Translation of Usage Control Policies

Semester Thesis

Lukas Mathias Novosad

June, 2007

ETH Zürich
Department of Computer Science
Chair of Information Security

SUPERVISORS:
PROF. DR. DAVID BASIN
DR. ALEXANDER PRETSCHNER
MANUEL HILTY

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Usage control enhances access control by not only specifying how the content may be accessed but also by defining what may or may not happen to the data afterwards.

Currently there are many DRM systems which provide enforcement mechanisms for usage control policies.

We take OSL, a general purpose language for usage control, and translate it to and from XrML, which is a rights expression language from the DRM community. The purpose of this approach is to use DRM mechanisms to enforce OSL policies and to enhance the interoperability between different DRM standards. We present a proof of concept using Microsoft's RMS (Rights Management Services).

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem space . . . 1
1.3 Contribution . . . 2
1.4 Document overview . . . 3

2 Background **5**
2.1 OSL . . . 5
2.2 XrML . . . 6
2.3 RMS . . . 6

3 Implementation **7**
3.1 OSL grammar adaptation . . . 7
3.2 Design Overview . . . 8
3.3 Code layout . . . 9
3.4 Supported subset of XrML . . . 10
3.5 From XrMLc to OSL . . . 11
3.5.1 General Idea . . . 12
3.5.2 Events needed . . . 12
3.5.3 Basic structure . . . 12
3.5.4 The Grant . . . 13
3.5.5 The KeyHolder . . . 13
3.5.6 Exercise Limit . . . 13
3.5.7 The Resource . . . 13
3.5.8 Conditions . . . 14
3.5.9 Rights . . . 14
3.5.10 Fees . . . 14
3.6 From OSL to XrMLc . . . 14
3.6.1 General Idea . . . 18
3.6.2 Rights needed . . . 18
3.6.3 Basic structure . . . 18
3.6.4 The Event . . . 19
3.6.5 TimeDefault . . . 19

3.6.6 The Obligation 19

4 Application **21**
4.1 Preparation 21
4.2 Implementation 21
4.3 Functionality 22
4.4 RMS - Overview 23
4.4.1 Hierarchy 23
4.4.2 Licenses 23
4.5 Package format 24
4.6 User identification 24
4.7 Content creation 24
4.8 Content consumption 25

5 Conclusion **27**
5.1 Contribution 27
5.2 Limitations 27
5.3 Future Work 27
5.4 Personal Experience 28

Bibliography **29**

List of Figures **33**

A Relax NG Schema **34**

B OSL example policy **36**

C XrML example policy **38**

D RMS Content Processing **40**

Chapter 1

Introduction

1.1 Motivation

Data management and protection in general is a very important topic in computer science. Access Control (AC) is a widely spread and well-known paradigm. It catches the essence of who may access specific data and how exactly this is realized. Recently the need for a more sophisticated approach to content control has appeared: Usage Control (UC) [1]. UC takes AC even further by specifying what may or may not happen to the content data after it has been actually accessed. This includes for example redistribution of content. If a policy states that a legitimate user may access a document for viewing but should not be able to paste a copy on some web page which is accessible to anybody then AC alone clearly cannot enforce this policy. That is exactly what UC was designed for, namely it provides security measures against legitimate or even malevolent users who might expose, modify or even damage sensitive data by accident or deliberately on purpose.

1.2 Problem space

Digital Rights Management (DRM) systems are widely spread and use different standards to implement the needed UC policies. Usually the type of language used to describe the policies in a DRM system is a Rights Expression Language (REL). A widely used REL is XrML - eXtensible rights Markup Language. As the name suggests, it is based on a XML-based sytanx and hence easily processable by existing software libraries. The Obligation Specification Language (OSL) developed at the ETH Zürich may be used to specify usage control policies.

We want to translate between OSL and a chosen REL. For the direction from OSL to the REL this demonstrates that OSL policies can be enforced in currently existing DRM mechanisms. The other direction from

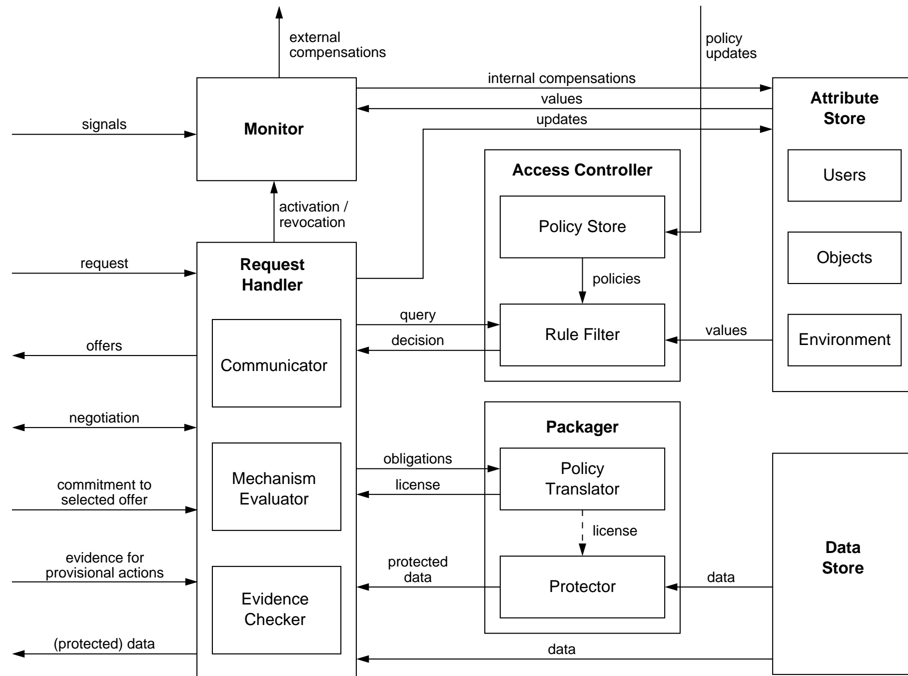

Figure 1.1: Provider components overview

the chosen REL to OSL shows that OSL may potentially serve as an intermediate format for the translation between different RELs and thus also DRM systems.

1.3 Contribution

This thesis provides a framework which can translate usage control policies from one language to another. It is able to translate policies between OSL and XrML. A similar approach for the translation between OSL and ODRL (Open Digital Rights Language) is depicted in [2]. This demonstrates the expressiveness of OSL and the potential to use it as an intermediate format for the translation of usage control policies written in almost any standard available at the moment. Hence for n given control policy formats it may reduce the amount of needed translation types from n^2 to n. This makes it very interesting for any digital content provider supplying several different end-user platforms each possibly using a different DRM standard.

Furthermore we also show that OSL policies may actually be enforced in current DRM systems by providing a prototype application using Microsofts RMS - Rights Managment Services - which uses XrML to describe

its policies.[1] This enables us to use the previously developed translation component.

In Figure 1.1 you see a possible arrangement of provider components as it is introduced in [1]. Our translation component corresponds to the Policy Translator and the Protector in our prototype application is implemented with Microsoft's RMS SDK.

1.4 Document overview

The structure of this document is as follows:

Background After the introduction we provide more detailed descriptions on the background of this thesis. This will include important features of OSL, XrML and the parser framework provided by Vania Bättig's Master Thesis "Monitoring Usage Control Requirements" [3] and the CoCo parser framework used therein.

Implementation Following the background section we describe details about the implementation of the translations for both directions. This should give the reader a good overview and provide better insight. Anybody interested in particular details should of course read the actual implementation code.

Application The part about the implementation is followed by a chapter on the prototype application where we use our translation framework to successfully translate an usage control policy from OSL to XrML and then enforce it in the prototype by using Microsoft's RMS.

Conclusion At the end we draw some conclusions about the experiences made and the difficulties encountered while writing this semester thesis. We also provide an outlook on possible future work which could be done in this area of research.

[1] Microsoft's RMS homepage, http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx

Chapter 2

Background

There are three main technologies used in this thesis and we give an overview of these in the following sections.

