MASTER

UconXACML
an implementation of UCON in XACML

van Schoubroeck, A.H.L.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
UconXACML: An implementation of UCON in XACML

André van Schoubroeck
Eindhoven University of Technology, Eindhoven, the Netherlands

24 November 2014
Abstract

Since the emergence of the time sharing computer model, over forty years ago, protection mechanisms to prevent unauthorised access of information have been researched and implemented. Such a system that protects information against unauthorised access is called an access control system.

The field of access control systems has matured. As a result, industry standards for access control, such as XACML have emerged. XACML is a access control policy language. A policy contains the conditions which must hold in order to access protected information.

Computing technology is evolving at a rapid rate. Everyone uses computing technology and everything is connected to the Internet. In a world where everything is connected to the Internet new questions arise regarding the protection of information.

Where in the past access control, an evaluation of conditions prior to the release of information, was sufficient, the computing environment of today needs more control. The new requirements regarding protecting information are captured by usage control. Usage control evaluates conditions during the entire time the information is being used: the usage.

Usage control is a relatively new field of computer science. There are many approaches to usage control. As a consequence there is are no standardised policy languages yet. UCON is an approach to usage control. UCON extends attribute based access control with continuity of policy enforcement and mutability of attributes.

The goal of this project is to extend XACML, a standardised access control policy language with features introduced by UCON. The extensions required to encode the UCON features in a XACML policy should not require the XML schema to be modified. There is room in the XACML language specification to encode new features without modification to the XML schema. Keeping the XML schema in tact ensures interoperability with existing XACML solutions. In order to extend XACML with UCON features the XACML architecture has to be extended. We provide an mapping of UCON features that meets this requirement, and present a software implementation of an extended XACML architecture capable of evaluating and enforcing XACML policies using the proposed UCON extensions.
## Contents

1 Introduction
   1.1 Setting ................................................................. 1
   1.2 Problem ............................................................... 1
   1.3 Contribution .......................................................... 2
   1.4 Outline ................................................................. 2

2 Preliminaries
   2.1 Historical overview on access control ................................. 3
   2.2 Overview of UCON
      2.2.1 Concepts of UCON ............................................. 4
      2.2.2 UCON Models ................................................. 5
      2.2.3 UCON Updates .............................................. 9
   2.3 Overview of XACML ...................................................... 10
      2.3.1 XACML Language ............................................... 10
      2.3.2 XACML Policy Evaluation ................................... 11
      2.3.3 XACML Reference Architecture ............................... 13

3 Analysis of UCON and XACML .............................................. 17
   3.1 Analysis of UCON ..................................................... 17
   3.2 Comparison of UCON and XACML .................................... 19

4 Intermediate Policy Language ............................................. 21
   4.1 Concepts ............................................................... 21
   4.2 Policy Evaluation .................................................... 22

5 Mapping from UCON to XACML .............................................. 27
   5.1 UCON to Intermediate Policy Language .............................. 27
      5.1.1 Allowed rules ................................................. 27
      5.1.2 stopped rules .............................................. 28
      5.1.3 Updates .................................................... 29
      5.1.4 Policies ................................................... 29
   5.2 Intermediate Policy Language to XACML ............................ 30
      5.2.1 Rules ........................................................ 30
      5.2.2 Updates .................................................... 31
      5.2.3 Policies ................................................... 31

6 Implementation ............................................................ 37
   6.1 HerasAF XACML Core ................................................. 37
   6.2 UconXACML Overview ................................................ 37
   6.3 UconXACML Classes .................................................. 38
   6.4 Evaluation Request Code Flow ..................................... 41

7 Validation ..................................................................... 45
   7.1 Scenario Description .................................................. 45
   7.2 Policy Specification ................................................... 46
   7.3 Policy Evaluation ..................................................... 48
   7.4 Discussion .............................................................. 51
8 Related Work

8.1 Other XACML UCON Implementations ........................................ 55
8.2 XACML Implementations ....................................................... 55
  8.2.1 Axiomatics (https://www.axiomatics.com/) .............................. 55
  8.2.2 enterprise-java-xacml (https://code.google.com/p/enterprise-java-xacml/) .............................. 56
  8.2.3 HerasAF XACML Core (http://www.herasaf.org) .......................... 56
  8.2.4 PicketBox XACML (http://picketbox.jboss.org) ............................ 56
  8.2.5 Sun XACML (http://sunxacml.sourceforge.net/) ........................ 56
  8.2.6 WSO2 Balana (https://svn.wso2.org/repos/wso2/trunk/commons/balana/) ....................... 57
  8.2.7 xacml4j (http://www.xacml4j.org/) ....................................... 57
  8.2.8 XACMLight (http://xacmllight.sourceforge.net/) ......................... 57
  8.2.9 XEngine ................................................................. 57
  8.2.10 Conclusions ............................................................ 57

8.3 Other models for usage control ........................................... 57

9 Conclusions and Recommendations ........................................... 59

9.1 Conclusions ................................................................. 59
9.2 Future Work ............................................................... 59

Appendices ..................................................................... 63

A Implementation: libraries and classes .................................. 65
  A.1 Implementation: code fragments ........................................... 65

B XACML policies .............................................................. 69
  B.1 updateRequestNew.xml ..................................................... 69
  B.2 paidCall.xml ................................................................. 70
  B.3 topupCall.xml ............................................................... 71
  B.4 minimalCredit.xml ......................................................... 72
  B.5 pastUsage.xml ............................................................... 72
  B.6 activeUsage.xml ............................................................. 73
  B.7 time.xml ................................................................. 73
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>UCON decision factors [13]</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>UCON decision timeline [16]</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>The UCON_ABC Family of Core models [13]</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>XACML XML Schema [12]</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>XACML Architecture [12]</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>UCON Policy evaluation</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Intermediate Policy Language Model, UML representation</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Policy Language Model, EBNF representation</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Policy Language Model, Example policy</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Activity Diagram</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Allowed Rule Conversion</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Stopped Rule Conversion</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Update Conversion</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Policy Combination</td>
<td>34</td>
</tr>
<tr>
<td>5.5</td>
<td>Condition Conversion</td>
<td>35</td>
</tr>
<tr>
<td>5.6</td>
<td>Action Conversion</td>
<td>35</td>
</tr>
<tr>
<td>5.7</td>
<td>Encoding of the re-evaluation trigger in XACML</td>
<td>35</td>
</tr>
<tr>
<td>5.8</td>
<td>Update Conversion</td>
<td>36</td>
</tr>
<tr>
<td>5.9</td>
<td>Encoding of timer based update activation in XACML</td>
<td>36</td>
</tr>
<tr>
<td>5.10</td>
<td>Encoding of time based update activation in XACML</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Activity Diagram for UCON XACML Request</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>Class diagram of UconXACML</td>
<td>39</td>
</tr>
<tr>
<td>6.3</td>
<td>UconPEP Class</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>RequestHandler Interface</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>UconPIP Class</td>
<td>41</td>
</tr>
<tr>
<td>6.6</td>
<td>UconOS Interface</td>
<td>41</td>
</tr>
<tr>
<td>6.7</td>
<td>Update Class</td>
<td>42</td>
</tr>
<tr>
<td>6.8</td>
<td>Trigger Interface</td>
<td>42</td>
</tr>
<tr>
<td>6.9</td>
<td>UconPAP Class</td>
<td>42</td>
</tr>
<tr>
<td>6.10</td>
<td>Extractor Class</td>
<td>42</td>
</tr>
<tr>
<td>6.11</td>
<td>UpdateMarshaller Class</td>
<td>42</td>
</tr>
<tr>
<td>6.12</td>
<td>Sequence Diagram for simple Request</td>
<td>43</td>
</tr>
<tr>
<td>6.13</td>
<td>Sequence Diagram for registering re-evaluation trigger</td>
<td>43</td>
</tr>
<tr>
<td>6.14</td>
<td>Sequence Diagram for registering updates</td>
<td>43</td>
</tr>
<tr>
<td>6.15</td>
<td>Sequence Diagram for calling pre-updates</td>
<td>44</td>
</tr>
<tr>
<td>6.16</td>
<td>Sequence Diagram for applying updates and re-evaluation</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>UCON Ongoing Authorisations and Ongoing Updates: phone call policy</td>
<td>47</td>
</tr>
<tr>
<td>7.2</td>
<td>UCON Post Updates: top-up call policy</td>
<td>48</td>
</tr>
<tr>
<td>7.3</td>
<td>UCON Ongoing Authorisations: Minimal credit required policy</td>
<td>49</td>
</tr>
<tr>
<td>7.4</td>
<td>UCON Pre Obligations: Must accept license before downloading</td>
<td>49</td>
</tr>
<tr>
<td>7.5</td>
<td>UCON Ongoing Obligations: Keep watching advertisements policy</td>
<td>50</td>
</tr>
<tr>
<td>7.6</td>
<td>UCON Pre Conditions: time limitation policy</td>
<td>50</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

7.7  UCON XACML Validation, UCON Ongoing Authorisations and updates  ................. 52
7.8  UCON XACML Validation, UCON Ongoing Authorisations  .................................. 52
7.9  UCON XACML Validation, UCON Pre Obligations  .............................................. 53
7.10 UCON XACML Validation: UCON Ongoing Obligations  ...................................... 53

A.1  UconPEP: Initialisation code  ................................................................................. 66
A.2  UconPEP: addRequest(): evaluating request by PDP  ........................................... 66
A.3  UconPEP: addRequest(): handling obligations returned by the PDP  ................. 66
A.4  UconPIP: fetchSubjectAttributes(): fetching subject attributes from database  ... 67
A.5  UconPIP: updateSubjectAttribute(): updating subject attributes in database  .... 67
A.6  UconPIP: updateSubjectAttribute(): re-evaluation queue  ............................... 67
A.7  RequestHandler: onStopped(): post updates  ....................................................... 68
A.8  Update: evaluateValue()  ...................................................................................... 68
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Concepts common for all UCON Models</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Pre Authorisations concept</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Ongoing Authorisation concepts</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Pre Obligation concepts</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Ongoing Obligation concepts</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Pre Condition concepts</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Ongoing Conditions concepts</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>Pre Update concepts</td>
<td>9</td>
</tr>
<tr>
<td>2.9</td>
<td>Ongoing Update concepts</td>
<td>10</td>
</tr>
<tr>
<td>2.10</td>
<td>Post Update concepts</td>
<td>10</td>
</tr>
<tr>
<td>2.11</td>
<td>XACML Policy Truth Table [12]</td>
<td>13</td>
</tr>
<tr>
<td>2.12</td>
<td>XACML PolicySet Truth Table [12]</td>
<td>13</td>
</tr>
<tr>
<td>2.13</td>
<td>XACML Rule Truth Table [12]</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison UCON and XACML</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Possible values for Time Elements</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>System States</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Mapping Intermediate Policy Language to XACML</td>
<td>32</td>
</tr>
<tr>
<td>7.1</td>
<td>Initial state of attributes</td>
<td>51</td>
</tr>
<tr>
<td>7.2</td>
<td>System state after Bob’s top-up call</td>
<td>51</td>
</tr>
<tr>
<td>8.1</td>
<td>XACML Implementations</td>
<td>56</td>
</tr>
</tbody>
</table>
Acronyms

**ABAC** Attribute Based Access Control. 3, 4

**ACL** Access Control List. 9

**CH** Context Handler. 14, 15

**DAC** Discretionary Access Control. 3, 4

**DRM** Digital Rights Management. 1, 57

**MAC** Mandatory Access Control. 3, 5

**OS** Obligation Service. 14

**OSL** Obligation Specification Language. 57

**PAP** Policy Administration Point. 13, 14

**PDP** Policy Decision Point. 14, 15

**PEP** Policy Enforcement Point. 13, 15

**PIP** Policy Information Point. 13, 15

**RBAC** Role Based Access Control. 3, 4

**UTC** Coordinated Universal Time. 51

**XACML** eXtensible Access Control Markup Language. 10, 11, 14, 17, 19, 20, 27

**XML** Extensible Markup Language. 10
Chapter 1

Introduction

1.1 Setting

Since the emergence of computing technology, over forty years ago, computer science was faced with security related questions. A notable field of security related questions in computer science is access control. The focus of access control lies on protecting sensitive information. It answers the question, when a user requests to access certain information, should this be permitted? Access control is a mature field in computer science. This has resulted in standardised access control policy languages such as XACML [12]. As a standard XACML has been adopted by industry and academia. Due technical innovations, the computing landscape has changed significantly: computing technology is available to virtually everyone and is interconnected through the Internet. These changes in the computing environment have created opportunities but also challenges for the industry. Information processed by computing systems has become more complex in nature. Asking the question whether a user is permitted to access certain information only once is not sufficient any more. Once a user has permission to access information, access control has no means to revoke access. As a response to these challenges the industry created systems such as Digital Rights Management (DRM) and systems that are capable to revoke previously granted access, such as credit-based systems, where traditional access systems only decide at request time.

Traditional access control systems and the new approaches introduced by industry have one common factor: they decide whether a user is permitted to access a resource. Usage control is an approach to unify traditional access control and later approaches in a single model. The term usage rather than access emphasises the fact decisions are not limited to the request time, but may happen during the entire usage of a resource.

UCON [13] is an attribute based usage control model. UCON introduces mutable attributes. Attribute values may be modified as side-effect of using a resource. Furthermore, UCON introduces the concept of continuous evaluation. When an attribute value is updates, a policy can be re-evaluated, and the permission to access an object might be revoked.

1.2 Problem

Opposed to access control, which is a mature field in computer science, usage control is a relatively young field. As such, standardised usage control languages have not yet emerged. The absence of standardised languages makes it difficult for systems from different vendors to interoperate.

The goal of this project is to extend an XACML with UCON features. Most notable are mutability of attributes and continuous evaluation of policies. These features are not natively supported by the XACML architecture, therefore the XACML architecture must be extended as well. The goal is to implement these extensions in such a way XACML’s XML schema is preserved, to preserve interoperability with existing XACML based systems.
1.3 Contribution

Our contributions to address the problems we have defined are

- A mapping from UCON to XACML
- An extension to the XACML architecture to support UCON features
- An implementation of the extended XACML architecture

We have defined a mapping from UCON to XACML. By a mapping from UCON to XACML we mean a definition to encode UCON features, such as attribute updates and policy re-evaluation triggers using structures provided by the XACML policy language, while keeping the XML schema in tact. We have performed this mapping using XACML Obligations. XACML Obligations are included in the result of a policy evaluation and may include arbitrary data. We use this feature of XACML to encode updates and re-evaluation triggers.

We have implemented software capable of evaluating and enforcing XACML policies that contain the obligations we have defined. Our implementation is named UconXACML. This implementation is written in Java and uses HerasAF as XACML policy evaluation engine. This software includes a simple GUI front end that is capable of making requests and displaying the evaluation result. The GUI front end can easily be replaced by code that integrates our implementation as evaluation engine of a system protecting resources.

1.4 Outline

Chapter 2 presents a historical overview on access control. Access control models such as DAC and MAC are discussed in this overview. Furthermore, we provide an overview of UCON and XACML, the model and policy language we are using.

Chapter 3 analyses of UCON and XACML to reveal the conceptual differences and similarities.

Chapter 4 presents an intermediate notation, used as a bridge between UCON and XACML during conversion.

Chapter 5 presents a mapping from a UCON policy to a XACML policy, using the Intermediate Policy Language.

Chapter 6 discusses the implementation details of the software parsing and evaluating XACML encoded UCON policies.

Chapter 7 validates the implementation by creating a testing scenario and comparing the results with expected behaviour.

Chapter 8 discusses related work, such as an overview of XACML implementations and other approaches of usage control.

Chapter 9 presents our conclusions and directions for future work.
Chapter 2

Preliminaries

2.1 Historical overview on access control

In the late 1960’s the time sharing computing model was introduced. Time sharing allows multiple users to interact with a computer system concurrently. The emergence of time sharing called for mechanisms that protects a user’s data from being accessed by other users. Such a mechanism is called an access control system. When a user attempts to access a resource, an access control system decides whether this access is permitted. In an access control system we refer to active entities as subject, resources protected by the system as objects, and to actions a subject is allowed to perform on the object as rights.

In one of the earliest works Lampson [10] wrote a proposal for access control. The proposal consisted of an access matrix that explicitly lists the rights a certain subject has on a certain object. Such an access control system is called a Discretionary Access Control (DAC). An access matrix enumerates the subjects and objects with the system. Each combination of subject and object is an entry in this matrix. Each entry in the matrix contains access rights that the subject is allowed to perform on the object. In practice storing a full matrix is inefficient. To this end, different implementations of access matrices have been proposed, for instance an Access Control List (ACL) a list of subjects allowed to access is assigned to an object. Alternatively a list of objects allowed to be accessed may be assigned to a subject, this is called a capability list.

