) $fcn(?V, ?V_M) = True$.	$\wedge$	hasValue($?M, ?V_M$),
indetM($?M, ?RQ$)	:–	indetAD($?AD, ?RQ$)	$\wedge$	hasAD($?M, ?AD$).
matchM($?M, ?RQ$)	:–	selected($?AS, ?RQ, ?V$) fcn($?V, ?V_M$) $= True$.	$\wedge$	hasValue($?M, ?V_M$),
indetM($?M, ?RQ$)	:–	indetAS($?AS, ?RQ$)	$\wedge$	hasAS($?M, ?AS$).
matchCM($?CM, ?RQ$)	:–	$\bigwedge_{M_i \in CM}$(matchT($?M_i, ?RQ$))		
indetCM($?CM, ?RQ$)	:–	indetM($?M, ?RQ$)	$\wedge$	hasM($?CM, ?M$).
matchDM($?DM, ?RQ$)	:–	matchCM($?CM, ?RQ$)	$\wedge$	hasCM($?DM, ?CM$).
indetDM($?DM, ?RQ$)	:–	$\bigwedge_{CM_i \in DM}$ (indetCM($?CM_i, ?RQ$))		
matchT($?T, ?RQ$)	:–	$\bigwedge_{DM_i \in T}$(matchT($?DM_i, ?RQ$))		
indetT($?T, ?RQ$)	:–	indetDM($?DM, ?RQ$)	$\wedge$	hasDM($?T, ?DM$).

Table 5.2: Matching a request (?RQ) against a target (?T) in Datalog.

indetR($?R, ?RQ$)	:–	indetT($?T, ?RQ$)	$\wedge$	hasT($?R, ?T$).
ϵR($?R, ?Q$)	:–	matchT($?T, ?RQ$) hasEffect($?R, \epsilon$).	$\wedge$	hasT($?R, ?T$) $\wedge$
naR($?R, ?RQ$)	:–	$\neg$indetR($?R, ?RQ$) $\neg$denyR($?R, ?RQ$) $\neg$permitR($?R, ?RQ$).	$\wedge$ $\wedge$	

Table 5.3: Matching a `Request` (?RQ) against a `Rule` (?R) in Datalog.

Proof Sketch The proof is fairly simple since the Datalog rules presented in this chapter are a rewrite of the natural deduction rules from Chapter 4. The correspondence between the deduction rules used to infer that $\mathrm{R}_{id}, \mathrm{RQ}_{id} \models \epsilon$ and the Datalog rules used to infer $\mathcal{P} \models \epsilon\mathrm{P}(\mathrm{R}_{id}, \mathrm{RQ}_{id}$ is shown in Table 5.4. For brevity, instead of the whole rules, only references to their definitions are provided.

Natural Deduction Rule	Datalog Rule
Rule 1 and Rule 2, Table 4.5	Rule 1, Table 5.1
Rule 4, Table 4.5	Rule 2, Table 5.1
Rule i ($1 \leq i \leq 10$), Table 4.6	Rule i ($1 \leq i \leq 10$), Table 5.2
Rule i ($1 \leq i \leq 3$), Table 4.7	Rule i ($1 \leq i \leq 3$), Table 5.3

Table 5.4: Correspondence of Natural Deduction rules and Datalog rules for XACML request matching

□

Lemma 2 $\mathcal{P}_R$ is a Datalog program with a unique minimal model and polynomial data complexity.

Proof 1 The lemma is proved by showing that $\mathcal{P}_R$ is a Datalog program with stratified negation. First, notice that in all rules in $\mathcal{P}_R$, negated predicates occur only in the body. Second, it can be shown that the negation is stratified by presenting a valid stratification, as presented in Table 5.5.

Stratum	Predicate
0	`hasAttribute`, `hasT`, etc. (EDB predicates)
1	`matchAD`
2	`indetAD`

3	`matchM`, `indetM`
	`matchCM`, `indetCM`
	`matchDM`, `indetDM`
	`matchT`, `indetT`

4	`indetR`, `permitR`, `denyR`
5	`naR`

Table 5.5: **Strata ordering of $\mathcal{P}_R$.**

5.1.2.2 Matching Requests to Policies

In this section, I show how access decisions are propagated from `Rules` to `Policies`. This section is categorized by the type of rule-combining algorithm.

Permit-Overrides and **Deny-Overrides** Note that `Permit-` and `Deny-Overrides` are essentially symmetrical in their meaning. Thus, in Table 5.6 I only show one overriding algorithm – a placeholder variable ϵ is used to stand for either `Permit` or `Deny`, and $\bar{\epsilon}$ to stand for the opposite of ϵ.

$$\begin{array}{llll}
\epsilon\mathrm{P}(?P, ?RQ) & :- & \mathtt{hasT}(?P, ?T) & \wedge\ \mathtt{matchT}(?T, ?RQ) \wedge \\
 & & \mathtt{hasRule}(?P, ?R) & \wedge\ \epsilon\mathrm{R}(?R, ?RQ) \wedge \\
 & & \mathtt{hasComb}(?P, \epsilon\text{-Overrides}). & \\
\mathtt{indetP}(?P, ?RQ) & :- & \neg\epsilon\mathrm{P}(?P, ?RQ) & \wedge\ \mathtt{indetR}(?R, ?RQ) \wedge \\
 & & \mathtt{hasT}(?P, ?T) & \wedge\ \mathtt{matchT}(?T, ?RQ) \wedge \\
 & & \mathtt{hasRule}(?P, ?R) & \wedge\ \mathtt{hasEffect}(?R, \epsilon) \wedge \\
 & & \mathtt{indetR}(?R, ?RQ) & \wedge\ \mathtt{hasComb}(?P, \epsilon\text{-Overrides}). \\
\bar{\epsilon}P(?P, ?RQ) & :- & \neg\epsilon\mathrm{P}(?P, ?RQ) & \wedge\ \neg\mathtt{indetP}(?P, ?RQ) \wedge \\
 & & \mathtt{hasT}(?P, ?T) & \wedge\ \mathtt{matchT}(?T, ?RQ) \wedge \\
 & & \mathtt{hasRule}(?P, ?R) & \wedge\ \bar{\epsilon}R(?R, ?RQ) \wedge \\
 & & \mathtt{hasComb}(?P, \epsilon\text{-Overrides}). &
\end{array}$$

Table 5.6: Mapping `Permit-` and `Deny-Overrides` rule-combining algorithm to Datalog.

First-Applicable Again, ϵ is used to stand for `Permit`, `Deny` or `Indeterminate`, to avoid repeating similar rules. Given a policy P with n rules, for each rule at position i, the rule from Table 5.7 is generated.

$$\begin{array}{lll}
\epsilon\mathrm{P}(?P, ?RQ) & :- & \mathtt{hasT}(?P, ?T) \wedge \mathtt{matchT}(?T, ?RQ) \quad \wedge \\
 & & \mathtt{hasComb}(?P, \text{First-Applicable}) \quad \wedge \\
 & & \mathtt{hasRule}(?P, ?R_i) \wedge \epsilon\mathrm{R}(?R, ?RQ) \quad \wedge \\
 & & \bigwedge_{\mathtt{hasRule}(?P, ?R_j), j<i} \left(\mathtt{naR}(?R_j, ?RQ)\right).
\end{array}$$

Table 5.7: Mapping `First-Applicable` rule-combining algorithm to Datalog.

Only-One-Applicable If a `Policy` P has n rules, then for each rule at position i, the Datalog rules in Table 5.8 are generated.

Lemma 3 For a XACML `Policy` element P, a request R, and `result` one of { `indeterminate`, `notapplicable`, `permit`, `deny` }

$$\mathcal{P}_R \models \mathtt{resultP}(\mathrm{P}_{id}, \mathrm{R}_{id}) \leftrightarrow \mathrm{P}_{id}, \mathrm{R}_{id} \models \mathtt{result}.$$

ϵP(?P, ?RQ)	:–	hasT(?P, ?T)	$\wedge$	matchT(?T, ?RQ) $\wedge$
		ϵR(?R_i, ?RQ)	$\wedge$	hasRule(?P, ?R_i) $\wedge$
		hasComb(?P, Only-One-Applicable).		
		$\bigwedge_{\texttt{hasRule}(?P,?R_j), j \neq i} \left(\texttt{naR}(?R_j, ?RQ)\right)$.		
indetP(?P, ?RQ)	:–	hasT(?P, ?T)	$\wedge$	matchT(?T, ?RQ) $\wedge$
		hasComb(?P, 'Only-One-Applicable')	$\wedge$	hasRule(?P, ?R) $\wedge$
		indetR(?R, ?RQ).		
indetP(?P, ?RQ)	:–	hasT(?P, ?T)	$\wedge$	matchT(?T, ?RQ) $\wedge$
		hasComb(?P, Only-One-Applicable)	$\wedge$	hasRule(?P, ?R_i) $\wedge$
		ϵR(?R_i, ?RQ)	$\wedge$	hasRule(?P, ?R_j) $\wedge$
		ϵR(?R_j, ?RQ), $i \neq j$		.

Table 5.8: Representation of `Only-One-Applicable` rule-combining algorithm in Datalog. ϵ stands for either a `Permit` or `Deny`.

Proof Sketch Similarly to the `Rule` case, the proof is fairly simple since the Datalog rules presented in this chapter are a rewrite of the natural deduction rules from Chapter 4. For brevity, instead of the whole rules, only references to their definitions are provided. in Table 5.9.

Natural Deduction Rule	Datalog Rule
Rule i ($1 \leq i \leq 3$), Table 4.8	Rule i ($1 \leq i \leq 10$), Table 5.6
Rule 1, Table 4.9	Rule 1, Table 5.7
Rule i ($1 \leq i \leq 3$), Table 4.10	Rule i ($1 \leq i \leq 10$), Table 5.8

Table 5.9: Correspondence of operational semantics and Datalog rules for XACML policy matching

□

Lemma 4 $\mathcal{P}_P \cup \mathcal{P}_R$ is a Datalog program with a unique minimal model and polynomial data complexity.

Proof 2 Similarly to the unique model proof for $\mathcal{P}_R$, I will show that $\mathcal{P}_P \cup P_R$ has a unique, minimal model since it is a program with stratified negation. Again, notice in all of the rules in $\mathcal{P}_P$, negation occurs only in the body of the rules. Moreover, most of the rules in

$\mathcal{P}_P$ have negation of predicates of $\mathcal{P}_R$, which suggests a possible valid stratification. We show that the negation is stratified by presenting the strata of $\mathcal{P}_P$ (Table 5.10).

combining algorithm	stratification
permit-overrides	$\mathcal{P}_R \prec$ permitP $\prec$ indetP $\prec$ denyP $\prec$ naP
deny-overrides	$\mathcal{P}_R \prec$ denyP $\prec$ indetP $\prec$ permitP $\prec$ naP
first-applicable	$\mathcal{P}_R \prec$ denyP, permitP, indetP $\prec$ naP
only-one-applicable	$\mathcal{P}_R \prec$ denyP, permitP, indetP $\prec$ naP

Table 5.10: **Strata of** $\mathcal{P}_R \cup \mathcal{P}_P$.

5.1.2.3 Matching Requests to PolicySets

Unlike `Policy` elements, which only contain `Rules`, a `PolicySet` can contain `Policies` or *other* `PolicySets`. Given this, in Table 5.11, I present the Datalog rules for `Permit-Overrides` and `Deny-Overrides`. The `First-Applicable` and `Only-One-Applicable` policy combining algorithms have the same semantics as their rule combining counterparts discussed above, so they are omitted here.

$$\begin{array}{llll}
\epsilon\text{PS}(?PS, ?RQ) & :- & \texttt{hasT}(?PS, ?T) & \wedge \ \texttt{matchT}(?T, ?RQ) \wedge \\
 & & \texttt{hasP}(?PS, ?P) & \wedge \ \epsilon\text{P}(?P, ?RQ) \wedge \\
 & & \texttt{hasComb}(?PS, \epsilon\text{-Overrides}). & \\
\bar{\epsilon}\text{PS}(?PS, ?RQ) & :- & \texttt{hasT}(?PS, ?T) & \wedge \ \texttt{matchT}(?T, ?RQ) \wedge \\
 & & \texttt{hasP}(?PS, ?P) & \wedge \ \bar{\epsilon}\text{P}(?P, ?RQ) \wedge \\
 & & \texttt{hasComb}(?PS, \epsilon\text{-Overrides}) & \wedge \ \neg\epsilon\text{PS}(?PS, ?RQ). \\
\epsilon\text{PS}(?PS, ?RQ) & :- & \texttt{hasT}(?PS, ?T) & \wedge \ \texttt{matchT}(?T, ?RQ) \wedge \\
 & & \texttt{hasPS}(?PS, ?PS_1) & \wedge \ \epsilon\text{PS}(?PS_1, ?RQ) \wedge \\
 & & \texttt{hasComb}(?PS, \epsilon\text{-Overrides}). & \\
\bar{\epsilon}\text{PS}(?PS, ?RQ) & :- & \texttt{hasT}(?PS, ?T) & \wedge \ \texttt{matchT}(?T, ?RQ) \wedge \\
 & & \texttt{hasPS}(?PS, ?PS_1) & \wedge \ \bar{\epsilon}\text{PS}(?PS_1, ?RQ) \wedge \\
 & & \texttt{hasComb}(?PS, \epsilon\text{-Overrides}) & \wedge \ \neg\epsilon\text{PS}(?PS, ?RQ).
\end{array}$$

Table 5.11: Representation of `Permit-` and `Deny-Overrides` policy combining algorithms in Datalog.

Lemma 5 For a XACML `PolicySet` element PS, a request R, and ϵ one of {`indeterminate`, `notapplicable`, `permit`, `deny`}:

$$\mathcal{P} \models \texttt{resultP}(\mathrm{PS}_{id}, \mathrm{R}_{id}) \leftrightarrow \mathrm{PS}_{id}, \mathrm{R}_{id} \models \epsilon.$$

Proof Sketch Similarly to the `Rule` and `Policy` case, the proof is fairly simple since the Datalog rules presented in this chapter are a rewrite of the natural deduction rules from Chapter 4. Note that there are two sets of rules in Table 5.11: one where decisions are propagated from child `Policies` (rules 1 and 2), and another set where decisions are propagated from child `PolicySets` (rules 3 and 4). This is because in the datalog mapping there are different intensional predicates used to infer access decisions for `Policies` and `PolicySets`. In the natural semantics rules (Table 4.11 in Chapter 4), there is no distinction on the type of child policy element.

It is straightforward to see that Each of the four rules in Table 5.11 corresponds to a rule with the same index in Table 4.11 (operational semantics rules).

□

Unlike $\mathcal{P}_R$ and $\mathcal{P}_P$, the Datalog program that captures the semantics of `PolicySets` ($\mathcal{P}_{PS}$) is not stratifiable. This follows from the fact that arbitrary references between `PolicySets` are allowed in XACML. Consider the rules in Table 5.11; observe that $\neg\epsilon$PS and $\bar{\epsilon}$PS occur in both the body of the rules, thus in cases of cyclical references among PolicySets negation and recursion will be intertwined and the program will not be stratifiable. The discussion about the computational properties of XACML with PolicySets is presented in a Section 5.2.

5.1.3 Administrative Profile

In this section, I will show how the administrative profile of XACML can also be mapped to Datalog.

5.1.3.1 Reduction Rules

In XACML, administrative requests are generated by reducing access requests against policies. For this purpose, for each `PolicySet` or `Policy` *P* I introduce two predicates: `reducep`(AR_{id}, R_{id}, P_{id}) and `reduced`(AR_{id}, R_{id}, P_{id}), where R_{id} is the ID of the original request, and AR_{id} is an ID for the administrative request that is generated when *R* is reduced against *P*. The predicate `reduced`(AR_{id}, R_{id}, P_{id}) means that the access request R_{id} has been reduced to the administrative request AR_{id} using the policy P_{id} according to the semantics of the reduction procedure defined in XACML's Administrative Profile. The Datalog version of the reduction rules is shown in Table 5.12.

`hasDelegate`(?*AR*, ?*I*)	:–	`reduceϵ`(?*AR*, ?*RQ*, ?*P*)	∧	`hasIssuer`(?*P*, ?*I*).
`hasAttr`(?*AR*, ?*AT*)	:–	`reduceϵ`(?*AR*, ?*RQ*, ?*P*)	∧	`hasAttr`(?*R*, ?*AT*) ∧
		`hasCat`(?*RQ*, ?*AT*, delegated).		
`hasAttr`(?*AR*, ?*AT*) ∧ `hasCat`(?*AR*, ?*AT*, `concat`('delegated', ?*C*))	:–	`reduceϵ`(?*AR*, ?*RQ*, ?*P*)	∧	`hasAttr`(?*R*, ?*AT*)∧
		`hasCat`(?*RQ*, ?*AT*, ?*C*),		?*C* ≠ delegated.

Table 5.12: Datalog version of reduction rules in XACML 3.0.

Note that I have extended the `hasCat` predicate to include the request where the attribute belongs; this was done to avoid conflating `hasCat` predicates belonging to different requests (e.g., the predicates in the RHS and LHS of the last rule above).

The Datalog rules used to capture the semantics of generating edges in the reduction graph are shown in Table 5.13.

$$\begin{array}{lll}
\epsilon\mathtt{P}(?P_i, ?P_j)\text{:-} & \mathtt{hasParent}(?P_i, ?par) & \wedge\mathtt{hasParent}(?P_j, ?par) \wedge \\
 & \mathtt{reduce}\epsilon(?AR, ?RQ, ?P_i) & \wedge\mathtt{permitP}(?AR, ?P_j). \\
\epsilon\mathtt{I}(?P_i, ?P_j)\text{:-} & \mathtt{hasParent}(?P_i, ?par) & \wedge\mathtt{hasParent}(?P_j, ?par) \wedge \\
 & \mathtt{reduce}\epsilon(AR, RQ, P_i) & \wedge\mathtt{indetP}(?AR, ?P_j).
\end{array}$$

Table 5.13: Rules to generate edges in reduction graph.

5.1.3.2 Propagation Rules

Propagation is done by a breadth-first-search along *PP* edges until a policy with a trusted issuer is encountered. Predicates `visitedPP` and `startingPolicy` are initialized to contain only the node whose access decision it is being authorized. The Datalog rules for authorizing `Permit` decisions are shown in Table 5.14; The rules for authorizing `Deny` decisions are omitted since they are equivalent to the `Permit` ones.

$$\begin{array}{lll}
\mathtt{visitedPP}(?P_j)\text{:-} & \mathtt{visitedPP}(?P_i) & \wedge\mathtt{PP}(?P_i, ?P_j). \\
\mathtt{visitedPI}(?P_j)\text{:-} & \mathtt{visitedPP}(?P_i) & \wedge\mathtt{PI}(?P_i, ?P_j). \\
\mathtt{visitedAll}(?X)\text{:-} & \mathtt{visitedPP}(?X). & \\
\mathtt{visitedAll}(?X)\text{:-} & \mathtt{visitedPI}(?X). & \\
\epsilon\mathtt{Auth}(?X)\text{:-} & \mathtt{visitedPP}(?P_i) & \wedge\mathtt{trusted}(?P_i)\wedge \\
 & \mathtt{starting}(?X) & \wedge\neg\mathtt{visitedPI}(?P_i). \\
\mathtt{indetAuth}(?X)\text{:-} & \mathtt{visitedAll}(?P_i) & \wedge\mathtt{trusted}(?P_i)\wedge \\
 & \mathtt{starting}(?X) & \wedge\neg\epsilon\mathtt{Auth}(?X).
\end{array}$$

Table 5.14: Authorization rules for access decision in reduction graph.

The program above is stratified, so it has polynomial data complexity. However, worst case running time increases by a factor of n^3 (n corresponds to the number of policies in the set), since additional evaluation is required for administrative policies. This is because for each evaluation of a request against a policy, first a reduction graph needs to be generated by evaluating the request against each pair of policies of the set (which takes $O(n^2)$ where n is the number of policies). Then, the access decision needs to be authorized by performing a breadth-first search on the generated reduction graph (following the rules in Table 5.14), which is $O(n)$. Thus, the total computational overhead of using the administrative profile is $O(n^2) + O(n)$ which is $O(n^2)$.

5.1.4 Functions in XACML

One of the most powerful features of XACML is its wide array of functions available to compare attributes. These functions are available as part of the `Condition` element in `Rules`, and act as *constraints* on the access request. There are different types of constraints available: from datatype comparison, arithmetic and set-oriented to higher level functions.

Adding constraint functions to Datalog is non-trivial – even if the constraints are decidable, coupling them with rules can easily lead to an undecidable combination [86]. Extending Datalog with constraints has attracted a great amount of interest from the research community; however, in most proposals the expressiveness of the constraint domains is severely limited and certainly not expressive enough to cover all of the functions supported by XACML.

Fortunately, there are two implicit assumptions in XACML regarding its support for functions/constraints:

- Functions only occur in the `Condition` and `Match` elements, and in both cases they return a boolean value

- Functions do not have any side-effects, i.e., they do not modify the values of any attribute in the request or policy.

The above observations indicate it is possible to avoid doing any constraint *solving* at run-time. Instead, given that the functions are boolean and assuming that all of the arguments are ground, much simpler constraint *checking* can be performed. This approach was used in SecPal [27], where the authors defined a safety condition for their language: each variable that goes in the constraints has to be bound before evaluating the constraints.

It can be easily shown that XACML satisfies SecPal's safety condition. This is because the only input variables `Condition` and `Match` are given using `AttributeSelector` and `AttributeDesignator` elements. In order to make sure all input arguments are ground, the policy can be pre-processed by evaluating each `AttributeSelector` and `AttributeDesignator` against the request and replacing it with a bag of constant values selected. In both cases, the variable referenced in the function is bound to a bag of attribute values, which can then be used as input to the function. Thus, while `Condition` elements support a wide range of functions, during run-time all of the input arguments for those functions are bound to bags of constant values, effectively satisfying the groundness safety condition and simplifying function evaluation.

5.2 Complexity of XACML

5.2.1 XACML with cyclical `PolicySets`

If cyclical `PolicySet` references are allowed, then the access decision of a policy for a given request will be sometimes be inconsistent – depending on the order of evaluation, different access decisions might be produced. Consider the following example.

Example 5.2.1.1 There are two `PolicySets`, PS_1 and PS_2. PS_1 has a `DenyOverrides` combining algorithm, whereas PS_2 is `PermitOverrides`. In addition to their joint references, they also contain a policy each, as shown in the diagram below.

$$\begin{tikzcd}
& PS_1\text{(Deny-Overrides)} \arrow[dl] \arrow[d, leftrightarrow] \\
P_1 & PS_2\text{(Permit-Overrides)} \arrow[d] \\
& P_2
\end{tikzcd}$$

Now, consider what would happen if a request came in such that P_2 yielded a `Deny` and P_1 a `Permit`. Depending on the order of evaluation of PS_1 and PS_2, different results would be obtained for the same request. If I choose to evaluate PS_1 first, then PS_1 would yield a Permit and PS_2 would also return a Permit (since it is Permit-Overrides and PS_1 returned a Permit). However, if PS_2 is evaluated first, it would return a Deny, and PS_1, having a Deny-Overrides combining algorithm, would return a Deny itself.

The Datalog program produced by mapping the example above is presented below.

$$
\begin{array}{lll}
\texttt{permitPS}(?X, PS2) & :- & \texttt{permitPS}(?X, PS1). \\
\texttt{permitPS}(?X, PS2) & :- & \texttt{permitP}(?X, P2). \\
\texttt{denyPS}(?X, PS2) & :- & \texttt{denyPS}(?X, PS1), \neg\texttt{permitPS}(?X, PS2). \\
\texttt{denyPS}(?X, PS2) & :- & \texttt{denyP}(?X, P2), \texttt{permitPS}(?X, PS2). \\
\texttt{denyPS}(?X, PS1) & :- & \texttt{denyP}(?X, P1). \\
\texttt{denyPS}(?X, PS1) & :- & \texttt{denyPS}(?X, PS2). \\
\texttt{permitPS}(?X, PS1) & :- & \texttt{permitP}(X, P2), \neg\texttt{denyPS}(X, PS2). \\
\texttt{permitPS}(?X, PS1) & :- & \texttt{permitPS}(X, PS2), \neg\texttt{denyPS}(X, PS2). \\
\texttt{permitP}(R, P1). & & \\
\texttt{denyP}(R, P2). & &
\end{array}
$$

The logic program above is not stratified (nor locally stratified) since denyPS and permitPS occur negatively in the bodies of the rules above. To provide semantics for such unstratifiable programs, the most well-studied approaches from logic programming are stable models [54] and well-founded semantics [123].

The above program has multiple models under the stable model semantics, where both permit and deny decisions are inferred for PS1 and PS2 depending on the model. The possibility of multiple models for a logic program is highly undesirable for policies since it implies ambiguity, i.e., a policy that returns different access decisions for the same access request. This ambiguity, along with the co-NP data complexity [44] of computing stable models renders the stable model semantics impractical for XACML.

The well-founded semantics of a logic program partitions all ground atoms into three sets: true, false and unknown. The well founded model can be generated by an constructing an alternating fixpoint [53], that is, building up a set of negative conclusions until the lest fixpoint is reached, and then deriving the positive conclusions that follow (without deriving any further negative ones) using traditional Datalog semantics. The alternating fixpoint for the above program does not contain any inferences for PS1 or PS2, so the access decision of the above program will be unknown under the well founded semantics.

Thus, applying the two standard semantics for unrestricted negation in logic programs yields ambiguity in one case and underspecification in the other case. To show that this behavior is an issue with XACML itself, and is independent of the type of logic programming applied, I show a proof that in XACML with arbitrary references, the problem of determining if a set of policies yields a `Permit` for a request is NP-complete.

For a policy set PS and a request R, I refer to each distinct and valid order of evaluation of R against the policies in PS as a *model* of PS for R (written $M(PS, R)$). Obviously, depending on the cycles in the program there may be many possible models $M_i(PS, R)$, where some of them can have conflicting access decisions for R. Thus, in order to provide an unambiguous answer, the decisions of all the models have to be combined when processing the request – this can be done similarly to how decisions of `Rule` or `Policy` elements are combined.

Theorem 5.2.1 Give a cyclical XACML policy set PS with a root `PolicySet` PS_0 and request R, the problem of determining an evaluation order $M(PS, R)$ such that $permit(PS_0, R) \in M(PS, R)$ is NP-complete w.r.t to the size of PS.

Proof. **Membership**. If I were given an evaluation order $M(PS, R)$ and a request R it is easy to see that checking if PS yields a `Permit` for R under $M(PS, R)$ is polynomial: I simply follow the ordering in $M(PS, R)$, evaluate each policy element and combine the results. Since each policy element in $P \in PS$ is evaluated only once, testing if R yields a `Permit` for PS_0 under ordering M is also polynomial.

Hardness. This will be shown by reducing the well known boolean satisfiability (3SAT) problem to ours. In 3SAT, a formula F of the following type is given:

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge \ldots \wedge (x_{n1} \vee x_{n2} \vee x_{n3})$$

where each x_{ij} is a (possibly negated) boolean variable. (The same variable might repeat in the formula.) The 3SAT problem is to find an assignment of values for the variables such that F is true overall.

To represent a boolean variable x_{ij}, I generate the following combination of `Policy-Sets`:

$$\begin{tikzcd}
PS^0_{ij}\text{(Deny-Overrides)} \arrow[r, leftrightarrow] \arrow[d] & PS^1_{ij}\text{(Permit-Overrides)} \arrow[d] \\
P_1 & P_2
\end{tikzcd}$$

All of the policy elements above have an empty `Target` element that is empty, so they match all requests. Additionally, P_2 always returns `Deny` and P_1 always returns `Permit`. As explained in the example in Section 5.1, depending on the order of evaluation, this set will return either a `Permit` or `Deny` (e.g., evaluating PS^0_{ij} before PS^1_{ij} will yield a `Permit`). This set is used to simulate a truth assignment of variable x_{ij} – if the set returns

`Permit`, then the value of x_{ij} is true (if it returns Deny, then x_{ij} is false).

$$\begin{tikzcd}
 & & & C(\text{Permit-Overrides}) \arrow[dlll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
PS^{0}_{11} \arrow[r, leftrightarrow] \arrow[d] & PS^{1}_{11} \arrow[d] & PS^{0}_{12} \arrow[r, leftrightarrow] \arrow[d] & PS^{1}_{12} \arrow[d] & PS^{0}_{13} \arrow[r, leftrightarrow] \arrow[d] & PS^{1}_{13} \arrow[d] \\
P^{0}_{11} & P^{1}_{11} & P^{0}_{12} & P^{1}_{12} & P^{0}_{13} & P^{1}_{13}
\end{tikzcd}$$

Table 5.15: Representing a clause consisting of three variables with a XACML policy structure.

The variables corresponding to a 3-clause are combined in Table 5.15. Since each clause represents a disjunction of the atoms, a `PermitOverrides` combining algorithm is used – it will yield a `Permit` iff at least one of the variables in the clause is true. All of the clauses are referenced by the root policy set F, which represents the top level formula. The combining algorithm for F is `DenyOverrides`, since the F represents a conjunction of all its clauses. □

5.2.2 Tractable Fragments

Let us denote the fragment of XACML where cyclical references are not allowed with XACML^{-}. This assumption does not change the rules of program $\mathcal{P}_{PS}$, so it is still not stratifiable. However, I will show that the translation of the nested XACML^{-} policy sets comprises a *locally* stratifiable Datalog program, i.e., if I consider the ground atoms as propositional symbols, then the instantiated version of $\mathcal{P}_{PS}$ will be stratified. Locally stratified programs have the same nice properties as stratified Datalog: they have a unique minimal model that can be computed in quadratic time.

Since there are no cycles allowed in XACML^{-}, there is a partial order which can be used to evaluate `PolicySets`. Consider the policy set below:

$$\begin{tikzcd}
 & PS_1 \arrow[dl] \arrow[dd] \arrow[dr] & & \\
P_1 & & PS_2 \arrow[dl] \arrow[dr] & \\
 & PS_3 & & PS_4
\end{tikzcd}$$

In this example policy, a directed edge $P \rightarrow D$ means that PolicySet P references(includes) PolicySet (or Policy) D. Since their references do not form cycles, there exists a partial order among the policy elements in the set. In the scenario above, for example, PS_4, P_1 and PS_3 will be evaluated first, then PS_2 and in the last step, the root policy set PS_1. Because the Datalog rules follow this evaluation ordering, the instantiated intensional predicates (`permitPS`, `denyPS`) can be split in separate strata that match the partial order of the *references* relation for policy sets. As a result, $\mathcal{P}_{PS}$ is *locally* stratifiable.

Theorem 5.2.2 The Datalog program $\mathcal{P} = \mathcal{P}_R \cup \mathcal{P}_P \cup \mathcal{P}_{PS}$ has a unique and minimal model that can be computed in quadratic time.

Proof. We will show that $\mathcal{P}_{PS} \cup \mathcal{P}_P \cup P_R$ has a unique, minimal model since it is a variant of Datalog with locally stratified negation. It was shown above that $\mathcal{P}_{PS}$ is a locally stratified Datalog program. Notice that in the rules in $\mathcal{P}_{PS}$, negated IDB predicates from $\mathcal{P}_P$ can also occur (since `PolicySets` can refer to `Policies` as well.) Given this, $\mathcal{P}$ is stratified as follows:

$$\mathcal{P}_R \prec \mathcal{P}_P \prec \mathcal{P}_{PS}$$

Thus, the predicates from P_R will be evaluated first (for those predicates, the ordering in Table 5.5 holds). Then, the rules that propagate an access decision from a `Rule` to a `Policy` are evaluated – the ordering of those predicates depends on the combining algorithm and is described in Table 5.10. Finally, the results are combined in the root policy sets according to the Datalog rules presented in Section 5.1.2.3. In this case, the order of evaluation depends on the references among the policy sets (no cycles allowed in XACML$^-$). Since the ground atoms for $\mathcal{P}_R$, $\mathcal{P}_P$, $\mathcal{P}_{PS}$ are disjoint, and I have shown that they are all (locally) stratifiable, I can infer that the $\mathcal{P}$ is locally stratifiable as well.

Being a locally stratified Datalog program, the XACML$^-$ translation has a unique stable model [54]. Moreover, since a) in [25] it was shown that well founded semantics coincides with the stable model semantics, and b) well-founded semantics can be computed in O(N^2) where N is the size of data [123], it follows that XACML$^-$ has quadratic time data complexity.

□

5.3 XACML and Logic-Based Languages

In this chapter, a comprehensive mapping of XACML v3.0 to Datalog was presented. Using this mapping, I showed that a large subset XACML, i.e., XACML without cyclical references among policies, has polynomial data complexity and unique, minimal model property. The mapping also showed the overhead of using the Administrative Profile, which results in increasing the worst case complexity by a factor of $O(n^2)$, where n represents the number of policies in the set.

In addition to the complexity results, this Datalog-based mapping opens the door to 1) detailed comparison with well-studied academic and industry logic-based access control languages, and 2) possibility of extending the language with features from other Datalog-based languages. I will cover both these points in this section, first by comparing XACML with the Flexible Authorization Framework (FAF [71]), and then showing how features from FAF that are missing in XACML can be added to the latter.

The version of XACML without references and constraints is very similar to FAF – they both can be embedded in locally stratified Datalog, they both allow for specifying propagation and conflict resolution policies and they both have the unique minimal model property. An in-depth comparison is presented below:

- *Hierarchies*. While FAF fully supports subject and object hierarchies (including role hierarchies), the support of such features in XACML is limited. In particular, XACML does not support hierarchies among subject groups (e.g., stating that a user *John* is in group *GraduateStudents* which is a subgroup of *Students*) and has limited support for role hierarchies. Some of these limitations are illustrated with an example below.

- *Propagation* of access decisions. Both approaches support propagation of access decision, with the difference being that in FAF propagation is done along hierarchies, whereas in XACML, propagation is done along the parent-child relationship of policy elements (*hasRule* , *hasPolicy* and *hasPolicySet* predicates in my mapping).

- *Conflict Resolution*. XACML and FAF both provide similar functionality for conflict resolution: they both support permit-overrides and deny-overrides combining algorithms. Additionally, given the XACML-to-Datalog mapping, the conflict resolution algorithms that are available in FAF and not in XACML can be easily added to the latter as well.

- `Datatypes` and `Functions`. XACML has a very powerful built-in support for various functions through its `Condition` element. In FAF, data-types and datatype matching functions are not explicitly addressed.

To illustrate XACML's incomplete support for role hierarchies, consider the following policy. It has three roles: Doctor, Nurse and Administrator, where Doctor is a senior role of both Nurse and Administrator. There is a set of permissions PPS_N associated with Nurse role and a set of permissions PPS_A associated with the Administrator role. To represent the seniority of Doctor roles, in XACML the Doctor policy would look like:

```
<PolicySet PolicySetId="PPS_D"
  PolicyCombininAlgId="Permit-Overrides">
  <Target/>
  <PolicySetIdReference>
    PPS_A.xml
 </PolicySetIdReference>
 <PolicySetIdReference>
    PPS_N.xml
 </PolicySetIdReference>
</PolicySet>
```

Consider what happens if a new `PolicySet` is added (PPS_B), which is associated with people who have activated the roles of both Nurse and Admin. Since doctors inherit the privileges of nurses and admins in this example, they should be able to access this policy set. However, for this to happen, the doctor policy will have to be manually updated and the PPS_B set added to its list of references. It is unfortunate that every time a new permission is added, this manual updating is required to make sure the hierarchy still works. Obviously, an automatic approach would be preferred where the policy engine is smart enough to realize that a Doctor role inherits the privileges of both Nurse and Admin, and is therefore able to access PPS_B.