2.1 OSL

OSL is a language which specifies obligations on how resources may be used. It is developed at the ETH - Eidgenössische Technische Hochschule - in collaboration with DoCoMo Euro-Labs. We adapted the grammar to improve readability during the writing of this thesis as well as to form it to serve our needs. The first construct in an OSL policy is an event declaration block which starts with *EventDecl*. All the needed events, or rights respectively, are listed with all possible parameters such as user or device. After that a *TimeDefault* is listed which is used whenever a time unit is needed but not provided. After that the most important part follows, namely the *OblDecl* block containing all the obligations. Obligations are formulae describing the restrictions which implement the desired usage control. Now it is a good time to have a look at appendix B where our OSL example policy used throughout this documentation is listed.
Considering OSL the most interesting and valuable sources were [1] and [2]. We separated the parser framework in [3] which processes OSL policies from the monitoring functionality contained therein. This parser is created with the help of the CoCo parser framework.[1] The monitor is responsible for detecting usages not allowed in the corresponding OSL policy. In this thesis, however, we were only interested in translating usage control policies, so we decoupled the parser for the OSL policies from the whole monitor package and modified it a little bit to serve our needs. The demo application described in chapter 4 demonstrates that the OSL policies we process with our translator component are valid and may be used in corresponding applications.

[1] CoCo parser framework, http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

CoCo is a very useful library in the form of a .jar file. It is a parsing framework which takes as input a grammar file and automatically produces the corresponding scanner and parser classes. Additional variable declarations and initializations can be passed to CoCo in two user-provided files. One one hand we have the *scanner.frame* for the scanner class and on the other hand there exists the *parser.frame* for the parser class.

2.2 XrML

XrML is a markup language defining rights for users and resources. It is build upon the familiar XML and therefore easily understood and processed. The basic XrML construct is the `<license>` which contains the three abstract elements `<issuer>`, `<title>` and `<grant>`. The title simply serves as a readable form for humans, e.g. a description of the license, and the issuer is the party issuing the license. More important is the grant, which contains the four main constructs `<condition>`, `<principal>`, `<resource>` and `<right>`. These are all abstract and must be refined. More details can be found in the following chapter. As previously with OSL the reader is encouraged to take a look at our XrML policy used throughout the document which can be found in appendix C.

2.3 RMS

RMS - Rights Managment Services - is an Active Directory (AD) service for Microsoft Windows Server 2003. Since RMS is a product belonging to Microsoft, most related resources can be found on their web pages. Besides the usual content such as documentation or a FAQ site they also provide a developer blog and forum. The best way to get started is to install the RMS SDK (current version is SP2) and read the source examples provided with it. They describe most of the needed paradigms and API calls needed for an RMS-enabled application. The SDK as well contains an API documentation in the form of a help file.

Chapter 3

Implementation

In this chapter we describe the steps undertaken to implement the translator component. The first goal was to adapt the OSL grammar in order to enhance readability and to adapt it to serve our needs. After the adaption, we discussed how to design the translation framework. For this, we had to restrict ourselves to subsets of OSL and XrML, respectively. One reason for this is that it would have been too tedious to support the complete languages and we did not need the full functionality of the respective languages. Another reason is the fact that there are constructs and features in one language which are simply non-existent in the other language and vice versa. Similar approaches such as in [4] or [5] also restricted themselves in order to keep things simple and comprehensible. The steps needed to enhance a framework to support a wider language spectrum are are relatively easy once the basic framework is functional. We implemented the translator component using the Java programming language. An illustration is given in figure 3.1

3.1 OSL grammar adaptation

We made three main modifications to the grammar as it was used in [3]. The first was that we made the declaration of an event uniquely identifiable by its name only. Before that change, it was the case that an event was identified by both its name and the number of accompanying arguments. This of course also meant that an event was possibly declared several times, each time with a different number of input arguments. We came to the conclusion that this was actually redundant and we changed the event declaration accordingly.

The second adaptation we made was the improved readability of the OSL policy syntax. In particular we added the parameter types used for events in the OSL *Obl* (Obligation) constructs. For example the use of event `pay{Bob,CHF,5}` became `pay{recipient=Bob,currency=CHF,amount=5}`.

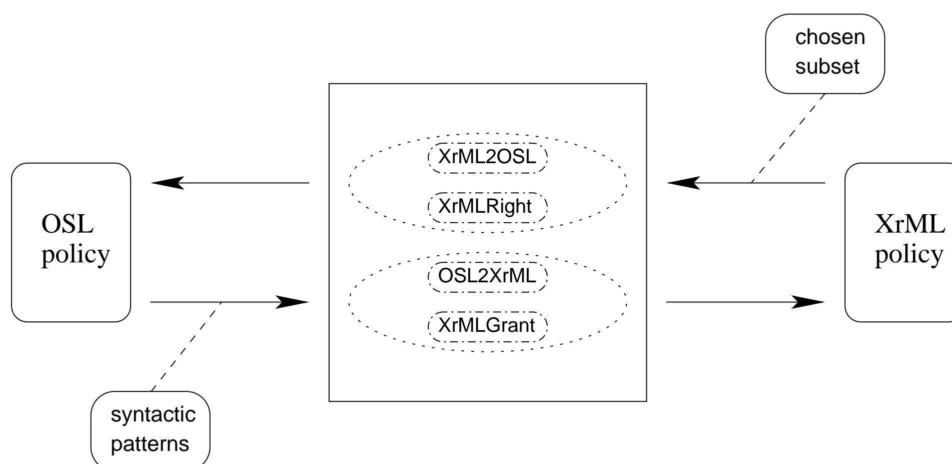

Figure 3.1: Translator component

Furthermore, we changed the *Event* declarations such that they now have a prefix "Event", similar to the *Obl* constructs which start with a "Obl" prefix.
The last change we made was the simplification for the *PermitEvParams* and *PermitEvNames* constructs, both in readability and processing. Refer to the listing in appendix B to see how the result looks like in our OSL example policy.

Readers interested in the actual attributive grammar declaration should have a look at the *obligations.ATG* file contained in the source code.

3.2 Design Overview

For the direction of translation from XrML to OSL a DOM tree must be constructed and traversed with the help of JAXP.[1] In order to create readable code, we distributed the treatment of the sub trees across several methods. The code is heavily based on if-then-else-constructs, but this is due to the nature of XML and DOM itself. A possibility would have been to introduce some sort of visitor pattern, but the housekeeping of the context with all the associated values such as boolean flags would have quickly become very cumbersome and most importantly hard to extend. And since this work only covers parts of XrML at the moment, an extension is definitively worth considering.
For the direction from OSL to XrML, we use the adapted OSL grammar and create a parser and scanner with the help of CoCo, similar to the approach

[1] Java API for XML Processing - http://java.sun.com/webservices/jaxp/

taken in [3]. The translator first reads through the block containing the event declarations and the processess each obligation from the *OblDecl* block separately.

3.3 Code layout

We wrote two classes for the direction from XrML to OSL. The main class is called *XrML2OSL.java* and it incorporates the whole conversion process. That is it first sets up the handles and buffers needed for I/O handling and then reads in the XrML policy by using classes from JAXP and also from the W3C DOM package. It uses a helper class *XrMLRight.java* to temporarily store important data needed to construct the *Obl* constructs for the OSL output policy. This class can also build a *PermitEvParams* formula, for example. While the DOM tree is parsed, the *Event* declarations are directly written to the OSL output policy and deduced *Obl* elements are written to a temporary buffer. These are written from the buffer to the output policy at the very end when the whole DOM tree has been parsed and all event declarations have been extracted from the grants. Then the output stream is closed and the policy is ready to be used by an encompassing framework.

From OSL to XrML, the conversion is implemented in the class *OSL2XrML.java*. Similar to the other direction, first buffers and handles needed for I/O are set up and then the OSL policy is parsed using the modified parser framework from the monitor package [3]. A hasthable contains instances of the helper class *XrMLGrant.java*, which is used for holding the bodies of each `<grant>` construct deduced from the input policy. They are written to the output policy at the end of the conversion process. First the *PermitEvNames* formula is processed and instances of XrMLGrant are created for each event found in the construct and stored in the above mentioned hashtable. Please note that we only allow one *PermitEvNames* construct to be present in the OSL policy. After that, all the other obligations are processed and the information extracted from them is stored in the appropriate XrMLGrant instances in the hashtable. At the end, when the parsing and processing is done, the `<grant>` bodies are written to the output policy. Finally, the output stream is closed and the XrML policy is ready to be used where it is needed.