Due to the hierarchical structure of the military, the military sector needs a different security approach. Rather than discretionary control and transfer of access, Mandatory Access Control (MAC) imposes certain restrictions based on a security class. A security class contains a security level, for example, secret or confidential, and a category representing for example an area of expertise. The security level and category are hierarchical in structure. Combined they form a lattice. Therefore, access control systems using these concepts are also called lattice based access control systems [15]. There are various models based on these principles. Most notably is the Bell-LaPadula [11] model. In this model a security level can be a classification for an object, and clearance for a subject. All subjects with a certain clearance level are allowed to access objects with a certain classification. The Bell-LaPadula model states that a subject can read an object when it’s clearance is at least the classification of the object, and write to an object if it’s clearance is at most the classification of the object. The philosophy behind these rules is preventing classified information to individuals without sufficient clearance. Information can never be written to objects with lower classification then the subject is able to read.

An approach that stands between DAC and MAC is Role Based Access Control (RBAC). In RBAC access rights are assigned to roles. Roles may be hierarchical in structure, in the sense that a superior role may inherit permissions from inferior roles. A role can represent a job function within an organisation. An individual is assigned one or more roles. Every individual in the same position has the same access rights. One of the main advantage of an RBAC system is easier administration as compared to previous access control proposals. Access rights can be modified per role rather than individually. Furthermore, RBAC can be used to prevent conflicting roles being assigned to an individual.

Attribute Based Access Control (ABAC) [9] is an approach of access control that can support DAC, MAC and RBAC. ABAC associates attributes to subjects and objects. Attributes are be key:value pairs, where
value can be a singleton or a set. Access decisions are based on the values of the attributes. Attributes can capture access control lists (DAC), clearance and classification (MAC), roles (RBAC), and other information, such as qualification or location.

2.2 Overview of UCON

Access control systems decide whether a subject is permitted to access an object. An access decision is a decision made at request time, a specific point in time. Usage control systems decide about usage. Decisions can be made, and changed, during the entire usage of a resource. After the initial decision to grant access, the usage can be revoked at a later point in time.

UCON [13] is an approach of usage control, capturing the demands in an open environment such as the Internet. UCON extends ABAC with mutable attributes and continuous evaluation. Attribute values may change as a side-effect of accessing an object. A policy may need to be enforced during the entire usage rather than only at request time. This means, an attribute value changed as a side-effect of accessing an object can cause a usage right to be revoked.

2.2.1 Concepts of UCON

UCON shares the concepts of subjects, objects and rights with traditional access models. Objects $O$ are passive entities or resources protected by UCON. Subjects $S$ are active entities. Attributes are assigned to Subjects and Objects, denoted by $ATT(S)$ and $ATT(O)$. Attributes are key:value pairs. A value may be a singleton or a set. A right is a usage function which allows a subject to access an object.

UCON has three decision factors: Authorisations (A), Obligations (B) and Conditions (C). Figure 2.1 gives a graphical representation of the decision factors used by UCON. UCON Authorisations are predicates on attribute values. UCON Obligations are mandatory requirements the subject has to perform in order to have a certain right on a certain object. UCON Conditions are predicates on environmental conditions. It is worth noting that environmental conditions do not depend on the subject, object or right and are not mutable by a policy.

UCON relies on two important concepts: mutability of attributes and continuity of policy enforcement. Figure 2.2 gives an overview of the timeline of a UCON evaluation. The UCON timeline consists of three phases, before the access begins (pre), during the usage (ongoing) and after the usage (post). Policy enforcement can take place in the pre and ongoing phase. Updates can take place in all three phases.
2.2. OVERVIEW OF UCON

UCON is a model to restrict access. As we will see in the following sections, UCON uses rules to specify the rights of a user. Adding rules can only result in more access restrictions. UCON has two types of rules, namely *allowed rules*, which are evaluated before the usage begins, and *stopped rules*, which are evaluated during the entire usage.

2.2.2 UCON Models

For each decision factor we have seen in previous section, Authorisations, Obligations and Conditions, an UCON core model exists. Core models can be combined to enable different decision factors in a decision. Figure 2.3(a) shows the possible combinations of the core models and their relationships. Each core model has a number of submodels, based on the decision time and the presence of updates. The possible submodels are shown in Figures 2.3(b), (c) and (d). The prefix *pre* represents decisions made at request time. The prefix *on* represents decisions that are re-evaluated continuously. The subscript 0 represents the absence of updates. The subscript 1, 2 and 3 represent pre-, ongoing- and post-updates respectively. Not all combinations are possible in the UCON model. The UCON model does not allow ongoing updates for policies that only have decisions at request time. This is visible in Figure 2.3(b) and (c) by the absence of pre models with subscript 2. Policies that depend on environmental conditions are also not allowed to have updates. This is visible in Figure 2.3(d) as the C models only have subscript 0. All UCON models share a number of common concepts, listed in Table 2.1. In each model there are objects (O) representing resources protected by UCON and subjects (S) representing active entities. Objects and subjects have attributes assigned, represented by ATT(O) and ATT(S). The distinctive details of each decision and update model are discussed in the following sections.

Pre Authorisation Model (preA)

The first model that we consider is the Pre Authorisation Model. The Pre Authorisation Model is used to evaluate attribute values before the usage begins, at request time. Table 2.2 lists the elements introduced in the Pre Authorisation model. A functional predicate, called *preA*, is introduced. This functional predicate takes a set of attributes ATT(S) and ATT(O),along with the right *r* as parameters and returns a decision. The Pre Authorisation Model support pre- and post-updates. (Section 2.2.3).

**Example 1** This example represents a model of [MAC] in UCON.
CHAPTER 2. PRELIMINARIES

Figure 2.3: The UCON_ABC Family of Core models

<table>
<thead>
<tr>
<th>PreA</th>
<th>Pre-Authorisations</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowed(s,o,r) ⇒ preA(ATT(S),ATT(O),r)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Pre Authorisations concept

Let $L$ be lattice of security labels with dominance relation $\geq$

- clearance $S \rightarrow L$
- classification $O \rightarrow L$
- $\text{ATT}(S)$ clearance
- $\text{ATT}(O)$ classification
- $\text{allowed}(s, o, \text{read}) \Rightarrow \text{clearance}(s) \geq \text{classification}(o)$
- $\text{allowed}(s, o, \text{write}) \Rightarrow \text{clearance}(s) \leq \text{classification}(o)$

The rights read and write are explicitly mentioned in the allowedrules. The allowedrules are only evaluated if the mentioned right matches the request. Clearance and classification as functions from subjects respectively objects to an element of the lattice $L$.

Ongoing Authorisation Model (onA)

The Ongoing Authorisation Model is used to evaluate attribute values during the entire usage. Table 2.3 lists the elements added in the Ongoing Authorisation Model. A functional predicate, called OnA, is introduced. OnA is continuously evaluated throughout duration of the usage. Once onA no longer holds, the right is revoked. The Ongoing Authorisation Model support pre-, ongoing- and post-updates (Section 2.2.3).

Pre Obligation Model (preB)

The Pre Obligation Model is used to verify a certain action has been taken in the past. Such an obligatory action is referred to as a Pre Obligation. Table 2.4 lists the elements added in the Pre Obligation Model. In the Pre Obligation Model model, a functional predicate called preB is introduced. PreB takes the history of previous usage and returns a decision. Auxiliary functions getPreOBL and preFulfilled assist onB to reach a decision. They select and evaluate the obligations that apply to the request.

The sets OBS, OBO and OB contain Subjects, Objects and Actions, respectively, which are used in Pre Obligations. OBS can be the same as S, but not necessarily. One can think of a policy where a supervisor must give permission. PreOBL is a set that contains pre-obligations, It consist of elements from OBS, OBO and OB. The Pre Obligation Model support pre- and post-updates (Section 2.2.3).
2.2. OVERVIEW OF UCON

Table 2.3: Ongoing Authorisation concepts

<table>
<thead>
<tr>
<th>OBS</th>
<th>Obligation subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBO</td>
<td>Obligation objects</td>
</tr>
<tr>
<td>OB</td>
<td>Obligation actions</td>
</tr>
<tr>
<td>preB</td>
<td>Pre Obligation predicates</td>
</tr>
<tr>
<td>preOBL:</td>
<td>$\subseteq OBS \times OBO \times OB$ (pre-obligation elements)</td>
</tr>
<tr>
<td>preFulfilled:</td>
<td>$OBS \times OBO \times OB \rightarrow {true, false}$</td>
</tr>
<tr>
<td>getPreOBL:</td>
<td>$S \times O \times R \rightarrow 2^{preOBL}$</td>
</tr>
<tr>
<td>preB(s, o, r) = $\bigwedge_{obs, obo, ob \in \text{getPreOBL}(s,o,r)} \text{preFulfilled}(obs, obo, ob)$</td>
<td></td>
</tr>
<tr>
<td>allowed(s, o, r) $\Rightarrow$ preB(s, o, r)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Pre Obligation concepts

Example 2 This example represents a scenario the user must accept a license agreement before using a service. There are two possible licenses, namely a high and low license agreement. Depending on the object’s attribute level, the applicable license agreement is selected, and verified if the subject has agreed. The function getPreOBL returns the applicable license agreement based on the object’s level attribute.

$OBS = S$
$OBO = \{high\_license\_agreement, low\_license\_agreement\}$
$OB = \{agree\}$
$level = O \rightarrow \{high, low\}$
$ATT(o) = level$
$\text{getPreOBL} = \begin{cases} (s, high\_license\_agreement, agree) & \text{if level}(o)='high' \\ (s, low\_license\_agreement, agree) & \text{if level}(o)='low' \end{cases}$
$\text{allowed}(s, o, r) \Rightarrow \text{preFulfilled}($getPreOBL$(s,o,r))$

Ongoing Obligation Model (onB)

The Ongoing Obligation Model is used when the user is required to perform a certain action during the usage of the requested resource. Such an obligatory action is referred to as an Ongoing Obligation. Table 2.5 lists the elements introduced in the Ongoing Obligation Model. A functional predicate $onB$ is introduced. $onB$ takes the currently active usage and returns a decision. Auxiliary functions $getOnOBL$ and $onFulfilled$ assist $onB$ to reach a decision. They select and evaluate the obligations that apply to the request. $onB$ is evaluated throughout the duration of the usage.

The way ongoing-obligations relate to pre-obligations is similar to how ongoing-authorisations relate to pre-authorisations. The sets $OBS$, $OBO$ and $OB$ contain subjects, objects and rights which are part of any Ongoing Obligations. Set $T$ is a set of times or event-based triggers which specify when an ongoing obligation must be fulfilled. $onB$ is evaluated during the entire usage. Once $onB$ no longer holds, the right is revoked. The Ongoing Obligation Model support pre-, ongoing- and post-updates (Section 2.2.3).

Example 3 The scenario is a free service, where the user has to keep an advertisement window open during the usage. Usage is allowed as long as an advertisement window is active.

$OBS = S$
$OBO = \{ad\_window\}$
$OB = \{keep\_active\}$
$T = \{always\}$
$\text{getOnOBL}(s,o,r) = \{ (s, ad\_window, keep\_active, always) \}$
$\text{stopped}(s, o, r) \Leftarrow \neg \text{onFulfilled}(s, ad\_window, keep\_active, always)$
CHAPTER 2. PRELIMINARIES

<table>
<thead>
<tr>
<th>OBS</th>
<th>Obligation subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBO</td>
<td>Obligation objects</td>
</tr>
<tr>
<td>OB</td>
<td>Obligation actions</td>
</tr>
<tr>
<td>T</td>
<td>A set of time or event elements</td>
</tr>
<tr>
<td>onB</td>
<td>Ongoing Obligation predicates</td>
</tr>
<tr>
<td>onFulfilled</td>
<td>$OBS \times OBO \times OB \times T \rightarrow {true, false}$</td>
</tr>
<tr>
<td>onOBL</td>
<td>$OBS \times OBO \times OB \times T$ (ongoing-obligation elements)</td>
</tr>
<tr>
<td>getOnOBL</td>
<td>$S \times O \times R \rightarrow 2^{\text{onOBL}}$</td>
</tr>
<tr>
<td>stopped</td>
<td>$\neg \text{onB}(s, o, r)$</td>
</tr>
</tbody>
</table>

Table 2.5: Ongoing Obligation concepts

<table>
<thead>
<tr>
<th>preCON</th>
<th>Pre Condition elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>preC</td>
<td>Pre Condition predicates</td>
</tr>
<tr>
<td>getPreCON</td>
<td>$S \times O \times R \rightarrow 2^{\text{preCON}}$</td>
</tr>
<tr>
<td>preConChecked</td>
<td>$\text{preCON} \rightarrow \text{true, false}$</td>
</tr>
<tr>
<td>preC(s, o, r) = $\bigwedge_{\text{preCON}_i \in \text{getPreCON}(s,o,r)} \text{preConChecked}(\text{preCON}_i)$</td>
<td></td>
</tr>
<tr>
<td>allowed(s, o, r) $\Rightarrow$ preC(s, o, r)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Pre Condition concepts

Pre Conditions Model (preC)

The Pre Conditions Model is used to evaluate environmental conditions before the usage begins, at request time. Table 2.6 lists the elements introduced in the Pre Conditions Model. In the Pre Conditions Model the preC functional predicate is introduced. preC takes environment variables and returns a decision. Auxiliary functions getPreCON and preConChecked assist preC to reach a decision. They select and evaluate the conditions that apply to the request. The Pre Conditions Model does not support updates.

Example 4 Allowed locations for students and faculty can be different and are selected according to the value of the subject attribute member. The group membership and the location are validated at request time.

studentAREA, facultyAREA (allowed area codes for student and faculty)
curArea is a current rendering device’s area code
$ATT(S) = \{\text{member}\}$

getPreCON = \{
    (\text{curArea} \in \text{studentAREA}) \quad \text{if member}(s) = \text{’student’}
    (\text{curArea} \in \text{facultyAREA}) \quad \text{if member}(s) = \text{’faculty’}
\}

allowed(s, o, r) $\Rightarrow$ preConChecked(getPreCON(s,o,r))

Ongoing Condition Model (onC)

The Pre Conditions Model is used to evaluate environmental conditions during the usage. Table 2.7 lists the elements introduced in the Ongoing Conditions Model. In the Ongoing Conditions Model the functional predicate onC is introduced. It takes environmental variables and returns a decision. Auxiliary functions getOnCON and preOnChecked assist PreC to reach a decision. They select and evaluate the conditions that apply to the request. OnC is evaluated during the entire usage. Once OnC no longer holds, the right is revoked. The Ongoing Conditions Model does not support updates.

Example 5 Usage is only allowed during the subject’s assigned shift. When a shift is over, usage is stopped immediately.

dayH, nightH (day-shift and night-shift office hours, mutually exclusive)
currentT is the current time
$ATT(s) = \{\text{shift}\}$
2.2. OVERVIEW OF UCON

<table>
<thead>
<tr>
<th>onCON</th>
<th>Ongoing Condition elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>onC</td>
<td>Ongoing Condition predicates</td>
</tr>
</tbody>
</table>

\[
\text{getOnCON} : S \times O \times R \rightarrow 2^{\text{onCON}}
\]

\[
\text{onConChecked} : \text{onCON} \rightarrow \text{true, false}
\]

\[
\text{onC}(s, o, r) = \bigwedge_{\text{onCON}_i \in \text{getOnCON}(s, o, r)} \text{onConChecked}(\text{onCon}_i)
\]

\[
\text{stopped}(s, o, r) \leftarrow \neg \text{onC}(s, o, r)
\]

Table 2.7: Ongoing Conditions concepts

<table>
<thead>
<tr>
<th>Pre Update</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT(O)</td>
<td>preUpdate(ATT(O))</td>
</tr>
<tr>
<td>ATT(S)</td>
<td>preUpdate(ATT(S))</td>
</tr>
</tbody>
</table>

Table 2.8: Pre Update concepts

\[
\text{getOnCON} = \begin{cases} 
  \{ \text{current}T \in \text{dayH} \} & \text{if shift(s)='day'} \\
  \{ \text{current}T \in \text{nightH} \} & \text{if shift(s)='night'} 
\end{cases}
\]

\[
\text{stopped}(s, o, r) \leftarrow \neg \text{onConChecked(getOnCON(s,o,r))}
\]

2.2.3 UCON Updates

The UCON Models can be extended with updates. Updates are assignments of new values to subject or object attributes. There are three update models, (1) pre-updates, (2) ongoing-updates and (3) post-updates. Attributes that can be changed by updates are called mutable attributes. Attributes that do not appear in any update are immutable attributes. Figure 2.3 shows the possible combinations of the UCON core models and UCON update models.

Pre Updates

Pre-updates are updates which are performed before the access to an object is granted. Table 2.8 lists the update functions introduced with Pre Updates.

The \text{preUpdate} function takes an object attribute or a subject attribute and assigns a new value to it, right before the access begins.

\textbf{Example 6} The credits of the subject will be decreased by the value of the object before the access is granted. We assign the identifier \text{u1} to a \text{PreUpdate} which decreases the subject’s credit by the object’s value. We assign the update to the rule.