There is support in FAF for this functionality, using *hierarchy* predicates. Given my XACML-to-Datalog semantics, these predicates can be added to XACML as well, essentially adding a new feature to the language without sacrificing computational properties. Adapting the hierarchy predicates from FAF has to be done at the attribute matching level in XACML, for example:

$$\begin{aligned}
&\texttt{hasAttrValue}(\textit{role}, \text{Nurse}) && \text{:- } \texttt{hasAttrValue}(\textit{role}, \text{Doctor})\\
&\texttt{hasAttrValue}(\textit{role}, \text{Admin}) && \text{:- } \texttt{hasAttrValue}(\textit{role}, \text{Doctor})
\end{aligned}$$

The above rules state that whenever a `Request` arrives with a value of Doctor for the *role* attribute, then the engine will automatically infer it has values Nurse and Admin as well, so the permissions written for nurses and admins will be inherited. With this simple addition, added for each role relationship, the policy engine would automatically infer that Doctor has access to PPS_B without any need to modify its policy.

Other useful feature of FAF is its *history table* , i.e., a table whose rows describe the access requests processed. A history table can be used to model various policy constraints such as the Chinese Wall security policy, and will not change the computational complexity of XACML if it were added. Additionally, FAF provides other suggestions for future extensions of XACML. For example, combining algorithms such as most-specific-overrides and no-overriding can also be easily adapted to XACML: most-specific-overrides can be performed by comparing the `Targets` of the two policy elements to be combined, whereas for no-overriding a clause can be added to throw `Indeterminate` whenever conflicting access decisions are returned. The mapping I have presented in this section provides a framework for experimenting with such extensions.

Chapter 6

XACML Analysis Services

In the previous chapter, I described a formal framework for XACML based on Datalog where access request checking was provided in PTIME in the case with no cyclical references between policies. That chapter, however, did not address the lack of support for comparing and debugging policies nor the lack of compile-time services for XACML in general. The following example scenario illustrates some of these services (note that formal verification services were discussed in Chapter 1).

Analysis Service Example Consider an access control policy for a bank for this example. There is a general, high-level access control policy assigned by the bank's main branch (headquarters), which must be followed by the policy of each branch. Each local branch policy *subsumes* the general one and extends it appropriately, essentially customizing the general policy for its own needs. The main requirement is that each local policy conforms to the general policy; namely, for any given access request, whenever the general policy returns `Permit` (or `Deny`), the local branch policy should also return `Permit` (or `Deny`). This scenario motivates the need for automated policy *comparison*. Checking policy subsumption, such as in this example, is one type of comparison, others include: checking policy *disjointness* (i.e., there is no access request s.t. both policies apply) and *equivalence* (for any access request, both policies will yield the same decision). For large policy sets written in an expressive language, performing these services manually would be very tedious and error-prone, so an automated approach is preferred [118].

Federated Policies Example Continuing on the access control policy of a bank example, consider what happens when access control is federated across multiple, (to some extent) independent branches of the bank. Each branch might use different attribute schemes to describe subjects, actions and resources that comprise its access policy. For example, keeping in mind that XACML is an attribute based language, one bank branch might use a boolean attribute *adultAge* to represent adult customers, whereas another might use a numeric constraint on an integer attribute *age*. In a large organization with many different departments, reconciling such vocabulary information is non-trivial. Thus, there exists a need for a mechanism that will simplify specification and management of policies from heterogeneous sources - and as importantly, support all of the common analysis services discussed here and in previous work: change analysis (policy comparison), formal verification and redundancy checking.

In this chapter, I present a logic-based analysis framework that can reason about expressive XACML policies and provide services such as above. As a basis for this framework I use Description Logics (DL), which are a family of formalisms that are decidable subsets of First-Order logic, and are the formal basis for the Web Ontology Language [45]. Because of the correspondence of policy analysis services to DL reasoning services (e.g., policy comparison can be reduced to concept subsumption, whereas formal verification can be reduced to concept satisfiability), the framework can leverage off-the-shelf DL reasoners optimized to provide the above-mentioned analysis services.

An important benefit of using a logic compatible with OWL is that we can leverage

OWL being a W3C standard for representing information on the Web. Thus, I extend previous work of coupling XACML with OWL ontologies [40] by providing a *unified* reasoning framework about that covers both XACML and OWL.

This chapter is organized as follows. In Section 6.1 the algorithm to map a XACML policy set to a Description Logic knowledge base is presented. The mapping covers XACML Policy Elements (PolicySet, Policy, Rule, Target, Request), along with Datatypes and Administrative Policies. Given this mapping, Section 6.2 describes how formal verification, policy compariosn and redundancy checking can be provided. In Section 6.3, the mapping is extended with support for OWL, which in turn provides a semantic extension of XACML (supporting data integration and rich policy models). Finally, Section 6.4 contains a discussion of limitations of the analysis framework as well as a comparison to other analysis approaches.

6.1 Mapping XACML

This section will provide details of the mapping of XACML policy elements to Description Logic (DL) concept expressions.

At the core of the mapping there are two translation functions. Given a DL KB K, decision type $\epsilon \in \{\texttt{Permit}, \texttt{Deny}, \texttt{Indeterminate}\}$, policy element $P \in \{\texttt{PolicySet}$, $\texttt{Policy}$, $\texttt{Rule}\}$, and `Target` element T, the translation functions are defined as:

$$\pi : \mathrm{T} \times \epsilon \Rightarrow \mathrm{C} \qquad\qquad \tau : \mathrm{P} \times \mathrm{K} \Rightarrow \mathrm{K}'$$

The function $\pi(\epsilon, \mathrm{T})$ takes an access decision and a target element T as input, and returns a concept expression C s.t. whenever the mapping of the access request is of type C, the original request will also yield the access decision ϵ when evaluated against T, and vice-versa. The translation function $\tau(\mathsf{P}, \mathsf{K})$ takes a policy element *P* and a DL KB K , and generates axioms in K to capture the semantics of *P*. More specifically, for each policy element P (`Rule`, `Policy` or `PolicySet`) three DL concepts are introduced: *Permit*-*P*, *Deny*-*P* and *Indeterminate*-*P*. Informally, whenever the mapping of the request is of type ϵ-*P*, that means that when evaluated against *P*, the original request will return ϵ, and vice-versa.

In the remainder of this section, I will present formal descriptions of π and τ. First, the translation function π will be discussed in detail; I will describe how a XACML access request and a `Target` are mapped to Description Logics. After that, the presentation will move on to τ and types of axioms added to the KB to capture the semantics of XACML combining algorithms. After these results for core XACML, I will discuss the more advanced parts of the language (administrative policies, datatypes, hierarchical resources) and their corresponding DL mapping. Note that from now on, unless specified differently, K is assumed to refer to the DL KB that we're mapping the XACML policies into.

6.1.1 Mapping `Requests`

XACML access requests represent a conjunction of attribute-value pairs. For illustrative purposes, an example of a request containing a single *subject-id* attribute and value is shown below:

```
<Request xmlns="urn:oasis:names:tc:xacml:3.0:schema:os"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
  <Attributes
    Category="...:tc:xacml:1.0:subject-category:access-subject">
    <Attribute AttributeId="...:tc:xacml:1.0:subject:subject-id"
      Issuer="med.example.com">
      <AttributeValue
        DataType=".../XMLSchema#string">Julius Hibbert
      </AttributeValue>
    </Attribute>
   </Attributes>
 </Request>
```

A XACML access request is mapped to a DL concept expression; details of the DL mapping are presented in Table 6.1. Note that no matching functions are allowed in requests, so the only data-type restriction used *type-equal*.

Syntax	Mapping π
RQ ::= (`Request` ATS*)	$\pi(\text{ATS}_1) \sqcap \ldots \sqcap \pi(\text{ATS}_n)$
ATS ::= (`Attributes` (AT content)* cat)	$\pi(\text{AT}_1, \text{cat}) \sqcap \ldots \sqcap \pi(\text{AT}_n, \text{cat})$
AT, cat ::= (`Attribute` AV* attr-ID Issuer)	$\exists r.\pi(\text{type}_{AV}\text{-equal, AV})$ where $r \in \mathcal{R}$ s.t. $name(r) = concat(\pi(\text{attr-ID}), \pi(\text{Issuer}), \pi(\text{cat}), \pi(\text{Type}_{AV}))$

Table 6.1: Mapping `Request` to a DL concept expression

This mapping is not complete because of the open world assumption present in Description Logics. The following additions to the concept expression corresponding to the request essentially close-off the information carried in a request:

1. To make sure that the request RQ is not augmented with additional attributes (apart from the ones it initially contains), for each attribute A that occurs in the policy P but not in RQ I add:

$$\pi(RQ) \leftarrow \pi(RQ) \sqcap \forall \pi(A).\bot$$

2. To make sure that the request RQ is not augmented with additional values for the attribute it already contains, for each attribute that occurs in RQ the following is added:

$$\pi(RQ) \leftarrow \pi(RQ) \sqcap =\text{nA}$$

, where n stands for the number of distinct attribute values for A that occur in RQ.

In a concise format, a full mapping of a Request RQ would be the following:

$$\begin{aligned}\pi(\text{RQ}) \equiv \bigsqcap_{AV \in RQ} (\exists \text{r}_{AV}.\pi(\text{type}_{AV}\text{-equal, AV})) \sqcap \bigsqcap_{AT \notin RQ,\ AT \in P} (\forall \text{r}_{AT}.\bot) \sqcap \\ \bigsqcap_{AT \in P,\ n = \#AV \ s.t.\ AV \in AT} (= n\text{r}_{AT})\end{aligned} \tag{6.1}$$

Given this mapping, to check whether a request RQ matches a target T, we only need to check whether $\mathsf{K} \models \{\pi(RQ) \sqsubseteq \pi(T)\}$ (equivalent to subsumption checking in Description Logics).

6.1.2 Mapping `Target`

`Target` represents the prerequisites that need to be satisfied by an access request for the policy element to apply - it is similar to the body of a rule. A `Target` element in XACML 3.0 is a conjunction of `DisjunctiveMatch` elements, where each `DisjunctiveMatch` is a disjunction of `ConjunctiveMatches`. A `Target` is matched if and only all `DisjunctiveMatches` are matched, and a `DisjunctiveMatch` is matched if at least one of its `ConjunctiveMatches` is matched. Finally, a `ConjunctiveMatch` is a (conjunctive) list of attribute-value pairs (`Match` elements).

Intuitively, I translate the `Target` element T to a DL concept expression C s.t. C captures all of the access requests that would match T. The main idea is that XACML attributes are mapped to DL roles (properties) and XACML attribute values are mapped to DL datatype values. Since in the formal semantics of XACML (discussed in Chapter 4), matching attributes depends on their datatype, id,issuer and category, in the mapping there is a one-to-one function that creates a unique DL role for each such combination of (datatype, id,issuer, category). Given this, attribute value pairs are mapped to existential restrictions – for example (*role Developer*) would be mapped to $\exists role.Developer$. Attribute-value pairs from different `ConjunctiveMatch` and `DisjunctiveMatch` are combined using the appropriate conjunctive and disjunctive DL constructs (resp. $\sqcap, \sqcup$).

The mapping function is split in two parts, depending on the access decision ϵ. Table 6.2 contains the semantics of $\pi(True, T)$ which covers the cases when $\mathrm{RQ}, \mathrm{T} \models$ `True`, which happens when a request RQ matches the a `Target` element T. Note that π(fcn, AV) maps the attribute value to a datatype expression in the logic depending on the matching function in XACML. The discussion of the datatype mapping warrants a separate section - which can be found in Section 6.1.7.

In Table 6.3 the mapping is presented for the cases when $\mathrm{RQ}, \mathrm{T} \models$ `Indeterminate`. According to the XACML semantics, the only case where `Indeterminate` can occur while matching a request against a `Target` is when there are no attribute values in the request that match the `AttributeDesignator`, and the `mustBePresent` attribute is true. Thus, in the mapping for indeterminate, a `Match` element is mapped as the following:

$$\pi(\texttt{Indeterminate},\ (\texttt{Match}\ (\mathrm{AV\ AD\ MatchFcn}))) = \forall \pi(\mathrm{AD}).\bot$$

The mapping will capture all requests that do not have a value for the attribute identified by AD. Also, note that in the case of an empty `Target`, $\top$ is used, which in DL represents the whole interpretation domain, so it will match any possible attribute value.

Syntax Elements E	Mapping π(True, E)
T ::= (`Target` DM*)	$\bigsqcap \pi(\text{True}, DM_i)$
DM ::= (`DisjunctiveMatch` CM$^+$)	$\bigsqcup \pi(\text{True}, CM_i)$
CM ::= (`ConjunctiveMatch` M$^+$)	$\bigsqcap \pi(\text{True}, M_i)$
M ::= (`Match` (AV AD fcn))	$\exists\pi(\text{AD}).\pi(\text{fcn}, \text{AV})$
AD ::= (`AttributeDesignator` Cat attr-ID Type Issuer$^?$ MustBePresent$^?$)	$r \in \mathcal{R}$ s.t. $name(r) = concat(\pi(\text{attr-ID}), \pi(\text{Issuer}), \pi(\text{cat}), \pi(\text{Type}))$

Table 6.2: Mapping access requests that match `Target` to a DL concept expression

Syntax Elements E	Mapping π(Indeterminate, E)
T ::= (`Target` DM*)	$\bigsqcup \pi(\text{Indeterminate}, DM_i)$
DM ::= (`DisjunctiveMatch` CM$^+$)	$\bigsqcap \pi(\text{Indeterminate}, CM_i)$
CM ::= (`ConjunctiveMatch` M$^+$)	$\bigsqcup \pi(\text{Indeterminate}, M_i)$
M ::= (`Match` (AV AD MatchFcn))	$\forall\pi(\text{AD}).\bot$
AD ::= (`AttributeDesignator` Cat attr-ID Type Issuer$^?$ MustBePresent$^?$)	$r \in \mathcal{R}$ s.t. $name(r) = concat(\pi(\text{attr-ID}), \pi(\text{Issuer}), \pi(\text{cat}), \pi(\text{Type}))$

Table 6.3: Mapping access requests that return `Indeterminate` when matched against `Target` to a DL concept expression

The next lemma states the correspondence between my DL mapping of `Target` elements and the proof-theoretic semantics from Chapter 4. In particular, the lemma shows that if the mapping of the `Request` π(RQ) is subsumed by the mapping of the `Target` element $\pi(T)$, then RQ will match T according to the proof-theoretic semantics (and vice-versa).

Lemma 6 For a Target element T, and a Request RQ:

$$\begin{aligned} \mathsf{K} &\models \pi(\text{RQ}) \sqsubseteq \pi(\texttt{True}, \text{T}) \Leftrightarrow \text{RQ}, \text{T} \models \texttt{True} \\ \mathsf{K} &\models \pi(\text{RQ}) \sqsubseteq \pi(\texttt{Indeterminate}, \text{T}) \Leftrightarrow \text{RQ}, \text{T} \models \texttt{Indeterminate} \end{aligned} \tag{6.2}$$

Proof. The proof of this lemma is in Appendix A. □

Lemma 7 For a `Target` element T, and a `Request` RQ:

$$\begin{aligned} \mathsf{K} &\models \pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{True}, \text{T}) \Leftrightarrow \text{RQ}, \text{T} \not\models \texttt{True} \\ \mathsf{K} &\models \pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{Indeterminate}, \text{T}) \Leftrightarrow \text{RQ}, \text{T} \not\models \texttt{Indeterminate} \end{aligned} \tag{6.3}$$

Proof.

If case We know that $\pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{True}, \text{T})$. This proof is done by contradiction: assume that if $\pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{True}, \text{T})$, then $\text{RQ}, \text{T} \models \texttt{True}$. However, in that case, according to Lemma 6, for the same request the following will hold as well: $\pi(\text{RQ}) \sqsubseteq \pi(\texttt{True}, \text{T})$. Note that $\pi(\text{RQ})$ cannot be subsumed by both $\pi(\texttt{True}, \text{T})$ and $\neg\pi(\texttt{True}, \text{T})$ at the same time since I have assumed that $\pi(\text{RQ})$ is satisfiable, so we have reached a contradiction. Thus,

$$\mathsf{K} \models \pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{True}, \text{T}) \Leftrightarrow \text{RQ}, \text{T} \not\models \texttt{True}$$

It can be shown for `Indeterminate` in the same manner.

Else case. Else case is more involved, since in DL KB $\not\models \pi(\text{RQ}) \sqsubseteq \pi(\texttt{True}, \text{T})$ (which can be easily shown) is not the same as $\mathsf{K} \models \pi(\text{RQ}) \sqsubseteq \neg\pi(\texttt{True}, \text{T})$ (which needs to be shown). The proof can be done by induction on the XACML deduction rules in Section 4.2. The full proof is available in the Appendix.

□

6.1.3 Mapping XACML `Rules`

For each XACML `Rule` *R*, I introduce a *Permit-R*, *Deny-R* and *Indeterminate-R* concept. The general idea is that for a request RQ, if $\pi(RQ)$ is of type *Permit-R*, then RQ would yield a `Permit` when evaluated against R (and vice-versa).

Rules in XACML consist of an identifier (ID), `Target` representing the types of request the rule matches, `Condition` representing additional conditions that need to be satisfied for the rule to fire and an `Effect`, representing the head of the rule (can be either `Permit` or `Deny`). The *Permit* and *Deny* concepts for a rule depend on the effect of the rule itself. For example, for a rule *R* that yields a `Permit`, the *Deny-R* concept will be equal to $\bot$, because there cannot be an access request where *R* will yield a `Deny`. Assuming there there is no `Condition` element, the *Permit-P* concept for *R* will be equal to the mapping of the `Target` of *R*.

Definition 5 Mapping a `Rule` Element.

For a rule (Rule ID T `Permit`), $\tau(K, R)$ is defined as follows:

$$\tau(\mathsf{K}, (\text{Rule ID T } \texttt{Permit})) = \mathsf{K} \cup \left\{ \begin{array}{l} \textit{Indeterminate-ID} \equiv \pi(\texttt{Indeterminate}, T_R) \\ \\ \textit{Permit-ID} \equiv \pi(\texttt{True}, T_R) \\ \\ \textit{Deny-ID} \equiv \bot \end{array} \right\}$$

Analogously, for a rule (Rule ID T Deny), $\tau(K, R)$ is defined as:

$$\tau(\mathsf{K}, (\text{Rule ID T }\texttt{Deny})) = \mathsf{K} \cup \left\{ \begin{array}{l} \textit{Indeterminate-ID} \equiv \pi(\texttt{Indeterminate}, T_R) \\ \\ \textit{Permit-ID} \equiv \bot \\ \\ \textit{Deny-ID} \equiv \pi(\texttt{True}, T_R) \end{array} \right\}$$

Example 6.1.3.1 I present here a running example (similar to Chapter 1) that will be used to illustrate the main concepts underlying this mapping. In this toy example, initially there are two security roles, *Manager* and *Developer*; one resource: *Report*; and two actions: *read*, *write*. The root policy set contains two policy sets which are combined using `First-applicable` combining algorithm. Assume there are no complicated matching functions; in each `Target`, the matching function is simply *string-equal*. The policy is presented in graphical form in Figure 6.1.

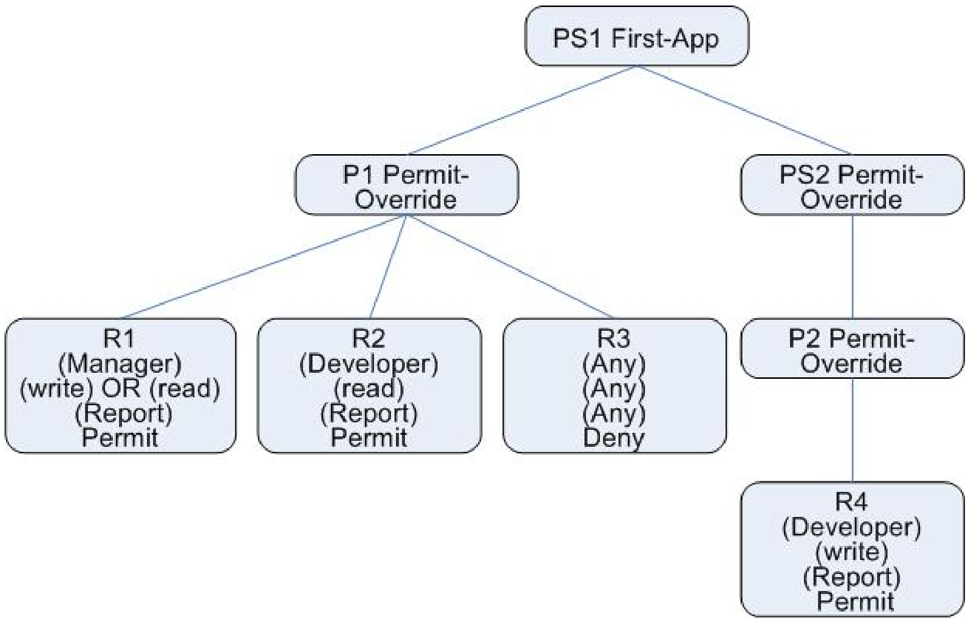

PS1 First-App
P1 Permit-Override
PS2 Permit-Override
R1 (Manager) (write) OR (read) (Report) Permit
R2 (Developer) (read) (Report) Permit
R3 (Any) (Any) (Any) Deny
P2 Permit-Override
R4 (Developer) (write) (Report) Permit

Figure 6.1: Example Policy

The `Rules` in this example are mapped to:

$$Permit\text{-}R_1 \equiv \exists role.Manager \sqcap \exists resource.Report \sqcap (\exists action.read \sqcup \exists action.write)$$

$$Deny\text{-}R_1 \equiv \bot \qquad Indeterminate\text{-}R_1 \equiv \bot$$

$$Permit\text{-}R_2 \equiv \exists role.Developer \sqcap \exists action.read \sqcap \exists resource.Report$$

$$Deny\text{-}R_2 \equiv \bot \quad Indeterminate\text{-}R_2 \equiv \bot$$

$$Permit\text{-}R_3 \equiv \bot \quad Indeterminate\text{-}R_3 \equiv \bot$$

$$Deny\text{-}R_3 \equiv \top$$

$$Permit\text{-}R_4 \equiv \exists role.Developer \sqcap \exists action\text{-}type.write \sqcap \exists resource.Report$$

$$Deny\text{-}R_4 \equiv \bot \quad Indeterminate\text{-}R_4 \equiv \bot$$

For brevity, this example is slightly simplified by omitting the data-type, category and issuer information for each attribute.

Lemma 8 For a `Rule` R and `Request` RQ:

$$\mathsf{RQ}, \mathsf{R} \models \mathtt{Effect} \Leftrightarrow \mathsf{K} \models \pi(\mathsf{RQ}) \sqsubseteq \mathtt{Effect}\text{-}\mathsf{R} \tag{6.4}$$

where $\mathtt{Effect} \in \{\mathtt{Permit}, \mathtt{Deny}, \mathtt{Indeterminate}\}$.

Proof. Proof can be found in the Appendix. □

Lemma 9 For a `Rule` R and `Request` RQ:

$$\mathrm{RQ}, \mathrm{R} \not\models \texttt{Effect} \Leftrightarrow \mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\texttt{Effect}\text{-R} \tag{6.5}$$

where $\texttt{Effect} \in \{\texttt{Permit}, \texttt{Deny}, \texttt{Indeterminate}\}$.

Proof.

If case This proof is done by contradiction: assume that if $\pi(\mathrm{RQ}) \sqsubseteq \neg\texttt{Effect}\text{-R}$, then $\mathrm{RQ}, \mathrm{R} \models \texttt{Effect}$. However, in that case, according to Lemma 6, for the same request RQ the following will hold: $\pi(\mathrm{RQ}) \sqsubseteq \texttt{Effect}\text{-R}$. Since RQ cannot be of type $\texttt{Effect}\text{-R}$ and $\neg\texttt{Effect}\text{-R}$ at the same time, we have reached a contradiction. Thus,

$$\mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\texttt{Effect}\text{-R} \Rightarrow \mathrm{RQ}, \mathrm{R} \not\models \texttt{Effect}$$

Else case. Else case is more involved; the proof is done by structural induction on the XACML semantics rules in Section 4.2, and is available in the Appendix. □

6.1.4 Mapping `Policies`

A `Policy` contains a `Target` element, a collection of rules, and a rule combining algorithm that specified how the access decision of the children `Rules` are to be combined. For a `Policy` P, the `Permit` and `Deny` concepts have to take into account the `Target`, and in addition the Permit- and Deny- concepts of P's children. This is because the access decision a `Policy` yields depends on the `Target` and the results of its children's decisions.

Intuitively, a Permit-P is matched by a request only if P's `Target` is matched and at least one of its children's `Permit` concepts is matched as well. However, depending on the overriding algorithm, for each child policy element, we might need to make sure that whenever it yields a `Permit` it will not be overridden by a `Deny` from a different child element. Given these considerations, the mapping function τ for XACML `Policies` is developed based on the semantics of the rule-combining algorithms (see Tables 4.8, 4.9, 4.10).

In this section, first I discuss the case of `Permit-Overrides` rule combining algorithm: P = (`Policy` `Permit-Overrides` T $R_1, \ldots, R_n$) . The logic axioms that result from the mapping and are added to K are presented in the definition below. The concept Permit-P has somewhat obvious semantics, since we only need to infer that at least one of the child `Rules` returns a `Permit`. However, the axiom for Deny-P is not as straightforward, since if a rule returns an `Indeterminate`, and the effect of the rule is `Permit`, then according to the XACML semantics, that rule will override all sibling rules that returned a `Deny`.

Definition 6 Mapping Axioms for P = (`Policy` `Permit-Overrides` T $R_1, \ldots, R_n$)

$$
\begin{aligned}
\text{Permit-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Permit-R}_i \right) \\
\text{Deny-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Deny-R}_i \sqcap \left(\bigsqcap_{R_j \in P} \neg\text{Permit-R}_j \right) \sqcap \left(\bigsqcap_{R_k \in P,\ \text{Effect}_{R_k} = \text{Permit}} \neg\text{Indet-R}_k \right) \right) \\
\text{Indet-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P, \text{Effect}_{R_i} = \text{Permit}} \text{Indet-R}_i \sqcap \left(\bigsqcap_{R_j \in P} \neg\text{Permit-R}_j \right) \right)
\end{aligned}
\tag{6.6}
$$

For `Deny-Overrides`, the axioms for the mapping concepts are very similar as in the `Permit-Overrides` case (it is enough simply to swap `Permit` and `Deny` in the definitions).

Definition 7 Mapping Axioms for P = (`Policy` `Deny-Overrides` T $\text{R}_1, \ldots, \text{R}_n$)

$$
\begin{aligned}
\text{Deny-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Deny-R}_i \right) \\
\text{Permit-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Permit-R}_i \sqcap \left(\bigsqcap_{R_j \in P} \neg\text{Deny-R}_j \right) \sqcap \left(\bigsqcap_{R_k \in P,\ \text{Effect}_{R_k} = \text{Deny}} \neg\text{Indet-R}_k \right) \right) \\
\text{Indet-P} &\equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P, \text{Effect}_{R_i} = \text{Deny}} \text{Indet-R}_i \sqcap \left(\bigsqcap_{R_j \in P} \neg\text{Deny-R}_j \right) \right)
\end{aligned}
\tag{6.7}
$$

For `First-Applicable`, the single axiom states that, for some access decision α made by a rule R_i, the parent policy will return that access decision only if all of the rules R_j *prior* to R_i did not not apply.

Definition 8 Mapping Axioms for P = (`Policy` `First-Applicable` T R$_1, \ldots,$ R$_n$)

$$\text{Effect-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Effect-R}_i \sqcap \left(\bigsqcap_{R_j \in P} \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indet-R}_j \right)\right)$$
$$\textit{where } i > j, \ \text{Effect} \in \{\text{Permit}, \text{Deny}, \text{Indet}\} \tag{6.8}$$

For `Only-One-Applicable`, the only way a policy to return a `Permit`(resp. `Deny`) is for *exactly* one rule to apply and return `Permit` (resp. `Deny`). In the case when two or more rules apply, `Indeterminate` is to be returned. Also, if at least one rule returns `Indeterminate`, then the parent policy will return `Indeterminate` as well.

Definition 9 Mapping Axioms for P = (`Policy` `Only-One-Applicable` T R$_1, \ldots,$ R$_n$)

$$\begin{aligned}
&\text{Indet-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Indet-R}_i \right) \\
&\text{Indet-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P,\ R_j \in P} ((\text{Permit-R}_i \sqcup \text{Deny-R}_i) \sqcap (\text{Permit-R}_j \sqcup \text{Deny-R}_j)) \right) \\
&\qquad \textit{where } i \neq j \\
&\text{Permit-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Permit-R}_i \sqcap \left(\bigsqcap_{R_j \in P,\ j \neq i} \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indet-R}_j \right)\right) \\
&\text{Deny-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup_{R_i \in P} \text{Deny-R}_i \sqcap \left(\bigsqcap_{R_j \in P,\ j \neq i} \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indet-R}_j \right)\right)
\end{aligned} \tag{6.9}$$

Lemma 10 For a $\mathtt{Policy}$ P and $\mathtt{Request}$ RQ:

$$\mathsf{RQ}, \mathsf{P} \models \mathtt{Effect} \;\Leftrightarrow\; \mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \pi(\mathtt{Effect}, \mathrm{P}) \tag{6.10}$$

Proof. Proof can be found in the Appendix. □

Lemma 11 For a $\mathtt{Policy}$ P and $\mathtt{Request}$ RQ:

$$\mathsf{RQ}, \mathsf{P} \not\models \mathtt{Effect} \;\Leftrightarrow\; \mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\pi(\mathtt{Effect}, \mathrm{P}) \tag{6.11}$$

where $\mathtt{Effect} \in \{\mathtt{Permit}, \mathtt{Deny}, \mathtt{Indeterminate}\}$.

Proof.

If case This proof is done by contradiction: assume that if $\pi(\mathrm{RQ}) : \neg\mathtt{Effect}\text{-P}$, then $\mathsf{RQ}, \mathsf{P} \models \mathtt{Effect}$. However, in that case, according to Lemma 10, for the same request RQ the following will hold: $\pi(\mathrm{RQ}) : \mathtt{Effect}\text{-P}$. Since RQ cannot be of type $\mathtt{Effect}\text{-P}$ and $\neg\mathtt{Effect}\text{-P}$ at the same time, we have reached a contradiction. Thus,

$$\mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\mathtt{Effect}\text{-P} \Rightarrow \mathsf{RQ}, \mathsf{P} \not\models \mathtt{Effect}$$

Else case. Proof is available in the Appendix. □

6.1.5 Mapping `PolicySets`

The `Permit` and `Deny` concepts for `PolicySets` are somewhat simpler from the `Policy` ones. In this section, I present the mappings for `DenyOverrides` and `Permit-Overrides` only, since the semantics for the other two combining algorithms is same as in the `Policy` case.

First, I will discuss the case of P = (`PolicySet` `Permit-Overrides` T $\mathrm{P}_1, \ldots, \mathrm{P}_n$). The mapping axioms that are to be added to K are presented below.

Definition 10 Mapping Axioms for PS = (`PolicySet` `Permit-Overrides` T $\mathrm{P}_1, \ldots, \mathrm{P}_n$)

$$\begin{aligned}
\text{Permit-PS} &\equiv \pi(T) \sqcap \left(\bigsqcup_{P_i \in PS} \text{Permit-P}_i \right) \\
\text{Deny-PS} &\equiv \pi(T) \sqcap \left(\bigsqcup_{P_i \in PS} \text{Deny-P}_i \sqcap \left(\bigsqcap_{P_j \in PS} \neg\text{Permit-P}_j \right) \right)
\end{aligned} \tag{6.12}$$

The axioms for `Deny-Overrides` are very similar with `Permit-Overrides`; only difference is that when a child element returns an `Indeterminate`, the parent must return a `Deny` access decision.

Definition 11 Mapping Axioms for PS = (`PolicySet` `Deny-Overrides` T $\mathrm{P}_1, \ldots, \mathrm{P}_n$)

$$\begin{aligned}
\text{Deny-PS} &\equiv \pi(T) \sqcap \left(\bigsqcup_{P_i \in PS} \text{Deny-R}_i \sqcup \text{Indet-R}_i \right) \\
\text{Permit-PS} &\equiv \pi(T) \sqcap \left(\bigsqcup_{P_i \in PS} \text{Permit-P}_i \sqcap \left(\bigsqcap_{P_j \in PS} \neg\text{Deny-P}_j \sqcap \neg\text{Indet-P}_j \right) \right)
\end{aligned} \tag{6.13}$$

Example 6.1.5.1 The policies and policy sets in our running example are mapped below. Note that a) for brevity, I have assumed that $\forall P : Indeterminate - P \equiv \bot$, and b) the targets of all of the policies and policysets in the examples are empty, thus $\pi(\mathrm{T}, \mathrm{True}) \equiv \top$ for all of them.

$$\begin{aligned}
\mathit{Permit\text{-}PS}\,1 &\equiv \top \sqcap \mathit{Permit\text{-}P}1 \sqcup (\mathit{Permit\text{-}PS}\,2 \sqcap \neg \mathit{Deny\text{-}P}1)\\
\mathit{Deny\text{-}PS}\,1 &\equiv \top \sqcap \mathit{Deny\text{-}P}1 \sqcup (\mathit{Deny\text{-}PS}\,2 \sqcap \neg \mathit{Permit\text{-}P}1)\\
\mathit{Permit\text{-}P}1 &\equiv \top \sqcap \mathit{Permit\text{-}R}1 \sqcup \mathit{Permit\text{-}R}2\\
\mathit{Deny\text{-}P}1 &\equiv \top \sqcap \mathit{Deny\text{-}R}3 \sqcap \neg(\mathit{Permit\text{-}R}1 \sqcup \mathit{Permit\text{-}R}2)\\
\mathit{Permit\text{-}PS}\,2 &\equiv \top \sqcap \mathit{Permit\text{-}P}2\\
\mathit{Deny\text{-}PS}\,2 &\equiv \top \sqcap \mathit{Deny\text{-}P}2\\
\mathit{Permit\text{-}P}2 &\equiv \top \sqcap \mathit{Permit\text{-}R}2\\
\mathit{Deny\text{-}P}2 &\equiv \top \sqcap \mathit{Deny\text{-}R}4
\end{aligned}$$

Lemma 12 For a `PolicySet` PS and `Request` RQ:

$$\mathrm{RQ}, \mathrm{PS} \models \texttt{Effect} \;\Leftrightarrow \mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \pi(\texttt{Effect}, \mathrm{PS}) \tag{6.14}$$

Proof. Proof can be found in the Appendix. □

Lemma 13 For a $\texttt{PolicySet}$ PS and $\texttt{Request}$ RQ:

$$\mathrm{RQ}, \mathrm{PS} \not\models \texttt{Effect} \Leftrightarrow \mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\pi(\texttt{Effect}, \mathrm{PS}) \tag{6.15}$$

where $\texttt{Effect} \in \{\texttt{Permit}, \texttt{Deny}, \texttt{Indeterminate}\}$.