For both directions, the TimeConverter.java class is used to convert the timing constraints in a reasonable manner. It makes use of the Joda library.[2] Since the XrML timing constraints are provided in ISO 8601 format, they can provide an accuracy of up to fractions of a second. OSL stores time data as a pair of a number and a time interval such as

[2] Joda Time - Java date and time API, http://joda-time.sourceforge.net/

"minute" for example. This information is stored in an auxiliary class called OSLTime.java and it is used by the TimeConverter class. The time converter can handle addition and subtraction of times in the ISO 8601 format and of course it converts between OSL and XrML time intervals.

We also wrote an interface class called *XrMLConstants.java*, which provides the currently supported event names and declarations. Furthermore, we wrote a class Tools.java which offers useful helper methods such as to check for a specific string in an array of strings or to find the first occurence of a specific child in a tree.

3.4 Supported subset of XrML

In order to keep the definition of the translation compact, we consider only a subset of XrML. We call this subset XrMLc, where the 'c' stands for *compact*. It was chosen in such a way that the structure and semantics of the translation can be understood very easily and in a straightforward manner. It is important to note that XrML consists of a core component - the "core schema" - and several different extensions. The extensions have a particular prefix, for example `<cx:play>`, whereas the core schema does not have any prefix, e.g. like the `<grant>` node. One extension we need is the XrML Standard Extensions which offers extensions to the abstract `<condition>` element, payment structures and timing functionality. Nodes from this extensions are depicted by the prefix **sx:**. The other extension we use is the XrML Content Extension. It provides refined rights and information considering digital resources and conditions using them. All elements from this extension start with the prefix **cx:**.

The `<license group>` element is a container for licenses and only useful when several licenses are related in some fashion. This also enables the licenses to issue other licenses by delegation of rights. Since we don't support delegation of rights, we do not need the `<license group>` and we restrict the number of elements of type `<license>` to one. The license contains an element `<title>` which is used for description and information in user interfaces. It also contains an `<issuer>` which describes the content distributor with an optional digital signature. We do not support digital signatures at the moment since it is considered to be a part of the system in which our translation component is embedded. The most important member in a license is the `<grant>` element. There can be several of them, one for each usage. A grant contains multiple elements. We require that the abstract `<principal>` child is the same for all grants in the policy provided. We refine this element with a `<keyHolder>` node. One element of the abstract type `<resource>` is used to depict the resource to which

the policy applies to and same as with the <principal> node it must be the same for all grants contained in the license. An element of the abstract type <right> must be defined for each grant. It describes a right that may be executed on the supplied resource. The XrML Content Extension, depicted by the round edged box in figure 3.2, is used for the following rights: backup, delete, pay, play, print, read. A grant can include an <sx:exerciseLimit> which serves to imply restrictions such as the maximum number of allowed play events of a music file. The abstract <condition> element may contain several refined nodes. We support two at the moment. The <validityInterval> node indicates a contiguous, unbroken interval of time, and the <validityTimeMetered> depicts an accumaleted time during which the correspoding right can be exercised. Finally the <sx:fee> element serves for payment definitions which must be carried out for certain rights. Refer to figure 3.2 for illustration.

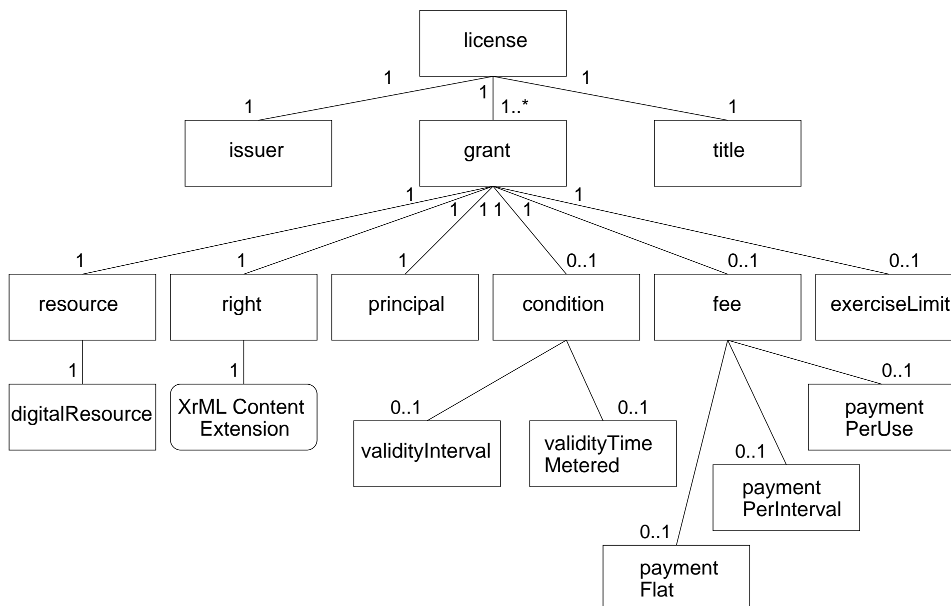

Figure 3.2: XrMLc overview

3.5 From XrMLc to OSL

First, we consider the direction of translation from XrMLc to OSL since it is fully defined due to the expressiveness of OSL. There are no restrictions for that direction and all XrMLc policies can be effectively translated to OSL.

3.5.1 General Idea

Since both types of policies are tree-structured, the translation works top-down on the XrMLc tree. As already mentioned, this means that a DOM tree is constructed using JAXP which is traversed while creating the appropriate OSL output policy. In the next sections, we describe some particular constructs and the way they are translated from XrMLc to OSL.

3.5.2 Events needed

We need the following events in order to be able to effectively map XrMLc licenses to OSL. They are defined at the beginning of an OSL license in the event declarations.

Event name	Class	Parameter name
backup	usage	object,device,user
delete	usage	object,user
display	usage	object,device,user
execute	usage	object,device,user
pay	other	recipient,currency,amount
play	usage	object,device,user
print	usage	object,device,user

Table 3.1: The needed OSL events with their usage class and arguments

3.5.3 Basic structure

The `<license>` element is mapped to an *Obligations* construct in OSL. Each of the included `<grant>` elements results in an *Event* declaration and several *Obl* declarations. The subnodes in each grant are translated by different methods we developed. Some can be translated in a straightforward manner. The `<sx:exerciseLimit>` , as an example, is mapped to a *RepMax* formula in OSL and the `<resource>` is used for object restrictions in the *PermitEvNames* formula. A more difficult case is the `<conditions>` element: One has to distinguish several cases for all the refined `<resource>` nodes which are present in order to construct the proper *Obl* elements in the OSL output policy. All the `<right>` elements are translated into the proper OSL representation as constructs of *Event* declarations and also supplied to the corresponding *Obl* formulae. Lastly the `<sx:fee>` element must be translated to a pay event based on its included children.

3.5.4 The Grant

This element can not be mapped to OSL straightforwardly in a one-to-one manner, but instead it is split up into several constructs from the Obligation Specification Language. These are described in the following few sections.

3.5.5 The KeyHolder

The `<keyHolder>` element stands for the user to which the right in the grant applies to. Its subelement `<dsig:keyName>` contains the user's name. It is extracted from the XrMLc policy and used for each *Obl* declaration in the OSL output policy. Of course it is possible to extend the data contained in the <keyHolder> construct with a digital signature for example.

3.5.6 Exercise Limit

The `<sx:exerciseLimit>`, if present, is translated into a *RepMax* formula build with the event deduced from the `<right>` element and the associated time interval for a single repetition.