Let \(M\) be a set of money amount

\[
\begin{align*}
\text{credit} & : S \rightarrow M \\
\text{value} & : O \rightarrow M
\end{align*}
\]

\[
\text{ATT}(S) \quad \text{credit} \\
\text{ATT}(O) \quad \text{value}
\]

\[
\text{allowed}(s, o, r) \Rightarrow \text{credit}(s) \geq \text{value}(o)
\]

\[
\text{preUpdate(credit(s))} : \text{credit}(s) = \text{credit}(s) - \text{value}(o)
\]

Ongoing Updates

Ongoing-updates are updates which are performed during the access. Table 2.9 lists the update functions introduced with Ongoing Updates.

The \text{onUpdate} function takes an object attribute or a subject attribute and assigns a new value to it during the access.
Post Updates

Pre-updates are updates which are performed after the access has finished. Table 2.10 lists the update functions introduced with Post Updates.

The postUpdate function takes an object attribute or a subject attribute and assigns a new value to it, right after the access has finished.

2.3 Overview of XACML

Many organisations are using legacy systems with built in authorisation logics. In the world of today many systems are interconnected through intranet or Internet. Users inside and outside organisations have the need to share resources in order to collaborate. Legacy systems with hard-coded or system-specific authorisation logic make it hard to manage the authorisation in a world of continuous changing demands. This calls for a standardised solution to manage authorisations.

XACML is an attribute based access control policy language using Extensible Markup Language (XML) syntax. XACML is a standard by OASIS for the specification of access control policies and has found a wide usage in both industry and academia. The XACML standard defines both a language for the specification of access control policies and a reference system architecture design. In this document version XACML 2.0 [12] is used. Even though version 3.0 has been released, we adopt version 2.0 due the availability of implementations.

2.3.1 XACML Language

The metamodel of a XACML policy is depicted in Figure 2.4. We will discuss the various elements found in the metamodel in this section.

A <Target> element serves as an applicability filter on subject, resource, action and environment attributes. A <Rule> consists of a <Condition>, an Effect and optionally a <Target> element. A <Condition> is a functional predicate on subject, resource, action and environment attributes. A functional predicate may contain nested functional predicates and use built-in and non-standard functions. Functions defined by the XACML standard are called build-in functions. Implementation specific functions are called non-standard functions.

Built-in functions include logical operators such as and and or, comparison functions such as less - than and equal. In XACML functions are typed: string - equal and integer - equal are different functions. Furthermore, there are set and bag functions such as, one-and-only, which extract the element of a bag containing only one element.

An <Obligation> element contains an ObligationId and a FulFillOn value. The ObligationId can contain an arbitrary value. The FulFillOn can be either Permit or Deny, indicating the evaluation result when the <Obligation> has to be released. Optionally, an <Obligation> may contain one or more arbitrary attributes.

A <Policy> element contains one or more <Rule> elements and optionally <Obligation> elements and a <Target> element. A <Policy> element contains a RuleCombiningAlgId value to indicate the Rule Combining Algorithm.
2.3. OVERVIEW OF XACML

A <Policy> element may contain other <PolicySet> elements, <Policy> elements, <Obligation> elements and a <Target> element. A <PolicySet> element contains a PolicyCombiningAlgId value to indicate the Policy Combining Algorithm.

Example 7 Consider the XACML policy in listing 2.1. This policy has a Target, which states the action-id must be a string and match the string eat. (lines 3 - 15) This policy contains two rules, diet and vegetarian. The diet Rule (lines 16 - 37). The effect of this Rule is Deny. This rule denied fastfood to subjects who are on diet. The vegetarian rule (lines 38 - 52) has an empty Target, meaning it applies to all requests. The decision is made using the Condition. A Condition is more flexible and allows comparison of subject and resource attributes. This rule states that vegetarian food is only served to vegetarians, and non-vegetarian food is only served to non-vegetarians.

2.3.2 XACML Policy Evaluation

In this section we discuss the evaluation process of a XACML request against a XACML policy. A XACML request contains of a set of subject, resource, action and environment attributes. A XACML policy contains the elements introduced in the previous section.

A <Target> element is a filter on the set of subject, resource, action and environment attributes of a XACML request. The evaluation of a <Target> element results in either Match or No-Match. A XACML request is evaluated against a <Rule> element. Table 2.13 shows the evaluation result for a <Rule> element. A <Rule> element is evaluated a decision. A decision is either Permit, Deny, NotApplicable.
or Indeterminate. If a <Rule> element contains a <Target> element, the <Target> element is evaluated first. If the <Target> element evaluates to No-Match, the evaluation result of the <Rule> element is NotApplicable. If the <Target> element evaluates to Match, the <Condition> element contained in the <Rule> element is evaluated. The <Condition> element contains a functional predicate, which may contain nested predicates and functions. If the functional predicate evaluates to true, the evaluation result of the <Rule> element is the value defined in its Effect. The Effect can be either Permit or Deny. If the functional predicate evaluates to false, the result is NotApplicable. In case an error occurs during the evaluation of the functional predicate, which can occur when the functional predicate refers to attributes which are not present, the evaluation result of the Rule is Indeterminate.

A XACML request is evaluated against a Policy element. Table 2.11 shows the evaluation result for a <Policy> element. A Policy element is evaluated to a decision. If a <Policy> element contains a <Target> element, the <Target> element is evaluated first. If the <Target> element evaluates to No-Match, the evaluation result of the <Policy> element is NotApplicable. If the <Target> element evaluates to Match, the <Rule> elements contained by the <Policy> element are evaluated. Based on the Rule Combining Algorithm specified by the RuleCombiningAlgId a decision is reached based on the evaluation results of the <Rule> elements. If the decision matches the FulFillOn of <Obligation> elements contained in the <Policy> element, those <Obligation> elements are included in the evaluation result.

A XACML request is evaluated against a PolicySet element. Table 2.12 shows the evaluation result for a <PolicySet> element. A PolicySet element is evaluated to a decision. If a <PolicySet> element contains a <Target> element, the <Target> element is evaluated first. If the <Target> element evaluates to No-Match, the evaluation result of the <PolicySet> element is NotApplicable. If the <Target> element evaluates to Match, the <Policy> elements contained by the <PolicySet> element are evaluated. Based on the Policy Combining Algorithm specified by the PolicyCombiningAlgId a decision is reached based on the evaluation results of the <Policy> elements. If the decision matches the FulFillOn of <Obligation> elements contained in the <PolicySet> element, those <Obligation> elements are included in the evaluation result.

Listing 2.1: XACML Example
### 2.3. OVERVIEW OF XACML

#### Target

<table>
<thead>
<tr>
<th>Match</th>
<th>At least one rule value is its Effect</th>
<th>Specified by the rule-combining algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>All rule values are NotApplicable</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Match</td>
<td>At least one rule value is Indeterminate</td>
<td>Specified by the rule-combining algorithm</td>
</tr>
<tr>
<td>No-match</td>
<td>Don’t care</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>Don’t care</td>
<td>Indeterminate</td>
</tr>
</tbody>
</table>

Table 2.11: XACML Policy Truth Table [12]

<table>
<thead>
<tr>
<th>Target</th>
<th>Policy Values</th>
<th>PolicySet Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>At least one policy value is its Decision</td>
<td>Specified by the policy-combining algorithm</td>
</tr>
<tr>
<td>Match</td>
<td>All policy values are NotApplicable</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Match</td>
<td>At least one policy value is Indeterminate</td>
<td>Specified by the policy-combining algorithm</td>
</tr>
<tr>
<td>No-match</td>
<td>Don’t care</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>Don’t care</td>
<td>Indeterminate</td>
</tr>
</tbody>
</table>

Table 2.12: XACML PolicySet Truth Table [12]

Possible Rule and Policy Combination Algorithms are (Ordered) Deny-overrides, (Ordered) Permit-overrides and First Applicable. The Deny-overrides algorithm returns Deny of one of the rules or policies evaluate to Deny, otherwise if one of the rules evaluated to Permit, it returns Permit. The Permit-overrides is the Permit-counterpart of Deny-overrides. The order of evaluation is not specified by these algorithms. If this behaviour is unacceptable, their Ordered variants should be used. This is the case when <Obligation> elements are involved. The Ordered variants ensure they follow the order in which the rules are written in the policy. The First Applicable also evaluates the rules or policies in the order they are written in the policy, and returns the result of the first applicable rule. A Combination Algorithm that only exists as Policy Combining Algorithm is Only One Applicable. All policies are evaluated. If only one is applicable, its result is returned, otherwise the return value is Indeterminate.

#### 2.3.3 XACML Reference Architecture

Apart from defining the syntax and semantics of the XACML Policy language, the XACML specifications also provide a reference system architecture design. In this section we provide an overview of the XACML reference architecture (Figure 2.5). In particular, we first present the components and then the data flow.

**PAP (Policy Administration Point)** The Policy Administration Point (PAP) is the component in the XACML reference architecture responsible for deploying Policies and PolicySets.

**PDP (Policy Decision Point)** The Policy Decision Point (PDP) is the component in the XACML reference architecture responsible for evaluating access requests against deployed policies and reaching an access decision.

**PEP (Policy Enforcement Point)** The Policy Enforcement Point (PEP) is the component in the XACML reference architecture responsible for forwarding access requests and enforcing the made access decision.

**PIP (Policy Information Point)** The Policy Information Point (PIP) is the component in the XACML reference architecture responsible for retrieving information necessary to evaluate policies.
<table>
<thead>
<tr>
<th>Target</th>
<th>Condition</th>
<th>Rule Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>True</td>
<td>Effect</td>
</tr>
<tr>
<td>Match</td>
<td>False</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Match</td>
<td>Indeterminate</td>
<td>Indeterminate</td>
</tr>
<tr>
<td>No-match</td>
<td>Don’t care</td>
<td>NotApplicable</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>Don’t care</td>
<td>Indeterminate</td>
</tr>
</tbody>
</table>

Table 2.13: XACML Rule Truth Table [12]

CH (Context Handler) The Context Handler (CH) is the component in the XACML reference architecture responsible translating data exchanged among components between native data formats and XACML data formats.

Obligation Services (OS) The Obligation Service (OS) is the component in the XACML reference architecture responsible for handling obligations. The interpretation of obligations is up to implementation.

XACML Reference Data Flow

Figure 2.5 shows a graphical representation of the XACML reference architecture and depicts the data flow. This reference architecture is not intended to restrict implementations. The steps shown in this data flow are

1. The PAP makes PolicySets available to the PDP
2. The Access Requester sends a request to the PEP
3. The PEP sends the request for access to the CH. This request is sent in its native request format. Optionally it may include attributes of the subject, resource, action and environment.
4. The CH creates an XACML request context and sends it to the PDP
5. The PDP requests additional attributes of the subject, resource, action and environment from the CH.
6. The CH requests the attributes from the PIP.
7. The PIP obtains the requested attributes.
8. The PIP returns the requested attributes.
9. Optionally, the CH includes the resource in the context.
10. The CH sends the attributes, and optionally the resource, to the PDP.
11. The PDP evaluated the attributes and sends the response context, including the access decision to the CH.
12. The CH translates the response context to the response format of the PEP and returns it.
13. The PEP fulfils the obligations.
14. If access is permitted, the PEP grants access to the resource, otherwise it denies access.
Chapter 3

Analysis of UCON and XACML

The goal of this work is to extend XACML in such a way we can express and enforce UCON policies. In this chapter we analyse UCON and XACML, map common features and identify conceptual differences. In Section 3.1 we discuss the details of UCON. For XACML such discussion is not needed as we follow the standard. Finally, in Section 3.2 we compare our findings about UCON and XACML.

3.1 Analysis of UCON

UCON is a family of usage control models which extend Attribute Based Access Control (ABAC) with the concepts of mutable attributes and continuity of policy enforcement.

In UCON two types of rules are used: allowed rules and stopped rules. Allowed rules are used for access decisions at request time. Stopped rules are used for continuous usage decisions. The allowed rule is defined as:

\[
\text{allowed}(s, o, r) \Rightarrow \text{constraints} \tag{3.1}
\]

This definition states that the decision to allow usage implies that the constraints are true.

In practice this representation is not appropriate, as we are interested in deriving a decision from the constraints. Therefore we rewrite (3.1) as

\[
\neg \text{constraints} \Rightarrow \neg \text{allowed}(s, o, r) \tag{3.2}
\]

Future references to \(\neg \text{allowed}(s, o, r)\) are denoted as \(\text{denied}(s, o, r)\) (denied-notation). The denied-notation is the logical transposition of the allowed-notation and is logically equivalent. Consider a rule stating access is allowed when the count is 3. The allowed and denied notation are logically equivalent:

\[
(\text{allowed}(s, o, r) \Rightarrow \text{count}(s) = 3) \equiv (\text{count}(s) \neq 3 \Rightarrow \text{denied}(s, o, r)) \tag{3.3}
\]

It states that when the constraints are not met, the access is denied. Stated differently, the denied-notation define the access constraints. Further references to constraints denote the negated instance.

When there are multiple denied-rules, it is sufficient for one of the constraints to hold to result in a denial of access. Equation 3.4 shows the logic behind this behaviour.

\[
((\text{constraint}1 \Rightarrow \text{denied}(s, o, r)) \land (\text{constraint}2 \Rightarrow \text{denied}(s, o, r))) \equiv \\
((\text{constraint}1 \lor \text{constraint}2) \Rightarrow \text{denied}(s, o, r)) \tag{3.4}
\]

Ongoing UCON policies use stopped rules. Stopped rules define the constraints for a usage. The stopped rule is defined as:

\[
\text{stopped}(s, o, r) \Leftarrow \text{constraints} \tag{3.5}
\]

When the constraints are no longer met, the usage is stopped. Since the decision to stop a usage follows from the constraints, no adjustment is required. The distinction between stopped rules and denied rules
Figure 3.1: UCON Policy evaluation

is whether they are to be evaluated at request time or continuously. A denied-rule denies access at request
time, while stopped rule stops a running usage.

Example 8 Consider a UCON policy which states the subject’s level must be at least the object’s required
level and the request must happen within opening hours.

openH (opening hours)
currentT (the current time)
level S → N
requiredlevel O → N
ATT (S) level
ATT (O) requiredlevel
allowed(s, o, r) ⇒ level(s) ≥ requiredlevel(o)
allowed(s, o, r) ⇒ currentT ∈ openH

In the denied notation the allowed rules are written as
level(s) < requiredlevel(o) ⇒ denied(s, o, r)
currentT /∈ openH ⇒ denied(s, o, r)

Figure 3.1 is a graphical representation of the evaluation of this policy. The area accented with horizontal
lines marks where level(s) < requiredlevel(o) is true. The area accented with vertical lines marks where
currentT /∈ openH is true. Access is denied when either level(s) < requiredlevel(o) or currentT /∈ openH
are true. This corresponds with all the accented regions in the figure. When both conditions are false
access is granted. This is represented by the unaccented area in the figure.

The evaluation result of a UCON policy is either to allow or deny access. Adding more rules to an
UCON policy can only put more restrictions on access. Therefore UCON rules define the deny-space.
This implies in the absence of rules, access is granted by default. When access is allowed, the result may
include pre-updates, ongoing-updates or post-updates.

The way updates are defined in UCON is limited. An update is performed if the policy allows usage.
This means there is no way to define an update only applies when a specific right is requested. The
update policy examples presented in the UCON definitions do not consider this possibility. Therefore we
propose an extension to the update definition that allows to define updates that are assigned to rules. A
list of update identifiers can be appended to a rule. In particular we redefine rules as

\[
\text{allowed}(s, o, r) \Rightarrow \text{constraints} : \text{update identifiers} \text{stopped}(s, o, r) \Rightarrow \text{constraints} : \text{update identifiers}
\]

We redefine updates as

\[
\ast \text{update(attribute, identifier)} : \text{expression}
\]
3.2 COMPARISON OF UCON AND XACML

where \( \ast \) update represents \( \text{preUpdate} \), \( \text{onUpdate} \) or \( \text{postUpdate} \). When there is a match between the update identifier at the rule, the update is performed.

**Example 9** Consider a policy that expresses two rules with different rights: read and write. We set a limit on the number of reads and writes on an object. Therefore we need to keep a read count and a write count. This is impossible in the standard UCON update definition. Our extension to the update definition consists of assigning an id to updates. Then we can assign updates to specific rules. Furthermore, we keep an access count at the subject. Subjects are limited on the number of accesses, being either read or write, they are allowed to perform. This is encoded using two allowed rules, one for the right read and one for the right write. The update for increasing the access count is assigned to both rules, while the update for increasing the read count is only assigned to the rule with right read, and the update for the write count only to the write rule.

\[
\text{allowed}(s,o,\text{read}) \Rightarrow \text{readCount}(o) \leq 10 \land \text{accessCount}(s) \leq 10 : u_1, u_3 \\
\text{allowed}(s,o,\text{write}) \Rightarrow \text{writeCount}(o) \leq 10 \land \text{accessCount}(s) \leq 10 : u_2, u_3 \\
\text{postupdate}(\text{readCount}(o),u_1) : \text{readCount}(o) + 1 \\
\text{postupdate}(\text{writeCount}(o),u_2) : \text{writeCount}(o) + 1 \\
\text{postupdate}(\text{accessCount}(s),u_3) : \text{accessCount}(s) + 1
\]

3.2 Comparison of UCON and XACML

Our goal is to express and enforce UCON in XACML. To this end we need to define a mapping from UCON policies to XACML policies. Table 3.1 shows a comparison between UCON and XACML.

