NTNU
Norwegian University of Science and Technology

Design and use of XML formats for the FRBR model

Anders Gjerde

Master of Science in Informatics
Submission date: June 2008
Supervisor: Trond Aalberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The objective of this thesis is to examine how XML can be used with the FRBR model as a framework to store bibliographic records. It should present design criteria for how an XML Schema Definition could be structured and also present different alternatives of implementation. The criteria and chosen implementation alternatives should be demonstrated using examples of their advantages over other alternatives. The areas of application and flexibility of such a format as well as reasons to adopt it should be thoroughly discussed. The assessment of which needs are present from such a format must be done as well as design choices in correlation to the needs.

The exact problem specification was originally as follows:

The examination of XML design criteria, analysis and evaluation of alternative XML implementations for the FRBR model of bibliographic information.

Throughout the thesis the problem specification was reevaluated and slightly reformulated into the following:

The identification of relevant XML design critera and the evaluation of different implementation alternatives for the FRBR model. Analysis of XML Schema properties, identification of needs related to a metadata format, and the application of statistics to support design choices.

Abstract

This thesis aims to investigate how XML can be used to design a bibliographical format for storage of records better in terms of hierarchical structure and readability. It first presents introductory theory regarding the techniques which make the fundament of bibliographical formats and what has previously been in use. It also accounts for the FRBR model which is the conceptual framework of the format presented here. Throughout the thesis, several important XML design criteria will be presented and examples as to why these are important to consider when constructing a bibliographical format with the use of XML. Different implementation alternatives will be presented, with their advantages and disadvantages thoroughly discussed in order to establish a solid foundation for the choices that have been made. After having done this study, an XSD (XML Schema Definition) has been made according to the best practices that have been uncovered.

The XSD is based on the FRBR Model, although it is slightly changed to accommodate the wishes and interests of librarians. Most noteworthy of these changes is that the Manifestation element has been made the top element with the Expression and Work elements hierarchically placed beneath Manifestation in that order. It maintains a MARC-based datatag structure, so that librarians who are already used to it will not have to readjust to another way of structuring the most common datafields. Relationships and other attributes however, are efficiently handled in language-based elements and the XSD accommodates new relationship types with a generic relation element.
XSLT has been used to transform an existing XML database to conform to the XSD for testing purposes. Statistics have been collected from the database to support design choices.

Depending on what the users' needs are, there are many different design choices. XML leads to more readable records but also takes up much space. When using XML to describe relational metadata, relationships can be expressed using hierarchical storing to a certain degree, but ID/IDREF will have to be used at some point to avoid infinite inclusion of new records. ID/IDREF may also be used to improve readability or save storage space. Hierarchical storing leads to many duplicated records, especially concerning Actors and Concepts. When using XML, one must choose the root element of the record structure according to which entity is the point of interest. In FRBR, there are several reasons to choose Manifestation as the root element as it is the focal point of a library record.

Preface

This work was done by Anders Gjerde during autumn 2007 and spring 2008 at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor, associate professor Trond Aalberg, who has given vital input throughout the entire period. His experience and knowledge with the concepts covered in this thesis helped to keep me on the right path and his guidance inspired me to think differently and see new opportunities.

Thanks also to my family and friends for their support and encouragement.
Ingen nevnt, ingen glemt.

Trondheim, 01.06.2008
Anders Gjerde

Table of Contents

1 Introduction....1
  1.1 Background and Motivation....1
  1.2 Problem Specification....1
  1.3 Goals....2
  1.4 Approach....2
  1.5 Outline....2

2 Theory....5
  2.1 Metadata....5
  2.2 FRBR....6
  2.3 XML....8
  2.4 Formats for storing bibliographic records....10
    2.4.1 MARC....10
    2.4.2 MARCXML....12
    2.4.3 MarcXchange....12
    2.4.4 MODS....12
    2.4.5 Dublin Core....14
  2.5 Schema Definition Languages....15
    2.5.1 DTDs....15
    2.5.2 XML Schema....16
    2.5.3 RELAX NG....19
  2.6 XPath....21
  2.7 eXtensible Stylesheet Language Transformations....21

3 Design Criteria....23
  3.1 XML as a metadata format....23
  3.2 Choosing the root element....24
  3.3 Expressing relationships in XML....25
    3.3.1 Hierarchical storing....25
    3.3.2 Using ID/IDREF....26
  3.4 Choices regarding use of ID/IDREF....27
    3.4.1 Use according to need....27
  3.5 Including records outside of hierarchy tree....29
  3.6 Exchange of bibliographic records....30
  3.7 Validation....31
  3.8 Readability of XML....32
  3.9 Expressing metadata fields....33

4 Application and examples of XML design criteria....35
  4.1 Choosing the root element....35
  4.2 Expressing relationships in XML....37
    4.2.1 Hierarchical storing between Manifestation, Expression and Work....38
    4.2.2 Using IDREF between Manifestation, Expression and Work....42
    4.2.3 Relationships seldom used....46
  4.3 Choices regarding use of ID/IDREF....46

4.4 Including records outside of hierarchy tree....48
4.5 Readability of XML....53
4.6 Expressing metadata fields....55
4.6.1 Using MODS....55
4.6.2 Using Dublin Core....57

5 The XML Schema Definition....59
5.1 Overview of the XML Schema Definition....59
5.2 Relation between elements....65

6 Analysis and Discussion....69
6.1 Application of the FRBR model....69
6.2 Application of MARC....70
6.3 Testing....71
6.3.1 Area of use....71
6.3.2 General Structure....71
6.3.3 Disadvantages....74
6.3.4 Application in this thesis....74
6.4 Expressing Relationships....75
6.5 Choices regarding use of ID/IDREF....77
6.6 Storage optimization using ID/IDREF....81
6.6.1 Statistics : Work is realized through Expression....83
6.6.2 Statistics : Expression is embodied in Manifestation....84
6.6.3 Statistics : Work has as subject Concept....84
6.6.4 Statistics : Person has created Work....85
6.6.5 Statistics : Duplication in full hierarchy....85
6.7 Size comparison: MARC vs. FRBRized....87