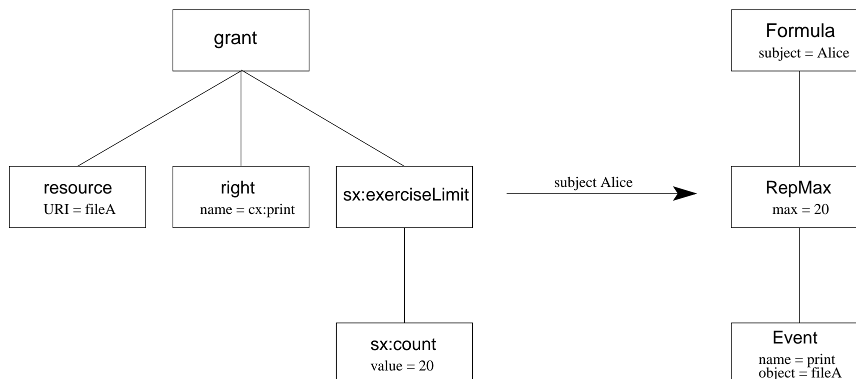

Figure 3.3: Translation of the exercise limit

3.5.7 The Resource

The abstract `<resource>` element corresponds to the *object* of the event and obligation declarations. Its implementation `<digitalResource>` identifies the digital resource by its location which is depicted with the help of a `<nonSecureIndirect URI=`*pathWithFilename*`/>` We do not support other types of resources at the moment. But as already mentioned, the framework can easily be expanded, and hence use all sorts of repositories and databases

if required. Like with the user extracted from the `<keyHolder>` element the resource is used for each obligation in the OSL output policy.

3.5.8 Conditions

Considering timing and validity there are two supported constructs at the moment. For an element of type `<validityInterval>`, the contiguous time interval has to be extracted an put into a *During* formula if it contains a child `<notBefore>`, and into an *After* formula if a child of type `<notAfter>` is present. For the other validity construct called `<validityTimeMetered>`, its subelement `<quantum>` stores the cumulated amount of time for which the resource may be used. Refer to figures 3.4, 3.5, and 3.6 for illustration. The `<cx:helper>` element corresponds to a *device* in OSL and it describes the software to be used, for example in the case of digital movie playback or print command. The elements we use from the XrML Content Extension are described in the following paragraph.

3.5.9 Rights

In case of the `<right>` element, the translation is as already mentioned rather easy. One creates an *Event* declaration in OSL. As mentioned before in section 3.3 this event is temporarily stored in an XrMLRight instance and the input arguments are added to it while the whole `<grant>` construct is parsed. After the whole conversion process, all the instances are written to the OSL output policy as *Event* declarations.

3.5.10 Fees

The `<sx:fee>` element can have one of two different children we currently support. All of them require the `<sx:to>` element, which contains the recipient of the money, as well as the `<sx:rate>`, which contains the amount of money to be payed.
The `<sx:paymentFlat>` corresponds to a flat rate and it is translated into an *Until* formula with a pay event. Unless the amount specified in this pay event has not been transfered, the event specified in the *Not* child of the *Until* formula mustn't be executed. A subelement of type `<sx:paymentPerUse>` is translated into an *Always* formula with the corresponding event to which the payment is bound. The exact amount is extracted from its `<sx:rate>` child. These translations are depicted in figures 3.7 and 3.8.

3.6 From OSL to XrMLc

For the other direction of translation, we need to restrict the possible OSL input policies such that they can be translated completely into our supported