UCON uses subject attributes, object attributes, environment state and other accesses as decision factors. XACML uses subject attributes, resource attributes and environment attributes as decision factors. Subject attributes can be mapped upon each other. Object attributes can be mapped on Resource attributes. The system/environment state can be mapped upon Environment attributes. UCON distinguishes mutable and immutable attributes, where the evaluation of a UCON policy may result in attribute updates of mutable attributes. XACML does not support updating of attributes, and therefore lacks the notion of mutable and immutable attributes too.

Both UCON and XACML have a concept called Obligations. However, they represent different concepts. A UCON Obligation is a certain action a user must have performed in the past, or must currently be performing. A XACML Obligation is a part of the evaluation result, therefore the Obligation has to be fulfilled after the request has been evaluated. A XACML Obligation is therefore performed in the future and by the system.

UCON policy evaluation can be either at request time or continuous, while XACML only evaluates a policy at request time. UCON has support referencing to past and current usage. There is no such concept in XACML. XACML provides a method of adding non-standard functions. A non-standard can be used to make past and current usage available to be used in policies. Such a function would return a bag of usage. Using build-in functions a set inclusion test can be performed on the returned bag.

The possible answers to an access request or decision set of UCON is Allow and Deny. The decision set of XACML is Permit, Deny, Not Applicable and Indeterminate. Allow can be mapped to Permit, Deny can be mapped to Deny.

When multiple rules are available in a UCON policy, access is denied if one of the rules denied access. As a result, access is granted in the absence of rules. This behaviour can be mapped to XACML using the DenyOverrides combining algorithm in combination with a default rule which always permits access.

In summary, the missing features in XACML are mutable attributes, continuous evaluation, and current/past accesses. Since XACML Obligations may contain arbitrary data included in the access evaluation result, XACML Obligation can be used to include these features in a policy. Current and Past Access can be made available as an environment variable. In order to support the mapping from UCON to XACML an intermediate policy language is defined. The intermediate policy language is discussed in Chapter 4.
<table>
<thead>
<tr>
<th></th>
<th>UCON</th>
<th>XACML</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Attribute based</td>
<td>Attribute based</td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td>Pre usage</td>
<td>Pre usage</td>
</tr>
<tr>
<td></td>
<td>Ongoing usage</td>
<td></td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Mutable</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Immutable</td>
<td></td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>Object Attributes</td>
<td>Resource Attributes</td>
</tr>
<tr>
<td></td>
<td>Subject Attributes</td>
<td>Subject Attributes</td>
</tr>
<tr>
<td></td>
<td>System/environment state</td>
<td>Environment Attributes</td>
</tr>
<tr>
<td></td>
<td>Current/past access</td>
<td>Action Attributes</td>
</tr>
<tr>
<td><strong>Multiple Rules</strong></td>
<td>Deny overrides</td>
<td>(Ordered) Deny-overrides,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Ordered) Permit-overrides</td>
</tr>
<tr>
<td></td>
<td></td>
<td>First Applicable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only One Applicable</td>
</tr>
<tr>
<td><strong>Decision Set</strong></td>
<td>Allow</td>
<td>Permit</td>
</tr>
<tr>
<td></td>
<td>Deny</td>
<td>Deny</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not Applicable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indeterminate</td>
</tr>
<tr>
<td><strong>Obligation</strong></td>
<td>By user</td>
<td>By system</td>
</tr>
<tr>
<td></td>
<td>Before or during usage</td>
<td>After request</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison UCON and XACML
Chapter 4

Intermediate Policy Language

Converting a UCON policy to XACML requires defining how UCON concepts are mapped to XACML concepts. To facilitate such mapping we define an Intermediate Policy Language. The intermediate policy language is designed in such a way it allows (nearly) mechanical conversion to the XACML policy language, while aiming to be better human readable than the resulting XACML. Section 4.1 introduces the concepts of the Intermediate Policy Language. Section 4.2 shows how a policy in the Intermediate Policy Language is evaluated.

4.1 Concepts

Figure 4.1 gives a UML [8] representation of the Intermediate Policy Language. An EBNF [6] representation of the Intermediate Policy Language is given in Figure 4.2.

A Policy may consist of a Rule, an Action and a set of Updates. All elements of a policy are optional. A Rule consists of an EvaluationTime and a Condition. EvaluationTime indicates when the condition has to be evaluated. Possible values for EvaluationTime are before usage for UCON pre-models and ongoing for UCON ongoing-models. In case of ongoing, there are three options: timer, representing a periodical timer activated re-evaluation, timestamp, which represent a re-evaluation at a specific time, and attribute change, representing a re-evaluation at the change of a certain attribute. Table 4.1 shows an overview of values for the EvaluationTime element.

Conditions are boolean predicates. The evaluation result of the boolean predicate determines when access is denied. Conditions are built from constants, subject attribute values, object attribute values, environment attribute values, or set inclusion tests on the PastUsage or CurrentUsage sets. CurrentUsage contains usage currently taking place. It is used to verify a certain usage is currently taking place. PastUsage contains usage that has ended and are available to be referred to in rules to validate future access requests. It is used to verify a certain usage has taken place in the past. Although not explicitly mentioned, the examples provided in [13] suggest that, once a rule has referenced an usage in the UCON pre obligations model, the same usage cannot be referred to again. Therefore, when a request is granted access, and the evaluated rules included a reference to a usage in PastUsage, the usage is removed from PastUsage. This behaviour is modelled in our Intermediate Policy Language evaluation process (Section 4.2), however, our XACML implementation leaves this behaviour optional. Allowed rules and stopped rules from the UCON model are encoded as conditions in the intermediate model.

An update contains an ExecutionTime and an Operation. An Operation assigns a new value to an object or subject attribute. The new value is calculated from current attribute values and constants. The ExecutionTime defines when the Operations have to be performed. Possible values are before usage, after usage or ongoing. This is the equivalent of pre, post and ongoing updates in UCON. In case of ongoing, there are three possibilities: timer, representing a periodical timer activated trigger, timestamp, which represent a trigger activated on a specific timestamp, and attribute change, representing a trigger on the change of a certain attribute. Table 4.1 shows an overview of values for the ExecutionTime element.

Optionally, an update may contain a Condition, containing a functional predicate. If a Condition is present, the functional predicate must evaluate to true to perform the update.
CHAPTER 4. INTERMEDIATE POLICY LANGUAGE

Figure 4.1: Intermediate Policy Language Model, UML representation

Policy = [Rule] [Action] [Updates]
Rule = EvaluationTime Condition
Updates = Update [Updates]
Update = ExecutionTime Operation [Condition]

Figure 4.2: Policy Language Model, EBNF representation

The concept of Effects, as seen in XACML, is not used in the intermediate model. An Effect in XACML represents the decision made when the condition is true. In the intermediate model access is denied when the condition is true. This behaviour follows UCON which only uses deny-based rules. (See Section 2.2). In other words, the effect of a rule in the intermediate model is implicitly deny.

Example 10 Consider a usage control policy for an online game. In order to play the game, a certain price has to be paid per minute. Playing is allowed as long as the subject has sufficient credit. Each time a subject has finished a game, their playcount is increased. Additionally, minors need permission from a parent in order to play the game. Every 5 minutes, the player has to watch an advertisement. The system is only available between 18:00 and 21:00. Figure 4.3 shows a policy encoding this scenario.

The syntax of the policy language is a JSON representation of a policy diagram (Figure 4.1) instance. Conditions and Operations use C syntax, extended with a set inclusion operator in. Time intervals are in the notation ISO 8601 defines for intervals.

Figure 4.3 shows a policy encoding the constraints described in the scenario, to be specific: Lines 1-15 encode the credit restriction. Lines 16-23 encode the age restriction. Lines 24-30 encode the advertisement requirement. Lines 31-36 encode the time restriction.

4.2 Policy Evaluation

Figure 4.4 shows a flow diagram of the policy evaluation process. During the evaluation process a temporary storage is used to store updates which are to be executed later. When a request arrives, the applicable policies are determined and queued for evaluation (1). A policy is considered applicable when it has no action specified, or when the specified action matches the action specified in the request.
The evaluation process of a policy consists of evaluating the boolean formula (2) in the condition of the policy. The policies are deny-based, which means, when the boolean formula evaluates to true, access will be denied. Once a policy denies access, this is the final decision (D1). If the rule being evaluated allows access, no decision has been reached until all applicable policies have been evaluated. If the rule contains updates, these updates are held in the temporary storage (3). The updates are stored at this point because they are only to be processed after all applicable policies have been evaluated. This procedure is repeated until a policy has denied access, or all applicable policies have allowed access. As a consequence, a request with no applicable policies will be granted by default.

A usage stored in PastUsage is only supposed to be referenced to in one access request. To prevent further access request from using the same usage, any past usages referenced in any applies rules are removed from PastUsage. (4).

Updates stored in step 3 which have an ExecutionTime before usage are executed (5). Updates alter the values of attributes on which other updates might depend, therefore updates are executed in order of definition. Since request is to be granted, the access request is considered a usage now. The usage is stored in the ActiveUsage (6), and the requesting subject is informed the access is granted (7).

During the usage, continuous updates are performed (8). Continuous policies are re-evaluated (9) continuously, until the subject ends the usage (10) or the re-evaluation of a continuous policy denies access (D2). When there are no continuous policies, evaluation result for continuous policies defaults to allow.

When the usage has ended, it is moved from ActiveUsage to PastUsage (11). This step may influence other access requests as they may reference to PastUsage or ActiveUsage. The final step in policy evaluation is applying updates with ExecutionTime after access. (12). Updates alter the values of attributes on which other updates might depend, therefore updates are executed in order of definition.

**Example 11** This example illustrates the evaluation process. In this example, objects and subjects are identified with a string stored as attribute ObjectID and SubjectID respectively.

The scenario for this example is a book store. In order to pick up a book, it must have been paid for. Paying and picking up books is only allowed after 6 PM.

```
"policy" : {
  "rule" : {
    "EvaluationTime" : "on access",
    "Condition" : "e.time < 18:00:00"
  }
},
"policy" : {
  "action" : "pay",
  "rule" : {
    "EvaluationTime" : "on access",
    "Condition" : "s.credit < o.price"
  }
},
```
In the discussion about the evaluation of the example, as a short-hand notation, the set of attribute/value pairs of the subjects and object and the set of ActiveUsage and PastUsage will be referred to as system state. Consider the initial system state as shown in Table 4.2 State A.

Consider a request (john, book, pay) made at 17:00. This request will be denied, since the time (in the environment) is before 18:00 hours. The system state has not been changed.
4.2. POLICY EVALUATION

Consider a request \((john, book, pickup)\) made at 19:00. This request will be denied, because \((john, book, pay)\) is not in PastUsage.

Consider a request \((john, book, pay)\) made at 19:00. This request is allowed because the time is past 18:00 and the subject’s credit is above the object’s price. As a result of this request, the subject’s attributes are updated. The system state becomes as shown in Table 4.2 State B.

Further requests in this example will be made at 19:00. We consider the pickup request \((john, book, pickup)\) again. \((john, book, pay)\) is in PastUsage now, so the request will be granted. As a result of performing this action, the PastUsage will be empty again. The system state becomes as shown in Table 4.2 State C.

Because the usage has been removed from PastUsage, further pickup requests will be denied.

Figure 4.4: Activity Diagram

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>State A</th>
<th>State B</th>
<th>State C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>price</td>
<td>5.00</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Subject</td>
<td>subjectID</td>
<td>john</td>
<td>john</td>
<td>john</td>
</tr>
<tr>
<td></td>
<td>credit</td>
<td>100.00</td>
<td>95.00</td>
<td>95.00</td>
</tr>
<tr>
<td>PastUsage</td>
<td>-</td>
<td></td>
<td>(john,book, pay)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: System States
Chapter 5

Mapping from UCON to XACML

This Chapter discusses the conversion from UCON to XACML policies, using the Intermediate Policy Language. The conversion consists of two steps. The first step is to convert the UCON policy to an intermediate policy language that serves as a bridge. The second step is to convert the policy in the Intermediate Policy Language to XACML. The Intermediate Policy Language is designed in such a way that the conversion to XACML can be done (nearly) mechanical, while keeping the policy more human-readable than the resulting XACML policy. Using the Intermediate Policy Language as an intermediate step eases the conversion process.

In Section 5.1 the conversion from UCON to the Intermediate Policy Language is explained. In Section 5.2 the conversion from the Intermediate Policy Language to XACML is discussed.

5.1 UCON to Intermediate Policy Language

This section explains the translation from UCON to the Intermediate Policy Language. The conversion of UCON allowed rule, stopped rules and updates are discussed in the following subsections.

5.1.1 Allowed rules

Recall the definition for a UCON allowed rule:

\[
\text{allowed}(s, o, r) \Rightarrow \text{constraints}
\]  

(5.1)

This definition states when access is allowed the constraints (a boolean expression) hold. Since we are interested in reaching a decision given the circumstances, we need to rewrite the rule in a form in which a decision follows from circumstances. (See Section 2.2)

\[
\neg \text{constraints} \Rightarrow \text{denied}(s, o, r)
\]  

(5.2)

The resulting \(\neg\text{constraints}\) is used in the intermediate model. The resulting boolean expression is the condition in the intermediate model. Notation to refer to attributes is adjusted to the notation defined in Chapter 4. Figure 5.1 shows a conversion from an UCON allowed rule to the intermediate model.

In UCON B and C models, the components defining the access conditions are divided over multiple elements of the UCON policy. Supporting functions such as \text{getPreOBL} are part of the decision logic. We need a single functional predicate without external functions in order to convert it to the Intermediate Policy Language. Therefore the references to supporting functions in an allowed rule need to be substituted by the definitions of the supporting functions. The \text{preFulfilled}() function is represented by \text{in PastUsage} in the condition.
OBS = S  
OBO = \{ high\_license\_agreement, low\_license\_agreement \}  
OB = \{ agree \}  
level : O \rightarrow \{ high, low \}  
ATT(O) : \{ level \}  
getPreOBL = \{  
( s, high\_license\_agreement, agree )  
  \text{ if level(o)='high'}  
( s, low\_license\_agreement, agree )  
  \text{ if level(o)='low'}  
\}  

allowed(s,o,r) \Rightarrow \text{preFulfilled(getPreOBL(s,o,r))}  

(a) UCON

"policy" : {  
"rule" : {  
"evaluationTime" : "before usage",  
"condition" : "  
( o. level == high \&\& !((s, high\_license\_agreement, agree) in PastUsage)) ||  
( o. level == low \&\& !((s, low\_license\_agreement, agree) in PastUsage))"  
  }  
}  

(b) Intermediate Language

Figure 5.1: Allowed Rule Conversion

Example 12 Consider the following UCON getPreOBL() and allowed() functions:
getPreOBL = \{  
( s, high\_license\_agreement, agree )  
  \text{ if level(o)='high'}  
( s, low\_license\_agreement, agree )  
  \text{ if level(o)='low'}  
\}  

allowed(s,o,r) \Rightarrow \text{preFulfilled(getPreOBL(s,o,r))}  

As a single allowed rule we write:
allowed(s,o,r) \Rightarrow ( \text{level(o)='high} \Rightarrow \text{preFulfilled((s, high\_license\_agreement, agree)))} \land  
( \text{level(o)='low} \Rightarrow \text{preFulfilled((s, low\_license\_agreement, agree)))})  

An UCON policy may consist of multiple allowed rules. This is the case when there are rules specifying explicit rights, or when there are rules from different UCON models. All allowed rules that specify the same right are to be combined as a conjunction (and-operator). In this context, ”the same right” may also refer to allowed rules that specify no right. The combining should be performed before converting the allowed rule to the Denied-notation. In case the update extension proposed in Section 3.1 is used, all allowed rules that specify the same right and the same set of updates are combined.

If an allowed rule specifies a right, it is encoded as an action in the Intermediate Policy Language. When there is no right specified, there is no action in the policy. An UCON allowed rule is evaluated before usage (at request time). This is encoded by an evaluationTime of before usage in the Intermediate Policy Language.