7 Conclusion....89
7.1 Summary....89
7.2 Results....89
7.3 Evaluation....89
7.4 Future Work....91

Bibliography....92

Appendix....94
XML Schema Definition code....94
Legend for Liquid XML....100
Glossary and abbreviations....101

1 Introduction

1.1 Background and Motivation

Metadata records has been a part of the library world for decades. Library metadata has been passed down from the cataloging records stored on paper card to the machine readable MARC records to the newer XML-based formats. Flexibility and readability has been improved. With the possibility to use XML and its connected set of tools, one has ample opportunity to design an XML-based format out of one's own desire.

The FRBR model has put the spotlight on functional requirements and properties which should exist within library metadata and this has made the weaknesses of earlier cataloging more apparent.

The motivation to conduct this study has been to produce an FRBR based XML format which could be an improvement from other formats of bibliographic storage. The most used format, MARC, is based on the traditional way of thinking in terms of cataloging and processing of library entries. However, it is a format which is dependent of machines to be fully understandable, hence the moniker **MA**chine **R**eadable **C**ataloging (MARC). With the advent of the **F**unctional **R**equirements for **B**ibliographic **R**ecords (FRBR) model, the bibliographic world has been made aware of new ways of thinking and structuring library records. Subsequently, the disadvantages of the MARC format, such as complexity, lack of readability, lacking types of relationships between records, have become more apparent. Formats which have not implemented the ideas found in the FRBR model, will not possess the granular distinction between different types of records and will also lack relationships.

Because of these shortcomings in other formats, this thesis takes aim to discover XML design criteria and use them as a basis to determine a well-designed XML Schema Definition.

1.2 Problem Specification

The original problem specification was the following:

The examination of XML design criteria, analysis and evaluation of alternative XML implementations for the FRBR model of bibliographic information.

Throughout working with the problem, certain ideas changed and the problem specification required a reformulation:

The identification of relevant XML design critera and the evaluation of different implementation alternatives for the FRBR model. Analysis of XML Schema properties, identification of needs related to a metadata format, and the application of statistics to support design choices.

1.3 Goals

The main goals in this thesis are:

- Identify relevant XML design criteria for the development of an XML Schema for FRBR
- Develop, test and evaluate an XML Schema Definition for storage and exchange of library metadata.
- Identify needs that correlate with design criteria.

These goals are the primary and essential objectives which this thesis aims to complete. These are the most important contributions in that they account for good design choices when using FRBR and XML. Before these can be fully complete, there are a set of other goals which the primary goals presuppose.

1.4 Approach

In order to approach the goals there are certain tasks that are involved:

- Investigate existing metadata formats.
- Familiarise one self with XML and available tools.
- Examine different XML Schema Definiton languages.
- Use a schema definition to validate the new XML format.
- Use FRBR in an XML context.
- Use XSLT in correlation with a test collection to verify the new XML format's applicability to the test collection.
- Use test collection as statistical material and as a basis to make decisions regarding implementation recommendations.

1.5 Outline

This thesis is the result of a one year long study done to examine how XML can be used to store bibliographic records with the FRBR model as the framework. Introductory theory concerning metadata, MARC and XML has been studied and important XML Design Criteria has been discussed to establish a fundament upon which an FRBRized XML Schema Definition (XSD) has been suggested. The benefits of such a format is to achieve a bibliographic record storage which is easier to use and easier to learn, with higher readability, greater focus on important information and which is in concord with the highly acclaimed FRBR model.

CHAPTER 1 introduces the background, motivation, problems, challenges and goals in the thesis.

CHAPTER 2 gives introductory theory concerning metadata, FRBR, existing XML formats, langugages for XML Schema Definition and XML transformation languages.

CHAPTER 3 gives XML design criteria and aspects to consider when designing a metadata format in XML whilst using FRBR as a conceptual model.

CHAPTER 4 gives examples of the XML design criteria.

CHAPTER 5 presents an XML Schema Definition of a format for storing bibliographic information.

CHAPTER 6 contains the analysis and discussion part. Here is performed an evaluation of how well design criteria would work in different situations and in different needs. Here is also described the test collection which has been used. It was used to check that every piece of information in it could be contained in the new XML Schema Definition. It was also used for statistical information regarding duplication.

CHAPTER 7 gives the summary, results, evaluation and future work.

2 Theory

2.1 Metadata

The word metadata is composed of the Greek 'meta' and the Latin 'data', where 'meta' means 'after'. Therefore, one can say that metadata is data which comes after the original data, either as an appendix detached from the data it is describing or as a part of the main data. All files on a computer are graced with additional information which complements the data itself. Documents, images, music and video to name a few, all have descriptive data which tells you something about its attributes. This could be time of creation, size, resolution, last edited, edited by whom, data format and so forth. In the library world, which is the main focus of this thesis, metadata mainly revolves around information such as title, author, publisher, number of pages, subject, concept and many more. To put it shortly, metadata is data about data.

So why do we need metadata? Is it important to have additional information connected to an object, be it digital or not? What would happen if there only was a name or an ID connected to a specific object? Could we really know anything about it?

Metadata is necessary to store knowledge about an object so that this knowledge is easily available at any time in the future. It is necessary when we have a huge collection of data and we want to easily be able to search, locate and retrieve a specific object. It is necessary when we want to find objects that belong to a certain category or are made by a specific person or corporation. It is necessary when we want to locate objects which belong to a certain age or a certain year. If we want to find objects of a certain format or objects placed at a certain location, metadata is necessary. In any case, metadata must be recorded and connected to an object from its very beginning until its expiration.