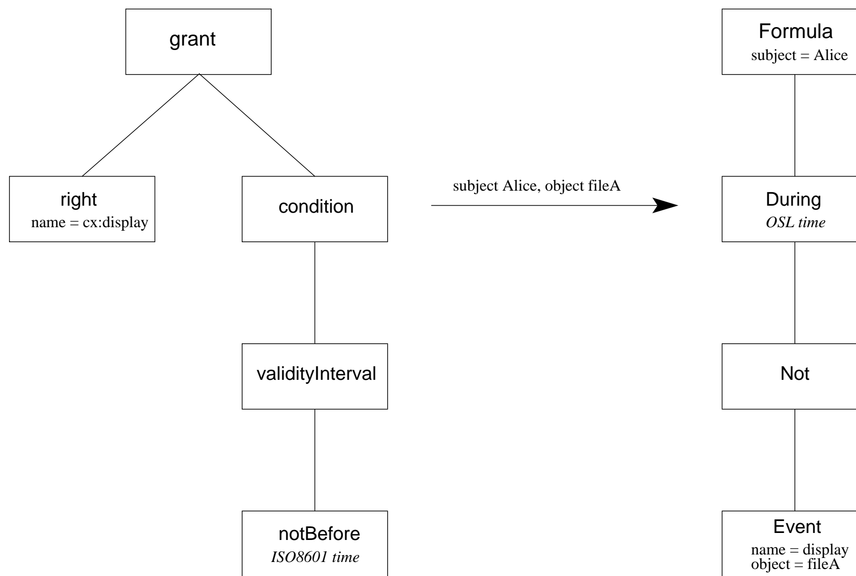

Figure 3.4: Translation of the `<notBefore>` node

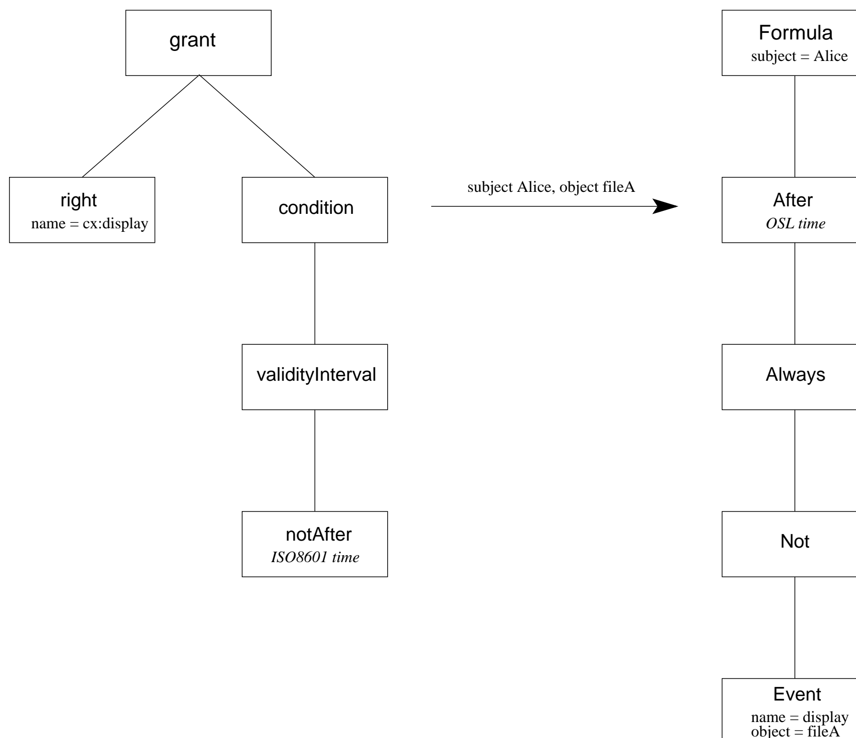

Figure 3.5: Translation of the `<notAfter>` node

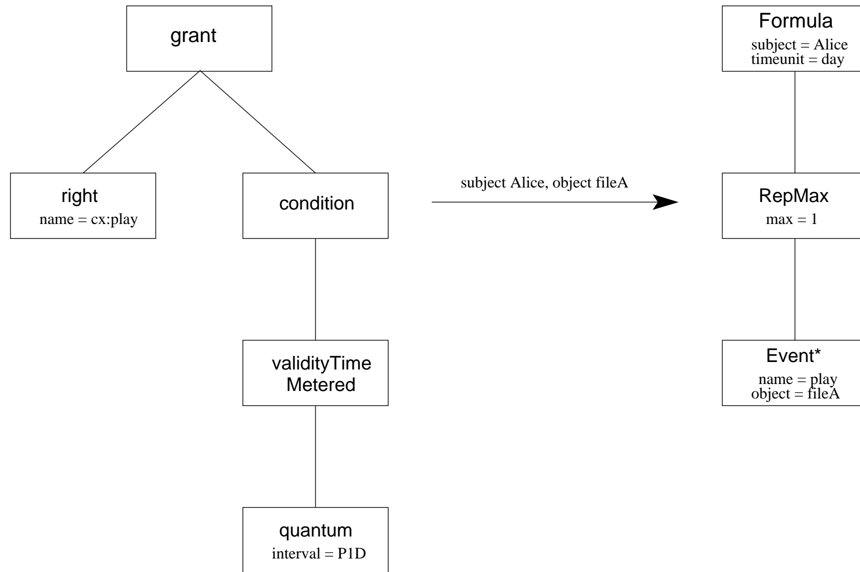

Figure 3.6: Translation of the `<validityTimeMetered>` node

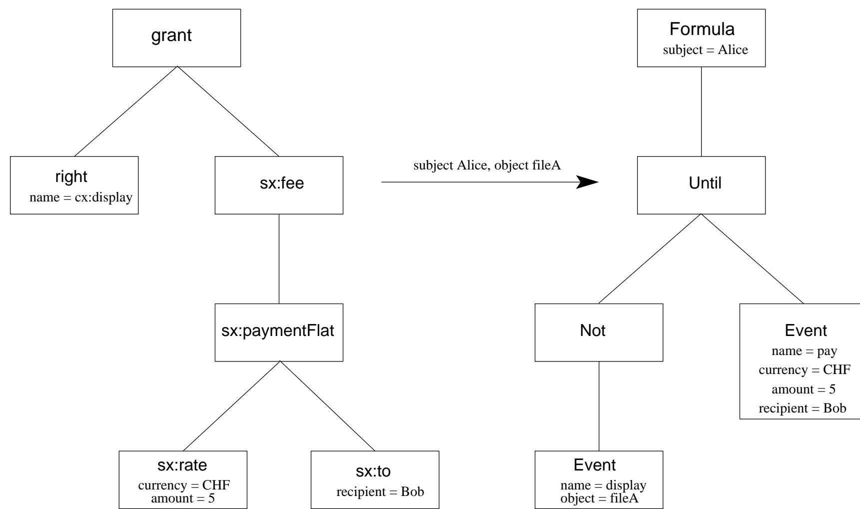

Figure 3.7: Translation of a flat payment.

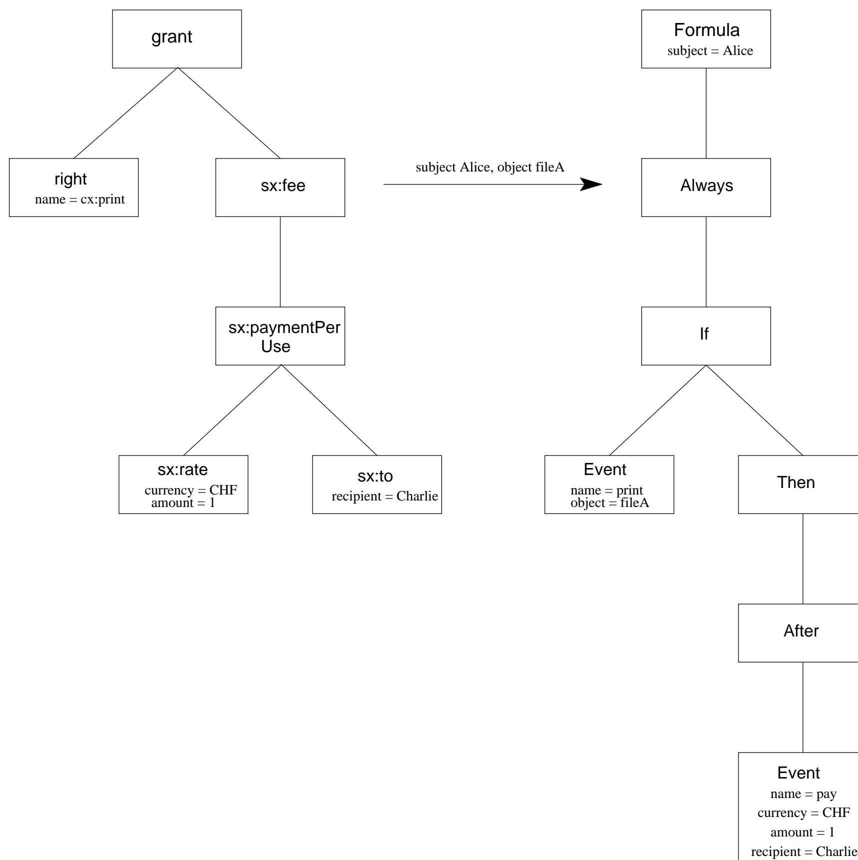

Figure 3.8: Translation of a payment per use.

XrMLc subset. Since we have started with the direction from XrML to OSL, we know how the OSL input must look like. Because OSL can express a specifc semantic meaning in several syntacticaly different ways, this provides us with a canonical from to which we restrict the possible range of OSL input policies.

3.6.1 General Idea

To parse the OSL policy we use the parser introduced in section 2.1. First, this parser builds a tree structure which is used for the translation. After that, the *EventDecl* and *OblDecl* blocks in the tree structure are processes by methods which we provide. The *PermitEvNames* formula from the the *OblDecl* section is looked up and for each usage found in it, a XrMLGrant instance is created and put into a hashtable. Then all the other *Obl* constructs are handled and the XrMLGrant elements in the hashtable are filled with the appropriate and needed data. At the end, each XrMLGrant is written into the XrML output policy. This is very similar to the approach taken for the other direction before in section 3.5.

3.6.2 Rights needed

In order to match the events listed in table 3.5.2 which we used before for the direction from XrML to OSL, we need the rights listed in table 3.2 taken from the XrML Content Extension. Each `<grant>` must contain one element of the abstract type `<right>` from this table.

<RIGHT> ELEMENT	SUBNODES TO BE CONSTRUCTED
<cx:backup>	keyHolder, digitalResource
<cx:delete>	keyHolder, digitalResource
<cx:display>	keyHolder, digitalResource, cx:helper
<cx:execute>	keyHolder, digitalResource, cx:helper
<cx:pay>	cx:paymentType
<cx:play>	keyHolder, digitalResource, cx:helper
<cx:print>	keyHolder, digitalResource, cx:helper

Table 3.2: The right elements with their the created sub nodes

3.6.3 Basic structure

Compared to the basic structure used for the other direction we simply turn it the other way around so to say. This means that the *Obligations* construct

from OSL is mapped to a `<license>` element in XrML and each *Event* declaration leads to the creation of a `<grant>` element. The *Obl* declarations are then parsed and the information for the grants is extracted from them and stored in the elements of the hashtable containing the XrMLGrant instances. A *RepMax* formula in OSL leads to the immediate construction of an `<sx:exerciseLimit>` construct and the *object* referred to in a *Obl* corresponds to the `<digitalResource>` needed for each `<grant>`. A *device* in an OSL policy maps to a `<cx:helper>` node.
A payment event in OSL depicted by a pay *Obl* is translated into a `<sx:fee>` with the appropriate type of payment. As already mentioned in the previous section we support `<sx:paymentFlat>` and `<sx:paymentPerUse>` at the moment.

3.6.4 The Event

The *Event* declarations are used merely for reference. As mentioned before in section 3.6.1, the needed XrML grants are extracted from the *PermitEvNames* obligation. The *EventDecl* must of course contain all the actions listed in the *PermitEvNames* formula, but it can have more events listed which are not used for any obligation in the *OblDecl* section. The XrML grants are filled with data which we extract from the obligations. This is described in the follow-up sections.

3.6.5 TimeDefault

Since there is no equivalent construct in XrML we do not handle it and simply read over it. Of course it could be used to set the default time unit used for all the rights if timing is involved and the `<grant>` containing the right does not provide its own time unit. But this will rarely, if at all be the case.

3.6.6 The Obligation

This construct is of course to most difficult and complex to handle. Each *Obl* formula is parsed and the associated event is used to look up the corresponding XrMLGrant in the hashtable containing the grants. Since we already know how to translate from XrMLc to OSL we take the structure of the output generated before to serve as a canonical form, or syntactic pattern respectively. This restricts the set of possible OSL input policies and we can detect the corresponding XrML constructs in each obligation and their corresponding meaning in XrML. The present information is extracted from the *Obl* construct and written into the `<grant>` body which is temporarily stored in the hashtable. At the end of the whole conversion process, all grant bodies are written to the XrMLc output policy.