5.1.2 stopped rules

Recall the definition for a UCON stopped rule:
\[
\text{stopped}(s, o, r) \Leftrightarrow \neg \text{constraints} \quad (5.3)
\]

The constraints (a boolean expression) define when access has to be stopped. Since the decision to stop access follows from the constraints, no rewriting is required. The constraints from the UCON stopped rule are the condition in the intermediate model. Figure 5.2 shows a conversion from an UCON stopped rule to the intermediate model. UCON stopped rules are evaluated continuously. This is encoded in the evaluationTime in the intermediate model. Like allowed rules, stopped rules may encode a right, which is encoded as the Action in the Intermediate Model.
5.1. UCON TO INTERMEDIATE POLICY LANGUAGE

stopped(s, o, call) ⇐ (credit(s) < price(o))

(a) UCON

"policy" : {
  "action" : "call",
  "rule" : {
    "evaluationTime" : "ongoing access attribute change, s.credit",
    "condition" : "s.credit < o.price"
  }
}

(b) Intermediate Language

Figure 5.2: Stopped Rule Conversion

In UCON B and C models, the components defining the access conditions are divided over multiple elements of the UCON policy. Supporting functions such as *getOnOBL* are part of the decision logic. We need a single functional predicate without external functions in order to convert it to the Intermediate Policy Language. Therefore the references to supporting functions in an *stopped rule* need to be substituted by the definitions of the supporting functions. The *onFulfilled()* function is represented by *in ActiveUsage* in the *condition*.

An UCON policy may consist of multiple *stopped rules*. This is the case when there are rules specifying explicit rights, or when there are rules from different UCON models. All *stopped rules* that specify the same right are to be combined as a disjunction (*or*-operator). In this context, "the same right" may also refer to *stopped rules* that specify no right. In case the update extension proposed in Section 3.1 is used, all *stopped rules* that specify the same right and the same set of updates are combined.

If a *stopped rule* specifies a right, it is encoded as an *action* in the Intermediate Policy Language. When there is no right specified, there is no *action* in the *policy*. An UCON *stopped rule* is evaluated during the usage. This is encoded by an *evaluationTime* of *ongoing usage* in the Intermediate Policy Language. The *evaluationTime* is accompanied with information to specify when the request should be evaluated, based on attribute change or a timer.

5.1.3 Updates

UCON defines three update functions: *preUpdate*, *onUpdate* and *postUpdate*. The distinction between these update functions is the moment in time when the updates are to be applied. Figure 5.8 shows the conversion of a UCON Pre-update to the intermediate model.

Recall the definition of an extended update:

\[
*update* \text{(attribute, identifier)} : expression \tag{5.4}
\]

*update* represents either *preUpdate*, *onUpdate* or *postUpdate*. Attribute is the attribute to be updated, and expression is an expression built from attributes, constants and arithmetic operators, which is to be evaluated to obtain the value which is to be assigned to the attribute. We extend the update function definition with an *identifier*. Using the identifier, an update can be associated to a rule.

In the intermediate model, the moment in time when the update is to be applied is encoded as *executionTime*. The attribute to be updated, and the expression which determines the update value is encoded as *operation*.

5.1.4 Policies

We have discussed the conversion of rules and updates. The fragments of Intermediate Policy Language resulting from the procedures discussed so far are parts of a policy. So far, we have encountered the
policy where we defined a right from an UCON rule to become an action of a policy in the Intermediate Policy Language. All Intermediate Policy Language rules resulting from UCON rules with the same right are combined in a policy, in which that right specified as action. Rules with different rights are placed in different policies.

In the case the UCON update extension, proposed in Section 3.1 is used, all Intermediate Policy Language rules originating from UCON rules specifying the same right and updates are combined in a policy. In the UCON update extension, an update may be assigned to multiple rules. If this is the case, the update will be included in all resulting Intermediate Policy Language policies.

Example 13 Consider a policy that allows a subject to access an object a limited number of times. An object is only accessible when it has been read or written at most 10 times, regardless which subject performs the action. A subject is only allowed to access (read or write) an object at most 10 times, regardless of the object accessed.

Figure 5.4 shows this policy in UCON and in the Intermediate Policy Language. In the Intermediate Policy Language we see two policies: one for the action read, and one for the action write. The update for accessCount has been duplicated and appears in both policies.

5.2 Intermediate Policy Language to XACML

The second step in the process to convert a UCON Policy to XACML is mapping from the Intermediate Policy Language to XACML. The Intermediate Policy Language is designed to fit XACML. Policies, rules, actions and conditions are mapped (nearly) one-to-one to their XACML counterparts. Updates and re-evaluation triggers are mapped as Obligations in XACML.

5.2.1 Rules

A rule from the Intermediate Model is mapped to a <Rule> in XACML with an Effect of deny. A XACML <Rule> should have an unique RuleId, but other then uniqueness, there are no other requirements for its value. A rule in the Intermediate Model may contain a condition: this is mapped to a <Condition> in XACML. Figure 5.5 shows an example conversion for a condition. The conversion process consists of converting the syntax and applying certain XACML specific functions to adjust for data types, such as a one-and-only function. When an attribute is referenced in XACML, using <SubjectAttributeDesignator> or <ResourceAttributeDesignator>, XACML returns a bag containing the attribute. Operators to compare attributes values require the attributes without bags. We use the one-and-only function takes the attribute out of the bag. In respect to data types, we use the one-and-only function to convert, for example, bag of strings to a string. The one-and-only function requires there is only one item in a bag, which is always the case on our setting.

A rule in the Intermediate Model may contain an Action. An Action is encoded as an <Action> element in XACML, which is contained in a <Target> element. In contrast to <Condition> elements, accessing
5.2. INTERMEDIATE POLICY LANGUAGE TO XACML

attributes in a <Target> does not require data type adjustment.

A rule in the Intermediate Model contains an EvaluationTime. If the EvaluationTime is before usage, no other action is required. If the EvaluationTime is ongoing, it is required to specify when the policy has to be re-evaluated. Figure 5.7 shows an example for a re-evaluation trigger on a change of the subject attribute credit. For each mutable attribute referred in the condition, a re-evaluation trigger has to be created. A re-evaluation trigger is encoded as a <Obligation> with ObligationId trigger. The Obligation contains an <AttributeAssignment> with an AttributeId of subjectAttribute or resourceAttribute, to identify a subject or resource attribute. The value of the attribute is the name of the attribute that triggers the re-evaluation. An AttributeId of triggerInterval indicates a re-evaluation at a certain timer interval, and triggerTime indicates a re-evaluation at a certain time.

5.2.2 Updates

We use an XACML Obligation to encode updates. Updates have an executionTime that is either before usage, ongoing or after usage. In XACML, this is encoded with an ObligationId of preUpdate, onUpdate or postUpdate respectively. The attribute to be updated is encoded in an <AttributeAssignment> with AttributeId subjectAttribute or resourceAttribute to encode subject attributes or resource attributes. The expression which evaluates to the value that the attribute is to be updated to is encoded as an embedded snippet of XACML code. This XML code exists as the value of an <AttributeAssignment> with AttributeId updateValue. In order to increase readability the snippet of XACML code is embedded as character data (<![CDATA[ ... ]]>). Like in the <Condition> element, the embedded XACML snippet requires one-and-only calls to correct the data type of attribute values.

Ongoing updates contain one more attribute to indicate when the ongoing attribute has to be performed. Possible update activation triggers are timer based or attribute change based. A timer based trigger is encoded by an attributeId of updateInterval. The value of this attribute is an time interval encoded in ISO 8601 notation. This is the notation defined by XACML to indicate a time duration. Figure 5.9 shows an update interval timer of 10 seconds. Attribute change triggered updates are encoded by an attributeId of updateResourceTrigger or updateSubjectTrigger, to indicate a trigger by a resource or subject attribute update. The value of this attribute is the name of the attribute whose change triggers the update activation. Note that the update is triggered by attribute changes of the resource or subject referred in the request that matches the rule associated with the update. Figure 5.10 shows an update triggered by the update of the subject attribute credit. When using attribute change triggered updates it is essential to prevent circular updates. An attribute that activates an update should not be affected by the update it activates. This includes chains of updates activating each other by updating attribute values. Policies including circular updates are considered invalid and are not supported.

In the case an update operation refers to an attribute not mentioned in the condition of the containing policy, an attribute guard is required in the resulting XACML Policy. The guard consists of a <Subject> or <Resource> in the <Target> to test the presence of the referred attribute. In this way, a policy will be rendered not applicable if an attribute required to perform the update is missing, in stead of a failing update.

An extension not provided by UCON are conditional updates. We provide support for conditions in updates. A condition is a functional predicate, like we use for rules. When the condition evaluates to true, the update is performed. We encode a condition as an <AttributeAssignment> with AttributeId of condition. The condition contains a snippet of XACML code, and is, just like the updateValue embedded as character data (<![CDATA[ ... ]]>).

5.2.3 Policies

Table 5.1 summarises the conversion from the Intermediate Policy Language to XACML. The left column indicates an element in the Intermediate Policy Language, the right column indicates the corresponding element in XACML. As discussed in previous Sections, the elements policy, action, rule and condition map (nearly) one-to-one to their XACML counterparts. Updates and continuous evaluation are encoded as XACML <Obligation>. The converted elements are assembled in a XACML <Policy> following the same structure as observed in the Intermediate Policy Language policy. Apart from the converted
CHAPTER 5. MAPPING FROM UCON TO XACML

<table>
<thead>
<tr>
<th>Intermediate Policy Language</th>
<th>XACML</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy</td>
<td>&lt;Policy&gt;</td>
</tr>
<tr>
<td>action</td>
<td>&lt;Action&gt;</td>
</tr>
<tr>
<td>rule</td>
<td>&lt;Rule&gt;</td>
</tr>
<tr>
<td>condition</td>
<td>&lt;Condition&gt;</td>
</tr>
<tr>
<td>EvaluationTime = continuous</td>
<td>&lt;Obligation&gt;</td>
</tr>
<tr>
<td>Update</td>
<td>&lt;Obligation&gt;</td>
</tr>
</tbody>
</table>

Table 5.1: Mapping Intermediate Policy Language to XACML

rules, a default rule is added to the policy. The default rule has an Effect of Permit and has no constraints.

The <Policy> uses the Deny Overrides rule combining algorithm. This combining algorithm is used because it resembles the behaviour of UCON defining the Deny-space. In the absence of applicable rules, access is to be granted by default. This behaviour can be reached by adding a default rule, without conditions, with an Effect of Permit. If any of the other rules are applicable, their evaluation result is Deny and the Deny Overrides rule combining algorithm will evaluate to Deny, otherwise, the default rule is the only applicable rule, resulting in Permit.

Domain Knowledge Policies

Updates are encoded as an <Obligation>, which are part of a a <Policy>. Therefore they are only released in the context of a request that is granted on behalf of the containing <Policy>. If we would like to define attribute change triggered updates which apply to all subjects and resources, we cannot do this by defining it as part of an access policy.

In order to enable attribute change triggered updates, which apply for all subjects and attributes, we introduce Domain Knowledge Policies. Domain Knowledge Policies are policies, encoded in the Intermediate Policy Language or XACML, that function to release attribute change triggered updates that apply to any subject or resource. A Domain Knowledge Policy is not used to derive an access decision. The resource or subject attribute to trigger update activation is encoded in the same way as with regular updates. Conditions can be used to limit the applicability of domain knowledge updates. In combination with multiple global applicable updates a custom attribute mapping can be defined.

Example 14 Consider a pre-paid environment. In this environment, colour codes are assigned to subjects depending on their amount of credit. When a subject has less then 25 credits, red will be assigned. When a subject has between 25 and 50 credits, yellow will be assigned. When a subject has over 50 credits, green will be assigned.

"policy": {
  "update": {
    "condition": "s.credit < 25",
    "operation": "s.colour = red",
    "ExecutionTime": "ongoing attribute change, s.credit"
  },
  "update": {
    "condition": "s.credit > 25 & s.credit < 50",
    "operation": "s.colour = yellow",
    "ExecutionTime": "ongoing attribute change, s.credit"
  },
  "update": {
    "condition": "s.credit > 50",
    "operation": "s.colour = green",
    "ExecutionTime": "ongoing attribute change, s.credit"
  }
}
The XACML encoding of this policy is listed in Appendix B.1.
allowed(s, o, read) ⇒ readCount(o) ≤ 10 ∧ accessCount(s) ≤ 10 : u1, u3
allowed(s, o, write) ⇒ writeCount(o) ≤ 10 ∧ accessCount(s) ≤ 10 : u2, u3
postupdate(readCount(o), u1) : readCount(o) + 1
postupdate(writeCount(o), u2) : writeCount(o) + 1
postupdate(accessCount(s), u3) : accessCount(s) + 1

(a) UCON

```
"policy": {
  "action": "read",
  "rule": {
    "evaluationTime": "before usage",
    "condition": "o.readCount > 10 || s.accessCount > 10"
  },
  "update": {
    "executionTime": "after usage",
    "operation": "o.readCount = o.readCount + 1"
  },
  "update": {
    "executionTime": "after usage",
    "operation": "o.accessCount = o.accessCount + 1"
  }
},
"policy": {
  "action": "write",
  "rule": {
    "evaluationTime": "before usage",
    "condition": "o.writeCount > 10 || s.accessCount > 10"
  },
  "update": {
    "executionTime": "after usage",
    "operation": "o.writeCount = o.writeCount + 1"
  },
  "update": {
    "executionTime": "after usage",
    "operation": "o.accessCount = o.accessCount + 1"
  }
}
```

(b) Intermediate Language

Figure 5.4: Policy Combination
5.2. **INTERMEDIATE POLICY LANGUAGE TO XACML**

"condition" : "s.credit < o.price"

(a) Intermediate Language

```xml
<Rule RuleId="example" Effect="Deny">
  <Condition>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
        <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer" AttributeId="credit" />
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
        <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer" AttributeId="price" />
      </Apply>
    </Apply>
  </Condition>
</Rule>
```

(b) XACML

Figure 5.5: Condition Conversion

"action" : "call"

(a) Intermediate Language

```xml
<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">call</AttributeValue>
    <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" />
  </ActionMatch>
</Action>
```

(b) XACML

Figure 5.6: Action Conversion

```xml
<Obligation ObligationId="trigger" FulfillOn="Permit">
  <AttributeAssignment AttributeId="subjectAttribute" DataType="http://www.w3.org/2001/XMLSchema#string">
    credit
  </AttributeAssignment>
</Obligation>
```

Figure 5.7: Encoding of the re-evaluation trigger in XACML
"update" : {
    executionTime : "pre",
    operation      : "s.credit = s.credit - o.value"
}

(a) Intermediate Language

```xml
<Obligation ObligationId="preUpdate" FulfillOn="Permit">
  <AttributeAssignment AttributeId="subjectAttribute">
    <AttributeAssignment AttributeId="updateValue">
      <![CDATA[
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-subtract"
          xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
            <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer" AttributeId="credit" />
          </Apply>
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
            <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer" AttributeId="price" />
          </Apply>
        </Apply>
      ]]>
    </AttributeAssignment>
  </AttributeAssignment>
</Obligation>
```

(b) XACML

Figure 5.8: Update Conversion

```xml
ATTRIBUTE_ASSIGNMENT AttributeId="updateInterval" DataType="urn:oasis:names:tc:xacml:2.0:data-type:dayTimeDuration">PT10s
</ATTRIBUTE_ASSIGNMENT>
```

Figure 5.9: Encoding of timer based update activation in XACML

```xml
ATTRIBUTE_ASSIGNMENT AttributeId="updateSubjectTrigger" DataType="http://www.w3.org/2001/XMLSchema#string">credit
</ATTRIBUTE_ASSIGNMENT>
```

Figure 5.10: Encoding of time based update activation in XACML
Chapter 6

Implementation

This chapter discusses the implementation of UconXACML. Section 6.1 discusses HerasAF XACML Core, the library which implements XACML policy evaluation used in this project. Section 6.2 provides an overview of UconXACML, our implementation of UCON in XACML, capable of evaluating and enforcing policies using the extensions discussed in Section 5.2. A more detailed view is provided in Section 6.3 which discusses the classes of UconXACML.

6.1 HerasAF XACML Core

UconXACML is based upon HerasAF XACML Core. HerasAF XACML Core is a XACML 2.0 implementation written in Java. It consists of a XACML policy evaluation engine and a number of supporting components, such as a component called SimplePDPFactory, which provides automatic initialisation of all needed components to perform an evaluation. Most notably is the SimplePDP component. This component represents a XACML Policy Decision Point (See Figure 2.5). It evaluates XACML requests against XACML policies. Another component provided by HerasAF XACML Core is called PolicyRepository. This component allows Evaluatables to be deployed. An Evaluatable can be either a <Policy> or a <PolicySet>.

One family of components are the Marshallers. Marshallers convert from plain text XACML to HerasAF objects and from HerasAF objects to plain text XACML. The Marshallers provide work on Evaluatables, Requests and Responses. The HerasAF XACML Core provides an interface to implement a PIP, which makes it possible to retrieve attribute values from external sources.

6.2 UconXACML Overview

UconXACML is an implementation of XACML extended with UCON features such as continuous evaluation and mutable attributes. This section shows the steps of the evaluation process of requests in UconXACML. The components referenced in this Section are elaborated in Section 6.3.

Figure 6.1 shows the data flow for a request in our implementation. The data flow is a standard XACML data flow extended with steps performed by the obligation services. In contrast to the XACML Reference data flow diagram (Figure 2.5), it does not include a Context Handler. A Context Handler is a component to translate between native and XACML formats. In HerasAF, the Context Handler functionality is integrated into the SimplePDP component.