Metadata could be structured and stored in any way thinkable. The most elementary design view of metadata is considering where you place it. Metadata can either be a part of the data it is describing or as a separete record.

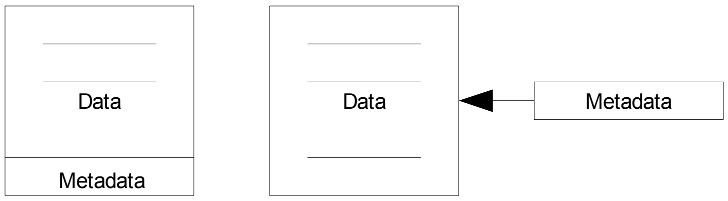


Figure 1: Metadata as a part of main data or as a separate record

Metadata in the digital world can easily be stored together with the data itself. In the library world however, when you have physical copies of books, manuscripts, compositions, drawings, etc. to deal with, metadata is stored separately. Nowadays, metadata is mostly stored in computers, and even data contained within books, compositions and drawings can be stored digitally. Why then store metadata separately from the original data?

When it comes to exchanging metadata records between different institutions, one will quickly see the advantages of keeping records separate. Exchanging only metadata, which takes up much less space than the actual data, is a lot less time consuming and requires much less resources. Another important reason to have metadata is to be able to easily exchange information across institutions and databases. A metadata format has to support properties enabling it to easily be exchanged, such as being able to be stored separately from the data it is describing and such as being able to be segmented in batches of metadata records so that you can exchange any given number of entries.

In order to give a better description of a format for storing bibliographic records and the features it must encompass, a look at the FRBR model must be taken.

2.2 FRBR

Functional Requirements for Bibliographic Records (FRBR)[6] is a conceptual model which was made by the International Federation of Library Associations and Institutions (IFLA) during the years 1990-1997. The model was conceived because of the lack of guidelines to clearly express what it was necessary for a bibliographic record to contain. Its purpose therefore, is to be a framework for what a bibliographic record should have of properties, and what is to be expected from such a record when it comes to relationships to other entities. The model has received a lot of attention from both within and outside of the library community, and is considered by many to be an important foundation to build upon when making the library systems of the future[2] . The former library community has mainly been preoccupied with keeping track of physical copies of books and the properties connected directly to what would be described as Manifestations in FRBR. Describing relations between Manifestations that have the same origin or the same inspiration has previously been less prioritized as well as describing a library entry's relations to other Works. Some of these weaknesses are what the FRBR models tries to remedy. As this model is the conceptual backbone of the work done in this thesis, it requires an introduction. The subjects concerning FRBR which will be described here are the main outlines, and will only provide a superficial view of the model. If the reader should wish to learn more about the FRBR model, the FRBR report made by IFLA[6], is strongly recommended.

The FRBR model

The FRBR model consists of three groups of entities. These represent the different abstract or physical objects which IFLA considers to be the most important ones for bibliographic entries. The three groups are:

Group 1: The product model
Group 2: The responsibility model – entities responsible for the product
Group 3: The subject model – entities which are the subjects of the product

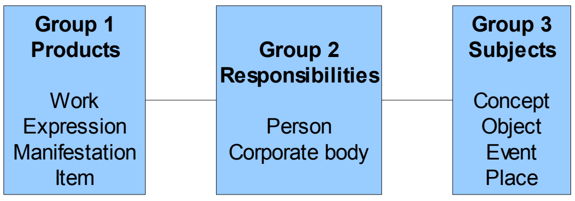


Figure 2: The FRBR conceptual data model

Group 1: The product model

This model consists of four entities; Work, Expression, Manifestation and Item.

Work is the distinct intellectual or artistic Work on an abstract level. This entity is intended to describe the idea or conceptualisation that is the fundament of a Work.

Expression is the intellectual or artistic realisation of the Work. For instance is the Concept behind the theatre play 'Pygmalion' and 'My Fair Lady' the same, therefore they are a part of the same Work. But since they differ in the artistic realisation, due to the fact that 'Pygmalion' is a play and 'My Fair Lady' is a musical, they are two different Expressions of the same Work. Likewise will the different translations of Henrik Ibsen's 'Et dukkehjem' be different Expressions of the same intellectual Work.

Manifestation is the physical design of an Expression of a Work. This includes such Manifestations as films, books, sound recordings and so forth. Dan Brown has released 'The Da Vinci Code' in different editions, at least a hardback, pocketbook and illustrated edition to name a few. The differences in format, layout, fonts and usage of illustrations make these editions three Manifestations of the same Expression.

Item is the physical copy of a Manifestation. In other words, the books in the libraries and the books you purchase in the store, are items in the FRBR sense.

Work and Expression are abstract entities and Manifestation and Item are physical entities. The reason for this distinction is to separate different properties and functionalities into appropriate entities. With this split into four entities, one is able to link Works to Expressions that draw on the concepts of or is a translation of that Work, link Expressions to Manifestations that display varieties in layout of that Expression, and link Manifestations to different physical copies, i.e. Items. Prior to the introduction of this model, these kinds of relations were not being expressed in a standardized way or even not expressed at all.