Chapter 4

Application

In order to derive a proof of concept, we decided to enforce our translated OSL policies in an existing DRM mechanism which uses XrML policies. For this we picked Microsoft's RMS - Rights Management Services - since it implements usage control with the help of XrML policies and it is available as a download from Microsoft.[1] This of course enabled us to use the previously developed translator component. In this chapter, we describe the development of the RMS-enabled prototype-like text editor. It was developed using the C++ programming language and the MFC - Microsoft Foundation Classes.

4.1 Preparation

First, we set up a Windows 2003 server R2 and installed the RMS server SP2 in the active directory (AD) structure. Then we installed the RMS client SP2 on the client machine. The RMS SDK was also installed on the client machine instead on a third computer in order to simplify the development process. After verifying that everything was set up properly, we started the development of the demo application.

4.2 Implementation

We extended an existing simple text editor from CodeProject[2] using the API from the RMS SDK. The API itself provides useful examples in order to quickly start developing. We added two new classes to the text editor called *RMSHelper* and *XrML4RMS*. The former takes care of the RMS functionality by wrapping the RMS API calls needed for our prototype. The latter prepares the data from our translated XrML policy so that it can actually be enforced in the text editor application. An illustration is given in figure 4.1.

[1] Microsoft's RMS homepage, http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx

[2] The CodeProject homepage, http://www.codeproject.com/

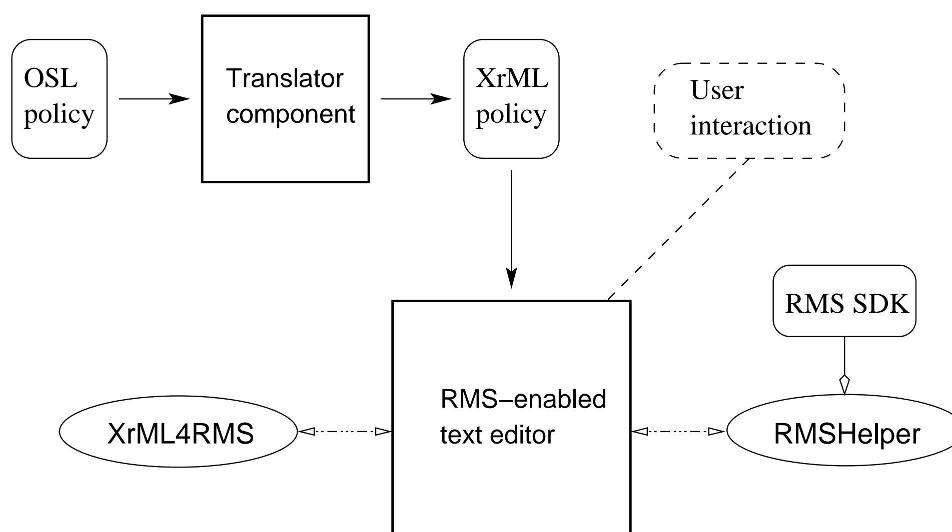

Figure 4.1: The RMS-enabled text editor

4.3 Functionality

The RMS-enabled editor uses the Windows ID to identify the currently loggend on user.

If a user is publishing content, she or he first writes the plain text and then specifies who may access the content and what exactly the consuming user is allowed to do with the content. This happens through the use of menu entries in the main menu bar.

There is a menu called *RMS Publish* to publish content and another menu called *RMS Consume* to consume content. Furthermore, there also exists a menu called *RMS User* where one can select the currently active user and a menu with the name *RMS Right* where one can select one of the currently supported rights. This is mainly used when content is being published. The last RMS-specific menu is called *RMS Mode*. It is used to enable or disable RMS functionality in the text editor.

If a user wants to consume content, she or he selects the RMS content to be accessed. The text editor then checks if the user is allowed to do so. If this is not the case, the access or action is denied, otherwise it is allowed.

Finally, the *XrML* menu is designed to load an XrMLc policy and set values from it to be enforced in the RMS-enabled text editor. First, the

XrMLc policy to be enforced is selected, and all the required values such as user, right, or content are extracted. At the end, the content is protected by writing a RMS package, as it is desribed in section 4.5, to disk.

The right currently implemented and checked by the text editor is print. More rights might be implemenented in the future.

4.4 RMS - Overview

In order to get a better understanding of RMS, we give an overview of the most important components.

4.4.1 Hierarchy

First of all, the client computer and user must be activated in the same RMS hierarchy. Currently, there exist two hierarchy types. The production hierarchy is designed for stable releases and needs an application manifest which is signed by Microsoft. The other type is the pre-production hierarchy which is provided for development and its application manifest can be signed by the developer himself. It is important to note that each time the application is rebuilt, a new manifest must be provided. Otherwise, certain calls to RMS API simply won't succeed.

4.4.2 Licenses

It is important to know about the different licenses which RMS uses. We explain the five most important license types.

Machine certificate The machine certificate identifies the client machine on which the RMS application is running and also serves for machine activation purposes.

Rights account certificate The rights account certificate (RAC) is used to identify and activate the client user in one of the above mentioned hierarchies. As mentioned above, the user and the machine must be activated in the same hierarchy.

End-user license The next important license is the end-user license (EUL) which is used for exercising rights, content encryption, and content decryption.

Client licensor certificate Finally, the client licensor certificate (CLC) is used to create the most important license which is called issuance license.

Issuance license The issuance license contains crucial usage and content information. Among this, for example, are the users with their associated rights in the form of EULs, a unique content identifier, and an access point for revocation lists.

4.5 Package format

Our application stores the usage controlled data in a custom package format. The first three entries depict the lengths of the unique identifier, signed issuance license, and encrypted content. Then follows the original length of the unencrypted, original plain text. Finally the block containing the actual data is listed. The picture 4.2 below serves as an illustration.

Package format
-lenght unique identifier -length signed isssuance license -length encrypted content -length plain text -unique identifier -signed issuance license -encrypted content

Figure 4.2: Text editor RMS package format

4.6 User identification

RMS has several options to identify a user. Two of them are the Windows ID (SID) and Passport ID (PUID). It is also possible to use a custom user identification system. The one we use at the moment is the standard Windows ID.

4.7 Content creation

A typical creation session starts by using the RMSHelper class to check if the currently logged on user and the machine itself have been activated in RMS. If not already done, they are both activated and a machine certificate and RAC for the current user are stored. Next a secure RMS environment is initialized using the application manifest. After that, an empty issuance license is created from scratch which is used for the content to be published.

The content publisher then fills all the users with their associated rights, e.g. their EULs into the issuance license. The unique identifier is created by the RMS server and added to the issuance license. Most importantly, the content creator sets himself as the owner of the content. When all the data has been set in the issuance license, it is finally signed. Then the content is encrypted and stored a in package together with the now signed issuance license, the original plain text length, and a unique identifier. Refer to figure D.1 in appendix D to see an illustration of the RMS creation process.

4.8 Content consumption

The consumption session also starts by checking whether the user and machine are already activated. The the user points to the package he would like to access and use. The application then checks if the currently logged on user has an EUL in the signed issuance license. If not the access is denied. In case there is an end-user license, a more refined license type called "bound license" is created which contains the rights for the current user. It is also used to decrypt the encrypt content from the package. Finally, the user may access and use the content according to the rights in the EUL which he was granted by the content publisher. Refer to figure D.2 in appendix D to get a better insight about the content consumption in RMS.

Chapter 5

Conclusion

5.1 Contribution

We have successfully implemented the translation component for OSL and XrML. Together with the translation component from [2], which translates between OSL and ODRL, this strengthens our proof of concept that OSL may be used as an intermediate format for translation between different Right Expression Languages. By this, we can potentially reduce the number of needed translation components in a large DRM system, which has n different policy languages, from $O(n^2)$ to $O(n)$. Furthermore, we have effectively demonstrated that the translated policies can be used in existing DRM standards. The RMS-enabled text editor from chapter 4 serves as prototype and proof of concept.

5.2 Limitations

There are a few limitations considering the XrML support in RMS. Currently only XrML 1.2 is supported whereas the latest available specification has version 2.0 [6]. This means that some constructs and usage conditions such as the exercise limit for example are not directly supported. To circumvent this limitation and implement an exercise limit, one has to store the limit in a right specific array with additional, custom information called *extendedInfo*.