1. The RequestHandler forms a HerasAF request object and provides it to the UconPEP.
2. The UconPEP forwards the request to the SimplePDP.
3. The SimplePDP requests attribute values from the UconPIP.
4. The UconPIP retrieves attributes values from the UconDB.
5. The UconPIP returns attribute values to the SimplePDP.
6. The SimplePDP reaches a decision and sends it to the UconPEP.
7. The UconPEP forwards obligations to UconOS*.
8. (a) The UconOS UpdateParser registers an Update at the RequestHandler.
     (b) The UconOS TriggerParser registers a trigger at the UconPIP.
9. The UconPEP forwards the decision to the RequestHandler.

### 6.3 UconXACML Classes

UconXACML is an implementation of UCON features around the HerasAF XACML Core. It uses the SimplePDP provided by HerasAF XACML Core to evaluate XACML requests. UCON features are implemented through Obligation Services and the Policy Enforcement Point.

**UconPEP**

UconPEP is the central class in the UCON XACML implementation. Figure 6.3 shows the UML diagram for the UconPEP class. The class is responsible for the initialisation of all other components of the UconXACML implementation, such as pdp, pip and the obligation services (oss). The function `evaluateRequest()` (re-)evaluates requests provided by RequestHandlers.

**RequestHandler**

RequestHandler is an interface. Figure 6.4 shows the UML diagram for the RequestHandler interface. Realisations of this interface form XACML requests and provide the connection between the system or resources to be protected and the UconXACML implementation. This is done by implementing functions like `onAllow()`, `onDeny()`, and `onStopped()`, which are called when the corresponding request is allowed, denied, or stopped respectively.

Furthermore, a RequestHandler maintains lists of updates. A list of pre-updates is maintained. During the evaluation pre-updates can be enqueued. When the `onAllow()` function is called, the updates in the pre-update list are performed. Similarly, a list of post-updates is maintained. When the function `onStopped()` is called, the updates in this list are performed. Furthermore, a list of timer based ongoing updates is maintained. The updates in this list are performed at regular intervals from the moment `onAllow()` is called until `onStopped()` is called.
6.3. UCONXACML CLASSES

UconPIP

UconPIP is a realisation of the PIP interface provided by the HerasAF XACML Core package. Figure 6.5 shows the UML diagram for the UconPIP class. The UconPIP provides the functions declared in the PIP interface to retrieve attribute values. These functions are `fetchSubjectAttributes()` and `fetchResourceAttributes()`. Additionally it contains functions to update attribute values: `updateSubjectAttribute()` and `updateResourceAttribute()`. Furthermore, the UconPIP keeps a list of triggers. A trigger may cause a re-evaluation of a RequestHandler or perform an update, when a certain attribute is changed.

UconOS

UconOS is an interface for Obligation Services. Figure 6.6 shows the UML diagram for the UconOS interface. It consists of one function `HandleObligations()`. When an obligation is passed to this function, it returns whether the obligation has been handled. Realisations for this class are UconOSUpdateParser and UconOSTriggerParser. The UconOSUpdateParser parses update obligations are adds the updates to their corresponding lists at the RequestHandler. The UconOSTriggerParser parses triggers and sets re-evaluation triggers for the corresponding RequestHandler.

Update

The Update class represents updates. Figure 6.7 shows the UML diagram for the Update Class. An update stores the attribute name and the value to be updated as a snippet of XACML code. Distinction between Subject and Resource attributes is made by inheriting from the Update class. There are two classes inheriting this class, SubjectUpdate and ResourceUpdate. The inheriting classes implement an `apply()` function which calculates the value to be updated, using the UpdateMarshaller class, and updates the attribute with the new value by a call to `updateSubjectAttribute()` or `updateResourceAttribute()` of the UconPIP.
Trigger

The Update class represents triggers. Figure 6.3 shows the UML diagram for the Trigger Class. A trigger stores an attribute name (name) and an identification (id) of the subject or resource. The distinction between subject or resource attribute is made by inheritance. There are two classes inheriting from this class, SubjectTrigger and ResourceTrigger. A trigger may contain either a RequestHandler or an Update. Triggers are stored in a list at UconPAP. Whenever an attribute is changed, the list of triggers is checked. If the subject or resource and attribute name match, the RequestHandler is enqueued for re-evaluation, or the Update is performed.

UconPAP

UconPAP is a class that is responsible for deploying and undeploying policies. Figure 6.4 shows the UML diagram for the UconPAP class. UconPAP implements the functions deployPolicy() and undeployPolicy(). Deploying a policy means loading it from a file and making it available such that it is used when evaluating a policy. Undeploying means making the policy no longer available, so requests are no longer evaluated against the policy.

Extractor

Extractor is a helper class that extracts subject, resource and action identifiers from HerasAF requests. Figure 6.5 shows the UML diagram for the Extractor class. it implements the functions extractResourceId(), extractSubjectId() and extractActionId(). which extract the subject-id, resource-id and action-id from a request in HerasAF’s native format, and returns it as a string. It is used by Updates and Triggers to identify the subject and resource.

UpdateMarshaller

The UpdateMarshaller class is custom Marshaller class. Figure 6.6 shows the UML diagram for the UpdateMarshaller class. The Marshaller classes provided by the HerasAF XACML Core package can either convert plain text XACML evaluatables, requests and responses to HerasAF objects. The UpdateMarshaller
6.4. EVALUATION REQUEST CODE FLOW

Figures 6.5 and 6.6: UconPIP Class and UconOS Interface

is a Marshaller that can generate objects from any snippet of plain text XACML, rather than only the object types supported by the provided marshellers. It is used in updates. Updates encode the expression to determine the new value as a snippet of XACML code. The UpdateMarshaller parses and evaluates this code and returns the resulting object, which can be for example an integer value or a string.

6.4 Evaluation Request Code Flow

In our implementation, a request is initiated by creating a RequestHandler object, and specify subject, resource and action parameters. Figure 6.12 shows the sequence diagram for a request that has an evaluation result without obligations. We call the evaluateRequest() function of the UconPEP class with the RequestHandler object. evaluateRequest() calls the SimplePDP with a HerasAF request generated by the RequestHandler. The SimplePDP evaluates the request and returns the result to the UconPEP. Eventual obligations contained in the evaluation result are passed to the obligation services. If the evaluation result is to allow access, the UconPEP calls onAllowed() of the RequestHandler. In case access is denied, the UconPEP calls onDenied() at the RequestHandler.

Figure 6.13 shows the sequence diagram for a result containing a trigger obligation. This diagram contains a part of the calling sequence. To put the diagram in context, we show the evaluation request to the PDP, as seen in Figure 6.12. If the result from the PDP contains a trigger obligation, UconPEP calls handleObligations() function of the UconTriggerParser. The UconTriggerParser parses the obligation and calls addReEvaluationTrigger() of the UconPIP, enlisting the trigger defined in the obligation.

Figure 6.14 shows the sequence diagram for result containing a update obligation (either preUpdate, onUpdate or postUpdate). As with the diagram for triggers, this is only a partial diagram, showing the flow starting at the evaluation request made to the PDP. If the result from the PDP contains an update obligation, UconPEP calls handleObligations() function of the UconUpdateParser. UconUpdateParser parses the update. In case the update is a preUpdate, postUpdate or a timer based onUpdate, UconUpdateParser enlists it at the RequestHandler using a call to either addPreUpdate(), addOnUpdate() or addPostUpdate(). This adds the update to the pre-, ongoing- or post-update list. If the update is a preUpdate with an attribute change trigger, the update is associated with a trigger object and enlisted at the UconPIP using an addUpdateTrigger() call.
CHAPTER 6. IMPLEMENTATION

When the UconPEP retrieves the evaluation result from the PDP, and obligations are handled as discussed above. After the obligations are handled UconPEP calls the `onAllowed` function of the RequestHandler. Figure 6.15 shows a sequence diagram for calling pre-updates. When `onAllowed()` is called, the RequestHandler calls `apply()` of any updates in the pre-update list. Similarly, at the end of the usage, when `onStopped()` is called, the RequestHandler calls `apply()` of any updates in the post-update list. When `onAllowed()` is called timers are activated for all updates in the ongoing-update list. When `onStopped()` is called, all timers for ongoing updates are stopped.

Figure 6.16 shows a sequence diagram for applying updates and calling for re-evaluation. When the `apply()` function of an update is called, the update value is calculated. Attribute values required to calculate the update value are fetched from the UconPIP. The Update calls either `updateSubjectAttribute` or `updateResourceAttribute` at the UconPIP to update the attribute with the new value. If the updated attribute triggers a re-evaluation (a trigger added by `addReEvaluationTrigger()` the UconPIP calls `addToReEvaluationQueue()` at UconPEP. When the RequestHandler has applied all updates, it calls `startReEvaluation` at the UconPEP. The re-evaluation is started only after all updates have been applied, since unprocessed updates may change attributes relevant to the enqueued re-evaluation. If the updated attribute triggers an attribute update (a trigger added by `addUpdateTrigger()` it is performed immediately.
6.4. EVALUATION REQUEST CODE FLOW

Figure 6.12: Sequence Diagram for simple Request

Figure 6.13: Sequence Diagram for registering re-evaluation trigger

Figure 6.14: Sequence Diagram for registering updates
CHAPTER 6. IMPLEMENTATION

Figure 6.15: Sequence Diagram for calling pre-updates

Figure 6.16: Sequence Diagram for applying updates and re-evaluation
Chapter 7

Validation

In this chapter validate our UconXACML implementation. We present a number of scenarios demonstrating UCON features. In each of the presented scenarios we show the evaluation of a series of requests by our UconXACML implementation, demonstrating the evaluation results are as according to the expected results.

7.1 Scenario Description

UCON Authorisations and updates

In the first scenario we demonstrate the implementation of the UCON Authorisations model. In particular, we show ongoing evaluation of policies, post- and ongoing updates. We show a request is denied when initially the conditions do not hold, that a previously granted right is revoked when the conditions no longer hold. We show a re-evaluation only happens for requests that might be affected by an attribute update. Furthermore, we show timer based ongoing updates being performed at regular intervals, and post updates being performed at the end of a usage.

Consider a phone company. The phone company provides pre-paid services to its customers. The service provided by the phone company consists of enabling their customers to make phone calls. In order to make a phone call, the customer calls a phone number. The phone company keeps records of costs associated to phone numbers. A customer has a certain credit. The costs for making a phone call are paid from this credit. A phone number is not required to have costs associated. There are service numbers at the phone company for their customers to top up more credit. When modelling this company, customers are represented by subject, and phone numbers are represented by objects. Making a phone call is represented by the taking the action call.

An object representing a paid phone number has an attribute price assigned. The value of this attribute is the amount of credits per tick to be paid. An object representing a credit top up phone number has an attribute topupValue assigned. The value of this attribute is the amount of credits the subject’s credit is increased by upon completion of the call. Furthermore, an object may contain a minimalCredit attribute, specifying a minimal required amount of credit to call the phone number.

Subjects are only allowed to make calls to paid phone numbers when their credit is at least the price of one tick. During a phone call, every 10 seconds a subject’s credit will be decreased by the object’s price. In the situation when this results in a credit value lower then the price of a tick, the phone call will be terminated.

In the following scenarios we assume two subjects: Bob and Jane. In the initial scenario Bob has 0 credits and Jane has 50 credits. There is a paid phone number 1, which has a price of 10 credits per tick. There is a top up phone number 8, which has topupValue of 50 credits per call. Furthermore, there is a phone number 4, which has a minimalCredit value of 40.

The first request demonstrates that a request will be denied if initially the conditions do not hold. Recall that in the initial scenario, Bob has no credits. When Bob attempts to call 1, this request will be denied
due insufficient credit.

Next, we will demonstrate post-updates. Recall that the topping up of credit is performed after finishing the top-up call. Bob calls 8 to top up his credit. After finishing the call, Bob’s credit will be updated to 50.

The following request shows ongoing updates and ongoing evaluation. When Bob calls 1 again, the request will be granted. The initial conditions, a greater credit then the price per tick, hold. Bob’s credit will be decreased with regular intervals (ongoing updates). When Bob’s credit is depleted, the usage will be stopped. (continuous evaluation).

We show re-evaluations do not require an update to caused by the same usage. For this request, we assume Bob has 50 credits. Bob calls 4 and 1 simultaneously. We will see the call to 4 will be re-evaluation when an update is caused by the call to 1.

To demonstrate only relevant requests are re-evaluated. We assume both Bob and Jane have 50 credits. Jane calls 4 and Bob calls 1 simultaneously. We see Jane’s request is not being re-evaluation when Bob’s credit is updated.

**UCON Obligations**

In the second scenario we demonstrate the implementation of the UCON Obligations model. In particular, we demonstrate the Pre Obligations model and the Ongoing Obligations model.

Consider a free computer game. The game is advertisement sponsored, therefore the user is required to watch advertisement while playing the game. Prior to downloading the game, the user must accept the license agreement. In this scenario we consider only one subject: John.

We demonstrate a download request is denied before accepting the license, and granted after accepting the license. We show the playing of the game is denied when the user does not watch the advertisement. We show that, when the user stops watching the advertisement while playing the game, the game is stopped as well.

**UCON Conditions**

In the third scenario we demonstrate the implementation of the UCON Conditions model. In the UCON conditions model, access decisions are made based on environmental conditions. In XACML, one of the attributes available in the environment is the current time.

Consider a soup kitchen. The soup kitchen needs to know for how many people they have to cook in advance, as they need to buy fresh ingredients and prepare the meal in time. Therefore one has to order between 14:00 and 15:00. We demonstrate a request within this time interval is accepted, and a request outside this time interval is denied.

**7.2 Policy Specification**

In the scenario description, we have defined three possible attributes objects can have: price, topupValue and minimalCredit. The effects of these attributes only apply when the attributes are available. Therefore, we define three UCON policies, each of them specifying one of the attributes and defining the rules describing the behaviour as sketched in the scenario description.

Calling a phone number with a price attribute is encoded in the policy shown in Figure 7.1. The XACML encoding of this policy is listed in Appendix B.2. The UCON Policy specifies the subject should have a credit attribute and the object should have a price attribute. This requirement renders this policy not applicable when the requested object has no price attribute. In the translated policies, the presence of a price attribute is implicit. A missing attribute in XACML results in an evaluation to Indeterminate, which has the desired effect: no access decision is made and no obligations are released.

The constraints stated in the UCON stopped rule define that access has to be stopped when the subject’s credit is less then the object’s value. This corresponds with the condition in the Intermediate Policy
7.2. POLICY SPECIFICATION

credit = S \rightarrow \mathbb{N}
price = O \rightarrow \mathbb{N}
ATT(S) = \{ credit \}
ATT(O) = \{ price \}
allowed(s,o,r) ⇒ true
stopped(s,o,call) ⇐ credit(s) < price(o) : u1
onUpdate(credit(s),u1) : credit(s) = credit(s) - price(o)

(a) UCON

"policy" = {
  "action" = "call",
  "rule" = {
    "evaluationTime" = "ongoing attribute change, s.credit",
    "condition" = "s.credit < o.price",
    "update" = {
      "executionTime" = "ongoing timer, PT10s",
      "operation" = "s.credit = s.credit - o.price"
    }
  }
}

(b) Intermediate Language

Figure 7.1: UCON Ongoing Authorisations and Ongoing Updates: phone call policy

Language. A stopped rule corresponds with ongoing evaluation. The re-evaluation triggers are left implicit in the UCON policy. The constraints of the stopped rule refer to two attributes, the subject’s credit and the object’s price. The object’s price is immutable, while the subject’s credit is mutable, therefore a re-evaluation should be triggered by a change of the subject’s credit. This is encoded as the evaluationTime in the Intermediate Policy Language. The right defined in the UCON stopped rule are encoded as an action in the Intermediate Policy Language.

The UCON onUpdate states that the subject’s credit should be decreased by the object’s value during the usage. In the Intermediate Policy Language the credit change is encoded as operation. UCON only specifies the credit change should happen during the usage, but does not otherwise specify when. In the Intermediate Policy Language we can be more specific. We define a a timer based update activation with an interval of 10 seconds. This is encoded as executionTime in the Intermediate Policy Language.

The UCON allowed rule always permits access, and corresponds with the default rule. It corresponds with the absence of allowed rules and as such, there is no corresponding element in the Intermediate Policy Language.

Calling a phone number with a topupValue attribute is encoded in the policy shown in Figure 7.2. The XACML encoding of this policy is listed in Appendix B.3. In this policy, an object attribute topupValue is required to be present to make the policy applicable.

The allowed rule in the UCON policy states that access is always allowed. In the Intermediate Policy Language this is encoded with the absence of a condition. The postUpdate states that the subject’s credit is to be increased by the object’s topupValue after the usage. The credit change is encoded as operation in the Intermediate Policy Language. The execution of the update after usage is encoded as executionTime in the Intermediate Policy Language.