Proof.

If case This proof is done by contradiction: assume that if $\pi(\mathrm{RQ}) \sqsubseteq \neg\texttt{Effect}\text{-PS}$, then $\mathrm{RQ}, \mathrm{PS} \models \texttt{Effect}$. However, in that case, according to Lemma 12, for the same request RQ the following will hold: $\pi(\mathrm{RQ}) \sqsubseteq \texttt{Effect}\text{-PS}$. Since RQ cannot be of type $\texttt{Effect}\text{-PS}$ and $\neg\texttt{Effect}\text{-PS}$ at the same time, we have reached a contradiction. Thus,

$$\mathsf{K} \models \pi(\mathrm{RQ}) \sqsubseteq \neg\texttt{Effect}\text{-PS} \Rightarrow \mathrm{RQ}, \mathrm{PS} \not\models \texttt{Effect}$$

Else case. Else case is more involved; the proof is done by structural induction on the XACML deduction rules in Section 4.2, and is available in the Appendix. □

6.1.6 Mapping Administrative XACML

The Administrative Policy Profile[113] adds support for administrative/delegation policies to XACML 3.0. In this version of XACML, after a policy element yields an access decision, that decision might need to be authorized by an administrative policy. In contrast, in previous versions of XACML all access decisions were assumed to be authorized by default.

An overview of administrative policies in XACML is provided in Section 2.1.3, and the formal semantics is given in Chapter 4. In this section, I will discuss their translation to DL. The mapping uses the same data structure, a *reduction graph*, as the official XACML Administrative Profile Specification.

Definition 12 Reduction Graph A reduction graph is a directed graph $\mathcal{G} = \{V, E\}$ where each $v \in V$ corresponds to a XACML `Policy` and has four labels: $target$, $issuer$, $delegate$ and $trusted$. For a node v, $v.target$ corresponds to the situation (target) to which v applies, $v.issuer$ holds the attributes that describe the issuer of v and $v.delegate$ contains information about the issuer to whom v can be delegated. In addition, $v.trusted$ is a boolean field that indicates if v was issued by a trusted issuer.

For two nodes v, w that correspond to two sibling policy elements, a directed edge $(v, w) \in E$ exists if and only if $v.issuer \sqsubseteq w.delegate$.

Intuitively, the edges in $\mathcal{G}$ indicate whether the issuer and delegate of two policy nodes are compatible. If $(v, w) \notin E$, that means it is not possible to reduce an administrative request from v to w. `Target` elements of each node in the reduction graph are mapped to DL concept expressions in a similar manner as `Targets` of access policies (presented in Section 5.1). In cases when the node is an access policy, the delegate field is empty.

6.1.6.1 Extending the XACML Mapping

Permit-P, Deny-P and Indet-P concept definitions are augmented with the constraints from the administrative policies in the following manner:

$$\textit{Permit-P-Auth} \equiv \textit{Permit-P} \sqcap \textit{get\_admin\_expr}(P, \mathcal{G})$$

$$\textit{Deny-P-Auth} \equiv \textit{Deny-P} \sqcap \textit{get\_admin\_expr}(P, \mathcal{G})$$

The function *get_admin_expr* generates a concept expression that is a disjunction of all unique reduction paths from the current policy (P) to a policy with a trusted issuer. Since according to the official semantics of XACML, this is the only way a policy decision can be authorized, the result of *get_admin_expr* is added as a constraint to the existing Permit-P and Deny-P concepts.

Details of *get_admin_expr* are shown in Algorithm 1. The function takes the reduction graph $\mathcal{G}$ and the node corresponding to policy being authorized P as input, and generates a disjunction of all unique possible paths from P to a trusted policy, following the XACML semantics. The algorithm that generates the paths (lines 5-11) performs a depth first search on the reduction graph. It starts with the policy being authorized, and tries to build a path consisting of administrative policies to a *trusted* node, accumulating constraints along the way. These constraints come from the $Target$ elements of each administrative policy, since administrative policies can restrict the cases where they are applied. When the accumulated constraints become unsatisfiable at a point in the path, the algorithm backtracks and tries a different node. All unique paths are added to the *Permit-P-Auth* or *Deny-P-Auth* concept.

Algorithm 1 *get_admin_expr*$(node, \mathcal{G})$

Input:

node: node in graph corresponding to input policy

$\mathcal{G}$: reduction graph of composed of *node*'s sibling policies

Output:

b: returns an expression that corresponds to all possible ways *node* can be authorized

```
1:  $paths \leftarrow []$
2:  if $node$ has trusted issuer then
3:      $paths$.add($constraint$)                ▹ found a new path, remember constraint
4:  else
5:      for nbor in n.getnbors() do
6:          if !$nbor$.getVisited() and $constraint \sqcap nbor.target \not\models \perp$ then
7:              $nbor$.setVisited(true)
8:              $searchGraph(nbor, constraint \sqcap nbor.target, paths)$
9:              $nbor$.setVisited(false)
10:         end if
11:     end for
12: end if
13: $result \leftarrow \perp$
14: for expr in paths do
15:         $result \leftarrow result \sqcup expr$
16: end for
17: return $result$
```

6.1.7 XACML Datatypes and Functions

XACML supports the XML schema data-types and in addition it defines four datatypes of its own: ipAddress, dnsName, rfc822Name, x500Name. As part of the processing model, it also supports a wide variety of functions over these datatypes, ranging from comparison, arithmetic, regular expression and Xpath matching to higher order map functions. In this section I will discuss how these datatypes and functions are handled by this XACML-to-DL mapping.

Datatype Support in Description Logics Horrocks and Sattler [64] presented an approach of combining DLs and types systems that allows for deriving new datatypes from existing ones. DLs already provide support for the various datatype comparison functions in XACML (such as datetime-greater-than) by way of *user-defined* XML schema datatypes. This support is currently being standardized in OWL 1.1 [105] and is implemented in Fact++ and Pellet. User-defined (restricted) datatypes are supported through the datatypeRestriction constructor, which creates a restricted range by applying a facet to a particular data range. Built-in XML schema facets include: length, minLength, maxLength, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, totalDigits, and fractionDigits. These facets cover the numeric and non-numeric comparison functions in the XACML specification (Appendix A.3 in [113]).

Since DL reasoners such as Pellet[109] and Fact++[120] already have built-in reasoning support for XML schema datatypes, mapping the encountered datatypes attribute values in XACML is trivial: for each datatype attribute value I create the same datatype value in the DL KB. Given this, the mapping supports the following XML schema datatypes: string, boolean, integer, double, time, date, datetime, anyURI, hexBinary and base64Binary.

More detailed information about the mapping is presented in Table 6.1.7. For brevity, the table only contains integer datatypes; comparison functions involving the other XML schema datatypes can be mapped in the same manner. Since the OWL functional syntax is already standardized for user-defined datatypes, it is used instead of a more concise DL syntax. For a XACML attribute matching function fcn, attribute a, and value v, I map a to a DL *datatype* role and map v to a datatype value (with the appropriate type) in the KB.

In addition to the above unary datatype predicates, some DL reasoners also support

Matching function fcn	$\pi(\text{AV}, \text{fcn})$
type-equal	DatatypeRestriction (*type* value$_{AV}$)
type-greater-than	DatatypeRestriction (*type* minExclusive value$_{AV}$)
type-greater-than-or-equal	DatatypeRestriction (*type* minInclusive value$_{AV}$)
type-less	DatatypeRestriction (*type* maxExclusive value$_{AV}$)
type-less-than-or-equal	DatatypeRestriction (*type* maxInclusive value$_{AV}$)
type-regexp-match	DatatypeRestriction (*type* pattern value$_{AV}$)
xpath-node-match	DatatypeRestriction (xpath-expression pattern value$_{AV}$)

Table 6.4: Mapping matching functions.

more advanced datatype functions such as n-ary datatype predicates. For example, the DL reasoner RacerPro [124] supports natural numbers, integers, reals, complex numbers and strings. The types of predicates supported are shown in Table 6.1.7. The predicates below can capture a subset of XACML's functions; e.g., linear inequality predicates cover the XACML integer comparison and arithmetic functions.

Datatype Domain	Predicates
$\mathbb{N}$	linear inequations with order constraints and integer coefficients
$\mathbb{Z}$	interval constraints
$\mathbb{R}$	linear inequations with order constraints and rational coefficients
Strings	equiality and inequality

Table 6.5: Predicates supported by Racer (table taken from RacerPro user guide [82]).

Example 6.1.7.1 Mapping the XPath-Node-Match Function.

The *xpath-node-match* function is used to write policies that apply to multiple nodes. This function is used as part of the `ResourceMatch` object, as shown below:

```
<ResourceMatch MatchId="...:xpath-node-match">
 <AttributeValue DataType="...#string">
  /md:record
 </AttributeValue>
 <ResourceAttributeDesignator
  AttributeId="...:resource:xpath"
```

```
   DataType="...#string"/>
  </ResourceMatch>
```

In the above example, both the attribute value of resource:xpath and "/md:record" are treated as XPath expressions; the matching succeeds iff the at least one of the nodes selected by "/md:record" matches at least one node selected by the ResourceAttributeDesignator. A resource match element with an attribute value of AV, an attribute ID of $AttrID$ and an xpath-node-match function would be mapped to $\exists\pi(AttrID).\pi(AV)$, where $\pi(AttrID)$ generates a DL role to correspond to the attribute. To map AV, I generate a unary predicate in the XPath concrete domain.

Note that I am able to use XPath data-types in the static analysis framework since it was shown in [55] that the XPath fragment covered by $XPATH$ is closed under negation and the problem of satisfiability of a conjunction of predicates is decidable. A prototype XPath reasoner (developed by Geneves et al. [55]) was used as a data-type reasoner for the purpose of my analysis framework.

6.1.7.1 AttributeSelector

Another element of XACML that warrants discussion in this section is the `AttributeSelector`, which selects attribute values from a request (similarly to an `AttributeDesignator`). The main difference is that the `AttributeSelector` uses an XPath expression to locate the attribute value in the request. Consider the following example:

```
<Resource>
  <ResourceMatch
   MatchId="...:string-equal">
```

```
    <AttributeValue
       DataType="...#string">
       application=webservicesJwsSimpleEar,
       contextPath=/jws_basic_simple,
       webService=SimpleSoapPort
    </AttributeValue>
    <AttributeSelector
     RequestContextPath="//md:patient-number/text()"
     DataType="...#string" />
   </ResourceMatch>
</Resource>
```

In this case, the expression in `RequestContextPath` is applied against the XML content element that is piggybacked on the access request and a bag of values is returned. The match is successful if at least one of the returned values matches the above attribute value.

A resource match element with an attribute selector (with an xpath expresion X) and an attribute value AV is mapped to the following DL expression:

$$\exists\pi(\mathit{Attr\text{-}ID}).P_{AV},$$

which looks the same as the mapping for xpath-node-match. However, note that there might be some interaction (e.g., subsumption, disjointness) between the different xpath expressions used as in the attribute selectors. As an example, consider the following two attribute selectors:

```
    <AttributeSelector RequestContextPath="/descendant::Role"
     DataType="...#string" />

    <AttributeSelector RequestContextPath="/descendant-or-self::Role"
     DataType="...#string" />
```

In this case, the first expression selects all descendants of the root node that are of type Role, whereas the second expression will select all Role descendants of root, and additionally the root itself, if it is a Role node. The result set of the second expression will always subsume the result set of the first one - this type of relationship will not be captured by the mapping described above.

To capture these subsumption relations between Attribute Selectors, I pre-process the policy set, and for each pair of Selectors I compare their XPath expressions using an XPath reasoner [55]. Then, if there is a subsumption, i.e., the expressions in AS_1 and AS_2 subsume each other, I add the following axiom in the DL KB:

$$\pi(AS_1) \sqsubseteq \pi(AS_2)$$

6.2 Services

This section will discuss the analysis services provided by the DL mapping described in the previous section.

6.2.1 Formal Verification

Formal verification is the most commonly offered analysis service for access policies. In most previous work, the security property to be verified is specified programmatically using the particular analysis tool's API, thus requiring users to be familiar with the internals of the tool. My approach, on the other hand, allows users to specify their security properties in XACML and also presents the verification results back in XACML. To accomplish this, the verification service is exposed as an adaptation of the well known software engineering technique of unit testing.

Traditional unit testing applied to policies amounts to users creating an access request (the unit test) and specifying its expected value (*Permit*, *Deny*, *NotApplicable* or *Indeterminate*). Then, whenever changes are made to the access policy, all of the test requests are run against a XACML engine to make sure no bugs (security holes) are introduced. However, writing such unit tests in this manner is tedious and error-prone, since it is difficult to think of all possible conditions that need to be tested. Consider our running example (Figure 1.1 in Chapter 1): it could happen that a user logs in with both a *developer* and a *manager* role – activating both of these grants him *write* access to *reports*. Since the unit test I used in the example in Chapter 1 only tests for a request containing the *developer* role by itself, this violation will not be caught.

A verification-based approach to testing overcomes the above limitations. Instead of taking the access request literally and performing a shallow test based on the explicitly mentioned attributes in the request, the verification approach proposed in this thesis attempts to build a logic model where the attributes in the original request produce a test failure. While building this model the DL reasoner explores all possible combinations of *additional* attributes in the request that could lead to a test failure. The test condition holds only when such a model cannot be built.

The input for this service consists of a XACML policy file *P* (for the access policy being tested), a XACML file that contains the test condition *T*, and a string denoting the type of test condition. Supported types of properties are *alwaysPermit*, *alwaysDeny*, *neverPermit* and *neverDeny* - these are similar to assertions for unit tests. Details on how these are reduced to entailment checking in Description Logics are presented in Table 6.2.1. If the policy fails, output is in the form of a XACML access request, which consists of the counter example that brings about the test failure.

Type of Property	Entailment Check
alwaysPermit	$\pi(T.target) \sqcap \neg Permit\text{-}P \models \perp$
alwaysDeny	$\pi(T.target) \sqcap \neg Deny\text{-}P \models \perp$
neverPermit	$\pi(T.target) \sqcap Permit\text{-}P \models \perp$
neverDeny	$\pi(T.target) \sqcap Deny\text{-}P \models \perp$

Table 6.6: **Mappings of Common Types of Policy Tests**. *T* refers to the test policy (containing the security property) and *P* refers to the top level policy set. Property holds if and only if the entailment holds, i.e., the concept expression is *not* satisfiable.

6.2.1.1 Tracing

In cases when the test fails, the analysis framework extracts the counter example directly from the logic model, maps it back to XACML and presents it as an access request. The procedure that extracts a counter-example XACML policy from a DL concept expression is shown in Algorithm 2.

However, with large policy sets, it can be difficult to find out exactly which policy elements were responsible for the error. For this purpose, I provide a 'stack trace' output of every policy that was fired while generating the counter example. Generating the trace is straightforward, assuming the counter example XACML request is available - the request is run against the policy using a XACML engine (Sun's reference implementation[1] is used), and the access decisions at each policy element are stored.

[1] `http://sunxacml.sourceforge.net`

Algorithm 2 *extract_request*(x)

Input:
x: individual used generated to check satisfiability of concept

Output:
L_{AT}: list of attributes and values that represent a valid XACML `Request`

```
1: if type expressions of the form $\exists P.Q \in \mathcal{L}(x)$ then
2:     for each $\exists P.Q \in \mathcal{L}(x)$ do
3:         $\mathcal{P} = getAttribute(P)$    ▷ from the name of the DL role $P$ we retrieve the important attribute information:category, issuer, ID and datatype
4:         $\mathcal{V} = getValue(Q)$    ▷ if Q is a datatype predicate that maps to multiple values, then a value is randomly selected from the value space characterized by Q
5:         $L_{AT} = L_{AT} \cup (\mathcal{P}, \mathcal{V})$
6:     end for
7: else    ▷ no $\exists P.Q$ occur in concept expression
8:     Select a DL role P corresponding to a attribute in the policy s.t. $\mathcal{L}(x) \cap \exists P.Q$ is satisfiable
9:     $\mathcal{P} = getAttribute(P)$    ▷ from the name of the DL role $P$ we retrieve the important attribute information:category, issuer, ID and datatype
10:    $\mathcal{V} = getValue(Q)$    ▷ if Q is a datatype predicate that maps to multiple values, then a value is randomly selected from the value space characterized by Q
11:    $L_{AT} = L_{AT} \cup (\mathcal{P}, \mathcal{V})$
12: end if
13: return $L_{AT}$
```

6.2.2 Policy Comparison

The *Permit-P* and *Deny-P* concepts defined previously allow us to easily compare the behaviors of two policies. For example, we can check for policy *subsumption*: P_2 subsumes P_1 iff if whenever P_1 produces access decision α, P_2 also yields the same access decision. We can restrict our attention to *Permit*, *Deny* or both. In my framework, policy subsumption is reduced to checking subsumption between the *Permit* and/or *Deny* concepts. For example, to check if P_2 subsumes P_1 w.r.t `Permit`, we ask the DL reasoner if $\mathsf{K} \models$ *Permit*-$P_1 \sqsubseteq$ *Permit*-P_2.

To illustrate the service, consider adding a new role , *LeadDeveloper*, to our running example. The updated policy now contains an additional `Rule` (R_3):

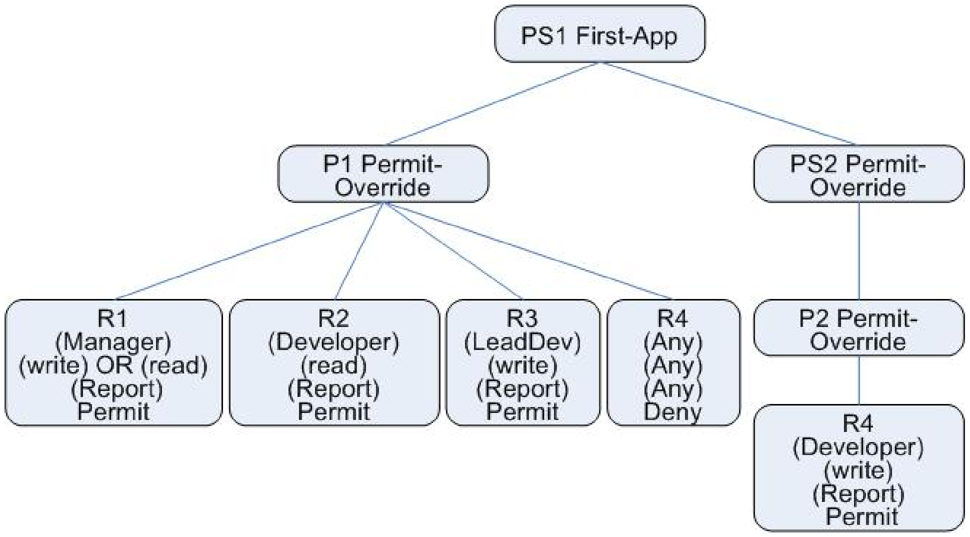

PS1 First-App
P1 Permit-Override
PS2 Permit-Override
R1 (Manager) (write) OR (read) (Report) Permit
R2 (Developer) (read) (Report) Permit
R3 (LeadDev) (write) (Report) Permit
R4 (Any) (Any) (Any) Deny
P2 Permit-Override
R4 (Developer) (write) (Report) Permit

Figure 6.2: Updated Policy (with LeadDev role)

To check whether we have given any unintended access to the other roles, we use the policy subsumption algorithm, that is, we generate the following concept expressions: Permit-PS$_{old}$, Deny-PS$_{old}$ and Permit-PS$_{new}$, Deny-PS$_{new}$. Subsumption holds only if both of the following hold:

$$Permit\text{-}PS_{old} \sqsubseteq Permit\text{-}PS_{new}$$

$$Deny\text{-}PS_{old} \sqsubseteq Deny\text{-}PS_{new}$$

The analyzer reports the first subsumption holds , which is rather obvious from since the `Rule` added in PS_{new} yields a `Permit`). However, the analyzer reports subsumption does not hold w.r.t. `Deny`.

In cases of non-subsumption, it is useful to know what are the counter examples, i.e., to show the user a request where PS_{new} and PS_{old} would yield different decisions. Since I use a tableau-based DL reasoner for policy analysis, to check whether $A \sqsubseteq B$, the reasoner tries to build a model for $A \sqcap \neg B$. If a model *can* be built, it means the subsumption does not hold. In that case the model that was just built can easily be extracted from the internals of the reasoner and used as a counter example. Here we get a number of counter-examples:

1) role=LeadDev, action=read, resource=report, action=write,
   resource=report

2) role=LeadDev, action=write, resource=report

3) role=LeadDev, role=Developer, action=write,
   resource=report

The first two are expected (because of the new `Permit` rule), however the third counter example represents a potentially dangerous access leak to a person who is a member of role *Developer*. It is possible to fix this bug by adding a separation of duty constraint for the roles of *Developer* and *LeadDev*. The constraint is presented below, in DL syntax.

$$\exists role.LeadDev \sqsubseteq \neg\exists role.Developer$$

Note that separation of duty constraints can also be serialized in XACML. To accomplish this, a new `Policy` should be created, having the two disjoint roles in its `Target`. Since we want to prohibit a requester that has both of the roles, the effect of this new is `Deny`. Finally, this new policy needs to be set with highest priority (e.g., add it at the beginning of the root policy set and use `First-Applicable` algorithm) to ensure that the separation of duty constraint can never be overridden.

The technique used for policy subsumption can be generalized to policy *comparison*. For two policies P_1 and P_2, we first specify the access decisions we are interested in (say, `Deny` for the first policy and `Permit` for the second), and then check satisfiability of the corresponding concept expressions for those decisions :

$$\mathit{Deny}\text{-}P_1 \sqcap \mathit{Permit}\text{-}P_2$$

If the above expression is *not* satisfiable, then there cannot be an access request s.t. the first policy yields a `Deny` and the second one yields a `Permit`. If it is satisfiable, there is such a request, and we can extract the counter example from the reasoner. To get all counter examples, we need to retrieve *all* consistent models that the concept expression admits; this involves saturation of the tableau, a technique for which DL reasoners are not particularly optimized.

Change verification was introduced in [50], and I show here that it can be accomplished in DL as well. Because the policy differences are expressed as concept descriptions, performing verification of changes is no different than performing verification of policies themselves. The safety properties to be verified are simply added to the change expression. For example, if we want to verify that all changes from `Permit` to `Deny` in the above policy involved the *LeadDev* role, we could test the following concept expression:

$$\mathit{map}(\mathit{Permit}\text{-}\mathit{PS}_{old}) \sqcap \mathit{map}(\mathit{Deny}\text{-}\mathit{PS}_{new}) \sqcap \forall \mathit{role}.\neg \mathit{LeadDev}$$

6.2.3 Redundancy Checking

Another service provided by the analysis framework is finding redundant `Rules`[2]. A redundant rule is one that whenever fires, it is always overridden by some other rule or policy with higher priority. A simple way to check redundancy of a rule r is to perform

[2]The technique can be easily generalized to `Policies` or `PolicySets` – for brevity I focus on `Rules` only.

Algorithm 3 *is_redundant*$(r, \mathcal{D})$

```
Input:
    $r = (ID \quad T \quad \alpha)$: XACML Rule
Output:
    $b$: returns true if $r$ is redundant, false otherwise
1:  $J \leftarrow \bot$
2:  $r_{old} \leftarrow r$                                              ▷ cache $r$
3:  while r $\neq$ null do
4:      for policy element $q$ s.t. $q \neq r$ and $q.parent = r.parent$ do
5:          if $overrides(Permit, q, r)$ then          ▷ Permit decision of $q$ overrides $r$
6:              $J \leftarrow J \sqcup$ Permit-q
7:          end if
8:          if $overrides(Deny, q, r)$ then            ▷ Deny decision of $q$ overrides $r$
9:              $J \leftarrow J \sqcup$ Deny-q
10:         end if
11:     end for
12:     $r \leftarrow r.parent$
13: end while
14: if $r_{old}.target \sqsubseteq J$ then                         ▷ request is subsumed
15:     return true
16: else
17:     return false
18: end if
```

change impact analysis for a policy with and without the rule. Here I present a more guided approach for checking redundancy, by building a concept expression that ignores the policy elements that *cannot* override the rule being checked. Algorithm 3 contains the pseudo-code.

The function starts with an input `Rule` *r* and works its way up to the root policy element. At the same time, it builds a disjunction that consists of the concept expressions for every `Policy` or `PolicySet` that *can override* the access decision made by *r*. In line 5, $overrides(Permit, q, r)$ functions returns true if when q yields a Permit decision, it will always override the decision made by r (no matter what it is). If *r* is subsumed by this disjunction of overriding policies, then the access decision of *r* will always be covered by some policy element. In this case, *r* is redundant.

Redundant rules do not have to be evaluated, and can be safely removed from a policy file. This simplifies the policy and improves runtime performance of the policy evaluator because there are less rules to match requests against.

Please note that in a policy set, there may be multiple possible minimal non-redundant subsets. As an example, consider a policy set containing of three policies: P1 applies to Students of GraduateStudents, P2 applies to GraduateStudents of Professors, and P3 applies to Professors or Students. In this example, {P1,P2} and {P2,P3} are minimal redundant subsets. To find such minimal redundant subsets, a modified version of the minimal set cover algorithm can be used, which is NP-complete. Finding and evaluating minimal redundant subsets of a policy set is not handled by my thesis, although it is an interesting area of future work and is discussed in Chapter 9.

6.3 Analyzing Heterogeneous XACML Policies

Given that one of the crucial requirements for this analysis framework was reasoning about heterogeneous policy domains, in this section I will show how this can be accomplished by leveraging Semantic Web technologies.

Recently, there has been a great amount of interest in extending the expressive power of XACML with Semantic Web technologies [40, 41, 21] for the purposes of heterogeneous policy integration. This is because the Semantic Web provides a data sharing and re-use framework for applications across enterprise and community boundaries. One of the foundational languages of the Semantic Web is the Web Ontology Language (OWL)

[45], which is a W3C standard for representing shared information.

Previous work has focused on extending XACML with support for rich ontology-based models representing subject and resource information. These extensions were accomplished by adding new matching functions and extended conditions to refer to the background ontologies. While these extensions do increase the expressive power of XACML, they do not offer analysis support for such ontology-extended XACML policies.

In this section, I will present a generalization of previous work by providing a *unified* formal framework for OWL ontology-extended XACML policies. To add the functionality, I extend the XACML policy syntax with a new matching function, called *ontology-match*. Following is an example of a Match element with a *semantic-match* function.

```
<Match MatchId="...:function:ontology-match">
   <AttributeValue DataType=".../XMLSchema#string">
      http://policy#Administrator
   </AttributeValue>
   <SubjectAttributeDesignator
    AttributeId="http://policy#role"
    Category="...:Subject"
    DataType="URI"/>
</SubjectMatch>
```

The semantics of the above extension is explained here: consider that we have already mapped the policy set P to DL using the above mapping , so there exists a DL KB K that contains the mapped concepts for each policy element in P and the request is mapped to a DL individual i in K . Given this, the `Match` element M above is mapped to a triple: $(i\ \pi(AT_{AD_M})\ \pi(AV_M))$ where i is the individual corresponding to the request, $\pi(AT_{AD_M})$ is the DL role corresponding to the attribute id, and $\pi(AV_M)$ is the individual corresponding to the attribute value. Then, to check if the *ontology-match* applies, we simply check if:

$$\mathsf{K} \models (i \;\; \pi(AT_{AD_M}) \;\; \pi(AV_M))$$

The *ontology-match* function extends XACML with support for expressive policy vocabularies based on OWL ontologies. Given the XACML mapping to Description Logics (DL) provided in this section, and the fact the semantics of OWL is grounded in DL, it follows that all of the analysis services described in this section can be seamlessly applied to ontology-extended XACML policies as well. To the best of our knowledge, the DL mapping provided in this paper combined with the ontology-based XACML extension discussed above is the first unified analysis framework that can reason about XACML extended with ontology support. Following I will discuss how some common policy idioms that are not easily expressible in XACML can be captured in these ontology-based policy models.

6.3.1 Representing Policy Idioms

In this section, I will discuss how a domain ontology can be used to provide semantic descriptions for the entities used in the access policy. For the policy in our running example, we could develop an ontology that describes the company domain, and link the policy entities with concepts in the ontology using subclass relationships. For example, we can state that a *Manager* is of type *Employee* who is a boss of at least one *Person*:

$$Manager \sqsubseteq Employee \sqcap \exists boss.Person$$

Using such ontologies, I show how common policy idioms can be expressed in Description Logics. DL syntax is used for brevity, but the following can be easily serialized in OWL:

1. *Role hierarchies* are easily captured with subclass axioms. For example ,stating that a *LeadDeveloper* inherits all of the access privileges of the *Developer* role can be expressed as:

$$\exists role.LeadDeveloper \sqsubseteq \exists role.Developer$$

2. Hierarchies on Attributes, can be captured using property hierarchies in DL. For example, to state that if a person is a CIO of a company, that means he is also an employee of that company, we write:

$$CIO\text{-}of \sqsubseteq employee\text{-}of$$

3. *Separation of duty* constraints can be captured with disjoint axioms. To state separation of duty for two role types *A* and *B*, we use:

$$\exists role.A \sqsubseteq \neg\exists role.B$$

4. *Cardinality constraints* can be expressed on any given attribute. To state that the *role* attribute cannot have more than *k* values, we can write:

$$\geq k\ role.\top \sqsubseteq \bot$$

   We can even specify maximum number of users that a role can have, with a combination of inverses and cardinality constraints. For example, the following says that a role cannot have more than *k* users:

$$\geq k\ role^{-}.\top \sqsubseteq \bot$$

6.4 Discussion

This chapter provided a framework for practical analysis services for XACML based on a mapping to Description Logics. Using this mapping, I showed how formal verification, change analysis and redundancy checking can be provided. The mapping covers XACML v3.0 including delegation policies and attribute matching functions using datatypes. Additionally, I showed how the framework can also reason about ontology-extended XACML policies, which have attracted attention in previous work.

One issue not fully covered in this chapter is all of XACML's 200+ datatype functions (I showed how subsets of these functions can be handled using OWL-DL datatypes). Because of the expressiveness of the functions in XACML, covering all of them can easily lead to undecidability. For example, using only the following three functions:

urn:oasis:names:tc:xacml:1.0:function:integer-multiply

urn:oasis:names:tc:xacml:1.0:function:integer-add

urn:oasis:names:tc:xacml:1.0:function:integer-equal

it is possible to construct a system of non-linear Diophantine equations, which have been shown to be undecidable. Thus, there cannot exist a sound, complete and terminating reasoning procedure that covers *all* XACML functions.

Note that in this chapter, I showed how there exists a mechanism that allows us to couple data-type reasoners with Description Logics reasoners. My implementation of a XACML analyzer, does not provide a separate data-type reasoner for all possible decidable data-type domains; instead, it is designed such that XACML developers can extend it with their own specific data-type reasoners. The only restriction to keep in mind is that the data-type domain has to be closed under negation and there needs to be an algorithm that checks for satisfiability of arbitraty conjunctions of predicates in the domain.

Another valid question is how OWL-DL reasoners fare as XACML policy engines. Description Logics suffer from bad worst case complexity - even the limited subset we need for this mapping ($\mathcal{ALCQ}$) is EXPTIME. This seems significantly worse than the polynomial data complexity of the formalization in the previous section - however, different problems are tackled this section. In the previous chapter, the main problem we investigated was access request checking - which is done at run-time, so fast performance is crucial. The services discussed in this section are meant to be performed at compile-time, *before* the policies are even used, so I assume that the performance requirements for my analysis tool are not as stringent. Given this, the empirical evaluation described in Chapter 8 shows that OWL-DL reasoners are surprisingly efficient at reasoning about XACML policies, and are comparable even to binary decision diagram (BDD)-based policy engines.

Chapter 7

Formalizing and Analyzing Web Service Policies

So far in thesis, the formal methods and techniques have been applied to access control (authorization) policies only. This chapter generalizes the approach by applying it to a different domain: web service policies.

In the web services (WS) domain, WS-Policy [129] has emerged as a standard for specification of constraints and capabilities (i.e., *policies*) of WS clients and service providers. In WS-Policy, a policy is defined to be a collection of one or more *assertions*; WS-Policy defines operators that build policy expressions on top of these assertions. A policy assertion represents an individual requirement placed on a web service - it contains domain-specific information (e.g., a client needs to use a particular encryption in order to be able to access the web service). Specifying syntax and semantics for assertions is not covered by the WS-Policy standard.

WS-XACML [95] provides an expressive, domain-independent language for specification of WS-Policy assertions. WS-XACML is a profile of XACML: it essentially specifies ways to use XACML in the context of web services for authorization, access control and privacy policies. While WS-Policy provides the processing model and operators for combining individual assertions, WS-XACML provides a language for specifying the assertions themselves.

WS-Policy and WS-XACML provide a run-time model for matching policies and assertions, respectively. However, there is no defined ’compile-time’ support for analyzing and debugging web service policies. Similarly to the XACML issues, because of the expressiveness of WS-Policy and WS-XACML, it is non-trivial for a policy developer to understand the implications of all of the policies in the system. Mapping both WS-Policy and WS-XACML languages into the same background formalism could potentially provide the first static analysis and verification tool that covers both languages. Additionally, by providing a formal foundation for these policy languages it is possible to acquire a clear semantics, as well as a good sense of the computational aspects.

It was shown in Chapters 5 and 6 how a logic-based formalization of policy languages (XACML) can provide a theoretical and practical foundation for static analysis of access control policies. In this chapter, the analysis framework is applied to web service policies by presenting an OWL-DL mapping for both WS-Policy and WS-XACML. The mapping is built on top of the XACML transformation described in Chapter 6, and it provides the same services: consistency checking, policy verification and policy comparison (containment).

Organization of this chapter is as follows: Section 7.1 presents the WS-Policy to OWL-DL mapping. This includes discussion of the WS-Policy operators (wsp:ExactlyOne and wsp:All), the operations supported (merge and intersection) along with the analysis services provided by the mapping. The second half of this chapter provides a similar treatment for WS-XACML: the syntax and semantics of the language is presented in Sections 7.2.1 and 7.2.2, and the OWL mapping is shown in Section 7.2.3. Finally, in Section 7.3 I show how both of the mappings can be integrated in a single reasoning framework where WS-Policy specifies the high-level processing model for policies and WS-XACML specifies individual policy assertions.

7.1 WS-Policy

WS-Policy provides a general purpose model and syntax to describe the policies of a Web service. It specifies a base set of constructs that can be used and extended by other Web service specifications to describe a broad range of service requirements and capabilities. This section will present the WS-Policy mapping to OWL-DL that allows us to use off-the-shelf OWL editors and reasoners to do policy administration and processing tasks.