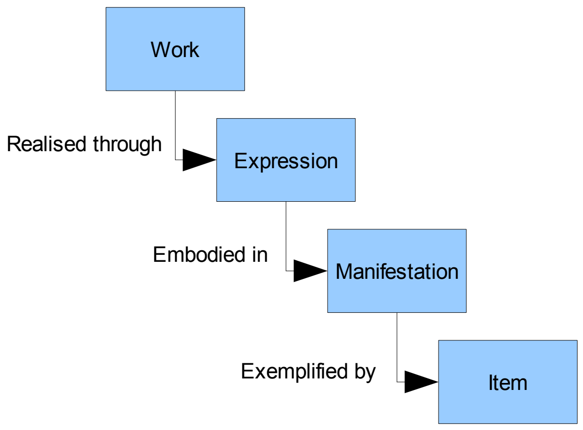


Figure 3: The FRBR product model

Group 2 : The Responsibility Model

This model consists of two entities; Person and Corporate body

Person is an entity which comprises all of the individuals described in bibliographic catalogues. This could be writers, co-writers, composers or people who are the subject of the Work.

Corporate body is the group of people who are identified with a specific name and act as a unit. Correspondingly with the entity Person, the Corporate body entity includes corporations in a bibliographic entry, no matter what function they have.

This model is associated to the product model, in that a Person or Corporate body is responsible for the creation of a Work, the realisation of an Expression, the production of a Manifestation and the ownership of an Item.

Group 3 : The Subject Model
This model consists of four entities; Concept, Place, Event and Object.

Concept is an entity which can contain any type of abstract Concept in connection to a Work. This could be ideologies, philosophies, areas of knowledge and science, theories, processes, techniques and so forth.
Object is any Object which is in connection to the Work.
Event is to describe any type of action or Event in connection to the Work, not only short-term ones, but also long-term historic events, eras and periods of time.
Place comprises all kinds of Places, terrestrial or extra terrestrial, historic and contemporary, geographic or geopolitical jurisdictions.

The FRBR conceptual model is a suitable starting point when developing a format for metadata storage. The decision as to which way the format could be stored was determined by an evaluation of a set of tools and languages that lay before us. Concerning a lot of metadata formats today, XML is used and there are many flexible tools available to operate it. Subsequently, XML became the choice that was most convenient when developing, testing and implementing a format for metadata.

2.3 XML

XML was developed in 1996 by an XML Working Group under the supervision of the World Wide Web Consortium (W3C)[3]. It is a restricted form of the Standard Generalized Markup Language (SGML). Some of the design goals for XML are that it should be easy to create, read and edit for humans, be straightforwardly usable and support a wide variety of applications both on and off the Internet, and that it should be kept simple and be quick to use.
XML documents are made up of storage units called elements which contain data. These elements consist of two tags, a start tag and a end tag. Other names for element could be entity, but the word 'element' will be used here. XML is very similar to HyperText Markup Language (HTML), which is quite logical since they both derive from SGML, with the big difference that you are able to define your own elements in XML. These elements can be organised hierarchically, giving them the possibility to act as *parent, child, sibling, ancestor* or *descendant.*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<recordstore>
  <record>
    <title>Dark Side of the Moon</title>
    <artist>Pink Floyd</artist>
    <year>1973</year>
  </record>
  <record>
    <title>Automatic for the People</title>
    <artist>R.E.M.</artist>
    <year>1992</year>
  </record>
</recordstore>
```

Figure 4: An XML document

The first line contains the XML declaration which defines the XML version and character encoding used in the document.

The root element <recordstore>, as well as being the root of the document, is also the *parent* of <record> and *ancestor* of <title>. <title>, <artist> and <year> are the *children* of <record>, and they are therefore *siblings*, as well as being the *descendants* of <recordstore>.

An XML-document can have a Document Type Definitions (DTDs) or XML Schema Definition (XSD) attached, which gives restrictions on its structure. One is for example able to define the hierarchical organization of the elements and how many occurrences each element is allowed to have. When an XML-document conforms to the DTD or XSD, it is considered *valid*. When an XML-document conforms to the general rules of structure and syntax which are inherent from the W3C XML 1.0. Recommendation, it is considered *well-formed*. An example is the following:

```
<title>Dark Side of the Moon<title>.
```

This is syntactically incorrect because the closing-tag is missing a backslash, therefore this is not well-formed and is in fact not considered to be XML. Let us look at another example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<recordstore>
  <record>
    <title>Dark Side of the Moon</title>
    <artist>Pink Floyd</artist>
    <artist>Roger Waters</artist>
    <year>1973</year>
  </record>
</recordstore>
```

Figure 5: An XML document

Let us assume that we have a DTD or an XML Schema which only allows one occurrence of the <artist> element within the <record> element. An explicit example of this will be given in 2.5.
Regarding the example above, the <artist> element occurs twice, which is in conflict with our DTD or Schema. However, since this sample of XML conforms to all the Well-Formedness Constraints, it is well-formed. One is thereby able to have documents which are well-formed, but not necessarily valid. When dealing with XML, this is a distinction one must be aware of. We will look at examples of DTDs and XSD in later sections to illustrate some of the differences between the formats.
But first, it is appropriate to look at some XML-based formats for metadata.

2.4 Formats for storing bibliographic records

There is a vast quantity of metadata formats in use today. Some have become a *de facto* standard due to their gradual growth over time and they have in turn become the basis of newer formats which have added new properties to an already well-functioning paradigm. The common thing shared by metadata formats is that they contain information, but structured in different ways. In recent years, the insufficiencies of the old metadata giants have become apparent, thus leading to a specification of requirements called the FRBR model which will be described in the next chapter. Firstly, let us examine some important metadata formats.

2.4.1 MARC

MARC (Machine-Readable Cataloguing)[1] is a format for storage and exchange of bibliographic records in a machine-readable format. This format was originally developed by the Library of Congress in the period of 1965-66 and was then called LCMARC, later known as USMARC. However, the format known as MARC 21 today, was created in 1997 through the unification of USMARC and CANMARC, the latter being the Canadian MARC-format. The MARC 21 format is now a widely used format for bibliographical records interchange and is maintained by the Library of Congress, in conjunction with other user communities. National variants have emerged, such as UKMARC in the United Kingdom, NORMARC in Norway and the aptly named DANMARC in Denmark. The Norwegian format NORMARC[2] is maintained by 'Den norske katalogkomité', BIBSYS-MARC is maintained by BIBSYS and BS-MARC is the exchange format of the 'Biblioteksentral'

The MARC structure

The structure of a MARC record is organised in three parts:
- the leader
- the directory
- the variable fields.