Since the object attribute topupValue is not present in the condition, the resulting XACML Policy requires an attribute guard, in the form of a <resource> in the <target>. This will render the policy not applicable when no topupValue attribute is present in the object.

The policy shown in Figure 7.3 allows access as long as the subject has at least the amount of credits specified by the object’s minimalCredit attribute. The XACML encoding of this policy is listed in Appendix B.4. This policy is equal to the paid phone call policy, except there is no updating of the
credit = S → N

topupValue = O → N

ATT(S) = { credit }

ATT(O) = { topupValue }

allowed(s,o,call) ⇒ true : u2

postUpdate(credit(s), u2) : credit(s) = credit(s) + topupValue(o)

(a) UCON

"policy" = {
    "action" = "call",
    "rule" = {
        "evaluationTime" = "before usage",
        "update" = {
            "executionTime" = "after usage",
            "operation" = "s.credit = s.credit + o.topupValue"
        }
    }
}

(b) Intermediate Language

Figure 7.2: UCON Post Updates: top-up call policy

subject’s credit.

The policies we discuss next are in the context of the free game, demonstrating UCON Obligations. The policy shown in Figure 7.4 shows a policy encoding the user must accept the license before he is allowed to download the game. The XACML encoding of this policy is listed in Appendix B.5. In UCON, we see the use of the `preFulfilled()` function with as parameter a triple of subject, license and right. The subject equals to the requesting subject. The subject, object and right in the `preFulfilled()` must be the subject, object and right of a usage that took place is the past. In the Intermediate Policy Language, this is encoded as a set inclusion test on the set PastUsage. The UCON Policy lists the sets OBS, OBO and OB to indicate what subject, objects and rights may be used in calls to `preFulfilled()`. This information is not encoded in the Intermediate Policy Language.

The policy shown in Figure 7.5 shows a policy encoding the user must watch advertisement while playing the game. The XACML encoding of this policy is listed in Appendix B.6. In UCON, this is encoded as a call to the `onFulfilled()` function. Like the `preFulfilled()` function, it takes a subject, object and right. Additionally, it takes a fourth parameter indicating when the obligation must be fulfilled. The subject, object and right in the `onFulfilled()` call must be the subject, object and right of a usage currently active. In the Intermediate Policy Language this is represented by a set inclusion test in the ActiveUsage set. The fourth parameter of the `onFulfilled()` call is a time parameter. In the Intermediate Policy Language this is represented by the `evaluationTime`.

The policy shown in Figure 7.6 shows a limitation of the time order requests can be made. The XACML encoding of this policy is listed in Appendix B.7. This is a policy in the UCON Pre Conditions model. The constructs required for this policy are readily available in XACML.

7.3 Policy Evaluation

For evaluation purposes, a GUI front-end to the UconXACML has been created. The front-end allows making requests with specified subject, resource and action ids. Once a request is made, the evaluation result is displayed along with a log showing details about the evaluation process. The policies presented in Section 7.2 have been loaded into this GUI front-end. Recall the definition of system state used in the Intermediate Policy Language evaluation discussion, a short-hand notation for the set of subject and
7.3. Policy Evaluation

\begin{align*}
\text{credit} &= S \rightarrow \mathbb{N} \\
\text{ATT(S)} &= \{ \text{credit} \} \\
\text{allowed}(s,o,r) &\Rightarrow \text{true} \\
\text{stopped}(s,o,\text{call}) &\Leftarrow \text{credit}(s) \leq \text{minimalCredit}(o)
\end{align*}

(a) UCON

"policy" {
  "action" = "call",
  "rule" = {
    "evaluationTime" = "ongoing attribute change, s.credit",
    "condition" = "s.credit \leq o.minimalCredit"
  }
}

(b) Intermediate Language

Figure 7.3: UCON Ongoing Authorisations: Minimal credit required policy

OBS = S
OBO = \{license\}
OB = \{accept\}
\text{allowed}(s, o, \text{download}) \Rightarrow \text{preFulfilled}(s,\text{license},\text{accept})

(a) UCON

"policy" {
  "action" = "download",
  "rule" = {
    "evaluationTime" = "before usage",
    "condition" = "!(s,\text{license},\text{accept}) \in \text{PastUsage})",
  }
}

(b) Intermediate Language

Figure 7.4: UCON Pre Obligations: Must accept license before downloading

resource attribute values. The initial system state is shown in Table 7.1.

Requests are represented as a tuple (subject, resource, action), analogous to the notation used in UCON, where rules are in the form allowed(s,o,r) and stopped(s,o,r). Following XACML specifications, subjects, resources and actions are identified by their subjectId, resourceId and actionId attributes.

Figure 7.7 shows the evaluation output of the GUI front-end for the following three requests discussed. Note that the oldest request is displayed at the bottom. In the initial state, a request (bob, 1, call) is denied. Resource 1 has a price attribute of 10. The credit attribute of subject Bob is 0. Therefore, this request will be denied, since 0 < 10 evaluates to true.

The next request is (bob, 8, call). This request is granted. Resource 8 has to price or minimalCredit attribute, therefore neither the paidCall nor the minimalCredit policies apply. Resource 8 has a topupValue attribute, which means the topupCall policy (Figure 7.2) is applicable. Since this policy has no condition, it will always be granted when applicable. This policy contains a post-update, which increases the subjects credit with the value specified at the resource attribute topupValue. In this case, Bob's will be increased to 50. Table 7.2 represents the state of the system after the top-up call has finished.

When we make the request (bob, 1, call) again, it will be granted, since 50 < 10 evaluates to false. When
OBS = S
OBO = {advertisement}
OB = {watch}

\[\text{stopped}(s, o, \text{play}) \iff \neg \text{onFulfilled}(s, \text{advertisement}, \text{watch}, 10s)\]

(a) UCON

"policy" {
  "action" = "download",
  "rule" = {
    "evaluationTime" = "ongoing timer, 10s",
    "condition" = "! ((s,advertisement,watch) in ActiveUsage)"
  }
}

(b) Intermediate Language

Figure 7.5: UCON Ongoing Obligations: Keep watching advertisements policy

\[\text{allowed}(s, o, \text{order}) \Rightarrow \text{currentTime} > 14:00 \land \text{currentTime} < 15:00\]

(a) UCON

"policy" {
  "action" = "order",
  "rule" = {
    "evaluationTime" = "before usage",
    "condition" = "! (e.currentTime < 14:00 \lor e.currentTime > 15:00)"
  }
}

(b) Intermediate Language

Figure 7.6: UCON Pre Conditions: time limitation policy

this request granted, it releases an ongoing update. At 10 second intervals, Bob’s credit is decreased by
the resource’s price. As Bob has 50 credits, and the credit is decreased by 10 every 10 seconds, Bob’s
credit is depleted after 50 seconds. When Bob’s credit reaches 0, the request is stopped.

Figure 7.8 shows the evaluation output of the GUI front-end for the next two requests that will be
discussed. We assume the system is in the state displayed in Table 7.2 To demonstrate a re-evaluation can
be triggered by an update induced by a different usage, we send \((\text{bob}, 4, \text{call})\) and \((\text{bob}, 1, \text{call})\). Figure 7.8a
shows the evaluation output for these requests. We notice the usage for \((\text{bob}, 4, \text{call})\) will be stopped once
the usage \((\text{bob}, 1, \text{call})\) has updated Bob’s credit to 30. To demonstrate a usage is only re-evaluated when
relevant attributes are changed, we send \((\text{jane}, 4, \text{call})\) and \((\text{bob}, 1, \text{call})\). Figure 7.8b shows the evaluation
output for these requests. We notice the usage for \((\text{jane}, 4, \text{call})\) is not being re-evaluated.

To demonstrate the capabilities of access request depending on previous or current usage, we consider
the setting of a free game. Figure 7.9 shows the evaluation output of the GUI front-end for the following
three requests discussed. When John requests to download the game, by sending a request \((\text{john}, \text{game}, \text{download})\) this request is denied, since there is no \((\text{john}, \text{license}, \text{accept})\) in the PastUsage available.
John sends a request \((\text{john}, \text{license}, \text{accept})\) and finished this usage. When John sends the request
\((\text{john}, \text{game}, \text{download})\) again, the request is granted.

Figure 7.10 show the evaluation output of the GUI front-end for the following three requests discussed.
The game is advertisement sponsored and requires the user to watch advertisements during gameplay.
When John wants to play the game, by sending a request \((\text{john}, \text{game}, \text{play})\) without watching the
advertisements, his request is denied. Once John starts watching advertisements, by sending a
7.4. Discussion

In this chapter, we have shown a number of UCON policies, demonstrating attribute updates, ongoing evaluation and references to other usages. We have provided the mapping of the UCON policies to the Intermediate Policy Language, and from the Intermediate Policy Language to XACML.

We have demonstrated our UconXACML implementation. We have verified a number of requests against the mapped XACML policies. The results from these requests are the expected evaluation results, except for the evaluation of a policy specifying time constraints. This policy is expressed in standard XACML without extensions, therefore it should work out-of-the-box. Time in XACML is assumed to be Coordinated Universal Time (UTC) unless otherwise specified. The unexpected results for the evaluation of time constraints seem unrelated to the time zone in the specification. We have tested times specified in both UTC and local time. We assume this is a bug in either HerasAF or Joda-Time.

In our design, we have assumed singleton attribute values, while UCON supports arbitrary data types, which may also be sets. Sets as data types are not supported. XACML, by default, returns a bag of attributes when referencing an attribute. This is the reason for the use of functions like one-and-only to adjust the data type. The reason behind this design choice are updates. An update calculates a new value and assigns that value to an attribute. If an attribute could contain multiple values, it is unclear how to define an update. The assumption of singleton attribute values has lead to a certain data storage implementation. We store our data in a relational database: SQLite3. Where the data retrieving code can be adjusted to support multiple attribute values by changing a few lines of code, the attribute updating mechanism depends on the assumption of singleton data types. All data is stored as a string in the database. Data type conversion is performed by HerasAF, which requires data to be provided as strings.

The SQLite3 database engine is a database engine not suitable for heavy loads, therefore it is not suitable for a large scale production environment. Our database implementation is based upon JDBC. JDBC is an database abstraction class provided by Java. Using JDBC means that switching to a different database engine costs as little as changing a few lines of code.
CHAPTER 7. VALIDATION

Figure 7.7: UCON XACML Validation, UCON Ongoing Authorisations and updates

(a) Re-evaluation on attributes changes by a different request
(b) Re-evaluation only happens when relevant attributes change

Figure 7.8: UCON XACML Validation, UCON Ongoing Authorisations

Other optimisations related to database usage might include differentiating mutable and immutable attributes. Immutable attributes, which are never updated by a policy, can be stored in a different storage optimised for read only access. Furthermore, cashing of frequent accessed immutable attributes may contribute to a performance gain.

In our design we have decided to implement the necessary features to implement UCON on top of XACML using \(<\textit{Obligation}>\)s. \(<\textit{Obligation}>\)s are part of the evaluation result. We have implemented handling \(<\textit{Obligation}>\)s in an extensible way. We define an array of obligation services. We present an obligation to each obligation service, starting with the first in the array, until one obligation service has handled it. Obligation services determine if it can handle an obligation by checking its obligation id. This implementation enables adding support for additional implementations by adding a new obligation service to the array.

One of the features encoded as \(<\textit{Obligation}>\)s are updates. In an update a new attribute value has to be calculated. Because ongoing- and post-updates are executed at a later time, its value cannot be determined at evaluation time. Therefore, an update is stored as an unevaluated snippet of XACML. We re-use parts of the HerasAF XACML evaluation to calculate the update values. In a similar way, we evaluate conditional updates. The condition is also a snippet of XACML which evaluates to a boolean value.

The other feature encoded as \(<\textit{Obligation}>\)s are triggers. A trigger is activated on an attribute change and can either re-evaluate a policy or execute an attribute change triggered update.
7.4. DISCUSSION

Figure 7.9: UCON XACML Validation, UCON Pre Obligations

(a) ActiveUsage: advertisement must be active to play (b) ActiveUsage: when advertisements are stopped, the game stops too

Figure 7.10: UCON XACML Validation: UCON Ongoing Obligations

A notable part in the design is the RequestHandler. A RequestHandler has a number of callback functions, which are called when a request is allowed, denied or stopped. The callback functions implement application specific actions that must be taken on such occasion. The RequestHandler is the connection between the UconXACML framework and the application interacting with users. Furthermore, the RequestHandler handles updates. The choice to make the RequestHandler responsible for updates is because the RequestHandler is aware when a usage begins and ends, which is relevant to ongoing- and post-updates in particular.
Chapter 8

Related Work

8.1 Other XACML UCON Implementations

U-XACML [11] is an extension of XACML capable of expressing UCON policies. For example, U-XACML adds \texttt{DecisionTime} to \texttt{<Condition>}. The most notable difference between our approach and U-XACML is the fact our approach does not extend XACML. Our approach encodes an UCON policy in the XACML 2.0 XML schema. Another difference is how mutable attributes are handled. U-XACML introduces an \textit{attribute retrieval policy} to determine when new attribute values have to be fetched for re-evaluation. Our approach triggers re-evaluation on the event of updating an attribute value.

Um-e-Ghazia et al. present a profile [2] of UCON in XACML 3.0. In this profile subjects are divided into categories, \textit{consumer}, \textit{provider} and \textit{identifyee}. Objects are divided into \textit{privacy sensitive}, \textit{privacy non sensitive} and \textit{derivative}. Rights are divided into \textit{consumer right}, \textit{provider right}, \textit{identifyee rights} and \textit{usage control}. Our implementation is against XACML 2.0 and does not implement such categorisation. Furthermore, the profile by Um-e-Ghazia et al. requires when an attribute is referred an indication whether an attribute is mutable or immutable. Our implementation does not provide such differentiation.

8.2 XACML Implementations

XACML is an open standard by the OASIS consortium. There are various implementations in existence. Table 8.1 provides an overview of XACML implementations which we will discuss in this section. In this table we provide the license, which is relevant when developing a derived product. For example, a GPL license is a copyleft license, which means a requirement to release the code of derived products under the same terms. Licenses such as the Apache License 2.0 and the BSD 2 Clause license have less strict terms on derived work. Furthermore, we look at the development type. Whether an individual, a research institute or commercial organisation is behind the development of a software product influences the stability of the development process. We also look at the latest release and commit dates when available. The latest release date represents the date when the development team has released a stable product, whereas the latest commit date represents the date the latest change has been made to the work-in-progress. Both the development type and the dates give insight to the stability of the development process.

8.2.1 Axiomatics (https://www.axiomatics.com/)

\begin{tabular}{|l|}
\hline
Name & Axiomatics \\
\hline
URL & \\
\hline
\end{tabular}

Axiomatics is a company based in Stockholm. Axiomatics provide commercial solutions for Attribute Based Access Control, based upon XACML 3.0. Currently, the white papers about the products Axiomatics offers are not available. Since the commercial nature of the Axiomatics XACML solution, this XACML implementation is not suitable for research purposes.
8.2.2 enterprise-java-xacml

Enterprise-java-xacml is an XACML 2.0 implementation. The project website mentions planned support for XACML 3.0 in 2012. However, the latest release dates from 2009, and the latest source commit from 2010. Therefore it appears this project is not being maintained any more. Obligations are not supported, which makes this project unsuitable for our purposes.

8.2.3 HerasAF XACML Core

HerasAF XACML Core is an XACML 2.0 implementation written by the University of Applied Science Rapperswil. The HerasAF project claims to be the de-facto reference implementation of XACML 2.0. HerasAF is designed to be adaptable which means it can be integrated with little changes. HerasAF supports Obligations which makes it suitable for our purposes.

8.2.4 PicketBox XACML

PicketBox XACML is part of the PicketBox security framework, developed by Red Hat. The latest release and commit date from over a year ago. Despite the fact of no recent activity, PicketBox is mentioned in recent Red Hat documentation, which suggest the project is currently supported. Obligations are supported, which makes PicketBox XACML suitable for our purposes.

8.2.5 Sun XACML

SunXACML is a XACML implementation by Sun Microsystems. It offers support for XACML versions 1 and 2. The latest release and source commit date from 2010. Therefore this project is currently unmaintained. Being unmaintained makes this XACML implementation less suitable for research purposes.
8.2.6 WSO2 Balana (https://svn.wso2.org/repos/wso2/trunk/commons/balana/)

WSO2 Balana is an XACML implementation based upon SunXACML. Compared to SunXACML it adds XACML 3.0 support. As Balana is a component integrated in WSO2’s products, a latest release date cannot be determined. By the date of the latest commit, WSO2 Balana appears to be a maintained project. Obligations are supported, which makes WSO2 Balana suitable for our purposes.

8.2.7 xacml4j (http://www.xacml4j.org/)

The xacml4j project has no webpage other than a code repository. Seeing there is only one code committer suggests this project is backed by only one individual. Xacml4j supports XACML 2.0 and 3.0, and has recent releases and code commits. Seeing there is only a single developer on this project gives little guarantees for future support.