7.1.1 Mapping WS-Policy Operators to OWL-DL

In this section, I present a formalization of the WS-Policy constructs by mapping a normal form policy expression to OWL class expression. A policy in a normal form is represented as an XML document, where a root Policy element enumerates each of its alternatives, and each alternative in turn enumerates its assertions. Following is a schema outline for the normal form of a policy expression:

```
<wsp: Policy>
   <wsp:ExactlyOne>
      [ <wsp:All> [<Assertion>  </Assertion>]* </wsp:All> ]*
   </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 1. Normal form of a policy expression

Policy expressions can also be represented in more compact forms, using additional operators such as `wsp:Optional`, however as shown in [129] the policy expressions can all be expanded to normal form. Therefore we only provide a mapping of the constructs used in a normal form policy expression: `wsp:ExactlyOne` and `wsp:All`.

First, policy assertions are mapped directly into OWL-DL class expressions. Instead of mapping separately each domain-specific assertion language into OWL, I provide a mapping of domain-independent WS-XACML instead. The mapping of WS-XACML is discussed in Section 7.2.

Mapping `wsp:All` to OWL is straightforward because `wsp:All` means that all of the policy assertions enclosed by this operator have to be satisfied in order for communication to be initiated between the endpoints. Thus, it is a logical conjunction and can be represented as OWL intersection. Each of the members of the intersection is a policy assertion, and the resulting class expression is a custom-made policy class that expresses the same semantics as the WS-Policy one.

Because the description of `wsp:exactlyOne` in the WS-Policy specification is ambiguous, there are here are two possible interpretations for the operator:

`wsp:ExactlyOne` as an exclusive OR `wsp:ExactlyOne` means that only one, not more, of the alternatives should be supported in order for the requester to support the policy. This is supported by the official specification[129], where it is stated that although policy alternatives are meant to be mutually exclusive, it cannot be decided in general whether or not more than one alternative can be supported at the same time. To cover this more complicated case, I translate `Wsp:ExactlyOne` in the following way: for n different policy

WS-Policy Construct	OWL Expression
Wsp:All (policies A and B)	owl:intersectionOf(A B)
Wsp:ExactlyOne (policies A and B)	intersectionOf(complementOf(intersectionOf(A B)) unionOf(A B))

Table 7.1: Mapping of WS-Policy Constructs to OWL

assertions, expressed as OWL classes themselves, `wsp:ExactlyOne` is the class expression consisting of those individuals in each separate policy class that do **not** also belong to another policy class. In OWL terms, it is the union of all of the classes with the complement of their pair-wise intersections. Because of the pair-wise intersections there is a quadratic increase in the size of the OWL construct that is used as a mapping for `wsp:ExactlyOne`.

wsp:ExactlyOne as an inclusive OR However, due to the open world assumption present in OWL-DL, the above mapping produces non-intuitive results. For example, if a request r comes in such that $r : A$, and the policy P contains only two alternatives, A and B, we will not be able to infer that the request r satisfies P (i.e., r is of type $(A \sqcup B) \sqcap \neg(A \sqcap B)$) unless we explicitly state that $r : \neg B$. To overcome this issue, I offer a simplified mapping to represent <wsp:exactlyOne> as logical disjunction (inclusive OR), and in addition I have made the classes representing the alternatives pair-wise disjoint, so even though a requester supports more than one alternative, he cannot use more than one at a time. This updated translation is more concise than the one presented above (compare $A \sqcup B$ with $(A \sqcup B) \sqcap \neg(A \sqcap B)$). In this scenario, if a requester comes in that is a member of two alternatives, a logical inconsistency will occur.

```
<wsp:Policy
    xmlns:sp=”http://schemas.xmlsoap.org/ws/2005/07/securitypolicy”
    xmlns:wsp=”http://www.w3.org/2006/07/ws-policy” >
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:RequireDerivedKeys />
      <sp:WssUsernameToken10 />
    </wsp:All>
    <wsp:All>
      <sp:RequireDerivedKeys />
      <sp:WssUsernameToken11 />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 7.1: Example policy

Example 7.1.1.1 Consider the example policy in Figure 7.1. For each policy assertion, a separate OWL class (RequireDerivedKeys, WssUsernameToken10, WssUsernameToken11) is generated. Then, each alternative is simply the conjunction of its constituent assertions.

$$Alt_1 \equiv RequireDerivedKeys \sqcap WssUsernameToken10$$

$$Alt_2 \equiv RequireDerivedKeys \sqcap WssUsernameToken11$$

Finally, the policy class P is equivalent to the disjunction of the alternative classes:

$$P \equiv Alt_1 \sqcup Alt_2$$

If, for example, these two alternatives have to be disjoint, then a disjoint axiom is added:

$$Alt_1 \sqsubseteq \neg Alt_2.$$

With this mapping, checking whether a web service requester satisfies a particular policy can then be reduced to instance checking: simply checking whether the OWL individual representing the requester is a member of the OWL class representing the policy.

To more compactly express complex policies, WS-Policy allows nesting of operators. To convert a policy from a compact to a normal form, the properties of `wsp:ExactlyOne` and `wsp:All` can be used. To show that the OWL translation correctly captures the meaning of `wsp:ExactlyOne` and `wsp:All`, I need to prove that the mappings from Table 7.1.1 have the same properties as the WS-Policy operators. `wsp:ExactlyOne` and `wsp:All` have the following properties: *commutativity*, *associativity*, *idempotency* and *distributivity*. It can be easily shown that the logical constructs corresponding to `wsp:ExactlyOne` and `wsp:All`, which are essentially a logical conjunction and disjunction, also satisfy these properties.

7.1.2 Mapping Policy Assertions to OWL

Web Services Policy assertions indicate domain-specific capabilities and requirements and are defined in separate specifications.The WS-Policy standard does not define the syntax and semantics of assertions - this task is delegated to experts in their respective domains.

Recently, there has been a proposal of a generic, domain-independent policy language that is able to capture the above domains. The proposal is WS-XACML, and it uses a subset of the functions of core XACML. Instead of formalizing and analyzing all of the domains separately, this chapter provides a mapping of domain-independent WS-XACML. The mapping is presented in Section 7.2.

7.1.3 WS-Policy Merge and Intersection

As part of the specification, WS-Policy also defines two operations on policies: *merge* and *intersection*.

Merge `Merge` is the process of combining sub-policies together to form a single policy. This operation is needed because a policy might be specified in a distributed way, having its fragments defined in separate files. It is necessary to combine all these policy fragments together to form a single merged policy which could be processed further.

`Merge` works on policies already converted to normal form. The merged policy is a Cartesian product of the alternatives in the first policy and the alternatives in the second policy. There is a straightforward way of doing the `Merge` operation in OWL-DL. First, we translate each of the input policies into OWL-DL as described above. Then, the merged policy is simply the *intersection* of the input policies. Thus, `Merge` also maps cleanly onto OWL-DL. An outline of the proof is shown in [84].

Intersection Policy intersection is used when a web service requester and provider both express policies and want to compute the compatible policy alternatives between them. Like in `Merge`, the process of coming up with an intersection is carried out in a cross product fashion, comparing each alternative from the first policy with every alternative from the other one. However, in the case of `Intersection`, if the two alternatives that are being combined do not agree on the same vocabulary, then the combined alternative is not added to the new policy. A vocabulary of an alternative is simply defined as the set of QNames of the assertions in that alternative. If the alternatives do agree on the same vocabulary, then as defined in [129], two alternatives are compatible if each assertion in the first alternative is compatible with an assertion in the second, and vice-versa.

Checking whether two assertions are compatible is outside of the scope of WS-Policy. To overcome this, I use WS-XACML to represent the individual policy assertions, since WS-XACML defines a processing model that checks if two assertions are compatible. More information is presented in Section 7.2.

7.1.4 Policy Processing

One benefit of expressing policies using OWL is the ability to reason about policy containment - i.e., checking if the requirements for supporting one policy are a subset of the requirements for another. That would allow us to be more flexible in determining whether a particular requester supports a policy, in the cases where the requester supports a superset of the requirements established by the policy. Policy containment is only one of the services supported out of the box; a full listing follows:

1. policy equivalence (A owl:equivalentTo B);
2. policy incompatibility (if x meets policy A then it cannot meet policy B; a.k.a, A owl:disjointWith B);
3. policy incoherence (nothing can meet policy A; a.k.a., A is unsatisfiable)
4. policy conformance (x meets policy A; a.k.a, x rdf:type A)
5. policy containment (if x meets policy A then it also meets policy B; a.k.a., A rdfs:subClassOf B);

There is an additional reasoning service that is useful for policies and warrants more discussion. It can been argued that explanation is a crucial requirement for a policy language. To address this requirement, we can use recent advances in the field of debugging OWL ontologies [80], esp. in providing explanations for both ontology inconsistencies and arbitrary entailments for OWL-DL. For example, if a user asks why the requester `r` satisfies (or does not satisfy) the policy `P`, then the debugging framework is simply asked to provide justification for the type assertion `r`:`P`. On the other hand, if a web service request causes an inconsistency (for example because of violating a domain disjointness constraint), then the debugging framework can provide explanation of why the inconsistency occurred. More specifically, if an OWL-DL ontology is inconsistent, the work by Kalyanpur et al. [80] provides an algorithm to extract the minimal set of axioms in the ontology that causes the inconsistency.

Thus, with a fairly simple mapping, one can use an off-the-shelf OWL reasoner as a policy engine and analysis tool, and an off-the-shelf OWL editor as a policy development and integration environment. OWL editors can also be used to develop domain specific assertion languages (essentially, domain ontologies) with a uniform syntax and well specified semantics. This mapping can also be used to experiment with extensions to WS-Policy, by using more expressive constructs from OWL as the policy language and assertion language level. We can experiment with policy language extensions without having to write yet another policy engine for them.

Furthermore, ontology development techniques can be useful for policy development as well. Iterative development is a popular ontology engineering approach [119], where specializations are added to the class tree over time. Similarly, we can build up our

XA	::=	(`XACMLAssertion` REQ* CAP*)
REQ	::=	(`Requirements` VocabRef (S | CONS*))
CAP	::=	(`Capabilities` VocabRef (RQ | polDoc | CONS*))
CONS	::=	(`AttributeDesignator` AV AD fcn) |
		(`AttributeSelector` AV AS fcn)

Table 7.2: Syntax of Web Service Profile of XACML.

policies from more general ones. A general policy could be very restrictive, setting tough guidelines for all organization's policies - then different departments could subsume and extend this general policy.

7.2 WS-XACML

This section will focus on WS-XACML, which is a XACML profile for web service policy assertions. Concise syntax and semantics is presented first, followed by a presentation of the WS-XACML to OWL-DL formal mapping.

7.2.1 Syntax of WS-XACML

The top-level element of a WS-XACML policy is an *assertion*. A XACML-based Web Services Policy Assertion is a description of an entity's Web Service's policy w.r.t. to some policy domain. An assertion in WS-XACML can be used to express both *requirements* and *capabilities*.

Requirements The requirements for an assertion element can be expressed in terms of a `Policy`, `PolicySet` or a `Constraint`. Additionally, the `Requirements` element contains a `VocabularyRef`, which contains the URI's associated with a given policy vocabulary. Each XACML attribute referenced in `Requirements` must be defined in one of the policy vocabularies specified by a `VocabularyRef`.

Capabilities Capabilities of a web service endpoint are expressed in terms of `Requests` or `Constraints`. `Requests` can be reduced to `Constraints` as follows: for each `Attribute` *A* in the Request a new `Constraint` is generated. The `Constraint` contains all attribute values occurring in the `Request` for *A*.

Constraint A `Constraint` is represented by a boolean function. The attribute referenced using an `AttributeDesignator` or `AttributeSelector` is one of the arguments to the function. A `Constraint` is satisfied if the function used in the constraint evaluates to True when evaluated against a given value for the Attribute. The set of values for which the function evaluates to True is the set of acceptable values for the policy vocabulary variable.Exactly one attribute must be referenced by each `Constraint`. Following is an example of a constraint:

```
<Constraint FunctionId="...:integer-less-than">
 <xacml:Apply FunctionId="...:integer-one-and-only">
   <xacml:AttributeDesignator AttributeId="max-data-retention-days"
   DataType="http://www.w3.org/2001/XMLSchema#integer"/>
 </xacml:Apply>
 <xacml:AttributeValue DataType="...#integer"/>
  90
 </xacml:AttributeValue>
</Constraint>
```

Only a limited number of comparison functions are allowed in `Constraints` and `Constraints` themselves cannot be nested. Finally, note that a `Constraint` can easily be translated to a XACML `Rule` element with the same semantics, simply by generating a `Rule` with an empty target and including the `Constraint` in the `Rule`'s `Condition`.

7.2.2 Semantics of WS-XACML

This section presents an intuitive description of the semantics of WS-XACML assertion matching, along with a more formal presentation based on natural deduction rules.

WS-XACML can be used to check if the constraints and capabilities of a web service provider are compatible with the constraints and capabilities with a web service consumer; thus, the main service WS-XACML provides is assertion *matching*. The rules for matching XACML Assertions are shown in Table 6.4. Two Assertions A and B match if at least one `Requirements` element in A matches one `Capabilities` element in B, and vice-versa. A missing `Requirements` (resp. `Capabilities`) in one assertion matches any `Capabilities` (resp. `Requirements`) element in the other assertion.

The semantics of matching `Requirements` against `Capabilities` is presented in Table 6.3. Intuitively, it is done in the following way:

- A match between `Policy` or `PolicySet` p from `Requirements` and a `Request` r from `Capabilities` is successful iff P yields a `Permit` on r.

- A match between `Constraint` c from `Requirements` and a `Request` r from Capabilities is successful iff the translation of c to a `Rule` P returns a `Permit` for r.

- A match between `Constraint` c_r from `Requirements` and a `Constraint` c_c from `Capabilities` is successful iff the intersection of the two is nonempty. In the presence of multiple `Constraint` elements in `Requirements`, they are treated as a logical AND, i.e., all `Constraints` must be satisfied in order for the `Requirements` element to be satisfied. Thus, for each `Constraint` in `Requirements`, there should be at least one matching `Constraint` from `Capabilities` for the match to be successful overall. Intersection of `Constraints` is defined in the next section.

$$\frac{\text{fcn-2(AD, AV-2)} \cap \text{fcn-1(AD, AV-1)} \not\models \bot}{(\texttt{Constraint}\ \text{AV-1 AD fcn-1})\ (\texttt{Constraint}\ \text{AV-2 AD fcn-2}) \models \textit{match}}$$

$$\frac{\begin{array}{c}\forall \text{CONS}_i : \\ \exists \text{CONS}_j \text{s.t.} \text{CONS}_i, \text{CONS}_j \models \textit{match}\end{array}}{(\texttt{Requirements}\ \text{VocabRef}\ (\text{CONS}^*)), (\texttt{Capabilities}\ \text{VocabRef}\ (\text{CONS}^*)) \models \textit{match}}$$

$$\frac{\begin{array}{c}\forall \text{CONS}_i : \\ \exists \text{RQ} \text{s.t.} \text{CONS}_i, \text{RQ} \models \texttt{Permit}\end{array}}{(\texttt{Requirements}\ \text{VocabRef}\ (\text{CONS}^*)), (\texttt{Capabilities}\ \text{VocabRef RQ}) \models \textit{match}}$$

$$\frac{\forall \text{S} : \quad \exists \text{RQ} \text{s.t.} \text{S}, \text{RQ} \models \texttt{Permit}}{(\texttt{Requirements}\ \text{VocabRef S}), (\texttt{Capabilities}\ \text{VocabRef RQ}) \models \textit{match}}$$

$$(\texttt{Requirements}), \text{CAP} \models \textit{match} \qquad \text{REQ}, (\texttt{Capabilities}) \models \textit{match}$$

Table 7.3: Matching `Requirements` with `Capabilities`.

7.2.2.1 Matching Constraints

If the constraints C_a and C_b that are matched do not refer to the same XACML attribute, the match still succeeds; however, the intersection is then defined as a conjunction of C_a and C_b. If the constraints refer to the same attribute (a), then matching is done by checking if the specified values for a are compatible. The constraints are compatible iff there exist some value v for a that satisfies both C_a and C_b.

Consider the following two constraints:

```
<Constraint FunctionId="...:integer-less-than">
 <xacml:AttributeDesignator AttributeId="...:max-data-retention-days"
   DataType="...#integer"/>
```

$$\frac{\begin{array}{c}\text{REQ}_i \in \text{REQ-1}, \text{CAP}_j \in \text{CAP-2}\\ \text{REQ}_i, \text{CAP}_j \models \textit{match}\end{array}}{(\texttt{XACMLAssertion}\ \text{REQ-1}), (\texttt{XACMLAssertion}\ \text{REQ-2 CAP-2}) \models \textit{match}}$$

$$\frac{\begin{array}{c}\text{REQ}_k \in \text{REQ-2}, \text{CAP}_l \in \text{CAP-1}\\ \text{REQ}_l, \text{CAP}_l \models \textit{match}\end{array}}{(\texttt{XACMLAssertion}\ \text{CAP-1}), (\texttt{XACMLAssertion}\ \text{REQ-2 CAP-2}) \models \textit{match}}$$

$$\frac{\begin{array}{c}\text{REQ}_i \in \text{REQ-1}, \text{CAP}_j \in \text{CAP-2}\\ \text{REQ}_i, \text{CAP}_j \models \textit{match}\\ \text{REQ}_k \in \text{REQ-2}, \text{CAP}_l \in \text{CAP-1}\\ \text{REQ}_l, \text{CAP}_l \models \textit{match}\end{array}}{(\texttt{XACMLAssertion}\ \text{REQ-1CAP-1}), (\texttt{XACMLAssertion}\ \text{REQ-2 CAP-2}) \models \textit{match}}$$

Table 7.4: Matching XACML Assertions.

```
 <xacml:AttributeValue DataType= "...#integer"/>90
 </xacml:AttributeValue>
</Constraint>
```

and

```
<Constraint FunctionId="...:integer-more-than">
 <xacml:AttributeDesignator AttributeId=":max-data-retention-days"
   DataType="...#integer"/>
 <xacml:AttributeValue DataType= "...#integer"/>45
 </xacml:AttributeValue>
</Constraint>
```

Intersecting the two constraints, we see the allowed values for *max-data-retention-days* are between 45 and 90, so the constraints are compatible. Since the allowed comparison functions are limited (mostly of *type-equal*, *type-less-than*, *type-greater-than*, etc.) and `Constraints` cannot be nested, computing the intersection of two `Constraints` is relatively straightforward. A detailed listing of supported constraint functions in WS-XACML and a decision procedure for computing intersection of Constraints can be found in Appendix A in the WS-XACML specification [95].

7.2.3 Mapping WS-XACML

Similarly to the XACML-to-DL mapping, I introduce a mapping function π that takes WS-XACML expresions as input and returns DL concept expressions. For a XACML assertion A we will refer to its set of requirements and capabilities as $A.req$ and $A.cap$, respectively. A XACML assertion

$$A = (\texttt{XACMLAssertion}\ (\mathrm{REQ}_1, \ldots, \mathrm{REQ}_n)\ (\mathrm{CAP}_1, \ldots, \mathrm{CAP}_n))$$

is mapped in the following manner:

$$\pi(A) = (\pi(\mathrm{REQ}_1) \sqcup, \ldots, \sqcup \pi(\mathrm{REQ}_n)) \sqcap (\pi(\mathrm{CAP}_1) \sqcup, \ldots, \sqcup \pi(\mathrm{CAP}_m))$$

Mapping `Requirements` There are three cases for a `Requirements` R element:

- R is empty; in which case $\pi(R) = \top$, since R being empty is compatible with everything.
- R is of type `Policy` or `PolicySet`; for this case, the mapping function $\pi(R)$ is already defined in Chapter 6.
- R contains one or more `Constraints` $c_1, \ldots, c_n$; $\pi(R) \equiv \pi(c_1) \sqcap, \ldots, \sqcap \pi(c_n)$. Multiple `Constraints` in a `Requirements` element are treated as a conjunction in the specification. The mapping function for constraints is defined in the next section.

Mapping `Capabilities` There are three cases for a Capabilities element C:

- C is empty; in which case $\pi(C) = \top$, since an empty Capabilities element is compatible with everything.

- C is of type Request; then $\pi(C)$ is already defined in Chapter 6.

- C contains one or more Constraints $c_1, \ldots, c_n$; $\pi(R) \equiv \pi(c_1) \sqcup, \ldots, \sqcup \pi(c_n)$. Multiple Constraints in a Requirements element are treated as a disjunction in the WS-XACML specification. The mapping function for constraints is defined in the next section.

Please note that WS-XACML also allows for PolicyDocument elements, that contain an XML document with domain specific policy vocabulary information. The semantics of PolicyDocument elements is outside the scope of the WS-XACML profile, hence it is not covered in this thesis.

7.2.3.1 Mapping Constraints

A Constraint is simply a datatype function f performed on a single XACML attribute a and attribute value(s) v. (I will use shorthand $f(a, v)$ to represent Constraints henceforth.) A WS-XACML Constraint is very similar to a Match element in XACML, with the only difference being the type of comparison functions supported. For the basic matching functions, the behavior (and hence the OWL-DL mapping) is the same. For example, given a Constraint where the comparison function f is of type *string-equal*, and the attribute value v is of type string, the mapping generates a concrete (datatype) OWL-DL property to capture the attribute a, and a user-defined string datatype to capture v. For other basic comparison functions that occur in Constraints and Match elements, the mapping is shown in Table 7.5.

Matching function fcn	π(AV, fcn)
type-equal	DatatypeRestriction (*type* value$_{AV}$)
type-greater-than	DatatypeRestriction (*type* minExclusive value$_{AV}$)
type-greater-than-or-equal	DatatypeRestriction (*type* minInclusive value$_{AV}$)
type-less	DatatypeRestriction (*type* maxExclusive value$_{AV}$)
type-less-than-or-equal	DatatypeRestriction (*type* maxInclusive value$_{AV}$)
type-regexp-match	DatatypeRestriction (*type* pattern value$_{AV}$)
xpath-node-match	DatatypeRestriction (xpath-expression pattern value$_{AV}$)

Table 7.5: Mapping matching functions.

However, there are some additional matching functions that are allowed in `Constraints`, but not in `Match` elements: *type-set-equals*, *type-subset*, *time-in-range*. Given a `Constraint` $f(a, v)$, *type*-subset is easiest to map since it simply corresponds to creating a conjunction of all of the attribute values in $v_i \in v$. This is because for a request to satisfy this constraint, *all* of the values in v must occur in that request (additional ones are allowed as well). *Type-subset-equals* is more involved, since we have to make sure no additional values for that attribute are present (apart from the ones in v). For this purpose, a cardinality constraint on attribute a is added , which essentially limits the number of possible values to $\#v$. Finally, *time-in-range* is handled by generating a user defined date-time datatype (using min and max facets of XML schema). Formal representation of mapping in Table 7.6.

7.2.3.2 Analysis Services

The OWL-DL mapping of WS-XACML presented in this section allows us to provide a variety of reasoning services for WS-XACML assertions out of the box, similarly

Matching function $f(a, v)$	$\pi(f(a, v))$
type-subset(a, v)	$\sqcap\exists\pi(a).v_i$ for each $v_i \in v$
type-set-equals	$\pi(\textit{type-subset}(a, v)) \sqcap =n\pi(a).\top$ where $n = \#v$
time-in-range(a, v_1, v_2)	(DataHasValue($\pi(a)$ DatatypeRestriction (*type* minInclusive v_1)) DataHasValue($\pi(a)$ DatatypeRestriction (*type* maxInclusive v_2))
must-be-present(a)	$\exists\pi(a).\top$
must-not-be-present(a)	$\forall\pi(a).\bot$

Table 7.6: Mapping `Constraint` functions.

to WS-Policy:

- assertion containment - given two XACMLAssertions A and B, checking containment (subsumption) is reduced to checking if $\pi(A) \sqsubseteq \pi(B)$
- assertion incompatibility - if a request matches assertion A , then it will not match assertion B. This is reduced to checking disjointness of $\pi(A)$ and $\pi(B)$: $\pi(A) \sqcap \pi(B) \models \bot$
- assertion satisfiability (nothing can match the assertion A, reduced to satisfiability check of $\pi(A)$)
- formal verification of a policy. In particular, we're presented with a set of WS assertions that must be satisfied and a set of assertions that have to be filtered by our XACMLAssertion. This is similar to verification for policies, where a set of tests (in form of access requests) and expected outcomes for those tests are provided. In order to make sure there is no possible instantiation of the XACMLAssertion that can break the properties, for a policy A and test input (in the form of an assertion,

too) C we perform the following satisfiability check:

$$\pi(A.req) \sqcap \pi(C.cap)$$

$$\pi(C.req) \sqcap \pi(A.cap)$$

If the assertion C needs to be compatible with A, then the verification test succeeds when both expressions above are satisfiable, whereas for an incompatibility test, verification is successful when at least one of the expressions above are be unsatisfiable

7.3 Putting it All Together

To illustrate how the mappings of WS-Policy and WS-XACML can be integrated, consider the following example

```
<wsp:Policy
      xmlns:sp=”http://schemas.xmlsoap.org/ws/2005/07/securitypolicy”
      xmlns:wsp=”http://www.w3.org/2006/07/ws-policy” >
  <wsp:ExactlyOne>
    <wsp:All>
      <ws-xacml:XACMLAuthzAssertion id=”XA-1” />
      <ws-xacml:XACMLAuthzAssertion id=”XA-2” />
    </wsp:All>
    <wsp:All>
      <ws-xacml:XACMLAuthzAssertion id=”XA-3” />
      <ws-xacml:XACMLAuthzAssertion id=”XA-2” />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 7.2: Example policy

Example 7.3.0.1 This example contains a revised version of the policy from Figure 7.1.

Instead of domain-specific assertions, here WS-XACML policy assertions are used (because of their verboseness, only the ID element is included in the description). Each WS-XACML assertion is mapped to an OWL class expression using the mapping function π described in this section. Given this, each WS-Policy alternative is simply a conjunction of its constituent assertions.

$$Alt_1 \equiv \pi(\text{XA-1}) \sqcap \pi(\text{XA-2})$$
$$Alt_2 \equiv \pi(\text{XA-3}) \sqcap \pi(\text{XA-2})$$

Finally, the policy class P is equivalent to the disjunction of the alternative classes:

$$P \equiv Alt_1 \sqcup Alt_2$$

As it can be seen from this example, given that both WS-Policy and WS-XACML are mapped to OWL-DL, their integration is straightforward.

7.4 Summary

This chapter generalized the framework presented in Chapters 4 and 5 by applying it to a different domain (web service policies). The main difference between the access control policies discussed earlier and the web service policies discussed here is that both web service requesters and web service providers can have policies of their own, unlike in access control where the requesters usually does not have an access policy. Because of this, checking whether two web service policies match each other in our framework is done using a satisfiability check, i.e., by trying to find a model where both policies will be satisfied.

Given this generalization the analysis framework was applied to web service policies by presenting an OWL-DL mapping for both WS-Policy and WS-XACML. The mapping was built on top of the XACML transformation described in Chapter 5 and it provides formal verification of policies, policy comparison (containment) and checking coherency (consistency) of web service policies.

Chapter 8

Implementation and Evaluation

In this chapter, I show the logic-based framework presented in Chapters 6 and 7 can provide analysis tasks such as formal verification, policy comparison and redundancy checking for large and expressive XACML policies in a practical manner. For this purpose, I have implemented the XACML mapping as a policy analysis tool that is based on OWL-DL reasoners. An overview of the analysis tool and its novel optimizations are discussed in Section 8.1.

Most of this chapter contains a discussion on the extensive empirical evaluation performed on my policy analysis tool. The empirical evaluation consists of two parts. The first part (Section 8.2) consists of a policy test suite where the performance of my approach is compared to two of the most scalable XACML analyzers: BDD-based Margrave[50] and a SAT-based analyzer by Hughes et al. [67]. The policy test suite contains 5 non-trivial real-world XACML policies with limited expressiveness so the other tools can process them; the evaluation shows that the performance of my analyzer is comparable to the other approaches for these policies.

The second part of the evaluation (Sections 8.3 and 8.4) uses two real world access control policies that employ more expressive features such as policy vocabulary domains, datatype functions and policy constraints: NASA Federated Data Access Use Case [118] and RBAC policies from the healthcare domain [62]. I converted these policies in

XACML and showed by empirical testing of formal verification and policy comparison service that my analysis tool can load and analyze these large policy sets in a practical manner.

8.1 Implementation

The three main components of the analysis framework are a *mapper*, which converts XACML policies to DL knowledge bases, an *analyzer*, which reduces policy analysis services to DL reasoning tasks, and an *output generator*, which extracts counter examples from the internals of the reasoner and converts them to XACML access requests. I have provided an API so different DL reasoners can be plugged in and used as analyzers; so far, I have used the open source reasoners Pellet [110] and Fact++ [120]. An architectural diagram of the implemented prototype is presented in Figure 8.1.

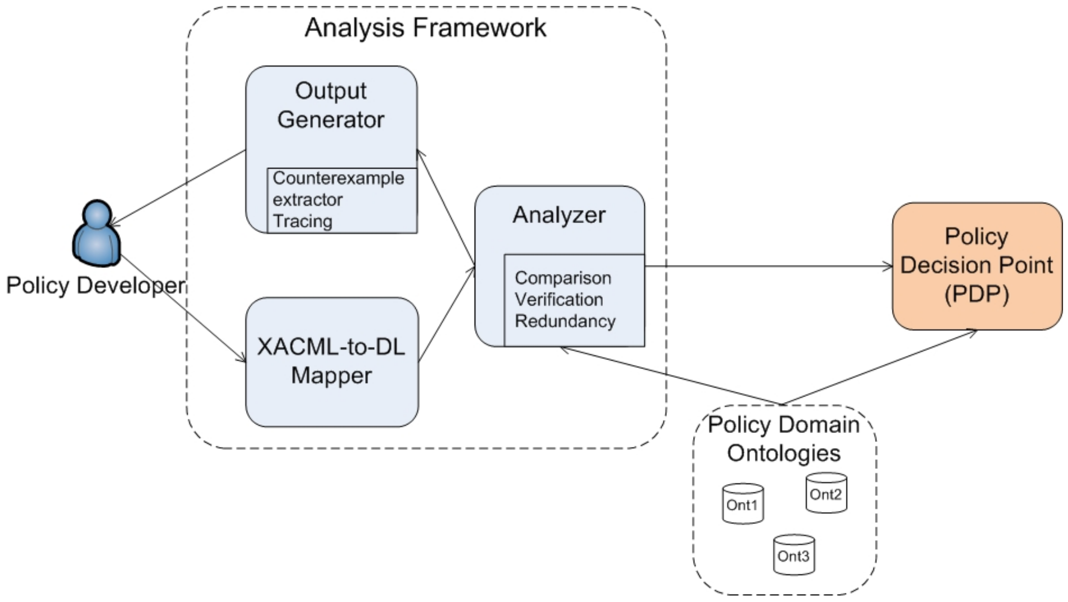


Figure 8.1: Architecture of Analysis Framework.

I also implemented a few optimizations in the mapper to improve performance of analysis services - these are briefly discussed below:

- *Grouping by categories* - A practical performance gain can be achieved by assuming the attributes for different categories are disjoint (which was the case for every XACML policy that I have used in the evaluation). This is because we can group the attributes based on the category they belong to, and then minimize unnecessary interaction between attributes in different categories (this idea is similar to conjunctive partitioning in model checking [130]). The partitioning is accomplished by introducing an additional (functional) DL role for each category like `Subject`, `Resource`, `Action`, and then adding the attribute values as role fillers of the corresponding category roles. To illustrate how it works, consider the mapping of a rule before:

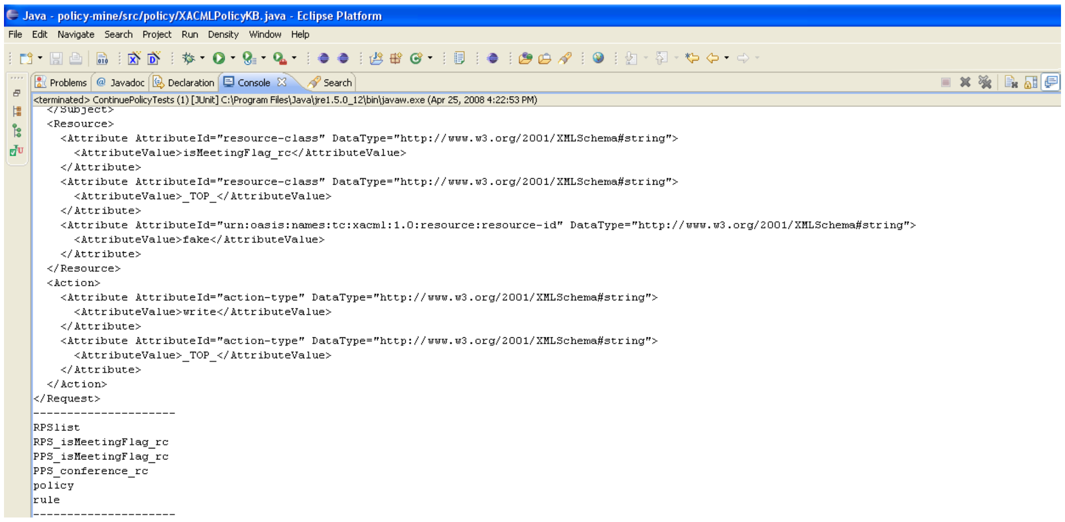
  $$\begin{aligned} \textit{Deny-}R_3 \equiv & \exists \textit{role}.(\textit{Manager} \sqcup \textit{Developer}) \sqcup \\ & \exists \textit{action-type}.(\textit{read} \sqcup \textit{write}) \sqcup \\ & \exists \textit{resource-type}.\textit{Report} \end{aligned}$$

  and after:

  $$\begin{aligned} \textit{Deny-}R_3 \equiv & \exists \textit{subject}.\exists \textit{role}.(\textit{Manager} \sqcup \textit{Developer}) \sqcup \\ & \exists \textit{action}.\exists \textit{action-type}.(\textit{read} \sqcup \textit{write}) \sqcup \\ & \exists \textit{resource}.\exists \textit{resource-type}.\textit{Report} \end{aligned}$$

- *Simplification* of concept expressions. To improve performance, tableau reasoners reduce concept expressions to a simplified normal form before checking for concept satisfiability. Due to the size of the *Permit-P* and *Deny-P* axioms returned from the mapping function, simplifying the DL concepts as they're generated by the mapping, before reasoning, reduces the amount of work the policy analyzer has to perform during reasoning.

An example of output of the tool is shown in Figure 8.2. The service ran is formal verification, and the analyzer found a counter-example which is presented in XACML syntax. Additionally, a stack trace of policies that produced the counter-example is provided.


Java - policy-mine/src/policy/XACMLPolicyKB.java - Eclipse Platform
File Edit Navigate Search Project Run Density Window Help
Problems
Javadoc
Declaration
Console
Search