5.3 Future Work

Considering future work, one might consider translations between OSL and other Rights Expression Languages. This would strengthen our proof of concept that OSL may be used as an intermediate for translations between different DRM license standards. This would immediately lead to the possibility to create a demo application capable of translating between several DRM standards. Digital content providers supplying different end-user platforms

ranging from the smallest cell-phone to a multimedia entertainment system could make use of this feature. Additionaly, the work on the RMS prototype in this thesis could be continued and the package format, for example, could be converted to be used in other DRM systems.

5.4 Personal Experience

After having achieved an overview of the research area based on the reference material and additional resources from the internet, the most difficult part was to understand the OSL grammar definition and to see how it is used in the existing monitor implementation. Once I grasped the whole structure, I relatively quickly adapted the grammar and made it more readable and general.

Next, the implementation was surprisingly easy and I was able to draw upon my programming experience, most importantly with Java. It was new to me to use generic types in Java. But since I already had used them in Eiffel, this posed no problem at all. It was interesting to combine several components. One component was the OSL grammar and the EBNF structure in it. The CoCo framework was used to construct a parser and scanner out of it. Another component was the XrML language and the JAXP. All together form this translation framework for usage control policies.

While I was developing the RMS-enabled text editor, I learned more about C++ (including MFC) and I made frequent use of the Microsoft's RMS forum and online documentation. I also found a RMS developer blog which gave me some clues to certain problems. I even found somebody at Microsoft through the email link on the blog website. The support and suggestions I received through this contact were extremely helpful for the development process. When I asked a question considering a certain API call, for example, I always received a quick, to the point answer, this was really amazing.

Furthermore, we ordered a RMS Evaluation Kit from Microsoft. Unfortunately, it arrived just at the completion of this thesis. Anybody interested to do future work on RMS is strongly suggested to have a look at the material supplied in the evaluation kit.

To sum it up, I can say that I really enjoyed writing this thesis, most of all because at some times it was really demanding and I was able to improve my programming skills and knowledge about the Microsoft server architecture and Active Directory system. Most notably I learned about

Usage Control, which will definitively be useful for my future career.

Finally, I would like to thank my mentors for the constant support, encouragement, and feedback I received.

Bibliography

[1] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed Usage Control. *CACM*, September 2006.

[2] Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In *Proc. 12th European Symposium on Research in Computer Security*. Springer-Verlag, 2007. To appear.

[3] V. Bättig. Monitoring usage control requirements. Master's thesis, Department of Computer Science, ETH Zürich, 2006.

[4] Jose Prados, Eva Rodriguez, and Jaime Delgado. Interoperability between different rights expression languages and protection mechanisms. In *Proc. of the First International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution (AXMEDIS'05*. IEEE Computer Society, 2005.

[5] ContentGuard. Profiling MPEG Rights Expression Language: Concept, Approach and Applications. Technical report, ContentGuard Holdings, Inc., 2003. Available at `http://www.xrml.org/reference/MPEG-REL-Profiling.pdf`.

[6] ContentGuard. XrML 2.0 Technical Overview. Technical report, ContentGuard Holdings, Inc., March 2002. Available at `http://www.xrml.org/reference/XrMLTechnicalOverviewV1.pdf`.

[7] Manuel Hilty, David Basin, and Alexander Pretschner. On obligations. In *Proc. 10th European Symposium on Research in Computer Security*, LNCS 3679, pages 98–117. Springer-Verlag, 2005.

[8] Brenton Cooper and Paul Montague. Translation of rights expressions. In *Procedings of the 4th Australasion Information Security Workshop (AISW2005)*, volume 44 of *Conferences in Research and Practice in Information Technology*, pages 137–144. Australian Computer Society, Inc., 2005.

[9] Xin Wang, Guillermo Lao, Thomas DeMartini, Hari Reddy, Mai Nguyen, and Edgar Valenzuela. XrML - eXtensible rights Markup Language. In *ACM Workshop on XML security (XMLSEC '02)*, pages 71–79. ACM Press, November 2002.

[10] CotentGuard. eXtensible rights Markup Language (XrML) 2.0 Specification. Technical report, ContentGuard Holdings, Inc., November 2001. Available at `http://www.xrml.org/get_XrML.asp`.

[11] Microsoft Corporation. Technical overview of windows rights management services for windows server 2003, April 2005. Available at `http://www.microsoft.com/windowsserver2003/techinfo/overview/rmenterprisewp.mspx`.

List of Figures

1.1 Provider components overview 2

3.1 Translator component 8
3.2 XrMLc overview 11
3.3 Translation of the exercise limit 13
3.4 Translation of the `<notBefore>` node 15
3.5 Translation of the `<notAfter>` node 15
3.6 Translation of the `<validityTimeMetered>` node 16
3.7 Translation of a flat payment. 16
3.8 Translation of a payment per use. 17

4.1 The RMS-enabled text editor 22
4.2 Text editor RMS package format 24

D.1 RMS Content Creation 41
D.2 RMS Content Consumption 42

Appendix A

Relax NG Schema

This is the Relax NG Schema [1] of the XrML subset, it is a RelaxNG compact syntax definition of XrMLc:

```
default namespace = "http://www.xrml.org/schema/2001/11/xrml2core"
namespace sx = "http://www.xrml.org/schema/2001/11/xrml2sx"
namespace cx = "http://www.xrml.org/schema/2001/11/xrml2cx"
namespace dsig = "http://www.w3.org/2000/09/xmldsig\#"


start =

license = element license { Issuer , Grant* }

Issuer  = element issuer { KeyHolder }

Grant = element grant { KeyHolder,
                          DigitalResource ,
                         (ExerciseLimit),
                          Right,
                         (Helper),
                         (Fee),
                         (Condition)
                        }

KeyHolder = element keyHolder { element info {KeyHolderName} }

KeyHolderName = element dsig:keyName { text }

DigitalResource = element digitalResource { ResourceLocation }

ResourceLocation = element nonSecureIndirect {attribute URI {text} }

ExerciseLimit = element sx:exerciseLimit { Count }

Count = element sx:count { text }

Right = element cx:backup |
        element cx:delete |
```

[1] Relax NG home page, http://relaxng.org/

```
        element cx:execute |
        element cx:display |
        element cx:pay |
        element cx:play |
        element cx:print

Helper = element cx:helper { KeyHolder }

Fee = element sx:fee { FeeType { FeeContent } }

FeeType = element sx:paymentPerUse |
          element sx:paymentFlat

FeeContent =  { element sx:rate {attribute currency { text }, text },
                element sx:to {text}
              }

Condition = element validityInterval { (ValidityContent) } |
            element validityTimeMetered { Quantum }

ValidityContent = { element notBefore { text } |
                    element notAfter { text } |
                    element notBefore { text },
                    element notAfter { text }
                  }

Quantum = { element quantum { text } }
```

Listing A.1: XrMLc Relax NG schema

Appendix B

OSL example policy

```
Obligations {

EventDecl {

  Event (print,usage) For (user,object,device)
  Event (display,usage) For (user,object,device)
  Event (delete,usage) For (user,object)
  Event (play,usage) For (user,object,device)
  Event (pay,other) For (recipient,currency,amount)

}

TimeDefault {day}

OblDecl {

  Obl [Alice] [day] {
    RepMax [20] {print(object=fileA)}
  }

  Obl [Alice] [day] {
    Always {If {print(object=fileA)}
      Then {Within [1] {pay(recipient=Charlie,currency=CHF,amount=1)} }
    }
  }

  Obl [Alice] [day] {
    RepMax [5] {display(object=fileA,device=WMediaPlayer)}
  }

  Obl [Alice] [day] {
    Until {Not {display(object=fileA,device=WMediaPlayer)},
        pay(recipient=Bob,currency=CHF,amount=5)
    }
  }

  Obl [Alice] [day] {
    And {During [10] {Not {display(object=fileA,device=WMediaPlayer)} },
      After [20] {Always {Not {display(object=fileA,device=WMediaPlayer)}}}
    }
  }
```

```
  Obl [Alice] [day] {
    PermitEvParams {WMediaPlayer} For {device} In {display(object=fileA)}
  }

  Obl [Alice] [day] {
    PermitEvParams {MyGarbageCollector} For {device} In {delete(object=fileA)}
  }

  Obl [Alice] [day] {
    RepMax[1] {*play(object=fileA)}
  }

  Obl [Alice] [day] {
    PermitEvNames {print,play,display,delete} For {(object=fileA)}
  }

}
```