8.2.8 XACMLight (http://xacmllight.sourceforge.net/)

XACMLight is an implementation of XACML 2.0. The latest release and code commit are from 2008, which makes this project unmaintained. Furthermore, the lack of obligation support make this project unsuitable as base for a project that depends on Obligation support.

8.2.9 XEngine

XACMLight is an implementation of XACML by the Michigan State University. It is not clear what version of XACML is implemented. The latest release is from 2010, however the latest code commit is from 2008. This suggests the latest code has not been released. However, seeing the latest release being from 2010 suggests this project is no longer maintained. Furthermore, the lack of obligation support make this project unsuitable as base for a project that depends on Obligation support.

8.2.10 Conclusions

We have decided to build the project using HerasAF. HerasAF is an XACML 2.0 implementation with Obligation support developed by a research institute. Therefore it fits the requirements of our project. Furthermore, Eindhoven University of Technology has already projects based upon HerasAF. Using HerasAF enables integration of our work with existing projects.

8.3 Other models for usage control

Apart from UCON there are other models for usage control. These models may have more expressive power on certain aspects of access restrictions. M. Hilty et al. [3] present Obligation Specification Language (OSL). OSL is a formal policy language that supports a wide range of usage control requirements. The OSL policy language supports expressing certain policies such as expressing a condition must hold during a certain interval, where UCON only supports at request time or during the entire usage. OSL also features obligatory tasks that have to be performed after an access, such as deleting a fine after 30 days. This is referred to as duties by Jaehong Park et al. and is not within the scope of UCON. Furthermore, OSL has a higher focus on DRM and provides translations between OSL and languages prominent within DRM such as XrML and ODRL. Another notable difference between UCON and OSL is the fact that UCON focusses on server-side control, whereas OSL focusses on distributed systems.

Alexander Pretschner et al. [14] present a tool that does a formal analysis of policies written in OSL. OSL is extended with features and restrictions to support re-distribution. The restrictions upon re-distribution include that a policy can only more restrictive or stronger then the original policy. The formal analysis tool can check if these restrictions hold.
Chapter 9

Conclusions and Recommendations

In this Chapter we present the conclusions of the thesis. Furthermore, we present possible directions for future work, related to the implementation of the project.

9.1 Conclusions

We have provided three contributions in this thesis:

- A mapping from UCON to XACML
- An extension to the XACML architecture to support UCON features
- An implementation of the extended XACML architecture

We have created a mapping from UCON to XACML. For this purpose we have analysed UCON and XACML in Chapter 3 and defined an Intermediate Policy Language in Chapter 4. The Intermediate Policy Language is defined in a way that its structure resembled XACML, while more better human readable. In Chapter 5 we presented a mapping of UCON features to XACML using the Intermediate Policy Language. For this purpose we have defined a number of XACML Obligations to encode the UCON features XACML cannot naturally express. We have defined Obligations that express when a policy is re-evaluated: timer-based triggers and attribute change triggers. We have defined Obligations that express an update of an attribute value. For updates we have also defined triggers for timer-based and attribute change based triggers.

Apart from UCON features we have introduced Domain Knowledge Updates, a feature that is beyond the scope of UCON. This feature enables the definition of global applicable updates. An update can be applied whenever an attribute is changed for any subject or resource.

In Chapter 6 we presented a framework that is capable of evaluating and enforcing policies specified in XACML with the defined obligations. The implementation uses HerasAF as XACML evaluation engine. Our work consists of components that parse the obligations we have defined, perform updates and re-evaluation of policies under the specified conditions, and enforces the results of the evaluations. We have created a GUI to demonstrate the evaluation of policies by our framework. In Chapter 7 we demonstrate the evaluation results given by our framework are the expected results.

9.2 Future Work

The UconXACML project is designed as a proof-of-concept. This means there have been no optimisations for performance. For example, we are using SQLite3 as our database engine. This database engine is suitable for small projects, and does not perform well in large systems. Due the database functionality
provided by Java, JDBC, switching to a different database engine takes little effort. Furthermore, the
database schemas and database usage are not optimised. For example, the current implementation stores
all ended usage as PastUsage. This could be optimised to only store usage in PastUsage that may be used
in future decisions. The current database does not distinguish between mutable and immutable attributes.
Introducing such a distinguishing could enable the possibility to cache immutable attributes. Cashing
immutable attribute values of frequently accessed attributes can improve performance. Apart from
database related improvements, the implementation is currently only fitted with a GUI for demonstration
purposes. An interface to a real-life scenario could be created, to demonstrate practical applicability of
the project.
Bibliography


<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>ActiveUsage (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>Authorisations, Ongoing (UCON)</td>
<td>6</td>
</tr>
<tr>
<td>Authorisations, Pre (UCON)</td>
<td>5</td>
</tr>
<tr>
<td>CH (XACML)</td>
<td>14</td>
</tr>
<tr>
<td>Combination Algorithm (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>Condition (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>Conditions, Ongoing (UCON)</td>
<td>8</td>
</tr>
<tr>
<td>Conditions, Pre (UCON)</td>
<td>8</td>
</tr>
<tr>
<td>Context Handler (XACML)</td>
<td>14</td>
</tr>
<tr>
<td>DAC</td>
<td>3</td>
</tr>
<tr>
<td>Discretionary Access Control</td>
<td>3</td>
</tr>
<tr>
<td>Effect (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>EvaluationTime (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>ExecutionTime (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>MAC</td>
<td>3</td>
</tr>
<tr>
<td>Mandatory Access Control</td>
<td>3</td>
</tr>
<tr>
<td>Obligation (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>Obligation Services (XACML)</td>
<td>14</td>
</tr>
<tr>
<td>Obligations, Ongoing (UCON)</td>
<td>7</td>
</tr>
<tr>
<td>Obligations, Pre (UCON)</td>
<td>6</td>
</tr>
<tr>
<td>OS (XACML)</td>
<td>14</td>
</tr>
<tr>
<td>PAP (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>PastUsage (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>PDP (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>PEP (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>PIP (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>Policy (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>Policy (UCON)</td>
<td>5</td>
</tr>
<tr>
<td>Policy (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>Policy Administration Point (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>Policy Decision Point (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>Policy Enforcement Point (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>Policy Information Point (XACML)</td>
<td>13</td>
</tr>
<tr>
<td>PolicySet (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>RBAC</td>
<td>3</td>
</tr>
<tr>
<td>Role Based Access Control</td>
<td>3</td>
</tr>
<tr>
<td>Rule (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>Rule (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>Target (XACML)</td>
<td>10</td>
</tr>
<tr>
<td>Update (imodel)</td>
<td>21</td>
</tr>
<tr>
<td>Updates, Ongoing (UCON)</td>
<td>9</td>
</tr>
<tr>
<td>Updates, Post (UCON)</td>
<td>10</td>
</tr>
<tr>
<td>Updates, Pre (UCON)</td>
<td>9</td>
</tr>
<tr>
<td>Usage Control (UCON)</td>
<td>4</td>
</tr>
</tbody>
</table>
Appendices
Appendix A

Implementation: libraries and classes

The following libraries are used for HerasAF and its dependencies:

- herasaf-xacml-core-1.0.1-RELEASE-SNAPSHOT.jar
- slf4j-api-1.7.7.jar
- slf4j-simple-1.7.7.jar
- joda-time-2.3.jar

Additionally sqlite-jdbc-3.7.2.jar is used to provide a database.

The UCON XACML Implementation consists of the following classes and interfaces:

- Extractor.java
- RequestHandler.java
- ResourceUpdate.java
- SubjectUpdate.java
- UconOS.java
- UconOsTriggerParser.java
- UconOsUpdateParser.java
- UconPAP.java
- UconPEP.java
- UconPIP.java
- Update.java
- UpdateMarshaller.java

Figure 6.2 gives an UML class diagram.

A.1 Implementation: code fragments
```java
pip = new UconPIP(this);
SimplePDPConfiguration config = new SimplePDPConfiguration();
config.setPip(pip);
config.setRootCombiningAlgorithm(new PolicyDenyOverridesAlgorithm());
config.setRespectAbandonedEvaluatables(true);
pdp = SimplePDPFactory.getSimplePDP(config);
PAP pap = new PAP(pdp);
oss.add(new UconOsUpdateParser());
oss.add(new UconOsTriggerParser());
```

Figure A.1: UconPEP: Initialisation code

```java
RequestType request = requestHandler.getRequest();
List<ResultType> results = pdp.evaluate(request).getResults();
```

Figure A.2: UconPEP: addRequest(): evaluating request by PDP

```java
for (int i = 0; i < obligations.size(); i++) {
    for (int j = 0; j < oss.size(); j++) {
        OsResult osresult = oss.get(j).handleObligations(requestHandler,
                obligations.get(i), firstEvaluation,
                (result.getDecision() == DecisionType.PERMIT));
        if (osresult == OsResult.HANDLED) {
            obligationsHandled = true;
            obligationsApplicable = true;
            break;
        }
        if (osresult == OsResult.NOTHANDLED) {
            obligationsApplicable = true;
        }
        obligationsHandled = false;
    }
    if (!obligationsHandled && obligationsApplicable) {
        result.setDecision(DecisionType.DENY);
    }
}
```

Figure A.3: UconPEP: addRequest(): handling obligations returned by the PDP
String sSubjectID = Extractor.ExtractSubjectId(request);
PreparedStatement ps = con
  .prepareStatement("SELECT attributeValue "
    + "FROM subjects WHERE SubjectID = ? AND attributeID = ?");
ps.setString(1, sSubjectID);
ps.setString(2, attributeId);
System.out.println("Requesting Subject :" + sSubjectID + "
  + attributeId);
ResultSet rs = ps.executeQuery();
if (rs.next()) {
  AttributeValueType avt = new AttributeValueType();
  avt.getContent().add(rs.getString(1));
  attributeValues.add(avt);
}
rs.close();

Figure A.4: UconPIP: fetchSubjectAttributes(): fetching subject attributes from database

psUpdate = con
  .prepareStatement("UPDATE subjects SET attributeValue=?" 
    + " WHERE attributeID=? AND subjectID=?");
psUpdate.setString(1, attributeValue);
psUpdate.setString(2, attributeId);
psUpdate.setString(3, subjectId);
if (psUpdate.executeUpdate() == 0) {
  psInsert = con
    .prepareStatement("INSERT INTO subjects "
      + "(subjectID,attributeID,attributeValue) VALUES (?,?,?)");
  psInsert.setString(1, subjectId);
  psInsert.setString(2, attributeId);
  psInsert.setString(3, attributeValue);
  psInsert.executeUpdate();
}

Figure A.5: UconPIP: updateSubjectAttribute(): updating subject attributes in database

for (int i = 0; i < reEvaluationTriggers.size(); i++) {
  Trigger trigger = reEvaluationTriggers.get(i);
  if (trigger.getClass().toString()
      .matches(SubjectTrigger.class.toString())) {
    if (trigger.getId().matches(subjectId) 
      && trigger.getName().matches(attributeId)) {
      pep.addToReEvaluationQueue(trigger.getRequester());
    }
  }
}

Figure A.6: UconPIP: updateSubjectAttribute(): re-evaluation queue
while (postUpdates.size() != 0) {
    postUpdates.get(0).apply();
    postUpdates.remove(0);
}
    pep.startReEvaluation();

Figure A.7: RequestHandler: onStopped(): post updates

ExpressionType et = ((JAXBElement<ExpressionType>) UpdateMarshaller
    .unmarshal(value)).getValue();

return et.handle(requester.getRequest(), ec);

Figure A.8: Update: evaluateValue()
Appendix B

XACML policies

B.1 updateRequestNew.xml

```xml
<Rule RuleId="defaultRule" Effect="Permit">
  <Obligations>
    <Obligation ObligationId="onUpdate" FulfillOn="Permit">
      <AttributeAssignment AttributeId="subjectAttribute" DataType="http://www.w3.org/2001/XMLSchema#string">
        colour
      </AttributeAssignment>
      <AttributeAssignment AttributeId="updateValue" DataType="http://www.w3.org/2001/XMLSchema#string">
        red
      </AttributeAssignment>
      <AttributeAssignment AttributeId="condition" DataType="http://www.w3.org/2001/XMLSchema#string">
        <![CDATA[
          
          <Apply xmlns="urn:oasis:names:tc:xacml:1.0:function" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-greater-than">
            <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">25</AttributeValue>
            </SubjectAttributeDesignator>
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">25</AttributeValue>
          </Apply>
        ]]>
      </AttributeAssignment>
    </Obligation>
  </Obligations>
</Rule>
```

The Domain Knowledge Policy should evaluate to Permit to release its Obligations.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.

The Domain Knowledge Policy contains onUpdate Obligations. The updateSubjectTrigger indicated the update is to be triggered when a subject's credit attribute changes. When an attribute change happens, the condition is evaluated. If the condition is true, the subjectAttribute (colour) is set to the value indicated by updateValue.
### B.2 paidCall.xml

```xml
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os" schemaLocation="urn:oasis:names:tc:xacml:2.0:policy-instance" version="1.0" encoding="UTF-8" standalone="yes">
  <Rule Id="paidCall" Effect="Deny">
    <Target>
      <Resource>
        <ResourceMatch>
          <MatchId>urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal</MatchId>
          <AttributeData>
            <AttributeId>credit</AttributeId>
            <AttributeValue/>
          </AttributeData>
        </ResourceMatch>
      </Resource>
    </Target>
    <Action>
      <Apply xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os" schemaLocation="urn:oasis:names:tc:xacml:2.0:policy-instance" version="1.0" encoding="UTF-8" standalone="yes">
        <AttributeAssignment>
          <AttributeId>updateSubjectTrigger</AttributeId>
          <DataReference />
        </AttributeAssignment>
      </Apply>
    </Action>
  </Rule>
</Policy>
```
### B.3. TOPUPCALL.XML

```
<xml version="1.0" encoding="UTF-8" standalone="yes"/>

<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <!-- This policy applies if the action-id is "call" The policy tests for the presence of a topupValue attribute for the resource by checking 0 <= topupValue -->
    <Target>
        <Resources/>
        <ResourceMatch>
            <ActionMatch>
                <MatchId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal"
                    <AttributeValue Data-type="http://www.w3.org/2001/XMLSchema#int">
                    0
                </AttributeValue>
            </ResourceMatch>
            <AttributeAssignment>
                <AttributeId="updateValue"
                    Data-type="http://www.w3.org/2001/XMLSchema#int">
                </AttributeAssignment>
            </ResourceMatch>
        </ResourceMatch>
    </Target>
    <Rule>
        <Obligation>
            <!-- This obligation encodes subject's credit has to be increased by the object's topupValue after the access has finished. -->
            <AttributeAssignment>
                <AttributeId="subjectAttribute"
                    Data-type="http://www.w3.org/2001/XMLSchema#string">
            </AttributeAssignment>
        </Obligation>
        <!-- This obligation encodes subject's credit has to be decreased by the object's price every 10 seconds. -->
        <Obligation ObligationId="onUpdate" FulfillOn="Permit">
            <AttributeAssignment>
                <AttributeId="subjectAttribute"
                    Data-type="http://www.w3.org/2001/XMLSchema#string">
            </AttributeAssignment>
        </Obligation>
    </Rule>
</Policy>
```
APPENDIX B. XACML POLICIES

B.4 minimalCredit.xml

```xml
  <Target>
    <Rule RuleId="minimalCredit" Effect="Deny">
      <Condition>
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal"/>
      </Condition>
    </Rule>
  </Target>
</Policy>
```

B.5 pastUsage.xml

```xml
  <Target>
    <Action>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than"/>
    </Action>
  </Target>
</Policy>
```
B.6 activeUsage.xml

```xml
<Policy version="1.0" encoding="UTF-8" standalone="yes">  
  <Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os">  
    <RuleCombiningAlgId id="urn:oasis:names:tc:xacml:1.0:rule:combining-algorithm:deny-overrides" />  
    <PolicyId id="testActiveUsage" />  
    <Condition>  
      <ActionMatch Id="urn:oasis:names:tc:xacml:1.0:function:string-equal" />  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:action:action" />  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:subject:subject-id" />  
    </Condition>  
    <Action>  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:action:action" />  
    </Action>  
    <Target />  
  </Rule>  
</Policy>
```

B.7 time.xml

```xml
<Policy version="1.0" encoding="UTF-8" standalone="yes">  
  <Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os">  
    <PolicyId id="timeLimitation" />  
    <Condition>  
      <ActionMatch Id="urn:oasis:names:tc:xacml:1.0:function:string-equal" />  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:action:action" />  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:subject:subject-id" />  
    </Condition>  
    <Action>  
      <AttributeId id="urn:oasis:names:tc:xacml:1.0:action:action" />  
    </Action>  
    <Target />  
  </Rule>  
</Policy>
```
<Rule RuleId="timeLimitation" Effect="Deny">
  <Condition>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:or">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than">
        <EnvironmentAttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#time"
          AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">
          14:00:00
        </AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than">
        <EnvironmentAttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#time"
          AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">
          15:00:00
        </AttributeValue>
      </Apply>
    </Apply>
  </Condition>
</Rule>