```
<terminated> ContinuePolicyTests (1) [JUnit] C:\Program Files\Java\jre1.5.0_12\bin\javaw.exe (Apr 25, 2008 4:22:53 PM)
  </Subject>
  <Resource>
    <Attribute AttributeId="resource-class" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>isMeetingFlag_rc</AttributeValue>
    </Attribute>
    <Attribute AttributeId="resource-class" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>_TOP_</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>fake</AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="action-type" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>write</AttributeValue>
    </Attribute>
    <Attribute AttributeId="action-type" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>_TOP_</AttributeValue>
    </Attribute>
  </Action>
</Request>
---------------------
RPSlist
RPS_isMeetingFlag_rc
PPS_isMeetingFlag_rc
PPS_conference_rc
policy
rule
---------------------
```

Figure 8.2: Sample Output of Formal Verification Service.

8.2 Policy Test Suite

This section presents the first part of the evaluation: applying the analysis framework to a data set consisting of publicly available real-world XACML policies. To evaluate the performance of the analysis tool, I tested the following services: formal verification, policy comparison (change analysis) and redundancy checking. The reason for testing only these services is that all the others (policy equivalence, disjointness, formal verification of policy differences) can be reduced to either formal verification or policy comparison; thus, the running time for the other services should not differ substantially.

The empirical dataset consists of XACML policies being used in various applications; each of these policies is briefly described below:

- *Continue* [11] is a web application for paper submissions, reviews, and PC meetings. Continue contains 26 policy files (each one representing a `PolicySet`) in the set, with a total of 13 attributes and 36 attribute values. Although it does not have many attributes and/or values, the policies are fairly interconnected and nested (up to 5 levels), which makes it non-trivial to analyze.

- *Fedora* [4] is an open source digital management repository. It provides an extensible architecture for digital asset management (DAM), upon which many types of digital library, institutional repositories, digital archives, and digital libraries systems might be built. Out-of-the-box, the Fedora repository is configured with a default set of XACML access control policies that provide for a highly restricted management service and an open access service for digital objects.

- Generic Authorization, Authentication and Accounting Framework (*GAAA* [1]) is aimed at developing a Web Services-based open source toolit that will enable application developers to incorporate access control functions as part of workflow management in a Grid environment. In particular, the research uses the problem of on demand provisioning of network connections across multiple domains as a proof of concept. As part of the prototype, a set of policies is developed and made publicly available.

- *eXist* [3] is an open source XML Database. It supports XACML as means of specifying access control for XML resources, and it provides a default XACML policy set to control access to Java methods from XQuery

- *Network* [67] is a set of non-trivial access control policies used in testing the SAT-based XACML analyzer developed by Hughes et al [67].

The general characteristics of the policies are shown in Table 8.1.

Name	Attributes/Values	`PolicySet` Elements	Depth
Continue	14/36	26	5
Fedora	7/15	14	2
eXist	9/37	5	2
Network	5/10	6	2
GAAA	3/47	2	2

Table 8.1: General Information on Policies in Test Suite.

In addition to testing my prototype, I also processed these policies using other XACML analyzers such as Margrave [50] and the analyzer described by Hughes et al [67] (referred to as HSAT from now on). These two were chosen since they are the fastest available XACML analyzers. Margrave is based on a mapping of a XACML policy to a multi-terminal binary decision diagram (MTBDD) which is a data structure used to compactly represent boolean formulas. Similarly to Margrave, HSAT also translates policy verification queries to Boolean satisfiability problems; the difference is that they use a SAT solver (zchaff [102]) to obtain an answer.

Both Margrave and HSAT are less expressive than my analyzer; in particular they lack the support for vocabulary domains in policies, data-types (HSAT has some incomplete support) and non-trivial `Condition` elements in `Rules`. The policies were chosen so that at least one other analyzer (other than mine) can process them – the goal being to investigate on how my approach would fare against analyzers optimized for less expressive policies. Note that the expressive policies discussed in Sections 7.3 and 7.4 cannot be handled by the other approaches.

The experiments where run on a Linux machine with 2Gb of RAM and a 3.06GHz Intel Xeon CPU. In all of the figures, the X-axis corresponds to the policy being analyzed, while the Y-axis is the average time in seconds for performing the analysis service in question.

8.2.1 Formal Verification Results

In order to perform formal verification, the following inputs are needed: a policy to be verified, a test case (represented as a XACML policy as well), and the expected outcome of the test case (e.g., *NeverPermit*, *AlwaysDeny*, etc.). Fortunately, the Continue policy contains 11 test cases and outcomes already specified by its security developers. For the other policies, however, I developed synthetic test cases based on the information in the input policy. Essentially, I generated test properties as boolean functions consisting of attribute and values taken from the input policy being tested. The generated boolean functions were serialized to a XACML policy set and an expected outcome also randomly chosen.

The results for verification are shown in Figure 8.3. Due to expressiveness limitations, Margrave was able to load only the Continue and Network policies, which consist of simple prerequisites in `Target` and no data-type functions. Please note that the loading – which includes parsing the policy and converting it to the corresponding logic format – for my approach is relatively stable; i.e., in in all cases it takes a few seconds to parse the policy and convert it to an OWL-DL knowledge base. The conversion time for HSAT, on the other hand, varies wildly: in the cases of eXist, Fedora and GAAA which use functions in the `Condition` element, it takes 100+ seconds (HSAT timed out while loading the GAAA policy after 16 minutes). With respect to verification time only, my approach exhibits the slowest performance, however notice that in all cases it still takes only around a second, which is acceptable performance for compile-time policy analysis; also, notice that the dominant component in all approaches seems to be policy loading and conversion time.

8.2.2 Policy Comparison Results

Policy comparison refers to checking for semantic differences between two policies, i.e., finding all possible access requests where the policies would return different decisions. To evaluate this service, for every policy in the test suite, I performed 1) a

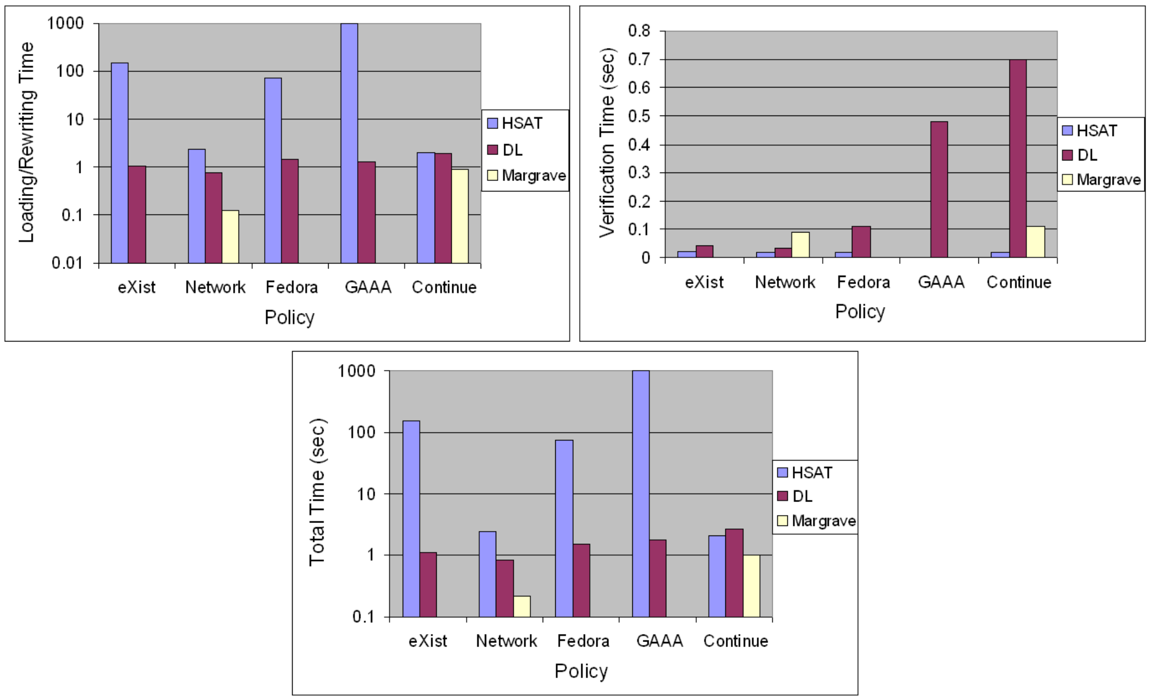


Figure 8.3: **Verification results for OWL-DL-based, HSAT and Margrave.** Times shown are for formal *verification* of security properties. Top left figure contains the time for parsing the XACML policy and loading/converting into the appropriate structure (BDD, conjunctive formula, OWL-DL ontology). The top right figure contains the verification time, once the structure is converted. Bottom figure contains aggregate (loading+verification) timings.

comparison of the policy against a copy of itself and 2) a comparison of the policy against a modified version of itself (5 randomly selected rules were removed to produce this modified copy). The results are shown in Table 8.4; they represent the average of 5 runs.

Notice that the results for policy comparison results are similar to the ones for verification, since in all approaches the services are reduced to the same basic reasoning tasks. The main difference between HSAT and my approach (DL) on one hand, and Margrave on the other is that Margrave finds all possible differences between policies in a very fast manner. My analysis framework is not optimized for finding all differences, so it only presents the first difference between the two policies that it finds (if a difference

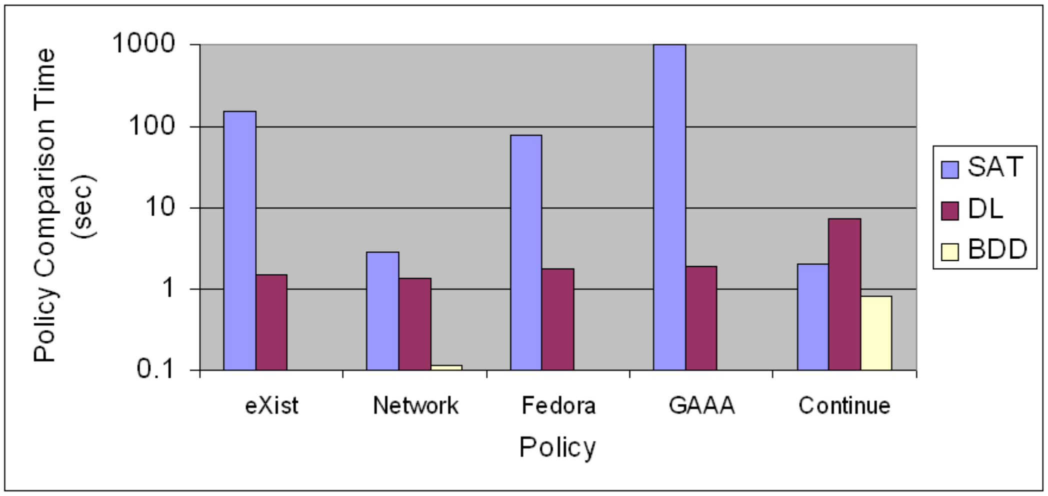


Figure 8.4: **Policy comparison results for OWL-DL-based, SAT and Margrave.** Times shown are for checking policy equivalence. The figure contains the aggregate (loading+comparing) timings.

exists).

8.2.3 Redundancy Checking Results

Finally, I also performed redundancy checking of each `Rule`, `Policy` and `PolicySet` element in the policy test suite. Since the other tools do not support this services, only the results of my analyzer are shown in Table 8.5. Notice that Continue takes significantly more time than the others because of its nesting and interlinking; a single `Policy` element might be included in different `PolicySets`, so it becomes more involved to check all of the possible implications if the particular `Policy` element is removed.

Interestingly, the analyzer did find a redundant `Policy` and `Rule` in Continue's policy: the third `Policy` in PPS_paper-assignments_rc `PolicySet`. Upon closer inspection, it seems that whenever the prerequisite of the third `Policy` is matched, the first `Policy` in that set will fire as well, and since the `PolicySet` is using the `First-Applicable` combining algorithm, the decision of the first `Policy` will always override the third one.

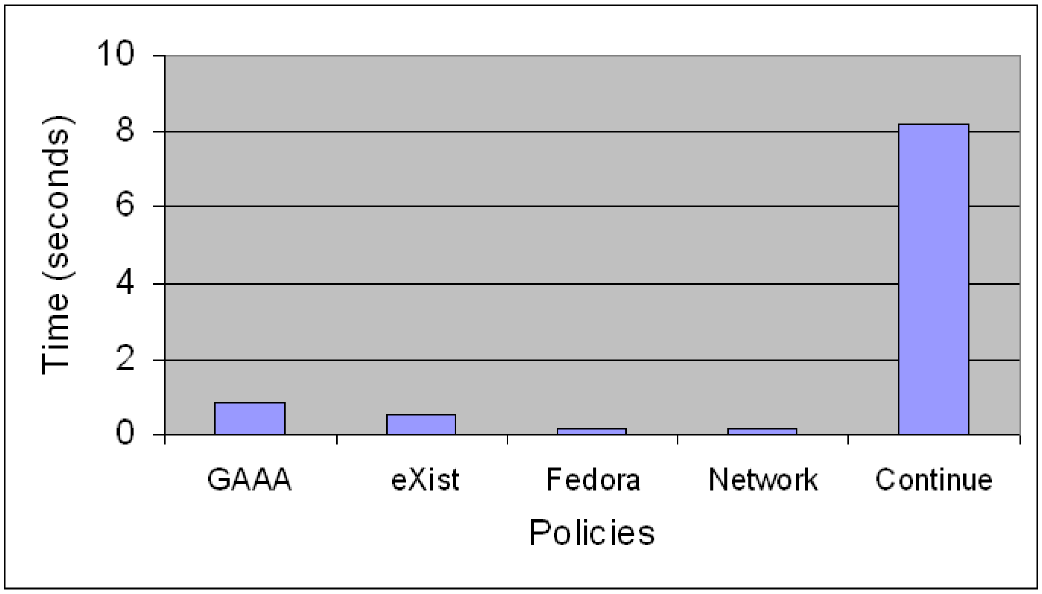


Figure 8.5: **Checking redundancy of `Rule`, `Policy` and `PolicySets`.** Times shown are aggregate (loading the policy, converting it to OWL-DL form and processing each policy element).

8.2.4 Extended Policies

Since the performance of my approach was on the order of few seconds for all of the policies in the suite, I decided to test its scalability for larger versions of the test suite policies. For this purpose, I implemented a tool that for a given input policy generates a 'slightly' modified version of the policy. In particular, the modified policy has the same structure as the original (same number of `Rules`,`Policies` and `PolicySets` and same relationships between them); the only difference is that the constants that are used as attribute values in the policies are randomly permuted and new attribute values are added. Using this tool, for each policy set (such as Continue), I generated a meta-policy that combines the original one with a number of slightly modified versions of it. An example of such meta-policy is shown below :

```
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  PolicySetId="top"
  PolicyCombiningAlgId="...:first-applicable">
  <Target />
  <PolicySetIdReference>continue</PolicySetIdReference>
  <PolicySetIdReference>continue1</PolicySetIdReference>
  <PolicySetIdReference>continue2</PolicySetIdReference>
  <PolicySetIdReference>continue3</PolicySetIdReference>
</PolicySet>
```

In the example above, continue refers to the original policy, whereas continue1, continue2, continue3 point to slightly modified versions. The goal of extending policies in such manner is to test the analyzers on larger policy sets that still preserve the structure of the original policy.

The formal verification results for extended versions of Continue and Fedora are shown in Tables 8.6 and 8.7. As the size of Fedora and Continue increases, my approach scales very well. Most of the performance degradation comes from increased verification time, while the loading and conversion time remains stable.

In the case of Continue, notice that Margrave is unable to load the larger policies. This is because each attribute/value pair from the policy is mapped to a node in the BDD structure used by Margrave. BDDs trade memory for speed, so the size of the structure is exponential to the number of variables (i.e., attribute-value pairs) in the worst case – this causes memory problems when loading larger policies.

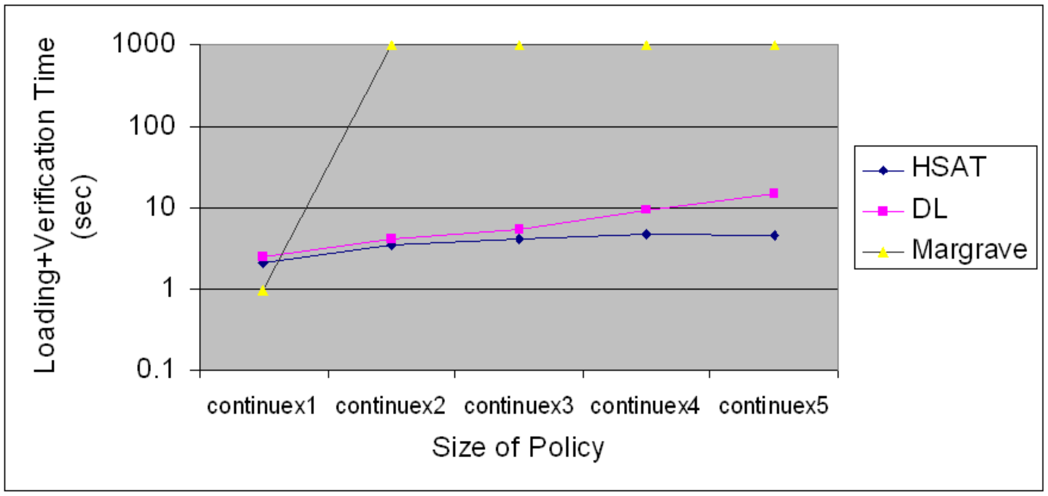


Figure 8.6: **Testing the policy analyzers on an extended version of Continue.** continuex1 refers to the original version of the policy, continuexn denotes a policy that is n times large than the original. The Y-axis denotes the total time needed to load the policy and verify all 11 security properties.

8.2.5 Summary

To show that the performance of my analysis framework is comparable to that of the fastest XACML analyzers available (Margrave and HSAT), in this first part of the evaluation I used a policy test suite consisting of 5 publicly available real XACML policies. The selected policies selected are relatively inexpressive but also non-trivial: they use a limited set of XACML features (so the other approaches can process them), but each policy contains dozens of XACML `Rule` and `Policy` elements and they are fairly nested and interconnected. The results from this section show that even for these limited fragments of XACML where the propositional-logic based approaches are expected to dominate, my tool performs surprisingly well and is comparable overall to the other approaches.

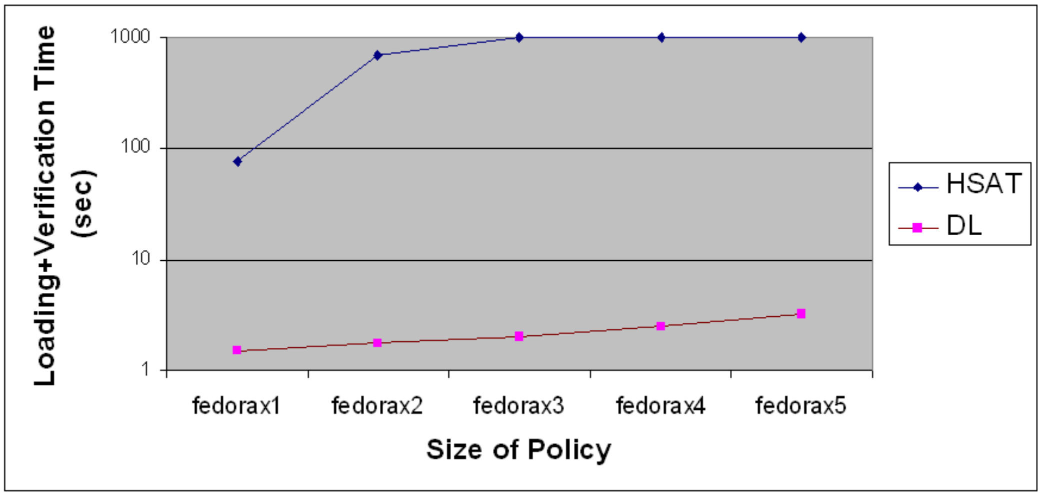


Figure 8.7: **Testing the policy analyzers on an extended version of Fedora.** fedorax1 refers to the original version of the policy and fedoraxn denotes a policy that is n times large than the original. The Y-axis denotes the total time needed to load the policy and verify 5 randomly generated security properties.

8.3 NASA Federated Data Access Use Case

In order to evaluate the more expressive features of the analysis framework, such as policy domains and datatypes, I collaborated with NASA HQ to develop an expressive set of realistic XACML policies [118]. Given the immense amount of heterogeneous data generated in different parts of the agency, information management is a major challenge for NASA. Several efforts are underway across the agency to use semantic technologies, including RDF and OWL, to respond to this information integration challenge. Some examples of Semantic Web applications at NASA include: BIANCA [2], an RDF-based data integration application, POPS [81] which provides faceted browsing across different domains and the NASA Taxonomy [49], which is a controlled vocabulary consisting of 7 hierarchies used to facilitate interoperability and search.

The difficulties of integrating and managing information across different organizations in the agency is manifested in the domain of access policy management. In particular, integrating from multiple data sources presents challenges in policy management since the data sources typically have heterogeneous access control policies. Determining how the policies of the constituent sources align and the subsequent specification of a policy for the integrated data is a labor-intensive process that is time consuming and error-prone. Because of these issues (referred to as the NASA HQ Federated Data Access Use Case) NASA HQ is interested in approaches that provide flexible and expressive policy management for heterogeneous and distributed policy sets [118].

For the purpose of this dissertation, I collaborated with representatives from NASA HQ to develop a set of policies for the Federated Data Access Use Case. The application we investigated was BIANCA (the name is an acronym for Business Impact Analysis for Networked Computer Assets). BIANCA provides a single integrated view of information about (including relationships between) applications, servers, network services, networks and change items for NASA HQ. BIANCA analyzes dependencies between these assets in order to provide services like repair plans, outage cost estimates, and dependency reports. Users can query across the federated information store and browse the data in a web browser in order, for example, to track the impact that a failure of one system, subsystem, or application would have on other systems and customers. BIANCA has an RDFS data reference model that can be reused by other applications.

8.3.1 BIANCA Policy Set

This section contains a description of the BIANCA Policy set that we developed. The policy follows the Role-Based Access Control (RBAC) model; thus, there is a separate XACML `PolicySet` that contains the set of permissions for each role. Additionally, the permissions are split among the different types of resources – these include the four broad categories of networks, network services, servers and applications. A fragment of the BIANCA policy is shown in Table 8.2.

Department	Role	Action	Resource	Decision
Financial Operations	Intern	Read	ExpenseReport	Permit
	Employee	Read OR Write	ExpenseReport	Permit
	Project Manager	Write or Read	Any	Permit
	Any	Any	Any	Deny
Missions & Projects	Scientists(Internal)	Read	InternalPaper	Permit
	Scientists(Internal)	Write	InternalPaper	Permit
	Scientists(External)	Read	PublicDoc	Permit
	Project Manager	Read OR Write	Any	Permit
	Any	Any	Any	Deny
Institutions and Management	CivilServant	Write	OrgHierarchy	Permit
	Employee	Read	OrgHierarchy	Permit
	Project Manager	Write	Any	Permit
	Any	Any	Any	Deny

Table 8.2: **Fragment of BIANCA Policy Set.**

We used the NASA Taxonomy [49] to describe policy entities such as types of resources. The Taxonomy includes seven facets, some of which can be used as policy vocabularies to express roles, access controls, competencies and organizations in NASA. Given that the taxonomy is in SKOS format, for this evaluation I converted it to OWL first and used the XACML-OWL coupling mechanism described in Chapter 6 to extend the XACML policy set.

In total, there are 4 main policy sets: a general, agency-wide policy representing rules that each department-specific policy has to conform to and three department policies for Institutions and Management, Missions & Projects and Financial Operations. Each departmental `PolicySet` in turns contains a policy for each type of role (the number of roles ranges from 5 to 50).

8.3.2 Empirical Results

Given the BIANCA policy set described above, the analysis services I tested were formal verification and policy containment. For formal verification, I used a number of queries that emerged from our discussion with representatives from NASA:

- Q1 – *PII*: For a policy P_a, check if there is a way for an access request to get a `Permit` from P_a without disclosing personally identifiable information (PII). PII is expressed as a XACML `Policy` containing a combination of attributes: SSN, street address, full name, IP address, email, etc.
- Q2 – Compliance of department-specific policies to general policy: whenever the base policy yields a `Permit` (resp. `Deny`), check if the department-specific policy will also return a `Permit` (resp. `Deny`).
- Q3 – *Disjointness* query : For two policies P_1 and P_2, check if there can exist an access request s.t. both policies apply to it.

The results of evaluating these queries are shown in Figures 8.8,8.9 and 8.10. Notice that even in the case with the largest policy set, the total processing time for all of the queries was less than 10 seconds, which the NASA engineers felt was more than acceptable given the size of the policies and the task at hand. Most of the running time is spent on loading and converting the Policy; this is because the policy refers to various segments of the OWL-based NASA Taxonomy, so the NASA ontologies are also loaded during this step. Because the ontologies are relatively inexpressive, reasoning about them is very fast (order of milliseconds), so the only noticeable overhead is during loading the policy.

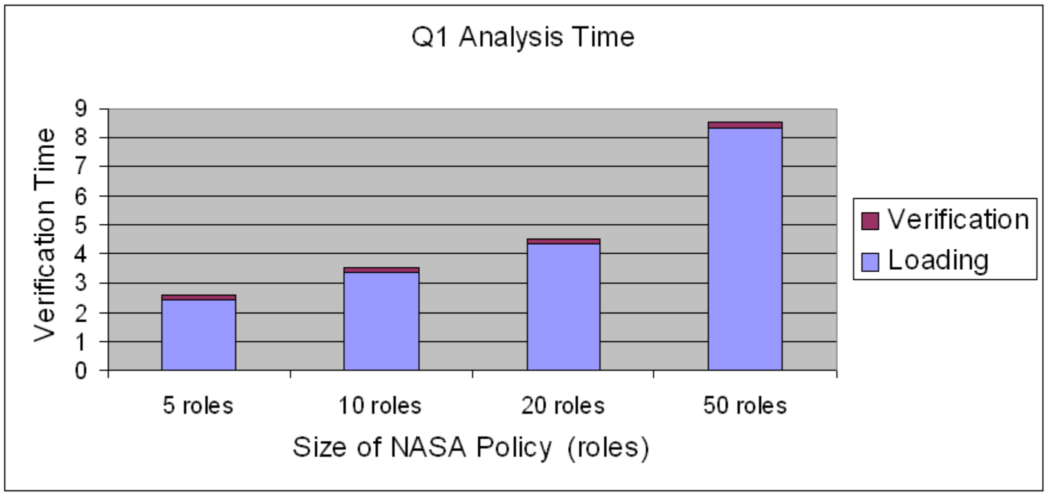


Figure 8.8: **Analysis time for the Personally Identifiable Information Query on various policies addressing the NASA HQ federated data access use case.**

8.4 Healthcare Access Policy

This final section of the chapter presents an empirical evaluation of my framework using a real-world access control policy from the healthcare domain. Similarly to the NASA access use case, the healthcare policy motivates the need for vocabulary domains and datatypes in policies; unlike the NASA policy, this use case is fully developed as part

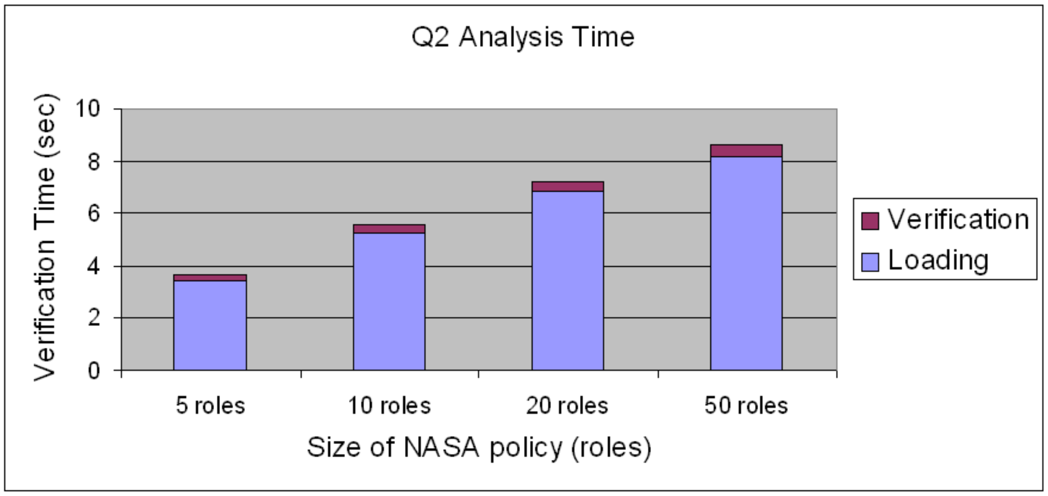


Figure 8.9: **Analysis time for the Policy Comparison (compliance) query on various policies addressing the NASA HQ federated data access use case.**

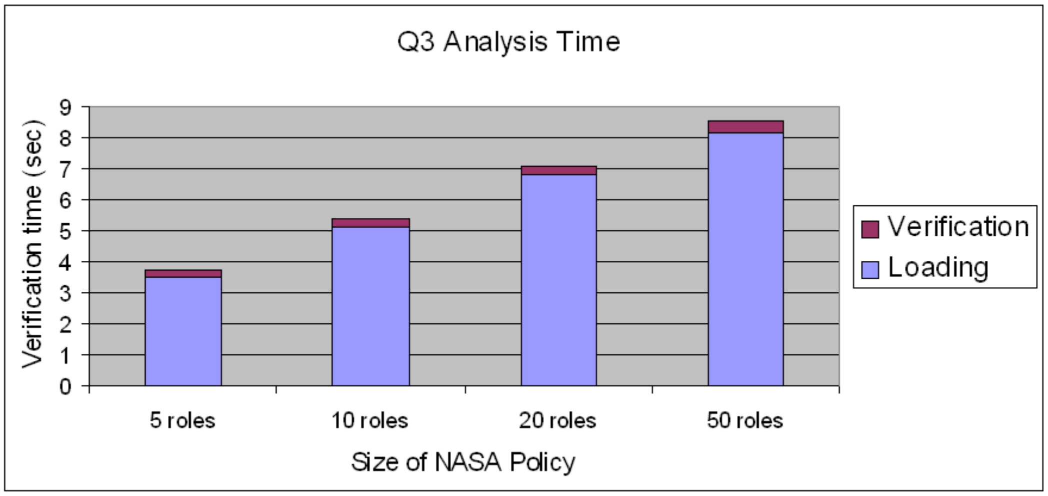


Figure 8.10: **Analysis time for the Policy Disjointness Query on various policies addressing the NASA HQ federated data access use case.**

of an ongoing international standardization effort (HL7 [1]). In the following, I provide a brief background on the developments of standards for accessing patient's electronic health records and then present the RBAC policy that is provided as part of the HL7 standard. I converted this RBAC policy to XACML form and used it to evaluate my XACML analysis tool. The empirical results along with some discussion are presented in Section 8.4.4.

8.4.1 Background: HL7 and Electronic Health Records

Health Level 7 (HL7) is an international community of healthcare subject experts and information scientists collaborating to create standards for the exchange, management and integration of electronic healthcare information. The goal of HL7 is to have a degree of interoperability among healthcare providers such that health information can be exchanged so that a patients medical information can be made portable and available to his/her clinicians (at least to the extent that the patient allows it to be).

The aspect of guarding patient's privacy and confidentiality is crucial in this setting. Because of this, a major component of HL7 called the Security Technical Committee[2] deals with specifying security policies for controlling access to patient's confidential information. The standardization work produced by this committee includes a set of role-based permissions [62] and recommended access policy scenarios [5] that were directly used to evaluate the performance of my policy analysis framework.

[1]Health Level 7 is an accredited Standards Developing Organizations operating in the healthcare arena. HL7s domain is clinical and administrative data.

[2]http://www.hl7.org/Special/committees/secure/index.cfm

8.4.2 HL7 RBAC Policy

Representatives from HL7 have worked closely with the US Department of Veteran Affairs (VA) to develop a standard set of permissions for healthcare access policies. The reason is that VA currently has the largest electronic health record system in the country, with more than 4 million patients, 180,000 medical personnel and deployed in more than 160 hospitals throughout the US [7].

Working with representatives from VA, the HL7 committee has published a standard of role-based access control (RBAC) permissions [62] recommended as building blocks for access control policies for health information systems. This set of permissions will be referred to as HL7-RBAC. An RBAC policy associates a role in the organization with a set of permissions for that role. Since users are not assigned permissions directly, but only acquire them through their role (or roles), management of individual user rights becomes a matter of simply assigning the appropriate roles to the user.

As part of HL7-RBAC, two general categories of roles are allowed: *structural* and *functional* roles. Functional Roles consist of all the permissions needed to perform a task. Functional role names are associated with groups of permissions for convenience in assigning to users. Structural roles, on the other hand, denote the placement of people in the organizational hierarchy as belonging to categories of healthcare personnel warranting differing levels of access control. Examples of structural roles include *Clinician*, *Patient*, *PhysicalAssistant*, etc. There are no permissions associated with structural roles – they are simply used as prerequisites for functional roles.

8.4.3 Converting HL7-RBAC to XACML

HL7-RBAC is specified using a tabular format in [62], so I converted the policy to XACML first in order to evaluate my analyzer. The conversion was done as follows: a functional role with an associated set of permissions was converted to a XACML `Policy` which contains the permissions as part of its `Target` elements. For example, the functional role *PPD-036* which contains create permissions for *New Patient/Family Preferences* is expressed in XACML as:

```
 <Policy  PolicySetId="PPD-036"
    RuleCombiningAlgId="...combining-algorithm:first-applicable">
 <Target>
   <Actions>
     <Action>
     <ActionMatch MatchId="...:string-equal">
        <AttributeValue DataType="...#string">
           create
         </AttributeValue>
        <ActionAttributeDesignator AttributeId="action-type"
           DataType="...#string"/>
      </ActionMatch>
   </Action>
   </Actions>
 <Resources>
     <Resource>
       <ResourceMatch MatchId="...:string-equal">
         <AttributeValue DataType="...#string">
            Patient_Preferences
          </AttributeValue>
         <ResourceAttributeDesignator
           AttributeId="resource-type"
           DataType="...#string"/>
       </ResourceMatch>
     </Resource>
   </Resources>
 </Target>
   <Rule RuleId="rule" Effect="Permit"><Target/></Rule>
</Policy>
```

Since structural roles are prerequisites for the permissions in functional roles, they are added to the `Target` element of a policy representing a functional role. Because XACML does not have built-in support to express the type of semantic hierarchies needed for structural roles [15], I used an OWL ontology to capture these hierarchical relationships. In addition to expressing relationships between structural roles, I also used OWL to provide integer and date-time datatype support (e.g., Regular Doctors can access Patient Records during business hours, whereas ER Doctors should have access at all times) and role-based cardinality constraints (e.g., Chief-of-Staff role has cardinality of 1).

8.4.4 Empirical Results

The standard documents discussed above present the basic building blocks of an access policy: a set of permissions and a set of roles that are to be associated with those permissions. Given these permissions, the Department of Veteran Affairs has presented a reference collection of RBAC scenarios [5] that denotes the access control policies used in their health information system. The reference collection contains 39 scenarios, and each of them describes a situation where a particular structural role (or a combination of roles) is presented and set of permissions is given that is required for that role.

I converted these RBAC scenarios to a XACML policy by associating the structural roles referenced in the scenarios with the corresponding functional roles described in the previous section. The end result of this conversion is a large and expressive XACML policy (referred to as HL7-RBAC) that was used to evaluated the analysis services provided by my framework.

HL7-RBAC contains a total of 107 `PolicySets`. It uses only 3 different attributes (*action-type*, *subject-role-type* and *resource-type*), however it has 102 different values: 5 values for action attributes (*create, read, update, delete, execute*), 51 attribute values to represent objects(resources) and 46 attributes to represent subject roles types (*Physician, Patient, Clerk,* etc.).

The two analysis services tested against HL7-RBAC were formal verification and policy comparison. The results for these services are shown in Figure 8.11 and Figure 8.12, respectively. Since for both of these services, the analyzer performed surprisingly well (a few seconds to verify properties and perform comparison), I also experimented with an extended version of the policy. Thus, HL7x2 and HL7x3 in the figures refers to a policy that is two and three times larger than the original HL7-RBAC. I should note that the the analysis framework scales very well as the size of these policies increases. Also, I should note that these are very large policies – HL7x5 for example, refers to a dataset with 535 `PolicySet` elements, and 250 attribute values.

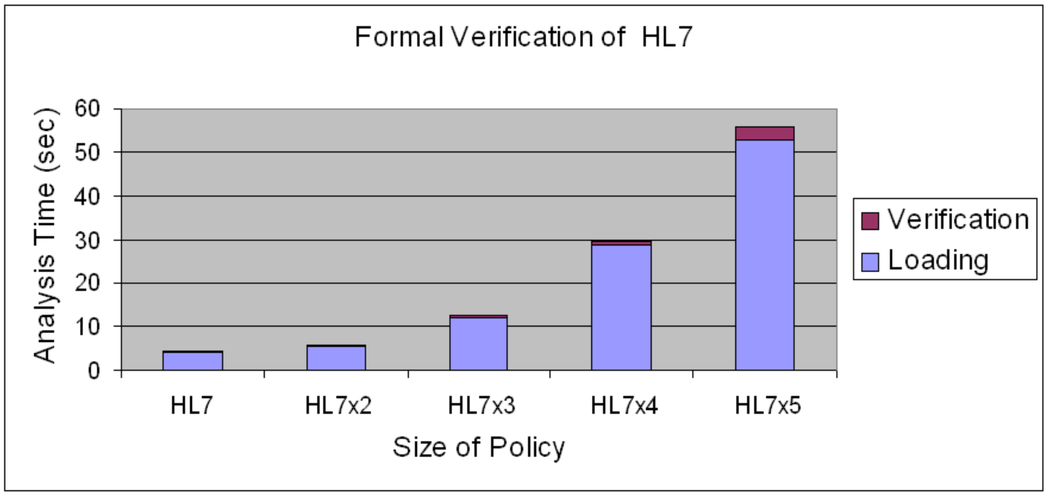


Figure 8.11: **Formal Verification of HL7 policy.** Please note that HL7x1 refers to the original policy whereas HL7xn refers to a meta-policy containing n policies of the same size and structure as HL7.

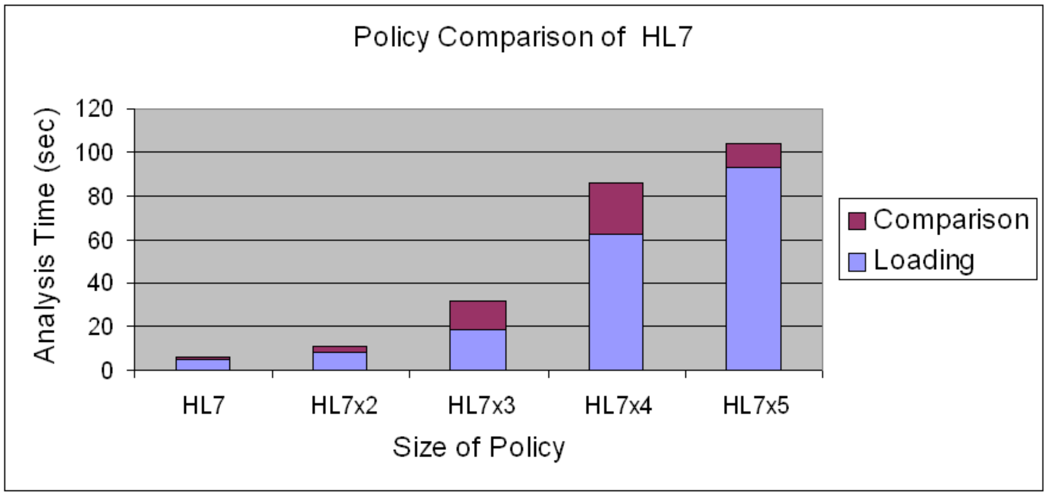


Figure 8.12: **Policy Comparison timings for HL7 policy.** Please note that HL7x1 refers to the original policy whereas HL7xn refers to a meta-policy containing n policies of the same size and structure as HL7.

8.4.5 Summary

The healthcare policy described in this section is a great use case for my analysis framework: it is large, with dozens of roles and hundreds of attribute values and it is expressive (with RBAC constraints, data-types and domain vocabularies for heterogeneous policies). By evaluating my analysis tool on this policy (after I converted it to XACML) I showed that my approach can formally verify and compare large, real-world policies in a practical manner. This use case also demonstrates the benefits of grounding the analysis framework on OWL-DL, since I used OWL ontologies to represent structural role hierarchies. Note that there has already been interest in formalizing the HL7 reference model in OWL [108] – I leveraged some of this previous work to develop the role hierarchies. Finally, I have made my XACML version of the HL7-RBAC policy publicly available [3] so it can be used as a benchmark to evaluate other XACML processors.

[3] Available at `http://www.mindswap.org/~kolovski/hl7rbac.zip`

8.5 Discussion

The second part of the evaluation used real world access control policies with expressive features such as policy vocabulary domains, datatype functions and policy constraints. The policies used (NASA HQ Data Access Use Case and Healthcare RBAC Policy) are a great match for the analysis framework since they both motivate the need for expressive background domains (ontologies) in order to facilitate information exchange and interoperability across different parts of an enterprise system. Both of the policies are quite large: 100s of `PolicySet` elements and 100+ attribute values in each case. I showed through empirical testing that the analysis framework scales very nicely as the size of these policies increases. Considering that the framework is supposed to be used at development time, before the policies are deployed (so the performance requirement is not as critical as a deployed policy engine), the empirical results shown in this section prove that my analyzer is practical for very large and expressive policy sets.

Chapter 9
Conclusions and Future Work

In this thesis, I identified some important challenges facing XACML; namely, a lack of understanding of its formal properties, and a lack of automated, design-time analysis services that provide support for distributed and heterogeneous policies.

To address the first issue above, in Chapter 4 a formal, proof-theoretic semantics for XACML 3.0 was given using natural deduction rules. The semantics covers the core of XACML along with its Administrative Profile, and represents a concise and formal version of the informal semantics given in the official language specification. To determine XACML's complexity properties, I provided a translation of various subsets of the language to locally stratified Datalog in Chapter 5, thus establishing its polynomial data complexity and close relationship to other logic-based languages such as the Flexible Authorization Framework. Additionally, I showed that access request checking in XACML with cyclical references between policies is NP-complete.

The other issue addressed in this dissertation is the lack of compile-time services that can detect inconsistencies in the presence of *distributed* and *heterogeneous* XACML policies. This issue has been motivated by recent interest in extending XACML with both distributed policies (Administrative Policy Profile in XACML v3.0) and rich policy domain models that provide additional expressiveness [13, 47] and support for integration of policies about different types of data resources [40, 21, 118].

Given these requirements, in Chapter 6 an analysis framework was presented that can reason about both distributed and heterogeneous XACML policies. In particular, I showed how with a mapping to Description Logics, we can do change analysis, formal verification and coverage checking for XACML policies using DL reasoners. Since Description Logics are the logical formalism underlying the OWL-DL sub-language, the analysis framework presented in this thesis can easily cover XACML policies extended with OWL ontologies [40]. This compatibility with OWL – which is a W3C standard for representing information on the Web – and the full mapping of the Administrative Profile presented in Chapter 6, enables the analysis framework to cover the expressive features of XACML that allow for distributed and heterogeneous policies.

In Chapter 7, I demonstrated that the analysis framework can be applied to domains other than access control. In particular, I mapped WS-Policy (a web services policy language) to OWL-DL and provided the same analysis services as for XACML.

Finally, in Chapter 8 I presented my prototype implementation of a XACML-to-DL mapper and policy analyzer. Since DLs in general have very bad worst case complexity, I showed through an extensive performance evaluation using real world policy sets that OWL-DL reasoners are practical for static policy analysis. The performance results were presented in Chapter 8.

9.1 Contributions

The contributions of this thesis are as follows:

- A formal, proof-theoretic semantics of XACML v3.0 that covers the core specification and the Administrative Policy Profile.
- A mapping of XACML to Datalog that provides a model-theoretic semantics and

computational complexity results for full XACML and various fragments. Additionally, an extensive comparison with other logic-based languages such as Flexible Authorization Framework and SecPal based on the Datalog mapping.

- A static analysis framework based on Description Logics that can reason about expressive XACML policies. Using a XACML-to-Description Logics mapping, a comprehensive set of services are provided: formal verification, policy comparison (change analysis) and redundancy checking.

- Demonstrated more general applicability of the framework by using it to formalize and analyze policies in the web services domain.

- An empirical evaluation of the scalability of the analysis framework using real world policy data sets.

9.2 Future Work

In this section, I will discuss the limitations and open issues of this dissertation. Additionally, I discuss possible avenues of future work, split in two sections: extensions of theoretical framework and improvements to analysis services support.

9.2.1 Theoretical Framework

9.2.1.1 Dynamic Environment and Obligations

In this dissertation, an assumption is made that the response to an access request is simply yes/no, i.e., it does not change the state of the policy environment. However, it has been recognized that a yes/no response to every scenario is insufficient for many modern systems and applications [33]. Many policies require certain conditions to be satisfied and actions to be performed before or after a decision is made. Thus, there has been great amount of interest in policy languages that support dynamic environments [33, 48, 69]. For example, XACML allows for policy *obligations*, meaning that after being granted access, the subject *must* perform the actions associated with the grant.

Most logic-based formalisms for dynamic policies are based on First-Order temporal logics [103, 94], which cannot be easily combined with Datalog (or Description Logics, for that matter). An interesting area of future work involves exploring decidable extensions of the logic-based formalization presented in this thesis with dynamic policies.

9.2.1.2 Extensions of XACML

In Chapter 5 I showed that there exists a polynomial reduction of XACML to a variant of Datalog. Given this mapping, and the existence of numerous policy languages coming from academia and industry that are also based on Datalog, one possible direction of future work is to investigate extending XACML with features from these languages.

A potentially useful feature of the Flexible Authorization Framework (FAF) is its *history table* , i.e., a table whose rows describe the access requests already processed. A history table can be used to model various policy constraints such as the Chinese Wall security policy, and will not change the computational complexity of XACML if it were added. Also, combining algorithms such as most-specific-overrides and no-overriding can also be easily adapted to XACML: most-specific-overrides can be performed by comparing the `Targets` of the two policy elements to be combined, whereas for no-overriding a clause can be added to throw `Indeterminate` whenever conflicting access decisions are returned.

Other, non-trivial features worth exploring are *delegation constraints*. While XACML does have an Administrative Policy Profile, there is no explicit support in it for common delegation constraints such as *depth-bounded delegation* (i.e., limiting the number of times an access decision can be delegated) or *threshold-constrained trust* (i.e., at least k out of n given principles must sign a given access request). Both of these features, among others, are supported by Datalog-based languages such as as SecPal [27] and Delegation Logic [87], so an interesting research problem would be how to incorporate them into XACML.

9.2.2 Analysis Services

9.2.2.1 Computing All Differences between Policies

One of the analysis services provided in Chapter 6 was change impact analysis. A limitation of the approach is that when comparing two policies, if the policies have differences among them, then the analyzer returns only one access request corresponding to one difference. As part of future work, I would like to extend this approach in order to provide comprehensive change impact analysis, as done in Margrave [50]. To provide this service, I plan to use a technique based on computation of all models that admit a Description Logic concept expression, which is done by tableau saturation. Saturation of the tableau completion graph is done by continuing application of the completion rules until all choice points are explored and no more rules are applicable. While most DL reasoner optimizations are not applicable during saturation, there are still some optimizations that can be used (such as absorption), and it is possible that other significant optimizations could be applied due to the nature of the XACML-to-DL mapping.

9.2.2.2 Policy Repair

Most of the analysis services discussed in the thesis help policy developers discover errors in their policies, be it unauthorized access after an update, or corner cases that do not satisfy given test cases. The next step is to help policy developers recover when they do find errors in their policies. To illustrate this service, consider the example in Figure 9.1 below.

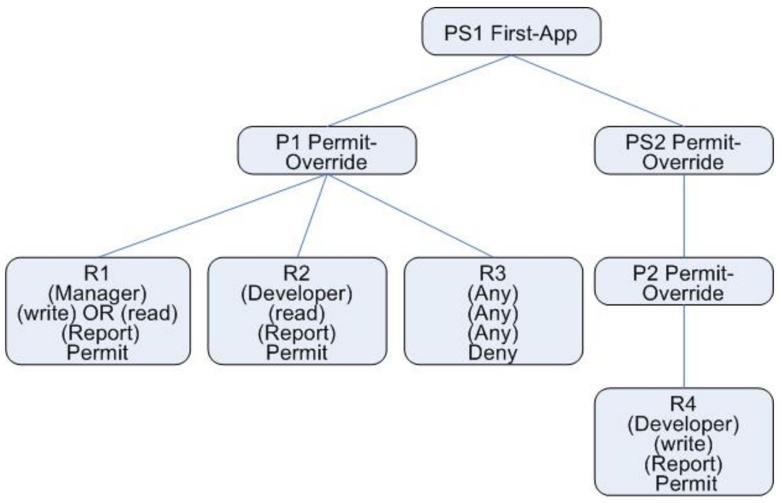

PS1 First-App
P1 Permit-Override
PS2 Permit-Override
R1 (Manager) (write) OR (read) (Report) Permit
R2 (Developer) (read) (Report) Permit
R3 (Any) (Any) (Any) Deny
P2 Permit-Override
R4 (Developer) (write) (Report) Permit

Figure 9.1: Example Policy

The security property is : *Developer* should not be able to *write* to *Report*. When checking the policy against this property, our analyzer returns two counter examples:

role=Manager, role=Developer, action=write, resource=report

role=Developer, action=write, action=read, resource=report

Thus, if a requester comes along that is a member of both roles (*Manager* and *Developer*), then she can gain *write* access to *Report*. The other way for a *Developer* to gain write access is if he tries to both *read* and *write* to *Report* at the same time. To prevent unauthorized access in this example, two constraints can be added; first one to make *Manager* and *Developer* mutually exclusive roles, and the second one to make sure that only a singleton value for the *action* attribute of a request is admitted.

However, there are also other ways to repair the policy. For example, if the goal is to add as few axioms as possible to fix the policy, then there exists a better solution, involving only one constraint:

$$\exists role.\{Developer\} \sqsubseteq \neg\exists action.\{write\}$$

This axiom explicitly prohibits developers from writing to *anything*. This constraint is adequate for our toy policy because there is only one type of resource that can be written to (*Report*). In a larger policy, we might not necessarily want to prevent Developers from writing to other types of resources. In that case, we will need somehow to measure and to compare the *impact* of each individual repair strategy.

The idea of policy repair was inspired by recent work by Kalyanpur et al. [79] in repairing unsatisfiable concepts in Description Logics. There the authors present a new DL service (ontology repair), which semi-automatically selects which axioms need to be removed (or rewritten) from the ontology in order to render the concept satisfiable. To select likely candidate axioms for removal, their work uses algorithms that rank the axioms depending on a set of criteria. There is one essential difference between the work by Kalyanpur et al. and the future work proposed here: whereas in their case axioms are *removed* to fix errors in ontologies, in my case axioms (representing policy constraints) will be *added* to the KB in order to fix access control errors in policies.

The problem of which axioms need to be added to make the policy concept unsatisfiable is hard since there are infinitely many solutions. However, for the purposes of access control policy repair, the search space could be constrained to DL axioms that correspond to policy *constraints* in the analysis framework. In that way, finding repair plans will be easier, and presenting them to policy developers will be easier since no knowledge of Description Logics will be required. As a first step, the following common types of policy constraints could be used: separation of duty, role cardinality constraints, singleton values for attributes and role hierarchies. As part of future work, I plan to devise an algorithm to compute sets of axioms that need to be added and then investigate possible ranking strategies similar to the ones in [79], but adapted for our access control scenario.

9.2.2.3 Extending and Evaluating Redundancy Checking

In Chapter 6 I presented an algorithm for determining if a policy is redundant. Additionally, I presented an algorithm for finding all non-redundant policy subsets for a given policy set. Considering that computing all of the non-redundant subsets is likely to be non-practical for large policies (e.g., with more than 20-30 `PolicySets`), a possible next step would be to devise an approach that considers user input on which of the redundant policies to remove. For example, users could rank different policies based on specific criteria: size of policy, number of children, etc., and the analysis tool would generate a non-redundant policy subset based on these criteria. A comprehensive evaluation (possibly through a user study) of such an approach would have to be performed to determine whether the tool provides enough information to guide users during the process of removing redundant policies.

9.2.3 Analyzing Business Rules

An interesting research problem to pursue would be applying the analysis framework to the domain of business rules. Business rules are much more general than access control policies; they specify the operations, definitions and constraints that apply to an organization. There exist a few business rule languages proposals [76, 6], and there has been a great amount of interest in rule engines that support verification, accountability and enforcement of business rules.

Appendix A

Proofs

*Proof of Lemma 6, **If** case.* For a request RQ, we need to show that:

$$\textbf{if}\ \mathrm{RQ}, \mathrm{T} \models_m \texttt{True}\ \textbf{then}\ \mathrm{KB} \models \pi(\texttt{True}, \mathrm{T})(\pi(\mathrm{RQ})) \tag{A.1}$$

The proof will be done by structural induction. We will show that (A.1) holds for Match elements first, and afterwards, by going through the XACML semantics rules, we will show it also holds for Target elements.

Match Element I will show that the following holds:

$$\textbf{if}\ \mathrm{RQ}, \mathrm{M} \models_m \texttt{True}\ \textbf{then}\ \mathrm{KB} \models \pi(\texttt{True}, M)(\pi(\mathrm{RQ})) \tag{A.2}$$

In order to show that $\mathrm{KB} \models \pi(\texttt{True}, \mathrm{M})(\pi(\mathrm{RQ}))$ we need to show that the expression $\pi(\mathrm{RQ}) \sqcap \neg\pi(\texttt{True}, \mathrm{M})$ is unsatisfiable. In Sections 6.1.2 and 6.1.1 the DL mapping was defined as:

$$\begin{aligned} \pi(\texttt{True}, \mathrm{M}) &\equiv \exists\pi(\mathrm{AD}_M).\pi(\mathrm{fcn},\ \mathrm{AV}_M) \\ \pi(\mathrm{RQ}) &\equiv \bigsqcap (\exists \mathrm{r}_{AV}.\pi(\mathrm{type}_{AV}\text{-equal},\ \mathrm{AV}))\ \text{ forall AV} \in \mathrm{AT}, \mathrm{AT} \in \mathrm{ATS}, \mathrm{ATS} \in \mathrm{RQ} \end{aligned} \tag{A.3}$$

To prove (A.2), it needs to be shown that:

$$\forall \pi(\mathrm{AD}_M).\neg\pi(\mathrm{fcn}_M,\ \mathrm{AV}_M) \sqcap \bigsqcap \left(\exists \mathrm{r}_{AV_i}.\pi(\mathrm{type}_{AV_i}\text{-equal},\ \mathrm{AV}_i)\right) \tag{A.4}$$

is unsatisfiable for some $\mathrm{AV}_i \in \mathrm{AT}, \mathrm{AT} \in \mathrm{ATS}, \mathrm{ATS} \in \mathrm{RQ}$.

Assume that the left hand side of the implication in Equation A.2 holds. By the inference rules in Table 4.6, $\mathrm{RQ}, \mathrm{M} \models_m \mathtt{True}$ means there exists an attribute value AV s.t. the following holds:

1. $AV \in AT, AT \in ATS, ATS \in RQ$
2. $\mathrm{type}_{AD_M} = \mathrm{type}_{AV}$, $\text{attr-id}_{AD_M} = \text{attr-id}_{AT}$, $\mathrm{issuer}_{AD_M} = \mathrm{issuer}_{AT}$, $\mathrm{cat}_{AD_M} = \mathrm{cat}_{ATS}$
3. $\mathrm{fcn}(\mathrm{AV}_M, \mathrm{AV}) = \mathtt{True}$

Condition 2) implies the DL role used in $\pi(\mathrm{AD}_M)$ will match the DL role used in the mapping of AV. (This is because the role names are generated as a function of attr-id, issuer, cat and Type, and they all match in this case.) Because of this match, the role filler of $\pi(\mathrm{AD}_M)$ will have the following DL expression occurring in its label:

$$\pi(\mathrm{type}_{AV}\text{-equal},\ \mathrm{AV}) \sqcap \neg\pi(\mathrm{fcn}(\mathrm{AV}_M)) \tag{A.5}$$

From condition 6) above we know that the value of AV belongs in the value space characterized by $\mathrm{fcn}(\mathrm{AV}_M)$. However, note that in our mapping $\pi(\mathrm{fcn}(\mathrm{AV}_M))$ creates a user defined datatype that has the same interpretation as $\mathrm{fcn}(\mathrm{AV}_M)$. This is because the user defined datatype in our mapping has the same base XML schema type, and the facets we use to limit the value space have the same semantics as the matching functions used in XACML (as shown in Table 6.1.7). Thus, $\pi(\text{type}_{AV}\text{-equal, AV})$ will belong in the space defined by $\pi(\text{fcn}(\text{AV}_M))$. That in turn implies that equation (A.5) will produce a clash, rendering the expression $\neg\pi(\texttt{True}, \text{M}) \sqcap \pi(\text{RQ})$ unsatisfiable and showing that:

$$\textbf{if } \text{RQ}, \text{M} \models_m \texttt{True} \textbf{ then } \text{KB} \models \pi(\texttt{True}, M)(\pi(\text{RQ}))$$

Indeterminate If $\text{RQ}, \text{M} \models_m \texttt{Indeterminate}$, that means that there is no attribute $AT \in ATS, ATS \in RQ$ that matches the attribute designator AD_M; additionally, the mustBePresent property must be set to true. As shown in Section 6.1.1, whenever an attribute AD_P is not present, we add (conjunct) the expression $\forall\pi(\text{AT}_P).\bot$ to $\pi(\text{RQ})$.

According to the mapping axioms in Section 6.1.2:

$$\neg\pi(\texttt{Indeterminate}, \text{M}) \equiv \exists\pi(\text{AD}_\text{M}).\top$$

Thus, to check satisfiability of $\neg\pi(\texttt{Indeterminate}, \text{M}) \sqcap \pi(\text{RQ})$, we have

$$\pi(\text{RQ}) \sqcap \exists\pi(\text{AD}_\text{M}).\top \tag{A.6}$$

Because $\text{RQ}, \text{M} \models_m \texttt{Indeterminate}$, we know that the attribute designated by AD_M does not occur in the request - thus, when checking for satisfiability of equation (A.6) we will get a clash, which proves that:

$$\textbf{if } \text{RQ}, \text{M} \models_m \texttt{Indeterminate} \textbf{ then } \text{KB} \models \pi(\texttt{Indeterminate}, M)(\pi(\text{RQ}))$$

Extending the proof to `ConjunctiveMatch`, `DisjunctiveMatch` and `Target` Elements. First I will show that

$$\textbf{if } \mathrm{RQ}, \mathrm{CM} \models_m \mathtt{E} \textbf{ then } \mathrm{KB} \models \pi(\mathtt{E}, CM)(\pi(\mathrm{RQ}))$$

where $\mathtt{E} \in \{\mathtt{True}, \mathtt{Indeterminate}\}$, and then inductively extend the proof for DM and T. I will only show for $\mathtt{E} = \mathtt{True}$ - the proof for the `Indeterminate` is very similar

Following the deduction rules in Table 4.2.1, if $\mathrm{RQ}, \mathrm{CM} \models_m \mathtt{True}$ then it must be that $\forall \mathrm{M}_{CM} : \mathrm{RQ}, \mathrm{M}_{CM} \models_m \mathtt{True}$. From the above proof for M, we also know that

$$\textbf{if } \mathrm{RQ}, \mathrm{M} \models_m \mathtt{True} \textbf{ then } \mathrm{KB} \models \pi(\mathtt{True}, \mathrm{M})(\pi(\mathrm{RQ}))$$

Taking the above axioms into account and using the mapping for $\pi(\mathtt{True}, \mathrm{CM})$ defined in Section 6.1.2, we can conclude that $\mathrm{KB} \models \pi(\mathtt{True}, \mathrm{CM})(\pi(\mathrm{RQ}))$.

Following the inference rules again, if $\mathrm{RQ}, \mathrm{DM} \models_m \mathtt{True}$ then exist at least one $\mathrm{CM}_i \in DM$ s.t. $\mathrm{RQ}, \mathrm{CM}_i \models_m \mathtt{True}$. From the previous proof for CM, we know that for this CM_i,

$$\mathrm{KB} \models \pi(\mathtt{True}, \mathrm{CM}_i)(\pi(\mathrm{RQ}))$$

Using the above result, and taking the mapping axiom for DM into account:

$$\pi(\mathtt{True}, \mathrm{DM}) \equiv \pi(\mathtt{True}, \mathrm{CM}_1) \sqcup \ldots \sqcup \pi(\mathtt{True}, \mathrm{CM}_n)$$

implies that $\mathrm{KB} \models \pi(\mathtt{True}, \mathrm{DM})(\pi(\mathrm{RQ}))$.

Finally, if $\mathrm{RQ}, \mathrm{T} \models_m \texttt{True}$ then $\forall \mathrm{DM}_T : \mathrm{RQ}, \mathrm{DM}_T \models_m \texttt{True}$. From the previous step for DM, we know that for all DM_T,

$$\forall \mathrm{DM}_T : \mathrm{RQ}, \mathrm{DM}_T \models_m \texttt{True} \rightarrow \forall \mathrm{DM}_T : \mathrm{KB} \models \pi(\texttt{True}, \mathrm{DM}_T)(\pi(\mathrm{RQ}))$$

Taking the mapping axiom for T into account:

$$\pi(\texttt{True}, \mathrm{T}) \equiv \pi(\texttt{True}, \mathrm{DM}_1) \sqcap \ldots \sqcap \pi(\texttt{True}, \mathrm{DM}_n)$$

it follows that $\mathrm{KB} \models \pi(\texttt{True}, \mathrm{T})(\pi(\mathrm{RQ}))$. □

Proof of Lemma 6, ***Else*** *case.* We break down the proof by going through the case when $\mathrm{KB} \models \pi(\texttt{True}, \mathrm{M})(\pi(\mathrm{RQ}))$ first and then for $\mathrm{KB} \models \pi(\texttt{Indeterminate}, \mathrm{M})(\pi(\mathrm{RQ}))$.

True case We know that $\mathrm{KB} \models \pi(\texttt{True}, \mathrm{M})(\pi(\mathrm{RQ}))$ In Sections 6.1.2 and 6.1.1 the DL mapping was defined as:

$$\pi(\texttt{True}, \mathrm{M}) \equiv \exists \pi(\mathrm{AD}_M).\pi(\mathrm{fcn},\ \mathrm{AV}_M) \tag{A.7}$$

Thus, $\pi(\mathrm{RQ})$ is of type $\exists \pi(\mathrm{AD}_M).\pi(\mathrm{fcn},\ \mathrm{AV}_M)$. In Section 6.2.1.1 we see that:

- $\pi(\mathrm{AD}_M)$ is mapped back to an attribute AT that matches (in datatype, ID, category and issuer) the one specified by the attribute designator AD_M.
- $\pi(\mathrm{fcn},\ \mathrm{AV}_M)$ is mapped nondeterministically to a value in the data value range that satisfies the function $\mathrm{fcn}(\mathrm{AV}_M)$.

Essentially, $\pi(\mathrm{AD}_M).\pi(\mathrm{fcn}(\mathrm{AV}_M))$ maps back to an attribute AT and attribute value AV that will match the designator AD_M and the attribute match function fcn, s.t.$\mathrm{M}, \mathrm{RQ} \models_m$ `True`.

To extend the proof to CM, consider that:

$$\pi(\texttt{True}, \mathrm{CM}) \equiv \pi(\texttt{True}, \mathrm{M}_1) \sqcap \ldots \sqcap \pi(\texttt{True}, \mathrm{M}_n)$$

Thus, $\pi(\mathrm{RQ})$ is of type $\pi(\texttt{True}, \mathrm{M}_1) \sqcap \ldots \sqcap \pi(\texttt{True}, \mathrm{M}_n)$. It was shown above that

$$\pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{M}) \rightarrow \mathrm{M}, \mathrm{RQ} \models_m \texttt{True}$$

so for each $\mathrm{M}_i \in \mathrm{M}$ above we know that RQ will match it. However, if RQ matches all of $\mathrm{M}_i \in \mathrm{M}$, then by virtue of rule 1 in Table 4.2.1:

$$\mathrm{CM}, \mathrm{RQ} \models_m \texttt{True}$$

To extend the proof to DM, we use the mapping axiom from Section 6.1.2:

$$\pi(\texttt{True}, \mathrm{DM}) \equiv \pi(\texttt{True}, \mathrm{CM}_1) \sqcup \ldots \sqcup \pi(\texttt{True}, \mathrm{CM}_n)$$

Thus, $\pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{CM}_1) \sqcup \ldots \sqcup \pi(\texttt{True}, \mathrm{CM}_n)$. It was shown above that

$$\pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{CM}) \rightarrow \mathrm{CM}, \mathrm{RQ} \models_m \texttt{True}$$

, so for at least one $\mathrm{CM}_i \in \mathrm{DM}$ it follows that $\mathrm{CM}_i, \mathrm{RQ} \models_m \texttt{True}$. However, if RQ matches at least one of $CM_i \in DM$, then by virtue of rule 4.2.1:

$$\mathrm{DM}, \mathrm{RQ} \models_m \texttt{True}$$

To extend to T, we use the mapping axiom from Section 6.1.2:

$$\pi(\texttt{True}, \mathrm{T}) \equiv \pi(\texttt{True}, \mathrm{DM}_1) \sqcap \ldots \sqcap \pi(\texttt{True}, \mathrm{DM}_n)$$

Thus, $\pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{DM}_1) \sqcap \ldots \sqcap \pi(\texttt{True}, \mathrm{DM}_n)$. It was shown above that

$$\pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{DM}) \rightarrow \mathrm{DM}, \mathrm{RQ} \models_m \texttt{True}$$

so for all $\mathrm{DM}_i \in \mathrm{T}$ above we know that $\mathrm{DM}_i, \mathrm{RQ} \models_m \texttt{True}$. However, if RQ matches all of $DM_i \in T$, then by following the inference rules in Table 4.2.1:$\mathrm{T}, \mathrm{RQ} \models_m \texttt{True}$. I have shown that:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\texttt{True}, \mathrm{T}) \rightarrow \mathrm{T}, \mathrm{RQ} \models_m \texttt{True}$$

Indeterminate case We assume the attribute mustBePresent is set to true. In this case, $\mathrm{KB} \models \pi(\texttt{Indeterminate}, \mathrm{M})(\pi(\mathrm{RQ}))$. The mapping axioms for indeterminate is defined in Section 6.1.2 as:

$$\pi(\texttt{Indeterminate}, \mathrm{M}) \equiv \forall \pi(\mathrm{AD}_\mathrm{M}).\bot$$

Thus, π(RQ) will map back to a request RQ that has no values for the attribute specified by $\mathrm{AD_M}$. However, if RQ has no values for AD_M then according to the inference rules in Table 4.2.1,

$$\mathrm{M}, \mathrm{RQ} \models_m \texttt{Indeterminate}$$

Extending the proof to CM, DM and T can be done in the same manner as for the `True` case above, we omit it here for brevity.

$\square$

Proof of Lemma 7. For a request RQ, we need to show that:

$$\textbf{if } \mathrm{RQ}, \mathrm{T} \not\models_m \texttt{True} \textbf{ then } \mathrm{KB} \models \neg\pi(\texttt{True}, \mathrm{T})(\pi(\mathrm{RQ})) \tag{A.8}$$

True Case We will start with M, and show that the following holds:

$$\textbf{if } \mathrm{RQ}, \mathrm{M} \not\models_m \texttt{True} \textbf{ then } \mathrm{KB} \models \neg\pi(\texttt{True}, M)(\pi(\mathrm{RQ})) \tag{A.9}$$

In order to show that $\mathrm{KB} \models \neg\pi(\texttt{True}, \mathrm{M})(\pi(\mathrm{RQ}))$ we need to show that the expression $\pi(\mathrm{RQ}) \sqcap \pi(\texttt{True}, \mathrm{M})$ is unsatisfiable. In Sections 6.1.2 and 6.1.1 the DL mapping was defined as:

$$\begin{aligned} \pi(\texttt{True}, \mathrm{M}) &\equiv \exists\pi(\mathrm{AD}_M).\pi(\mathrm{fcn},\ \mathrm{AV}_M) \\ \pi(\mathrm{RQ}) &\equiv \bigsqcap_{AV\in RQ} (\exists \mathrm{r}_{AV}.\pi(\mathrm{type}_{AV}\text{-equal},\ \mathrm{AV})) \sqcap \bigsqcap_{AT\notin RQ,\ AT\in P} (\forall \mathrm{r}_{AT}.\bot) \sqcap \\ &\quad \bigsqcap_{AT\in P,\ n=\#AV\ s.t.\ AV\in AT} (= n\mathrm{r}_{AT}.) \end{aligned} \tag{A.10}$$