The leader is a fixed-length field (24 characters) which contains parametrical definitions for the processing of the record.

The directory contains the tag, starting location and length of each field within the record.

The variable fields are where the data concerning the bibliographical entry is stored. These fields come in two varieties; variable control fields and variable data fields.

Variable control fields are fields that begin with a 00X-tag and consist of data and a field terminator. They contain either a single data element or a series of fixed-length data elements.

Variable data fields are all fields except 00X fields. These fields consist of the three character tag, an indicator which interprets or supplements the data, and subfield codes which identify data elements which may require separate manipulation.

```
00566cam  22001931  4500
001 4056496
005 20050304122526.0
008 730911s1957    nyu           000 1 eng
035    $9 (DLC)   57010033
906    $a 7 $b cbc $c orignew $d u $e ocip $f 19 $g y-gencatlg
010    $a    57010033
040    $a DLC $c DLC $d DLC
050 00 $a PZ3.R152 $b At $a PS3535.A547
100 1  $a Rand, Ayn.
245 10 $a Atlas shrugged.
260    $a New York, $b Random House $c [1957]
300    $a 1168 p. $c 23 cm.
655  7 $a Science fiction. $2 gsafd
991    $b c-GenColl $h PZ3.R152 $i At $p 00010154479 $t Copy 1 $w BOOKS
```

Figure 6: A MARC record of 'Atlas Shrugged' by Ayn Rand from the Library of Congress

If we consider this example of a MARC record of 'Atlas Shrugged', we can identify the aforementioned data fields. Notice that this representation of a MARC record has been modified to be more readable. A MARC record is normally stored on one line and its fields are separated with special assigned characters. The same MARC record is displayed below, only this time it is completely devoid of line breaks.

```
00566cam  22001931
45000010008000000050017000080080041000250350021000669060045000870100017001320400001
80014905000300016710000150019724500200021226000360023230000200026865500280028899100
05600316#4056496#20050304122526.0#730911s1957    nyu            000 1 eng  #
#9(DLC)   57010033#  #a7#bcbc#corignew#du#eocip#f19#gy-gencatlg#  #a   57010033 #
#aDLC#cDLC#dDLC#00#aPZ3.R152#bAt#aPS3535.A547#1 #aRand, Ayn.#10#aAtlas shrugged.#
#aNew York,#bRandom House#c[1957]#  #a1168 p.#c23 cm.# 7#aScience fiction.#2gsafd#
#bc-GenColl#hPZ3.R152#iAt#p00010154479#tCopy 1#wBOOKS##
```

Figure 7: A MARC record without line breaks

The readability is greatly compromised in this figure compared to the one above. Even so, despite the line breaks in the previous example, it takes some time getting used to reading a MARC record.

The variable fields start on the second line, where there are three lines with 00X-tags. These comprise this MARC-record's variable control fields. All of the following fields contain the data which is directly related to the library item. Of these, the most important ones to notice are:

100 tag	personal name main entry (author)
245 tag	title information (including title, other title information, statement of responsibility)
260 tag	publication information
300 tag	physical description
655 tag	genre/form

Upon closer inspection of the 245 tag in the MARC-record above, we see that it has both first and second indicators (1 and 0) and a subfield code ($a)

```
245 10 $a Atlas shrugged.
```

The first digit signifies whether or not a title entry has been made, whereas the second indicates whether or not initial character positions are disregarded. The first digit being 1, means that the title has been added (the opposite being 0). The second digit is 0, which means that no initial characters are to be disregarded.

```
245 14 $a The Lord of the Rings
```

In this example, character position 4 of the title field is considered the first one, and therefore the article 'the' is to be disregarded when it comes to sorting, filing and search processes.

The subfield code $a simply means 'title'. There could potentially be other subfield codes for this tag, such as $b - 'remainder of title'.

2.4.2 MARCXML

MARCXML[4] is an XML Schema Definition (XSD) which was developed by the U.S. Library of Congress in 2001 as a framework for working with MARC in an XML environment. The main objective of this framework is to offer a lossless round-trip conversion of an ISO 2709 MARC 21 record and an XML encoded MARC 21 record. The terms 'lossless' and 'round-trip' mean that one is able to convert back and forth between these formats without losing any data. It is intended to allow MARC data to be handled in a way that is more suited to the individual user. MARCXML keeps the original datafield, indicator and subfield structure from MARC21, so it is a way for users of MARC to easily make the transition to XML. This also increases record size considerably in comparison to ordinary MARC. Once such a format is designed, users are able to apply stylesheets to the XML records in order to present the data suited to their own needs. Other possibilities connected to XML include such as editing, conversion and validation of records.