Listing B.1: OSL example policy

Appendix C

XrML example policy

```
<?xml version="1.0" encoding="UTF-8" ?>
<license
 xmlns="http://www.xrml.org/schema/2001/11/xrml2core"
 xmlns:sx="http://www.xrml.org/schema/2001/11/xrml2sx"
 xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:cx="http://www.xrml.org/schema/2001/11/xrml2cx"
 xsi:schemaLocation="http://www.xrml.org/schema/2001/11/xrml2cx..\schemas\xrml2cx.xsd">

  <grant>
    <keyHolder><info>
      <dsig:keyName>Alice</dsig:keyName>
    </info></keyHolder>
    <digitalResource><nonSecureIndirect URI="fileA"/></digitalResource>
    <sx:exerciseLimit>
      <sx:count>20</sx:count>
    </sx:exerciseLimit>
    <cx:print/>
    <sx:fee>
      <sx:paymentPerUse>
        <sx:rate currency="CHF">1</sx:rate>
        <sx:to>Charlie</sx:to>
      </sx:paymentPerUse>
    </sx:fee>
  </grant>

  <grant>
    <keyHolder><info>
      <dsig:keyName>Alice</dsig:keyName>
    </info></keyHolder>
    <cx:display/>
    <digitalResource><nonSecureIndirect URI="fileA"/></digitalResource>
    <sx:exerciseLimit>
      <sx:count>5</sx:count>
    </sx:exerciseLimit>
    <cx:helper>
      <keyHolder>
        <info>
          <dsig:keyName>WMediaPlayer</dsig:keyName>
        </info>
      </keyHolder>
```

```xml
    </cx:helper>
    <sx:fee>
      <sx:paymentFlat>
        <sx:rate currency="CHF">5</sx:rate>
        <sx:to>Bob</sx:to>
      </sx:paymentFlat>
    </sx:fee>
    <validityInterval>
      <notBefore>2006-10-09T00:00:00</notBefore>
      <notAfter>2006-10-19T00:00:00</notAfter>
    </validityInterval>
  </grant>

  <grant>
    <keyHolder><info>
      <dsig:keyName>Alice</dsig:keyName>
    </info></keyHolder>
    <digitalResource><nonSecureIndirect URI="fileA"/></digitalResource>
    <cx:delete/>
    <cx:helper>
      <keyHolder>
        <info>
          <dsig:keyName>MyGarbageCollector</dsig:keyName>
        </info>
      </keyHolder>
    </cx:helper>
  </grant>

  <grant>
    <keyHolder><info>
      <dsig:keyName>Alice</dsig:keyName>
    </info></keyHolder>
    <digitalResource><nonSecureIndirect URI="fileA"/></digitalResource>
    <cx:play/>
    <validityTimeMetered>
      <quantum>P1D</quantum>
    </validityTimeMetered>
  </grant>

  <issuer>
    <keyHolder><info>
      <dsig:keyName>DataProvider</dsig:keyName>
    </info></keyHolder>
  </issuer>

</license>
```

Listing C.1: XrML example policy

Appendix D

RMS Content Processing

We included the two overviews from the RMS SDK help file. First one should have a look at the content consumption process since these steps are all also required for the content publishing process.

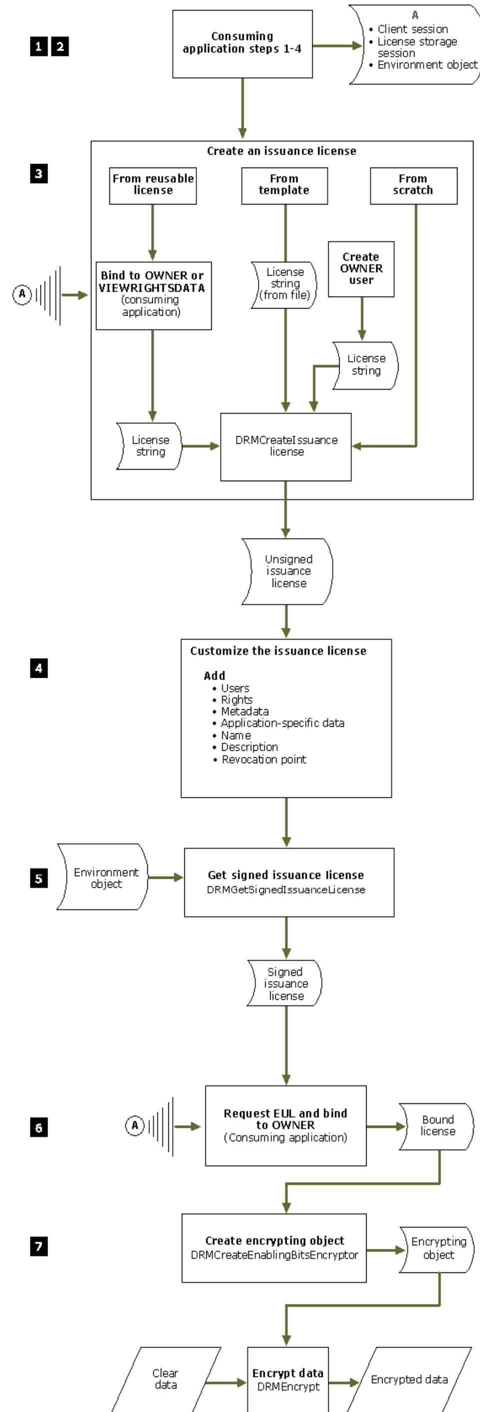

Figure D.1: RMS Content Creation

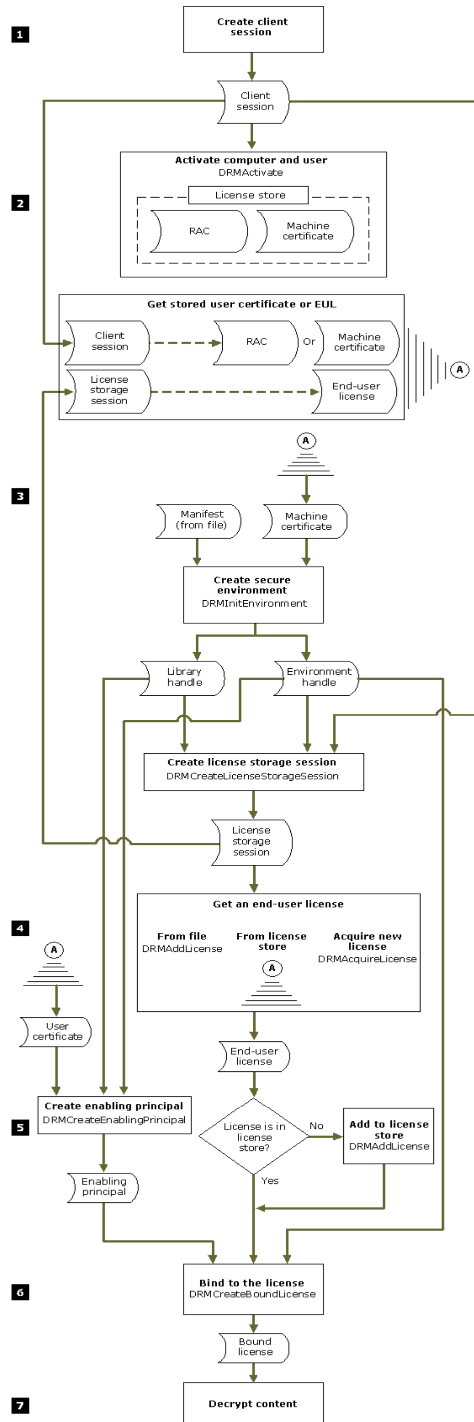

Figure D.2: RMS Content Consumption