Assume that the left hand side of the implication in Equation A.9 holds. By the inference rules in Table 4.6, whenever $\mathrm{RQ}, \mathrm{M} \not\models_m \texttt{True}$ that means one of the following holds:

1. $\mathrm{AD}_M, \mathrm{RQ} \not\models_m \mathrm{AV}_{\mathrm{AT}}$. That means that there is no attribute $AT \in RQ$ s.t. $\mathrm{AD}_M, \mathrm{RQ} \models_m \mathrm{AV}_{\mathrm{AT}}$, and mustBePresent is set to false. However, if there is no attribute that matches AD_M, then $\forall\pi(\mathrm{AD}_M).\bot$ will occur as a conjunct in $\pi(\mathrm{RQ})$ – which will clash with $\exists\pi(\mathrm{AD}_M).\pi(\mathrm{fcn},\ \mathrm{AV}_M)$, rendering the expression in (A.4) unsatisfiable.

2. $\mathrm{AD}_M, \mathrm{RQ} \models_m \mathrm{AV}_{AT} : \mathrm{fcn}_M(\mathrm{AV}_M, \mathrm{AV}_{AT}) = \texttt{False}$. This implies that there is at least one attribute $AT \in RQ$ s.t. AV_{AT} is returned by AD_M, however the comparison function fcn evaluates to false. Thus, there will be a subexpression in $\pi(\mathrm{RQ}) \sqcap \pi(\texttt{True}, \mathrm{M})$:

$$\exists\pi(\mathrm{AD}_M).\pi(\mathrm{fcn}_M,\ \mathrm{AV}_M) \sqcap \bigsqcap \Big(\exists \mathrm{r}_{AV_m}.\pi(\mathrm{type}_{AV_i}\text{-equal},\ \mathrm{AV}_i)\Big)$$

   where $\mathrm{name}(\mathrm{r}_{AV_m}) = \mathrm{name}(\pi(\mathrm{AD}_M))$. Now, we also take into account the cardinality constraint that we have added to requests, which allows for exactly as many distinct fillers for role r_{AV_m} as there are values for that attribute in the request. Notice that $\exists\pi(\mathrm{AD}_M).\pi(\mathrm{fcn}_M,\ \mathrm{AV}_M)$ will introduce another role filler for the role $\pi(\mathrm{AD}_M)$.When there are n+1 role fillers for a role that has cardinality n, the DL semantics is such that two of the role fillers will be nondeterministically selected and a merge will be attempted. There are two cases:

   - The DL reasoning algorithm will try to merge the role fillers that represent two values for the same attribute in the request. In that case, if they have different datatype values a clash will occur. We assume that we prune duplicate attribute values in the request beforehand, so a clash will always occur in this case.

- The DL tableau algorithm will select the role fillers that corresponds to the attribute value in M (AV_M), and try to merge it with some role filler corresponding to an attribute value (for the same attribute) in the request. Thus, the following expression

$$\pi(\text{fcn}_M,\ \text{AV}_M) \sqcap \pi(\text{type}_{AV_i}\text{-equal},\ \text{AV}_i) \tag{A.11}$$

  will occur in the label for some attribute values in RQ whose attributes matches the AD_M. However, since $\text{fcn}_M(\text{AV}_M, \text{AV}_i) = \texttt{False}$ it follows that $\text{AV}_i \notin \text{fcn}_M(\text{AV}_M)$, which implies that a clash will occur for all possible $\text{AV}_i \in RQ$ that match AD_M in

I have shown that in both cases, $\pi(\text{RQ}) : \neg\pi(\texttt{True}, \text{M})$

Indeterminate Case For a request RQ, we need to show that:

$$\textbf{if } \text{RQ}, \text{M} \not\models_m \texttt{Indeterminate} \textbf{ then } \text{KB} \models \neg\pi(\texttt{Indeterminate}, M)(\pi(\text{RQ})) \tag{A.12}$$

In order to show that $\text{KB} \models \neg\pi(\texttt{Indeterminate}, \text{M})(\pi(\text{RQ}))$ we need to show that the expression $\pi(\text{RQ}) \sqcap \pi(\texttt{Indeterminate}, \text{M})$ is unsatisfiable. In Sections 6.1.2 and

6.1.1 the DL mapping was defined as:

$$\pi(\texttt{Indeterminate}, \mathrm{M}) \equiv \forall \pi(\mathrm{AD_M}).\bot \tag{A.13}$$

Thus, we essentially need to show that:

$$\pi(\mathrm{RQ}) \sqcap \forall \pi(\mathrm{AD_M}).\bot \tag{A.14}$$

is unsatisfiable. Since $\mathrm{RQ}, \mathrm{M} \not\models_m$ $\texttt{Indeterminate}$ that means the attribute does occur in the Request (according to the semantics rules in Table 4.2.1). Thus, there will exist a conjunct $\exists r_{AD_M}.\pi(\text{type}_{AV_i}\text{-equal}, \ \mathrm{AV}_i)$ as part of $\pi(\mathrm{RQ})$, which will clash with $\pi(\mathrm{AD_M}).\bot$, thus implying that $\mathrm{KB} \models \neg\pi(\texttt{Indeterminate}, \mathrm{T})(\pi(\mathrm{RQ}))$.

Extending to CM First I will show that:

$$\textbf{if } \mathrm{RQ}, \mathrm{CM} \not\models_m \texttt{E} \textbf{ then } \mathrm{KB} \models \neg\pi(\texttt{E}, CM)(\pi(\mathrm{RQ}))$$

where $\texttt{E} \in \{\texttt{True}, \texttt{Indeterminate}\}$. I will only show for $\texttt{E} = \texttt{True}$ - the $\texttt{Indeterminate}$ case can be shown in the same manner. To show $\mathrm{KB} \models \neg\pi(\texttt{True}, CM)(\pi(\mathrm{RQ}))$ we need to show that the expression

$$\pi(\texttt{True}, CM) \sqcap \pi(\mathrm{RQ})$$

is unsatisfiable.

Following the deduction rules in Table 4.2.1, if $\mathrm{RQ}, \mathrm{CM} \not\models_m$ $\texttt{True}$ then there must be some $\mathrm{M}_i \in CM$ s.t. $\mathrm{M}_i : \mathrm{RQ}, \mathrm{M}_i \not\models_m \texttt{True}$. Thus the following holds:

$$\pi(\mathrm{RQ}) : \neg\pi(\mathrm{M}_i, \mathrm{True})$$

From the mapping axioms in Section 6.1.2 it is also known that:

$$\pi(\texttt{True}, CM) \equiv \pi(\mathrm{M}_1, \texttt{True}) \sqcap \ldots \sqcap \ \pi(\mathrm{M}_n, \texttt{True})$$

- this implies that the expression $\pi(\texttt{True}, CM) \sqcap \pi(\mathrm{RQ})$ will not be satisfiable, since there will be a conflict on the $\pi(\mathrm{M}_i, \mathrm{True})$ conjunct. Thus, $\mathrm{KB} \models \neg\pi(\texttt{True}, \mathrm{CM})(\pi(\mathrm{RQ}))$.

Extending to DM   Here I will show that:

$$\textbf{if } \mathrm{RQ}, \mathrm{DM} \not\models_m \texttt{E} \textbf{ then } \mathrm{KB} \models \neg\pi(\texttt{E}, DM)(\pi(\mathrm{RQ}))$$

where $\texttt{E} \in \{\texttt{True}, \texttt{Indeterminate}\}$.

- True case. To show $\mathrm{KB} \models \neg\pi(\texttt{True}, DM)(\pi(\mathrm{RQ}))$ we need to show that the expression

  $$\pi(\texttt{True}, DM) \sqcap \pi(\mathrm{RQ})$$

  is unsatisfiable.

  Following the deduction rules in Table 4.2.1, if $\mathrm{RQ}, \mathrm{DM} \not\models_m \texttt{True}$ then for all $\mathrm{CM}_i \in DM : \mathrm{CM}_i : \mathrm{RQ}, \mathrm{CM}_i \not\models_m \texttt{True}$. Thus the following holds for all $\mathrm{CM}_i \in$ DM:$\pi(\mathrm{RQ}) : \neg\pi(\mathrm{CM}_i, \mathrm{True})$

From the mapping axioms in Section 6.1.2 it is known that:

$$\pi(\texttt{True}, DM) \equiv \pi(\mathrm{CM}_1, \texttt{True}) \sqcup \ldots \sqcap\ \pi(\mathrm{CM}_n, \texttt{True})$$

- this implies that the expression $\pi(\texttt{True}, DM) \sqcap \pi(\mathrm{RQ})$ will not be satisfiable, since there will be a clash for each $\pi(\mathrm{CM}_i, \mathrm{True})$ disjunct. Thus, $\mathrm{KB} \models \neg\pi(\texttt{True}, \mathrm{DM})(\pi(\mathrm{RQ}))$.

- Indeterminate Case. To show $\mathrm{KB} \models \neg\pi(\texttt{Indeterminate}, DM)(\pi(\mathrm{RQ}))$ we need to show that the expression

$$\pi(\texttt{Indeterminate}, \mathrm{CM}) \sqcap \pi(\mathrm{RQ})$$

  is unsatisfiable.

  Following the deduction rules in Table 4.2.1, if $\mathrm{RQ}, \mathrm{DM} \not\models_m \texttt{Indeterminate}$ then there must be some $\mathrm{CM}_i \in DM$ s.t. $\mathrm{CM}_i : \mathrm{RQ}, \mathrm{M}_i \not\models_m \texttt{Indeterminate}$. Thus the following holds:

$$\pi(\mathrm{RQ}) : \neg\pi(\mathrm{CM}_i, \mathrm{Indeterminate})$$

  From the mapping axioms in Section 6.1.2 it is also known that:

$$\pi(\texttt{Indeterminate}, DM) \equiv \pi(\mathrm{CM}_1, \texttt{Indeterminate}) \sqcap \ldots \sqcap \pi(\mathrm{CM}_n, \texttt{Indeterminate})$$

  This implies that the expression $\pi(\texttt{Indeterminate}, DM) \sqcap \pi(\mathrm{RQ})$ will not be satisfiable, since there will be a conflict on the $\pi(\mathrm{CM}_i, \mathrm{Indeterminate})$ conjunct. Thus, $\mathrm{KB} \models \neg\pi(\texttt{Indeterminate}, \mathrm{DM})(\pi(\mathrm{RQ}))$.

Extending to T   Here I will show that:

$$\textbf{if } \text{RQ}, \text{DM} \not\models_m \texttt{E} \textbf{ then } \text{KB} \models \neg\pi(\text{E}, DM)(\pi(\text{RQ}))$$

where $\texttt{E} \in \{\texttt{True}, \texttt{Indeterminate}\}$.

- `True` case. Can be shown in same manner as the proof for the True case for CM - simply replace the references to CM with T.

- `Indeterminate` case. To show $\text{KB} \models \neg\pi(\texttt{Indeterminate}, T)(\pi(\text{RQ}))$ we need to show that the expression

  $$\pi(\texttt{Indeterminate}, T) \sqcap \pi(\text{RQ})$$

  is unsatisfiable.

  Following the deduction rules in Table 4.2.1, if $\text{RQ}, \text{T} \not\models_m \texttt{Indeterminate}$ then for all $\text{DM}_i \in T : \text{DM}_i : \text{RQ}, \text{DM}_i \not\models_m \texttt{Indeterminate}$. Thus the following holds for all $\text{DM}_i \in \text{T}{:}\pi(\text{RQ}) : \neg\pi(\text{DM}_i, \text{Indeterminate})$.

  From the mapping axioms in Section 6.1.2 it is also known that:

  $$\pi(\texttt{Indeterminate}, T) \equiv \pi(\text{DM}_1, \texttt{Indeterminate}) \sqcup \ldots \sqcap \pi(\text{DM}_n, \texttt{Indeterminate})$$

  - this implies that the expression $\pi(\texttt{Indeterminate}, T) \sqcap \pi(\text{RQ})$ will not be satisfiable, since there will be a clash for each $\pi(\text{DM}_i, \text{Indeterminate})$ disjunct. Thus, $\text{KB} \models \neg\pi(\texttt{Indeterminate}, \text{DM})(\pi(\text{RQ}))$.

□

Proof of Lemma 8.

If case We will consider the case there $\text{Effect}_R = \texttt{Permit}$ first. The mapping for a rule (Rule ID T `Permit`) is defined as:

$$\begin{gathered}\text{Permit-}ID \equiv \pi(\texttt{True}, T_R), \quad Deny\text{-}ID \equiv \bot \\ \text{Indeterminate-}ID \equiv \pi(\texttt{Indeterminate}, T_R)\end{gathered} \tag{A.15}$$

According to the inference rules in Table 4.7, the only way that $\text{RQ}, \text{R} \models \texttt{Permit}$ can be inferred is if $\text{RQ}, \text{T} \models \texttt{Permit}$. However, $\text{RQ}, \text{T} \models \texttt{Permit}$ and the mapping axioms in A.15 imply that $\text{KB} \models \pi(\texttt{Permit}, \text{R})(\pi(\text{RQ}))$. It can be shown in the same manner for `Deny` and `Indeterminate`.

Else case For a `Rule` R and decision Effect $\in \{\texttt{Permit}, \texttt{Deny}\}$, we are given that $\pi(RQ)$: Effect-ID, and Effect-ID $\equiv \pi(\text{T}_R, \text{True})$ (mapping axiom in Section 6.1.3). This, combined with Lemma 6 implies that $\text{T}, \text{RQ} \models_m True$. Finally, if $\text{T}, \text{RQ} \models_m True$ then by virtue of the XACML semantics rules in Table 4.7, and assuming Cond always returns True, it follows that: $\text{R}, \text{RQ} \models \text{Effect}$. It can be shown in the same manner for `Indeterminate`. □

Proof of Lemma 9. We will consider the case there $\text{Effect}_R = \texttt{Permit}$ first. The mapping will then be as follows:

$$\begin{gathered}\text{Permit-}ID \equiv \pi(\texttt{True}, T_R), \quad \textit{Deny-ID} \equiv \bot \\ \text{Indeterminate-}ID \equiv \pi(\texttt{Indeterminate}, T_R)\end{gathered} \tag{A.16}$$

We need to show that Permit-ID $\sqcap \pi(\text{RQ})$ is unsatisfiable. Notice that $\text{RQ}, \text{R} \not\models$ `Permit` – the only way it could occur is if $\text{RQ}, \text{T}_R \not\models$ `True`, which implies that

$$\pi(\text{RQ}) : \neg\pi(\text{T}_T, \texttt{True})$$

This in turn clashes with Permit-ID, so the expression Permit-ID $\sqcap\, \pi(\text{RQ})$ is unsatisifable. It can be shown in the same manner for `Indeterminate` and `Deny`. □

*Proof of Lemma 10, **If** case.* We will split the proofs depending on the type of combining algorithm.

Permit-Overrides We will consider the different access decisions separately:

- $\text{P}, \text{RQ} \models$ `Permit`. This means that there exist some rule $\text{R}_i \in P$ s.t. $\text{R}_i, \text{RQ} \models$ `Permit`, and also $\text{T}_P, \text{RQ} \models_m$ `Permit`. This, together with our previous proofs (Lemmas 8, 6), in turn implies that $\text{KB} \models \pi(\text{RQ}) : \pi(\text{Permit-R}_i$ and $\text{KB} \models \pi(\text{RQ}) : \pi(\text{T}_P, \text{True})$. Combining the two results we get:

$$\text{KB} \models \pi(\text{RQ}) : \pi(\text{T}_P, \text{True}) \sqcap \pi(\text{Permit-R}_i, \text{RQ})$$

  which implies $\pi(\text{RQ}) : \text{Permit-P}$.

- $\mathrm{P}, \mathrm{RQ} \models$ `Deny`. This means that there exist some rule $\mathrm{R}_i \in P$ s.t. $\mathrm{R}_i, \mathrm{RQ} \models$ `Deny`, all other rules R_j do not yield a `Permit` ($\forall \mathrm{R}_j : \mathrm{R}_j, \mathrm{RQ} \not\models$ `Permit`), and all other rules where the effect is Permit do not yield an `Indeterminate`: $\forall \mathrm{R}_k$ s.t. $\mathrm{Effect}_{R_k} =$ `Permit` : $\mathrm{R}_k, \mathrm{RQ} \not\models$ `Indeterminate` . Taking these prerequisites and Lemmas 8, 6 into account we have:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \mathrm{True}) \sqcap \pi(\text{Deny-R}_i, \mathrm{RQ})$$

$$\mathrm{KB} \models \forall j : \pi(\mathrm{RQ}) : \neg\pi(\text{Permit-R}_j, \mathrm{RQ})$$

$$\mathrm{KB} \models \forall \mathrm{k} \text{ s.t. } \mathrm{Effect}_{R_k} = \texttt{Permit} : \pi(\mathrm{RQ}) : \neg\pi(\text{Indeterminate-R}_k, \mathrm{RQ})$$

which implies $\pi(\mathrm{RQ})$: Deny-P.

- $\mathrm{P}, \mathrm{RQ} \models$ `Indeterminate`. This means that there exist some rule $\mathrm{R}_i \in P$ that has a `Permit` effect and yields `Indeterminate` $\mathrm{R}_i, \mathrm{RQ} \models$ `Indeterminate`; in addition, all other rules R_j do not yield a `Permit`: $\forall \mathrm{R}_j : \mathrm{R}_j, \mathrm{RQ} \not\models$ `Permit`. Taking these prerequisites into account we have:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \mathrm{True}) \sqcap \pi(\text{Indeterminate-R}_i, \mathrm{RQ})$$

$$\mathrm{KB} \models \forall j : \pi(\mathrm{RQ}) : \neg\pi(\text{Permit-R}_j, \mathrm{RQ})$$

which implies $\mathrm{KB} \models \pi(\mathrm{RQ})$: Indeterminate-P.

Deny-Overrides This case is very similar to the `Permit-Overrides`.

- $\mathrm{P}, \mathrm{RQ} \models \texttt{Deny}$. This means that there exist some rule $\mathrm{R}_i \in P$ s.t. $\mathrm{R}_i, \mathrm{RQ} \models \texttt{Deny}$, and also $\mathrm{T}_P, \mathrm{RQ} \models_m \texttt{Deny}$. This, together with our previous proofs (Lemmas 8, 6), in turn implies that KB $\models \pi(\text{Deny-R}_i, \mathrm{RQ})$ and KB $\models \pi(\mathrm{T}_P, \text{True})(\pi(\mathrm{RQ}))$. Combining the two results we get:

$$\pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \text{True}) \sqcap \pi(\text{Deny-R}_i, \mathrm{RQ})$$

which implies $\pi(\mathrm{RQ})$: Deny-P.

- $\mathrm{P}, \mathrm{RQ} \models \texttt{Permit}$. This means that there exist some rule $\mathrm{R}_i \in P$ s.t. $\mathrm{R}_i, \mathrm{RQ} \models \texttt{Permit}$, all other rules R_j do not yield a $\texttt{Deny}$ ($\forall \mathrm{R}_j : \mathrm{R}_j, \mathrm{RQ} \not\models \texttt{Deny}$), and all other rules where the effect is Deny do not yield an $\texttt{Indeterminate}$: $\forall \mathrm{R}_k$ s.t. $\text{Effect}_{R_k} = \texttt{Deny} : \mathrm{R}_k, \mathrm{RQ} \not\models \texttt{Indeterminate}$. Taking these prerequisites and Lemmas 8, 6 into account we have:

$$\text{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \text{True}) \sqcap \pi(\text{Permit-R}_i, \mathrm{RQ})$$

$$\text{KB} \models \forall j : \pi(\mathrm{RQ}) : \neg\pi(\text{Deny-R}_j, \mathrm{RQ})$$

$$\text{KB} \models \forall \text{k s.t. Effect}_{R_k} = \texttt{Deny} : \pi(\mathrm{RQ}) : \neg\pi(\text{Indeterminate-R}_k, \mathrm{RQ})$$

which implies $\pi(\mathrm{RQ})$: Permit-P.

- $\mathrm{P}, \mathrm{RQ} \models \texttt{Indeterminate}$. This means that there exist some rule $\mathrm{R}_i \in P$ that yields has a $\texttt{Deny}$ effect and $\mathrm{R}_i, \mathrm{RQ} \models \texttt{Indeterminate}$, and all other rules R_j do not yield a Deny : $\forall \mathrm{R}_j : \mathrm{R}_j, \mathrm{RQ} \not\models \texttt{Deny}$. Taking these prerequisites into account we have:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \mathrm{True}) \sqcap \pi(\text{Indeterminate-R}_i, \mathrm{RQ})$$

$$\mathrm{KB} \models \forall j : \pi(\mathrm{RQ}) : \neg\pi(\text{Deny-R}_j, \mathrm{RQ})$$

which implies $\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Indeterminate-P}$.

First-Applicable Since the deduction rules are the same regardless of the effect decison, we will simply use Effect in this proof as substitute for Permit, Deny or Indeterminate.

$\mathrm{P}, \mathrm{RQ} \models \texttt{Effect}$ implies two things (according to inference rules in Table 4.9):

- $\exists \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \models \texttt{Effect}$
- $\forall \mathrm{R}_j \in \mathrm{P}$ s.t. $j < i : \mathrm{R}_j, \mathrm{RQ} \not\models \mathrm{Permit} \wedge \mathrm{R}_j, \mathrm{RQ} \not\models \mathrm{Deny} \wedge \mathrm{R}_j, \mathrm{RQ} \not\models \mathrm{Indeterminate}$

From the above conditions (and Lemmas 10, we know that

$$\pi(RQ) : \text{Effect-R}_i$$

and

$$\forall \mathrm{R}_j \text{s.t. } j < i : \pi(RQ) : \neg\text{Permit-R}_j \sqcap \neg\text{Deny-R}_j \sqcap \neg\text{Indeterminate-R}_j$$

which in turn implies $\pi(RQ) : \text{Effect-P}_i$.

Only-One-Applicable

- $\mathrm{P}, \mathrm{RQ} \models \texttt{Indeterminate}$. This means that there exist some rule $\mathrm{R}_i \in P$ s.t.

$\mathrm{R}_i, \mathrm{RQ} \models$ `Indeterminate` or there is a pair of rules $R_j, R_k \in P$ s.t. $\mathrm{R}_j, \mathrm{RQ} \models \mathrm{E}$ and $\mathrm{R}_k, \mathrm{RQ} \models \mathrm{E}$ (where E_1, E_2 are either `Permit` or `Deny`). Thus, one of the two conditions holds:

$$\pi(RQ) : \text{Indeterminate-R}_i$$

or

$$\forall \mathrm{R}_j, \mathrm{R}_k \text{ s.t. } j \neq i : \pi(RQ) : (\text{Permit-R}_j \sqcup \text{Deny-R}_j) \sqcap (\text{Permit-R}_k \sqcup \text{Deny-R}_k)$$

which implies $\pi(RQ)$: Indeterminate-P_i

- $\mathrm{P}, \mathrm{RQ} \models$ `Effect`, where Effect is `Permit` or `Deny`. This implies that there exist some rule $\mathrm{R}_i \in P$ s.t. $\mathrm{R}_i, \mathrm{RQ} \models$ `Effect` and for all rules $R_j \in P$ where $j \neq i$: $\mathrm{R}_j, \mathrm{RQ} \not\models \text{Permit} \wedge \mathrm{R}_j, \mathrm{RQ} \not\models \text{Deny} \wedge \mathrm{R}_j, \mathrm{RQ} \not\models \text{Indeterminate}$. Thus, both of the following conditions hold:

$$\pi(RQ) : \text{Effect-R}_i$$

and

$$\forall \mathrm{R}_j \text{ s.t. } j \neq i : \pi(RQ) : \neg\text{Permit-R}_j \sqcap \neg\text{Deny-R}_j \sqcap \neg\text{Indeterminate-R}_j$$

which implies $\pi(RQ)$: Effect-P_i

□

*Proof of Lemma 10, **Else** case.* We will break down the proof based on the different types of rule combining algorithm available.

Permit-Overrides

- KB $\models \pi(\text{RQ}) : \text{Permit-P}$. Since

$$\text{Permit-P} \equiv \pi(\text{True}, T) \sqcap \left(\bigsqcup \text{Permit-R}_i\right)$$

  this means that $\pi(\text{RQ}) : \pi(T)$ and $\pi(\text{RQ}) : \text{Permit-R}_i$ for some rule $R_i \in P$. Combining these results with Lemmas 8 and 6 implies that RQ satisfies both of the prerequisites of inference rule 1 in Table 4.8, thus $\text{P}, \text{RQ} \models \texttt{Permit}$.

- KB $\models \pi(\text{RQ}) : \text{Deny-P}$. The mapping axioms for this case in Section 6.1.4 is defined as:

$$\text{Deny-P} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Deny-R}_i \sqcap \left(\bigsqcap \neg\text{Permit-R}_j\right) \sqcap (\bigsqcap \neg\text{Indeterminate-R}_k)\right)$$
$$\textit{where } \text{Effect}_{R_k} = \text{Permit} \tag{A.17}$$

  This implies all of the following:

  - $\pi(\text{RQ}) : \pi(T, True)$, which by virtue of Lemma 6 means one of the prerequisites of rule 2 in Table 4.8 is satisfied.
  - There exist some rule $\text{R}_i \in P$ s.t. $\pi(\text{RQ}) : \text{Deny-R}_i$. This satisfies another prerequisite of rule 2 in Table 4.8.
  - For the particular rule $\text{R}_i \in P$ where $\pi(\text{RQ}) : \text{Deny-R}_i$, for all of the other rules R_j, $\pi(\text{RQ}) : \neg\text{Permit-R}_j$. Additionally, for all rules $\text{R}_k \in \text{P}$ where $\text{Effect}_{R_k} = \text{Permit}$, $\pi(\text{RQ}) : \neg\text{Indeterminate-R}_k$. (This all follows directly

from the mapping axioms in Section 6.1.4). These statements together with Lemma 8 satisfy the last prerequisite of rule 2 in Table 4.8.

I have shown how with these three conditions all of the prerequisites of the rule that entails $\mathrm{P}, \mathrm{RQ} \models \mathrm{Deny}$ are satisfied, so:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Deny-P} \rightarrow (\mathrm{P}\ \text{Permit-Overrides}\ \mathrm{T}\ \mathrm{R}_i), \mathrm{RQ} \models \mathrm{Deny}$$

- $\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Indeterminate-P}$. Since

$$\begin{gathered}\text{Indeterminate-P} \equiv \pi(\mathrm{True}, \mathrm{T}) \sqcap \left(\bigsqcup \text{Indeterminate-R}_i \sqcap \left(\bigsqcap \neg\text{Permit-R}_j\right)\right) \\ \textit{where}\ \mathrm{Effect}_{R_i} = \mathrm{Permit}\end{gathered} \tag{A.18}$$

This implies all of the following:

  - $\pi(\mathrm{RQ}) : \pi(\mathrm{True}, \mathrm{T})$.
  - There exist some rule $\mathrm{R}_i \in P$ where $\mathrm{Effect}_{R_i} = \mathrm{Permit}$ s.t. $\pi(\mathrm{RQ}) : \text{Indeterminate-R}_i$.
  - For all other rules $R_j \in P$, $\pi(\mathrm{RQ}) : \neg\text{Permit-R}_j$.

The three conditions above satisfy all of the prerequisites of the rule in Table 4.8 that entails $\mathrm{P}, \mathrm{RQ} \models \mathrm{Indeterminate}$ are satisfied, thus:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Indeterminate-P} \rightarrow \mathrm{P}\ \text{Permit-Overrides}\ \mathrm{T}\ \mathrm{R}_i, \mathrm{RQ} \models \mathrm{Indeterminate}$$

Deny-Overrides Done in same manner as the `Permit-Overrides` case – it is enough simply to swap `Permit` and `Deny` wherever they are referenced in the proof.

First-Applicable Since the deduction rules are the same regardless of the effect decision, we will simply use Effect in this proof as substitute for Permit, Deny or Indeterminate.

KB $\models \pi(\mathrm{RQ})$: Effect-P implies two things (according to mapping axioms in Section 6.1.4):

- $\pi(\mathrm{RQ}) : \pi(T, True)$, which by virtue of Lemma 6 means one of the prerequisites of rule 1 in Table 4.9 is satisfied.
- $\pi(\mathrm{RQ}) : \left(\bigsqcup \text{Effect-R}_i \sqcap \left(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\right)\right)$, where i > j. Thus, there exists some rule $\mathrm{R}_i \in P$ s.t. $\pi(\mathrm{RQ})$: Effect-R_i. Additionally, for all $R_j \in P$ s.t. $j < i : \pi(\mathrm{RQ}) : \left(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\right)$, which in turn implies that $\mathrm{R}_j, \mathrm{RQ} \not\models$ Deny, $\mathrm{R}_j, \mathrm{RQ} \not\models$ Permit and $\mathrm{R}_j, \mathrm{RQ} \not\models$ Indeterminate for all $j < i$.

The above conditions , together with Lemma 8 satisfy all of the prerequisites for the rule in Table 4.9, thus:

$$\text{KB} \models \pi(\text{RQ}) : \text{Effect-P} \rightarrow \text{P First-Applicable T R}_i, \text{RQ} \models \text{Effect-P}$$

Only-One-Applicable In this case, we use Effect as a substitute for Permit or Deny.

- According to the mapping axioms in Section 6.1.4, KB $\models \pi(\mathrm{RQ})$: Effect-P implies the following holds:

  - $\pi(\mathrm{RQ}) : \pi(\mathrm{True}, \mathrm{T})$.
  - $\pi(RQ)$: Effect-R$_i$ for some $R_i \in P$.
  - $\pi(RQ)$: $\neg$Deny-R$_j$ $\sqcap$ $\neg$Permit-R$_j$ $\sqcap$ $\neg$Indeterminate-R$_j$ for all $R_j \in P$.

  The above three conditions together with Lemma 8 satisfy all of the prerequisites for rule 3 in Table 4.10, thus:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Effect-P} \rightarrow \text{P Only-One-Applicable T R}_i, \mathrm{RQ} \models \text{Effect-P}$$

- Indeterminate. According to the mapping axioms in Section 6.1.4, KB $\models \pi(\mathrm{RQ})$: Effect-P implies at least one of the following:
  - $\pi(RQ) : \pi(T)$ and $\pi(RQ)$: Indeterminate-R$_i$ for some $R_i \in P$.
  - $\pi(RQ) : \pi(T)$ and there exist $R_i, R_j \in P$, $i \neq j$ s.t. $\pi(RQ)$: Effect-R$_i$ and $\pi(RQ)$: Effect-R$_j$, where Effect-R$_i$, Effect-R$_j$ can be Permit or Deny.

  Satisfying either of the above conditions , together with Lemma 8 is enough to infer that:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Effect-P} \rightarrow \text{P Only-One-Applicable T R}_i, \mathrm{RQ} \models \text{Indeterminate}$$

After covering all of the rule combining algorithms , I proved that:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \text{Effect-P} \rightarrow \mathrm{P}, \mathrm{RQ} \models \text{Effect}$$

$\square$

Proof of Lemma 11. This proof has a few sections, depending on what type of rule combining algorithm is being considered.

Permit-Overrides

- $\mathrm{P}, \mathrm{RQ} \not\models$ `Permit`. Considering that semantics rule 1 is the only one that can entail $\mathrm{P}, \mathrm{RQ} \models$ `Permit` in Table 4.8, we have:

$$\mathrm{T}, \mathrm{RQ} \not\models \mathrm{True} \vee (\forall \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \not\models \mathrm{Permit})$$

  which together with Lemmas 7 and 9 implies that

$$(\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\pi(T, \mathrm{True})) \text{ OR } (\forall \mathrm{R}_i \in \mathrm{P} : \mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\text{Permit-R}_i)$$

  Since

$$\text{Permit-P} \equiv \pi(\mathrm{T}, \mathrm{True}) \sqcap \left(\bigsqcup \text{Permit-R}_i\right),$$

  when we try to evaluate satisfiability of $\text{Permit-P} \sqcap (\neg\pi(T, \mathrm{True}) \sqcup (\bigsqcap \neg\text{Permit-R}_i))$ we will always get a clash, proving that $\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\text{Permit-P}$

- $\mathrm{P}, \mathrm{RQ} \not\models$ `Deny`. The mapping axiom for `Deny` in Section 6.1.4 is defined as follows:

$$\text{Deny-P} \equiv \pi(\mathrm{True}, \mathrm{T}) \sqcap \left(\bigsqcup \text{Deny-R}_i \sqcap \left(\bigsqcap \neg\text{Permit-R}_j\right) \sqcap \left(\bigsqcap \neg\text{Indeterminate-R}_k\right)\right)$$
$$\textit{where } \text{Effect}_{R_k} = \text{Permit} \tag{A.19}$$

Since inference rule 2 in Table 4.8 is the only one that can entail $\mathrm{P}, \mathrm{RQ} \models \texttt{Deny}$, that means that at least one of the following holds:

- $\mathrm{T}, \mathrm{RQ} \not\models$ True, which implies $\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\pi(\mathrm{T}, \mathrm{True})$. Notice that this expression will clash with the first conjunct in equation (A.19), rendering Deny-P $\sqcap\, \pi(\mathrm{RQ})$ unsatisfiable.
- $\forall \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \not\models$ Deny. This means that for each $\mathrm{R}_i \in \mathrm{P}$, $\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg$Deny-$\mathrm{R}_i$, which means that there always will be a clash with the first item in the second conjunct in equation (A.19) ($\bigsqcup$ Deny-R_i), again rendering Deny-P $\sqcap\, \pi(\mathrm{RQ})$ unsatisfiable.
- $\exists \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \models$ Indeterminate where $\mathrm{Effect}_{R_i} = \mathrm{Permit}$. The mapping of this expression will clash with the last item in the second conjunct ($\bigsqcap \neg$Indeterminate-R_k), which is only focused on rules where the effect is `Permit`.
- $\exists \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \models$ Permit – this implies that for some $\mathrm{R}_i \in \mathrm{P}$, $\mathrm{KB} \models \pi(\mathrm{RQ}) :$ Permit-R_i which will clash with the second item in the second conjunct.

Since I showed that in all three cases the expression Deny-P $\sqcap\, \pi(\mathrm{RQ})$ will be unsatisfiable, it will always follow that $\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg$Deny-P for this case.

- $\mathrm{P}, \mathrm{RQ} \not\models$ `Indeterminate`. The mapping axiom for `Indeterminate` in Section 6.1.4 is:

$$\text{Indeterminate-P} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Indeterminate-R}_i \sqcap \left(\bigsqcap \neg\text{Permit-R}_j\right)\right) \qquad \text{(A.20)}$$
$$\textit{where } \mathrm{Effect}_{R_i} = \mathrm{Permit}$$

This is the case of inference rule 3 in Table 4.8. If that rule did not fire, that means one of the following conditions holds:

- $\mathrm{T}, \mathrm{RQ} \not\models \mathrm{True}$, which implies $\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\pi(\mathrm{T}, \mathrm{True})$. This expression will clash with the first conjunct in equation (A.20), thus the expression Indeterminate-P $\sqcap\, \pi(\mathrm{RQ})$ is unsatisfiable.
- $\forall \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \not\models$ Indeterminate whereEffect$_{R_i}$ = Permit. The mapping of this expression will be:

$$\forall \mathrm{R}_i \in \mathrm{P}\ : \mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\text{Indeterminate-R}_i \text{ where Effect}_{R_i} = \text{Permit},$$

  which will clash with $(\bigsqcup \text{Indeterminate-R}_i)$, which is only focused on rules where the effect is `Permit`.
- $\exists \mathrm{R}_i \in \mathrm{P} : \mathrm{R}_i, \mathrm{RQ} \models$ Permit – this implies that for some $\mathrm{R}_i \in \mathrm{P}$, $\mathrm{KB} \models \pi(\mathrm{RQ})$: Permit-R$_i$ which will clash with $\left(\bigsqcap \neg\text{Permit-R}_j\right)$, rendering Indeterminate-P$\,\sqcap$ $\pi(\mathrm{RQ})$ unsatisfiable.

Deny-Overrides To show that:

$$\mathrm{RQ}, \mathrm{P} \models \texttt{Effect} \textbf{ then } \mathrm{KB} \models \pi(\texttt{Effect}, \mathrm{P})(\pi(\mathrm{RQ}))$$

in the `Deny-Overrides` case the same approach as in `Permit-Overrides` can be used.

First-Applicable The mapping axiom for `First-Applicable` is defined as:

$$\text{Effect-P} \equiv \pi(T) \sqcap \Big(\bigsqcup \text{Effect-R}_i \sqcap \Big(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\Big)\Big)$$

$$\textit{where } i > j \tag{A.21}$$

Since the deduction rules are the same regardless of the effect decision, we will simply use Effect in this proof as substitute for `Permit`, `Deny` or `Indeterminate`.

$\text{P}, \text{RQ} \not\models \texttt{Effect}$ means one of the following things holds(according to inference rules in Table 4.9):

- $\text{T}, \text{RQ} \not\models \text{True}$. This implies $\text{KB} \models \pi(\text{RQ}) : \neg\pi(\text{T}, \text{True})$. This expression will clash with the first conjunct in equation (A.21), thus the expression $\text{Effect-P} \sqcap \pi(\text{RQ})$ is unsatisfiable.

- $\forall \text{R}_i \in P$ s.t. $\text{R}_i, \text{RQ} \not\models \texttt{Effect}$. This implies that

$$\forall \text{R}_i \in \text{P} : \text{KB} \models \pi(\text{RQ}) : \neg\text{Effect-R}_i$$

  Which means that there will be a clash between the term $\bigsqcup \text{Effect-R}_i$ from equation (A.21) and $\pi(\text{RQ})$.

- $\exists R_j$s.t. $\text{j} < \text{i} : \text{R}_j, \text{RQ} \models \texttt{Indeterminate} \vee \text{R}_j, \text{RQ} \models \texttt{Permit} \vee \text{R}_j, \text{RQ} \models \texttt{Deny}$ which means that

$$\exists \text{R}_j \in \text{P where } j < i : \text{KB} \models \pi(\text{RQ}) : \text{Indeterminate-R}_j \sqcup \text{Deny-R}_j \sqcup \text{Permit-R}_j$$

which will always clash with $\left(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\right)$, thus rendering Effect-P $\sqcap\, \pi$(RQ) unsatisfiable.

I showed in any of the three conditions that implies $\text{P}, \text{RQ} \not\models \texttt{Effect}$, $\text{KB} \models \pi(\text{RQ}) : \neg\text{Effect-P}$, thus

$$(\text{P}, \text{RQ} \not\models \texttt{Effect}) \rightarrow (\text{KB} \models \pi(\text{RQ}) : \neg\text{Effect-P})$$

Only-One-Applicable We will show for the Permit/Deny case first, then for Indeterminate.

- Permit/Deny. The mapping axiom for Permit/Deny in `First-Applicable` is defined as:

$$\text{Effect-P} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Effect-R}_i \sqcap \left(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\right)\right)$$
$$\textit{where } i \neq j \tag{A.22}$$