2.4.3 MarcXchange

MarcXchange[5] is a standard format for general exchange of MARC records. It was developed as an alternative to MARCXML becaue the latter format is tightly connected to MARC21, and there was a need for a format which supported all kinds of MARC dialects. MarcXchange is based on the MARCXML schema with the difference in that it was generalized so that it was applicable to all MARC formatted records. This was done without changing the basic MARC structure including its datafield, indicator and subfield elements. Its intended usage was to exchange MARC records and other metadata and act as a temporary format for data transformation, conversion, publication, editing and validation. Also, it is used in representing metadata for harvesting, as in OAI-PMH (The Open Archives Initiative Protocol for Metadata Harvesting)

2.4.4 MODS

Metadata Object Description Schema (MODS) [15] is another format which is maintained by the Network Development and MARC Standards Office. MODS is a schema for a bibliographic element set, mainly used for library applications and is intended to carry selected data from MARC 21 records and to create original resource description records. It includes a subset of MARC fields but it uses language-based tags instead of the numeric tags in MARC 21. It does not support round-tripability with MARC 21, which means that one may not convert back to MARC 21 from MODS without some loss of specificity or actual loss of data. With the language based tags, this format has achieved a higher degree of readability. This higher degree of readability was one of the main reasons for its development, as MARC is considered by many to be too complex. Dublin Core arose as a simplistic alternative to MARC, but was claimed by many to be too simple with its mere 15 metadata tags. MODS came about as a compromise between these two, offering legible tags but maintaining flexibility towards what it is able to express. However, it was not designed according to the principles of the FRBR model, thus it lacks the ability to express relations as those in the product model of FRBR.

MODS' top elements are as follows:

Element name	*Function*
titleInfo	Title of the item
name	Name of author
typeOfResource	Description of item, physical category
genre	Genre of item
originInfo	Origin of item
language	Language of item
physicalDescription	Information about form, media type, extent
abstract	Abstract of the item
tableOfContents	Table of Contents of the item
targetAudience	Group for which the item is targeted
note	A note connected to the item
subject	The item's topic
classification	The classification of the item
relatedItem	Any other MODS item which is related to the current item
identifier	Identifier of the item
location	Either physical location or URL of the item
accessCondition	Information about restrictions on an item
part	The designation of physical parts of an item
extension	To provide additional information not covered by MODS
recordInfo	Information pertaining to the creation of the record

The top elements of MODS have attributes and sub elements which cover each aspect of its belonging element and segment information further. What MODS offers instead of Dublin Core, is that flexibility is maintained whilst user friendliness is increased. Users familiar with MARC will recognize that these fields form a subset of MARC, only that they of course now have language based tags. Further differences from MARC is that MODS does not use field and subfield tagging and that there are some elements in MODS which have no corresponding tag in MARC. These differences lead to the non-round-tripability between the two formats.

2.4.5 Dublin Core

The Dublin Core Metadata Initiative (DCMI) [19] was conceived at an invitational workshop in Dublin, Ohio, USA in 1995. It is an element set of 15 elements which are quite concise and are intended to describe all kinds of information. The name 'Dublin Core' therefore refers to its place of origin and that it is supposed to encapsulate the core of all information needs. The purpose behind the DCMI is to promote the widespread usage of interoperable metadata standards and this format is now maintained by a large number of individuals. It bases itself upon development by a diverse community with people from many different backgrounds, located in organizations all over the world. This is in order to more efficiently harness the most important properties which Dublin Core is supposed to contain and to spread the ideology of a common framework for metadata description to all the parties involved and their associates. The 15 elements in the Dublin Core Metadata Element Set (DCMES) are as follows:

Element name	*Function*
Title	The title of the item
Creator	The entity responsible for creating the item
Subject	The topic of the item
Description	An account of the item
Publisher	An entity responsible for publishing the item
Contributor	A person, organization or a service responsible for making contributions
Date	A point or period of time associated with the item
Type	The nature or genre of the item
Format	The file format, physical medium or dimension of the item
Identifier	A unique identifier of the item
Source	The originative source of the item
Language	The language contained within the item
Relation	A related item
Coverage	The spatial or temporal topic, spatial applicability or jurisdiction of the item
Rights	Rights held in and over the item

These elements are all optional and repeatable.

Qualified Dublin Core is a further development of the original 15 elements which have now gained three additional elements (Audience, Provenance and RightsHolder). Qualified Dublin Core is an ongoing process to extend each element's potentiality. This is done by providing so-called 'qualifiers' which makes a user of Dublin Core able to refine and narrow down the meaning of each element. This specification of elements is an attempt to combat criticism from supporters of more refined element sets towards Dublin Core being too simplistic and too broad. According to 'Understanding Metadata'[20]as hosted by the NISO (National Information Standards Organization) there has been an ongoing conflict between these two general opinions. For those applications which are not designed for the use of qualifiers, there is the possibility to simply ignore them and regard the element as if it were unqualified. This is called the 'Dumb-Down Principle' and is a guiding principle for the qualification of Dublin Core elements. Of course the 'dumbing-down' of qualified Dublin Core will give you a loss of nuance and can in some cases lead to incorrectly retrieved items. If you are conducting a search based on a certain criterion, but you are not able to search using qualifiers on Dublin Core elements which are in fact qualified, you might get a loss of precision. That is to say, you will retrieve irrelevant items. If you have the following elements of Dublin Core for the English version of 'A Dolls House' by Henrik Ibsen;

Title="A Doll's House"
Title.original="Et dukkehjem"
Creator="Henrik Ibsen"
Language="English"
Language.original="Norwegian"

you will notice that two of these elements are qualified. If you have an application which does not handle qualifiers and searches for Works written by Henrik Ibsen in Norwegian, the above entry will wrongly be amongst the retrieved ones. This exemplifies a problem that could occur when using Dublin Core qualifiers. Even so, Dublin Core has received much attention and is definitely contributing to the future of metadata.

There are more formats available which make alternatives to these ones, but the focus has been put on the previous metadata formats which are the most important ones.

2.5 Schema Definition Languages

A schema definition language is used to define the structure of XML. The schema which defines the properties of a certain type of XML format is written used a schema definition language and is stored in a file. This file is then later used to validate that XML is structured in a way that conforms to the rules specified in the schema. An XML file which conforms to the schema is *valid*.

There are different types of schema definition language available with different properties and different levels of complexity, depending on what your exact need is. Below is given an account of the most important ones.