  $\text{P}, \text{RQ} \not\models \texttt{Effect}$ means one of the following things holds(according to inference rules in Table 4.10):

  - $\text{T}, \text{RQ} \not\models \text{True}$. This implies $\text{KB} \models \pi(\text{RQ}) : \neg\pi(\text{T}, \text{True})$. This expression will clash with the first conjunct in equation (A.22).

- $\forall \text{R}_i \in P$ s.t. $\text{R}_i, \text{RQ} \not\models$ `Effect`. This implies that

$$\forall \text{R}_i \in \text{P} : \text{KB} \models \pi(\text{RQ}) : \neg\text{Effect-R}_i$$

  Which means that there will be a clash between the term $\bigsqcup \text{Effect-R}_i$ from equation (A.22) and $\pi(\text{RQ})$.

- $\exists \text{R}_j \in P$ s.t. $\text{R}_j, \text{RQ} \models \text{Effect}_x$ where Effect_x is a Permit, Deny or Indeterminate. This in turn implies

$$\text{KB} \models \pi(\text{RQ}) : \text{Permit-R}_j \sqcup \text{Deny-R}_j \sqcup \text{Indeterminate-R}_j$$

  which will always produce a clash with $\left(\bigsqcap \neg\text{Deny-R}_j \sqcap \neg\text{Permit-R}_j \sqcap \neg\text{Indeterminate-R}_j\right)$.

• $\text{P}, \text{RQ} \not\models$ Indeterminate. The mapping axioms are defined in Section 6.1.4 as:

$$\begin{aligned}
&\text{Indeterminate-P} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Indeterminate-R}_i\right)\\
&\text{Indeterminate-P} \equiv \pi(T) \sqcap \left(\bigsqcup ((\text{Permit-R}_i \sqcup \text{Deny-R}_i) \sqcap (\text{Permit-R}_j \sqcup \text{Deny-R}_j))\right)\\
&\qquad\qquad\qquad \textit{where } i \neq j
\end{aligned} \tag{A.23}$$

According to the inference rules in Table 4.10 , there are only two cases s.t. $\text{P}, \text{RQ} \not\models$ Indeterminate:

- $\forall \text{R}_i \in \text{P} : \text{R}, \text{RQ} \not\models$ Indeterminate. This implies $\forall \text{R}_i \in \text{P} : \text{KB} \models \pi(\text{RQ}) : \neg\text{Indeterminate-R}_i$ which , in conjunction with the first axiom in (A.23), will produce a clash.

– $\forall \mathrm{R}_i, \mathrm{R}_j \in \mathrm{P}$ where $i \neq j$: $\mathrm{R}, \mathrm{RQ} \not\models \mathrm{Effect}_x \vee \mathrm{R}, \mathrm{RQ} \not\models \mathrm{Effect}_y$ where $\mathrm{Effect}_x, \mathrm{Effect}_y$ are either a Permit or a Deny. This, together with the results from the previous proofs implies that:

$$\forall \mathrm{R}_i, \mathrm{R}_j \in \mathrm{P} \text{ where } i \neq j : \mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\mathrm{Effect}_x \sqcup \neg\mathrm{Effect}_y$$

which in conjunction with the mapping axioms in (A.23) implies that the expression $\pi(\mathrm{RQ}) \sqcap$ Indeterminate-P will be unsatisfiable.

By covering all possible rule combining algorithms, I have shown that

$$(\mathrm{P}, \mathrm{RQ} \not\models \mathtt{Effect}) \rightarrow (\mathrm{KB} \models \pi(\mathrm{RQ}) : \neg\text{Effect-P})$$

$\square$

*Proof of Lemma 12, **If** case.* As in the previous proof, we itemize by the type of combining algorithm. Notice that we will only cover `Permit-Overrides` and `Deny-Overrides`, since the other two combining algorithms can be handled same as in the `Policy` proof.

Permit-Overrides

- $\mathrm{PS}, \mathrm{RQ} \models$ `Permit`. This means that there exist a policy $\mathrm{P}_i \in PS$ s.t. $\mathrm{P}_i, \mathrm{RQ} \models$ `Permit`, and also $\mathrm{T}_P, \mathrm{RQ} \models_m$ `Permit`. This, together with our previous proofs (10), in turn implies that $\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\text{Permit-P}_i)$ and $\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \mathrm{True})$.

Combining the two results we get:

$$\pi(\text{RQ}) : \pi(\text{T}_P, \text{True}) \sqcap \pi(\text{Permit-P}_i, \text{RQ})$$

which implies $\pi(\text{RQ})$: Permit-PS.

- PS, RQ $\models$ `Deny`. This means that there exist a policy $\text{P}_i \in PS$ s.t. $\text{P}_i, \text{RQ} \models$ `Deny`, and all other rules P_j do not yield a `Permit`: $\forall \text{P}_j : \text{P}_j, \text{RQ} \not\models$ `Permit`. Taking these prerequisites into account we have:

$$\text{KB} \models \pi(\text{RQ}) : \pi(\text{T}_P, \text{True}) \sqcap \pi(\text{Deny-P}_i, \text{RQ})$$

$$\text{KB} \models \forall j : \pi(\text{RQ}) : \neg\pi(\text{Permit-P}_j, \text{RQ})$$

which implies $\pi(\text{RQ})$: Deny-PS.

Deny-Overrides

- PS, RQ $\models$ `Deny`. This means that there exist a policy $\text{P}_i \in PS$ s.t. $\text{P}_i, \text{RQ} \models$ `Deny` or $\text{P}_i, \text{RQ} \models$ `Indeterminate`, and also $\text{T}_P, \text{RQ} \models_m$ `True`. Thus, one of the following conditions holds:

$$\text{KB} \models \pi(\text{Deny-P}_i, \text{RQ})$$

or

$$\text{KB} \models \pi(\text{Indeterminate-P}_i, \text{RQ})$$

which implies $\pi(\text{RQ})$: Deny-PS.

- PS, RQ $\models$ `Permit`. This means that there exist a policy $\mathrm{P}_i \in PS$ s.t. $\mathrm{P}_i, \mathrm{RQ} \models$ `Permit`, and all other rules P_j do not yield a `Deny` or `Indeterminate`: $\forall \mathrm{P}_j$: $\mathrm{P}_j, \mathrm{RQ} \not\models$ `Deny` $\vee$ $\mathrm{P}_j, \mathrm{RQ} \not\models$ `Indeterminate`. Taking these prerequisites into account we have:

$$\mathrm{KB} \models \pi(\mathrm{RQ}) : \pi(\mathrm{T}_P, \mathrm{True}) \sqcap \pi(\text{Permit-P}_i, \mathrm{RQ})$$

$$\mathrm{KB} \models \forall j : \pi(\mathrm{RQ}) : \neg\pi(\text{Deny-P}_j, \mathrm{RQ}) \sqcap \neg\pi(\text{Indeterminate-P}_j, \mathrm{RQ})$$

which implies $\pi(\mathrm{RQ})$: Permit-PS.

$\square$

*Proof of Lemma 12, **Else** case.* As in the `Policy` case, the proof is split based on the different types of policy combining algorithms.

Permit-Overrides

- KB $\models \pi(\mathrm{RQ})$: Permit-PS. Since

$$\text{Permit-PS} \equiv \pi(\mathrm{True}, \mathrm{T}) \sqcap \left(\bigsqcup \text{Permit-P}_i\right)$$

this means that $\pi(\mathrm{RQ}) : \pi(T)$ and $\pi(\mathrm{RQ})$: Permit-P_i for some policy $P_i \in PS$. Combining these results with Lemmas 10 and 6 implies that RQ satisfies both of the prerequisites of inference rule 1 in Table 4.11, thus PS, RQ $\models$ `Permit`.

- KB $\models \pi(\text{RQ}) : \text{Deny-PS}$. Since

$$\text{Deny-PS} \equiv \pi(\text{True}, \text{T}) \sqcap \left(\bigsqcup \text{Deny-P}_i \sqcap \left(\bigsqcap \neg\text{Permit-P}_j\right)\right) \tag{A.24}$$

  This implies all of the following:

  - $\pi(\text{RQ}) : \pi(\text{True}, \text{T})$, which by virtue of Lemma 6 means one of the prerequisites of rule 2 in Table 4.11 is satisfied.
  - There exist some policy $\text{P}_i \in P$ s.t. $\pi(\text{RQ}) : \text{Deny-P}_i$.
  - For all policies $P_j \in PS$, $\pi(\text{RQ}) : \neg\text{Permit-P}_j$. These statements together with Lemma 10 satisfy the last prerequisite of rule 2 in Table 4.11.

  I have shown how with the three conditions above all of the prerequisites of the rule that entails $\text{PS}, \text{RQ} \models \text{Deny}$ are satisfied, so:

$$\text{KB} \models \pi(\text{RQ}) : \text{Deny-PS} \rightarrow \text{PS Permit-Overrides T P}_i, \text{RQ} \models \text{Deny}$$

Deny-Overrides

- KB $\models \pi(\text{RQ}) : \text{Deny-PS}$. Since

$$\text{Deny-PS} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Deny-R}_i \sqcup \text{Indeterminate-R}_i\right)$$

  this means that $\pi(\text{RQ}) : \pi(T)$ and either $\pi(\text{RQ}) : \text{Deny-P}_i$ or $\pi(\text{RQ}) : \text{Indeterminate-P}_i$ for some policy $P_i \in PS$. Combining these results with Lemmas 10 and 6 implies that RQ satisfies both of the prerequisites of inference rule 3 in Table 4.11, thus $\text{PS}, \text{RQ} \models \texttt{Deny}$.

- $\text{KB} \models \pi(\text{RQ}) : \text{Permit-PS}$. Since

  $$\text{Permit-PS} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Permit-P}_i \sqcap \left(\bigsqcap \neg\text{Deny-P}_j \sqcap \neg\text{Indeterminate-P}_j\right)\right) \tag{A.25}$$

  This implies all of the following:

  - $\pi(\text{RQ}) : \pi(T, True)$, which by virtue of Lemma 6 means one of the prerequisites of rule 4 in 4.11 is satisfied.
  - There exist some policy $\text{P}_i \in PS$ s.t. $\pi(\text{RQ}) : \text{Permit-P}_i$.
  - For all policies $P_j \in PS$, $\pi(\text{RQ}) : \neg\text{Deny-P}_j$ and $\pi(\text{RQ}) : \neg\text{Indeterminate-P}_j$. These statements together with Lemma 10 satisfy the last prerequisite of rule 4 in Table 4.11.

  With the above three conditions above all prerequisites of the rule that entails $\text{PS}, \text{RQ} \models \text{Permit}$ are satisfied, so:

  $$\text{KB} \models \pi(\text{RQ}) : \text{Permit-PS} \rightarrow \text{PS Deny-Overrides T P}_i, \text{RQ} \models \text{Permit}$$

First-Applicable Can be done in same manner as the `Policy` proof.

Only-One-Applicable Can be done in same manner as the `Policy` proof.

After covering all of the rule combining algorithms , I proved that:

$$\text{KB} \models \pi(\text{RQ}) : \text{Effect-PS} \rightarrow \text{PS}, \text{RQ} \models \text{Effect}$$

□

Proof of Lemma 13. This proof has a few sections, depending on what type of rule combining algorithm is being considered.

Permit-Overrides

- $\text{PS}, \text{RQ} \not\models$ `Permit`. Since inference rule 1 is the only one that can entail $\text{PS}, \text{RQ} \models$ `Permit` in Table 4.11, if that rule doesn't hold then:

  $$\text{T}, \text{RQ} \not\models \text{True} \vee (\forall \text{P}_i \in \text{PS} : \text{P}_i, \text{RQ} \not\models \text{Permit})$$

  which together with Lemmas 7 and 11 implies that

  $$(\text{KB} \models \pi(\text{RQ}) : \neg\pi(T, \text{True})) \text{ OR } (\forall \text{P}_i \in \text{PS} : \text{KB} \models \pi(\text{RQ}) : \neg\text{Permit-P}_i)$$

  Since

  $$\text{Permit-PS} \equiv \pi(\text{True}, \text{T}) \sqcap \left(\bigsqcup \text{Permit-P}_i \right),$$

  when we try to evaluate satisfiability of $\text{Permit-PS} \sqcap (\neg\pi(T, \text{True}) \sqcup \neg\text{Permit-P}_i)$ we will always get a clash, thus proving that

$$\text{KB} \models \pi(\text{RQ}) : \neg\text{Permit-PS}$$

- $\text{PS}, \text{RQ} \not\models$ `Deny`. The mapping axiom for `Deny` in Section 6.1.5 is defined as:

$$\text{Deny-PS} \equiv \pi(\text{True}, \text{T}) \sqcap \left(\bigsqcup \text{Deny-P}_i \sqcap \left(\bigsqcap \neg\text{Permit-P}_j\right)\right) \tag{A.26}$$

  Since inference rule 2 in Table4.11 is the only one that can entail $\text{PS}, \text{RQ} \models$ `Deny`, that means that one of the following happened:

  - $\text{T}, \text{RQ} \not\models \text{True}$, which implies $\text{KB} \models \pi(\text{RQ}) : \neg\pi(\text{T}, \text{True})$. Notice that this expression will clash with the first conjunct in equation (A.26).
  - $\forall \text{P}_i \in \text{PS} : \text{P}_i, \text{RQ} \not\models \text{Deny}$. This means that for each $\text{P}_i \in \text{PS}$, $\text{KB} \models \pi(\text{RQ}) : \neg\text{Deny-P}_i$, which means that there always will be a clash with the first item in the second conjunct in equation (A.26) ($\bigsqcup \text{Deny-P}_i$).
  - $\exists \text{P}_i \in \text{PS} : \text{P}_i, \text{RQ} \models \text{Permit}$ – this implies that for some $\text{P}_i \in \text{PS}$, $\text{KB} \models \pi(\text{RQ}) : \text{Permit-P}_i$ which again will produce a clash so the expression $\text{Deny-PS} \sqcap \pi(\text{RQ})$ will be unsatisfiable.

  Since in all three cases the expression $\text{Deny-PS} \sqcap \pi(\text{RQ})$ will be unsatisfiable, it will always follow that $\text{KB} \models \pi(\text{RQ}) : \neg\text{Deny-PS}$ for this case.

Deny-Overrides The proof is slightly different than the `Permit-Overrides` case.

- PS, RQ $\not\models$ `Deny`. Since inference rule 3 is the only one that can entail PS, RQ $\models$ `Deny` in Table 4.11, that implies:

$$\text{T}, \text{RQ} \not\models \text{True} \vee (\forall \text{P}_i \in \text{PS} : \text{P}_i, \text{RQ} \not\models \text{Deny} \wedge \text{P}_i, \text{RQ} \not\models \text{Indeterminate})$$

  which together with Lemmas 7 and 11 implies

$$(\text{KB} \models \pi(\text{RQ}) : \neg\pi(T, \text{True})) \text{ OR}$$

$$(\forall \text{P}_i \in \text{PS} : \text{KB} \models \pi(\text{RQ}) : \neg\text{Indeterminate-R}_i \sqcap \neg\text{Deny-R}_i)$$

  Since

$$\text{Deny-PS} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Deny-R}_i \sqcup \text{Indeterminate-R}_i\right),$$

  when we try to evaluate satisfiability of

$$\text{Deny-PS} \sqcap (\neg\pi(T, \text{True}) \sqcup (\neg\text{Indeterminate-R}_i \sqcap \neg\text{Deny-R}_i))$$

  we will always get a clash, thus proving that

$$\text{KB} \models \pi(\text{RQ}) : \neg\text{Deny-ID}_{PS}$$

- PS, RQ $\not\models$ `Permit`. The mapping axiom for `Permit` in Section 6.1.5 is defined as follows:

$$\text{Permit-PS} \equiv \pi(T) \sqcap \left(\bigsqcup \text{Permit-P}_i \sqcap \left(\bigsqcap \neg\text{Deny-P}_j \sqcap \neg\text{Indeterminate-P}_j \right) \right) \tag{A.27}$$

Since inference rule 4 in Table 4.11 is the only one that can entail $\text{PS}, \text{RQ} \models$ `Permit`, that means that one of the following holds:

- $\text{T}, \text{RQ} \not\models \text{True}$, which implies $\text{KB} \models \neg\pi(\text{RQ}) : \pi(\text{T}, \text{True})$. Notice that this expression will clash with the first conjunct in equation (A.27), rendering $\text{Permit-PS} \sqcap \pi(\text{RQ})$ unsatisfiable.
- $\forall \text{P}_i \in \text{PS} : \text{P}_i, \text{RQ} \not\models \text{Permit}$. This means that for each $\text{P}_i \in \text{PS}$, $\text{KB} \models \pi(\text{RQ}) : \neg\text{Permit-P}_i$, which means that there always will be a clash with the expression in equation (A.27) ($\bigsqcup \text{Permit-R}_i$).
- $\exists \text{P}_j \in \text{PS} : \text{P}_j, \text{RQ} \models \text{Deny} \vee \text{P}_j, \text{RQ} \models \text{Indeterminate}$. This means that there exists a $\text{P}_i \in \text{PS}$ s.t. $\text{KB} \models \pi(\text{RQ}) : \text{Indeterminate-P}_i \sqcup \text{Deny-P}_i$, which means that there always will be a clash with the expression in equation (A.27) $\left(\bigsqcap \neg\text{Deny-P}_j \sqcap \neg\text{Indeterminate-P}_j\right)$.

Since I showed that in all three cases the expression $\text{Permit-P} \sqcap \pi(\text{RQ})$ will be unsatisfiable, that it will always follow that $\text{KB} \models \pi(\text{RQ}) : \neg\text{Permit-P}$ for this case.

First-Applicable Proof can be done in same way as for the `Policy` case (since the semantics is the same).

Only-One-Applicable Proof is same as for the `Policy` case..

□

Bibliography

[1] Aaauthreach project information page. Available at `http://staff.science.uva.nl/~demch/projects/aaauthreach/`.

[2] BIANCA Application. Available at `http://clarkparsia.com/clients/`.

[3] eXist: Open Source Native XML Database. Available at `http://exist.sourceforge.net/`.

[4] Fedora Authorization with XACML Policy Enforcement. Available at `http://www.fedora.info/download/2.1b/userdocs/server/security/AuthorizationXACML.htm`.

[5] Role-Based Access Control (RBAC) Scenarios. Available at `http://www.va.gov/RBAC/scenarios.asp`.

[6] Semantics of Business Vocabulary and Business Rules (SBVR). Available at `http://www.businessrulesgroup.org/sbvr.shtml`.

[7] Veterans Health Information Systems and Technology Architecture (VISTA). Available at `http://www.va.gov/vista_monograph/`.

[8] Dod trusted computer system evaluation criteria. Technical Report CSC-STD-001-83, Department of Defense, 1985. Available at `http://nsi.org/Library/Compsec/orangebo.txt`.

[9] A guide to understanding discretionary access control in security systems. Technical Report NCSC-TG-003, National Computer Security Center, 1987. Available at `http://www.fas.org/irp/nsa/rainbow/tg003.htm`.

[10] Role based access control. Technical Report ANSI INCITS 359-2004, NIST, 2004. Available at `http://csrc.nist.gov/rbac/`.

[11] Margrave Continue Example, 2005. Available at `http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/`.

[12] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[13] Dhiah el Diehn I. Abou-Tair, Stefan Berlik, and Udo Kelter. Enforcing privacy by means of an ontology driven xacml framework. *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 279–284, 29-31 Aug. 2007.

[14] Dakshi Agrawal, James Giles, Kang-Won Lee, and Jorge Lobo. Policy ratification. In *POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[15] A. Anderson. A comparison of two privacy policy languages: Epal and xacml, 2005.

[16] Anne Anderson. XACML References, v1.65. Available at `http://docs.oasis-open.org/xacml/references/xacmlRefsV1.65.html`.

[17] Anne Anderson. Core and hierarchical role based access control (rbac) profile of xacml v2.0, February 2005. Available at `http://www.docs.o\discretionary{-}{}{}asis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf`.

[18] Anne Anderson. Hierarchical resource profile of XACML, February 2005. Available at `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-hier-profile-spec-os.pdf`.

[19] Anne Anderson. Domain independent, composable web services policy assertions. In *IEEE POLICY Workshop*, 2006.

[20] Anne Anderson. Multiple resource profile of XACML, November 2007. Available at `http://www.oasis-open.org/committees/download.php/26154/xacml-3.0-hie%2Cmul%2Cdsig%2Cpriv-profiles-wd01.zip`.

[21] Claudio Agostino Ardagna, Ernesto Damiani, Sabrina De Capitani di Vimercati, Cristiano Fugazza, and Pierangela Samarati. Offline expansion of xacml policies based on p3p metadata. In *ICWE*, pages 363–374, 2005.

[22] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[23] Franz Baader and Philipp Hanschke. A scheme for integrating concrete domains into concept languages. Technical Report RR-91-10, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Erwin-Schrödinger Strasse
Postfach 2080
67608 Kaiserslautern
Germany, 1991.

[24] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[25] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics. *J. Autom. Reasoning*, 8(3):345–366, 1992.

[26] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and semantics of a decentralized authorization language. *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 3–15, 6-8 July 2007.

[28] David E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical report, The MITRE Corporation, July 1975.

[29] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[30] Elisa Bertino, Francesco Buccafurri, Elena Ferrari, and Pasquale Rullo. An authorization model and its formal semantics. In *ESORICS '98: Proceedings of the 5th European Symposium on Research in Computer Security*, pages 127–142, London, UK, 1998. Springer-Verlag.

[31] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[32] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. Authorizations in relational database management systems. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 130–139, New York, NY, USA, 1993. ACM.

[33] Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminda Wijesekera. Provisions and obligations in policy management and security applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 502–513. VLDB Endowment, 2002.

[34] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *Information and System Security*, 5(1):1–35, 2002.

[35] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation submitted 22 February 1999 `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[36] Jery Bryans. Reasoning about xacml policies using csp. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35, New York, NY, USA, 2005. ACM Press.

[37] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 01(1):146–166, 1989.

[38] Jan Chomicki, Jorge Lobo, and Shamin Naqvi. A logic programming approach to conflict resolution in policy management. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 121–132, San Francisco, 2000. Morgan Kaufmann.

[39] Brickley D. and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema/`, February 2004.

[40] Ernesto Damiani, Sabrina De Capitani di Vimercati, Cristiano Fugazza, and Pi erangela Samarati. Extending policy languages to the semantic web. In *ICWE*, pages 330–343, 2004.

[41] Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. New paradigms for access control in open environments. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, December 2005.

[42] Dan Lin and Prathima Rao and Elisa Bertino and Jorge Lobo and Ninghui Li. EXAM - a Comprehensive Environment for the Analysis of Access Control Policies, 2007.

[43] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *IEEE Conference on Computational Complexity*, pages 82–101, 1997.

[44] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[45] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 `http://www.w3.org/TR/2002/WD-owl-ref-20021112/`.

[46] John DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[47] Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, models, and languages for access control. In *In Proc. of the Workshop on Databases in Networked Information Systems*, 2005.

[48] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *3rd International Joint Conference on Automated Reasoning (IJCAR)*, 2006.

[49] Jayne Dutra. NASA Taxonomy 2.0. Available at `http://nasataxonomy.jpl.nasa.gov/`.

[50] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[51] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, 1997.

[52] Rita Gavriloaie, Wolfgang Nejdl, Daniel Olmedilla, Kent Seamons, and Marianne Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *European Semantic Web Symposium*, May 2004.

[53] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–10, New York, NY, USA, 1989. ACM.

[54] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[55] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of xml paths and types. In *PLDI*, pages 342–351, 2007.

[56] Benjamin N. Grosof. Diplomat: Compiling prioritized default rules into ordinary logic programs, for e-commerce applications. In *AAAI/IAAI*, pages 912–913, 1999.

[57] Dimitar P. Guelev, Mark Ryan, and Pierre-Yves Schobbens. Model-checking access control policies. In *ISC*, pages 219–230, 2004.

[58] Volker Haarslev and Ralf Möller. High performance reasoning with very large knowledge bases: A practical case study. In *IJCAI*, pages 161–168, 2001.

[59] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *In Proceedings of the Computer Security Foundations Workshop (CSFW'03)*, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[60] Joseph Y. Halpern and Vicky Weissman. A formal foundation for xrml. In *CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[62] HL7 Security Technical Committee. Role Based Access Control (RBAC) Healthcare Permission Catalog. Available at `http://www.va.gov/RBAC/docs/Stds_20071129_SW_22_5_HL7_RBAC_Healthcare_Permission_Catalog_v3_38.pdf`.

[63] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[64] I. Horrocks and U. Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[65] Ian Horrocks. The fact system. `http://www.cs.man.ac.uk/~horrocks/FaCT/`.

[66] Graham Hughes and Tevfik Bultan. Automated verification of access control policies (technical report). Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, September 2004.

[67] Graham Hughes and Tevfik Bultan. Automated Verification of XACML Policies Using a SAT Solver. In *Proceedings of the 7th International Conference on Web Engineering, Workshop on Web Quality, Verification and Validation (WQVV)*, pages 378–392, July 2007.

[68] Polar Humenn. The Formal Semantics of XACML, October 2003. Available at `http://lists.oasis-open.org/archives/xacml/200310/pdf00000.pdf`.

[69] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

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

[71] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *Database Systems*, 26(2):214–260, 2001.

[72] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Eliza Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 474–485, New York, NY, USA, 1997. ACM Press.

[73] Sushil Jajodia and Duminda Wijesekera. A flexible authorization framework for e-commerce. In *ICDCIT*, pages 336–345, 2004.

[74] Rasool Jalili and Mohsen Rezvani. Specification and verification of security policies in firewalls. In *EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, pages 154–163, London, UK, 2002. Springer-Verlag.

[75] Trevor Jim. Sd3: A trust management system with certified evaluation. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 106, Washington, DC, USA, 2001. IEEE Computer Society.

[76] Diane Jordan and John Evdemon. Business Process Execution Language (BPEL). Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[77] L. et al Kagal. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.

[78] Lalana Kagal, Tim Berners-Lee, Dan Connolly, and Daniel Weitzner. Using semantic web technologies for policy management on the web. In *21st National Conference on Artificial Intelligence (AAAI)*, 2006.

[79] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and Bernardo Cuenca-Grau. Repairing unsatisfiable concepts in owl ontologies. In *Proceedings of European Semantic Web Conference (ESWC)*, 2006.

[80] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics - Special Issue of the Semantic Web Track of WWW2005*, 3(4), 2005.

[81] Kendall Clark and Andrew Schain and Bijan Parsia. Semantic Web @ NASA. In *XTech Conference*, 2006.

[82] Racer Systems GmbH & Co. KG. Racerpro user guide, October 2007.

[83] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 677–686, New York, NY, USA, 2007. ACM.

[84] Vladimir Kolovski, Bijan Parsia, Yarden Katz, and James Hendler. Representing Web Service Policies in OWL-DL. In *Proc. of the Int. Semantic Web Conference (ISWC)*, 2005.

[85] N. Li, B. Grosof, and J. Feigenbaum. A nonmonotonic delegation logic with prioritized conflict handling, 2000.

[86] N. Li and J. Mitchell. Datalog with constraints: A foundation for trust management languages, 2003.

[87] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[88] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[89] Ninghui Li and John Mitchel. Understanding spki/sdsi using first-order logic, 2003.

[90] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[91] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information Systems Security*, 9(4):391–420, 2006.

[92] Ninghui Li, William H. Winsborough, and John C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *IEEE Symposium on Security and Privacy*, May 2003.

[93] Dan Lin, Prathima Rao, Elisa Bertino, and Jorge Lobo. An approach to evaluate policy similarity. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 1–10, New York, NY, USA, 2007. ACM.

[94] Chuchang Liu, Patrick McLean, and Maris A. Ozols. Combining logics for modelling security policies. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 323–332, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[95] Hal Lockhart, Bill Parducci, and Anne Anderson. Web services profile of xacml,(ws-xacml) version 1.0. Available at `http://www.oasis-open.org/committees/download.php/21490/xacml-3.0-profile-webservices-spec-v1.0-wd-8-en.pdf`.

[96] C. Lutz. NExpTime-complete description logics with concrete domains. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083, pages 45–60, Siena, Italy, 2001. Springer Verlag.

[97] C. Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics Volume 4*. King's College Publications, 2003.

[98] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS 2006)*, pages 139–158, December 2006.

[99] Fabio Massacci. Reasoning about security: A logic and a decision method for role-based access control. In *First International Joint Conference on Qualitative and Quantitative Practical Reasoning ECSQARU-FAPR*, pages 421–435, 1997.

[100] Imtiaz Mohammed and David M. Dilts. Design for dynamic user-role-based security. *Comput. Secur.*, 13(9):661–671, 1994.

[101] Moritz Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY '04: Proceedings of the Fifth IEEE*

International Workshop on Policies for Distributed Systems and Networks (POLICY'04), page 159, Washington, DC, USA, 2004. IEEE Computer Society.

[102] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.

[103] Till Mossakowski, Michael Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. *time-ictl*, 00:83, 2003.

[104] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesitt Karlsruhe, Germany, January 2006.

[105] Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. OWL 1.1 Web Ontology Language:Structural Specification and Functional-Style Syntax . Editor's Draft of 23 May 2007.

[106] Matunda Nyanchama and Sylvia Osborn. Role-based security, object oriented databases and separation of duty. *SIGMOD Rec.*, 22(4):45–51, 1993.

[107] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.*, 2(1):3–33, 1999.

[108] B. Orgun and J. Vu. HL7 ontology and mobile agents for interoperability in heterogeneous medical information systems. *Computers in Biology and Medicine: Special Issue on Medical Ontologies*, 36(7):817–836, 2006.

[109] Bijan Parsia and Evren Sirin. Pellet: An OWL DL reasoner. In *Third International Semantic Web Conference - Poster*, 2004.

[110] Pellet. Pellet - owl dl reasoner, 2003. http://www.mindswap.org/2003/pellet.

[111] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[112] Riccardo Pucella and Vicky Weissman. A Formal Foundation for ODRL. In *In Proceedings of the Workshop on Issues in the Theory of Security (WITS-04)*, 2006.

[113] Erik Rissanen. eXtensible Access Control Markup Language (XACML) Version 3.0 (Core Specification and Schemas), May 2007. Available at `http://www.oasis-open.org/committees/download.php/23950/xacml-3.0-core-wd-02.zip`.

[114] Erik Rissanen, Hal Lockhart, and Tim Moses. Xacml administrative policy version 1.0, working draft 17, May 2007. Available at `http://www.oasis-open.org/committees/download.php/23951/xacml-3.0-admininstration-v1-wd-17.zip`.

[115] Ronald Rivest and Butler Lampson. A Simple Distributed Security Infrastructure (SDSI).

[116] 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.

[117] Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2002. ACM Press.

[118] Michael A. Smith, Andrew J. Schain, Kendall Grant Clark, Arlen Griffey, and Vladimir Kolovski. Mother, May I? OWL-based Policy Management at NASA. In *Proc. of the Third International Workshop on OWL: Experiences and Directions (OWLED)*, June 2007.

[119] Christoph Tempich, Helena Sofia Pinto, and Steffen Staab. Ontology engineering revisited: An iterative case study. In *ESWC*, pages 110–124, 2006.

[120] Dmitry Tsarkov and Ian Horrocks. Description logic reasoner: System description. In *IJCAR*, pages 292–297, 2006.

[121] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM Press.

[122] A. Uszokand and J. Bradshaw. Kaos policies for web services. In *W3C Workshop on Constraints and Capabilities for Web Servies*, October 2004.

[123] Allen van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[124] V.Haarslev and R.Moeller. Racer system description. In *Proc. of the Joint Conf. on Automated Reasoning (IJCAR 2001). Volume 2083 of Lecture Notes in Artificial Intelligence, pages 701-705*, 2001.

[125] S. H. von Solms and Isak van der Merwe. The management of computer security profiles using a role-oriented approach. *Comput. Secur.*, 13(9):673–680, 1994.

[126] Daniel J. Weitzner, Jim Hendler, Tim Berners-Lee, and Dan Connolly. Creating a policy-aware web: Discretionary, rule-based access for the world wide web.

[127] Marianne Winslett, Charles C. Zhang, and Piero A. Bonatti. Peeraccess: a logic for distributed authorization. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 168–179, New York, NY, USA, 2005. ACM Press.

[128] Thomas Y. C. Woo and Simon S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[129] WS-Policy. Web services policy framework (ws-policy). http://www-106.ibm.com/developerworks/library/specification/ws-polfram/.

[130] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A performance study of bdd-based model checking. In *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 255–289, London, UK, 1998. Springer-Verlag.

[131] Nan Zhang, Mark D. Ryan, and Dimitar Guelev. Evaluating access control policies through model checking. In *Eighth Information Security Conference (ISC05)*, 2005.

[132] Chen Zhao, NuerMaimaiti Heilili, Shengping Liu, and Zuoquan Lin. Representation and reasoning on rbac: A description logic approach. In *ICTAC*, pages 381–393, 2005.