2.5.1 DTDs

Document Type Definitions (DTDs)[8] are used to express a schema via a set of declarations to describe the content of a class, and it is a method for validation of XML. Validation of XML means that you secure how your XML is organised. It defines the document structure with a list of legal elements and attributes. A valid XML document conforms to all the constrictions a DTD dictates. You are able to declare:

- Hierarchical structure of an XML document
- Element names
- Attribute names and placement within elements
- Number of occurrences (0 or 1, 1, zero or more, one or more)
- Data type contained within an element (Character data or Parsed Character Data)

These are the most important features. DTDs can be either included in the XML document itself or referenced, a so called external DTD.

Here is the 'Pink Floyd' example which was given earlier:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<recordstore>
  <record>
    <title>Dark Side of the Moon</title>
    <artist>Pink Floyd</artist>
    <artist>Roger Waters</artist>
    <year>1973</year>
  </record>
</recordstore>
```

Figure 8: An XML document

Now we will look at an example of a DTD. It defines the overall structure of the example above, with the limitation that there can only be one occurrence of the <artist> element.

```
<!ELEMENT recordstore (record)*>
  <!ELEMENT record (title*, artist, year)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT artist (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
```

Figure 9: A Document Type Definition (DTD)

This defines a recordstore element as the root of the document and it can contain zero or more occurrences of record elements. The star (*) signifies 0 or more in cardinality. Further, it defines the record element to contain zero or more occurrences of title, only one occurrence of artist and only one occurrence of year. All of the children of record are parsed character data (PCDATA). Notice that the title element is deliberately defined with cardinality zero or more, as this will be used in the next section. The next section will also thoroughly describe reasons for the conception of XML Schema and why DTDs in some cases are not enough.

2.5.2 XML Schema

An XML Schema, or rather an XML Schema Definition (XSD)is a way to define the structure and properties of the various elements in an XML file. XML Schema arose as a workgroup in W3C as a response to DTDs (Document Type Definition) being little flexible and narrow. David Gulbransen in his book *Using XML Schema*[8], explains that DTD offers a basic mechanism for specifying elements, attributes, entities and notations. DTD was originally conceived for SGML, but when DTD is used with XML there are more limitations as compared with using XML with SGML. These limitations include such as not being able to define specifically the exact minimum or maximum number of occurrences of an element. You are neither able to define which type of data an element should contain, such as Integer, String and Boolean, or define the exact format of a number concerning decimal places.
XSD however, has functionality which covers these aspects. Take note that using DTD is good enough in many circumstances, and was designed to be a simple way to express the framework of an XML file.

If we consider the recordstore example in the previous chapter, we had the following DTD:

```
<!ELEMENT recordstore (record)*>
  <!ELEMENT record (title*, artist, year)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT artist (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
```

Figure 10: A Document Type Definition (DTD)

This DTD lets us have zero or more occurrences of <title>. Now let us assume that we want to be able to have either one or two occurrences of the element <title>. On rare occasions, an album may acquire a nickname that is so famous that it almost replaces the original name. This goes for the album 'Metallica' by Metallica, which is almost more recognized by its handle 'The Black Album'. Let us express this in XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<recordstore>
  <record>
    <title>Metallica</title>
    <title>The Black Album</title>
    <artist>Metallica</artist>
    <year>1991</year>
  </record>
</recordstore>
```

Figure 11: An XML document

In the example the album may have two <title> elements. There could also be used attributes here to qualify the <title> elements, to differentiate them, but in this example it will be left out to simplify. Having just the DTD to validate this piece of XML, there is nothing stopping us from entering more occurrences of the <title> element. This is one of the mentioned weaknesses of DTDs which can be solved by XSDs:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="recordstore" type="recordstoreType" />
  <xs:complexType name="recordstoreType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="record"
type="recordType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="recordType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="2" name="title" type="xs:string" />
      <xs:element name="artist" type="xs:string" />
      <xs:element name="year" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 12: XML Schema Definition for the 'recordstore' example

The example above is quite more detailed and bigger than the DTD but also provides a much greater flexibility. Looking directly at the point of interest, namely the definition of the complexType 'recordType', we see that it defines a sequence of three elements, where the element 'title' is defined with a minimum of one and a maximum of two occurrences. Connecting this XSD to the 'Metallica example', ensures that only one or two occurrences of <title> will be accepted. Notice also that it is easy to define the datatype which is to be contained within the element, which here has been set to string in the eXtended Stylesheet-namespace called xs.

As previously mentioned, there are other benefits like being able to define datatype and format of data. A benefit from XML Schema which is used in this thesis, is the application of an extension, also called extension base. An extension is a way to inherit functionality from other types defined in the same schema. If you have elements which occur in several types, you can join them in a separate type and use this type as an extension. All the types extending or inheriting from this type will also contain the elements that type has. This simplifies the work of reusing functionality and maintaining consistency over elements which are used in a uniform fashion over a larger scope.

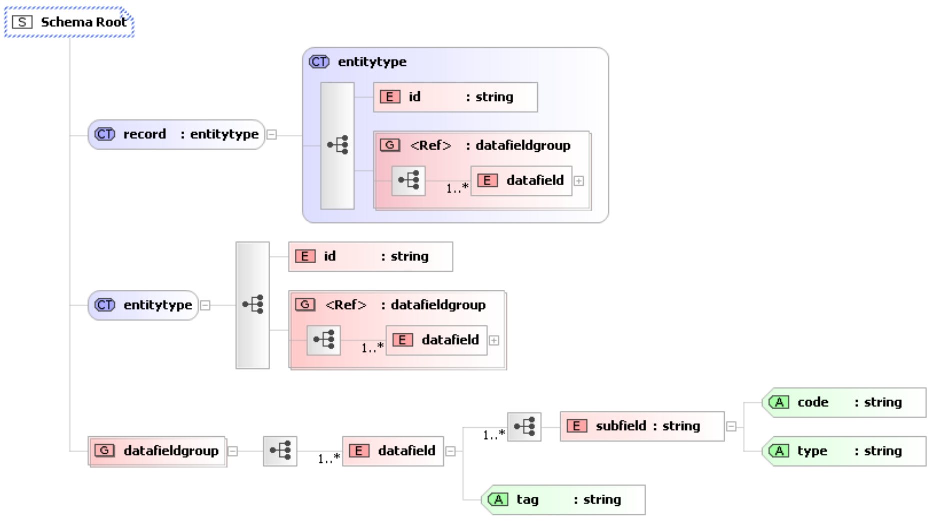


Figure 13: Re-use of functionality through groups and extensions

Rather than showing the code for this example, the model of the XML Schema code will be shown. The element <record> has the extension base 'entitytype'. This secures that the <record> element and any other element who has the same extension base will have to implement the id element and the datafield group. The group functionality can be used in order to gather elements which will be used in several circumstances, thereby eliminating the problem of inconsistencies as the group definition only occurs once. References can be made to groups in a complex type in addition to the extension. The difference between complex types and groups is that that complex types are used for bigger collections of elements, whilst groups have to contain either an <all>, <sequence> or <choice> element. These elements further contain child elements. Groups thereby exist as a means to making a reference to a repeating structure of elements. These properties of XML Schema were important criteria when deciding to use it in this thesis.

This way of thinking is also common in programming where one is able to extend attributes and functions from a super class. DTDs are unfortunately not able to reuse functionality in such a manner. Other functionalities such as support for namespaces or support for regular expressions are lacking in DTDs. These disadvantages of DTDs mentioned, especially not being able to define cardinality to minute detail, and the non-support for extension bases and namespaces, made XML Schema the natural choice for implementation. Additionally, the ability to make complex types which form the extension bases of other types is a great advantage when it comes to securing uniform design of repeating structures. The ability to make groups helps make design become more congruent due to the possibility to reuse code.

There was also another candidate for schema format, which is called RELAX NG.

2.5.3 RELAX NG

RELAX NG [21] is a schema format for XML which defines amongst others the structure, element names and cardinality of an XML document. It is based on two preceding formats which were called RELAX by Murata Makoto and TREX by James Clark. A RELAX NG schema is an XML document itself, but it also offers non-XML syntax. RELAX NG was developed at about the same time as the W3C specification for XML Schema. However, it stands out from the other mentioned schema formats by being simpler. It has mostly language based tags and is as previously stated mostly written in XML. It can be better explained through an example:

```
<element name="collection" xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="record">
      <element name="identifier">
        <text/>
      </element>
        <oneOrMore>
          <element name="title">
            <attribute name="language">
              <text/>
            </attribute>
            <text/>
          </element>
        </oneOrMore>
    </element>
  </zeroOrMore>
</element>
```

Figure 14: A RELAX NG schema definition

This example shows the simplicity of RELAX NG and its hierarchical structure. This schema defines a root element <collection> which can contain zero or more <record> elements. This <record> element contains exactly one <identifier> element and one or more <title> elements. The <title> element has an attribute called "language'.

This way of structuring a schema can get less readable when it gets bigger and the number of nested elements increases. It is possible to define named patterns which you reference in-line and thereby prevent the schema from expanding into a too large nested structure.

```
<grammar>
  <start>
    <element name="record"
      <oneOrMore>
        <element name="datafield">
          <ref name="datafieldContent">
        </element>
      </oneOrMore>
    </element>
  </start>

  <define name="datafieldContent">
    <attribute name="tag">
      <oneOrMore>
        <element name="subfield">
          <attribute name="code">
            <text/>
          </attribute>
          <attribute name="type">
            <text/>
          </attribute>
          <text/>
        </element>
      </oneOrMore>
      </text>
    </attribute>
  </define>
</grammar>
```

Figure 15: Defining a named pattern in RELAX NG

This example shows that one is able to define a named pattern called <datafieldContent>. The element called <grammar> is necessary here to be able to define several patterns. The <start> element is necessary in the definition of the grammar, as it indicates at which point is the pattern matching begins. The <datafield> element is repeatable, which the surrounding element <oneOrMore> indicates. The definition of datafieldContent dictates that it should have an attribute called 'tag' and 1 or more occurrences of 'subfield'. Furthermore, the subfield contains two attributes, 'code' and 'type'. Likewise as in the XML Schema example given previously, one is able to take advantage of datafieldContent several times and reuse its functionality.

You can also employ a compact syntax in RELAX NG which is somewhat similar to DTDs. This syntax offers exactly the same functionality as the ordinary syntax in RELAX NG, but is an alternative way to write the schema in order to combat problems of readability. This is of course also beneficial for users who are already familiar with DTDs and the transition to RELAX NG can be made smoother.

RELAX NG has the advantages of defining data types, using namespaces, naming patterns for referencing and re-use just like XML Schema. However, there are a few more properties in XML Schema, such as the ability to define exact number of occurrences and simply that it is more widespread in use, that made it more attractive in comparison with RELAX NG. XML Schema also has a greater flexibility when it comes to reusing functionality through extensions and groups. This lead to using XML Schema in this thesis.

2.6 XPath

XPath[16] is a W3C language belonging to the eXtensible Stylesheet Language (XSL) family which is used in order to locate elements and attributes in XML documents. As the name suggests, this language employs path expressions to navigate through XML documents. Using these path expressions, you are able to define parts of an XML document in order to retrieve the elements and attributes that you need. More specifically, the path expressions are used to select nodes or node-sets within the scope you define. As these are expressions, they will process and return a value. The language contains a library of standard functions and is an important part in eXtensible Stylesheet Language Transformations (XSLT). Among these are functions for string values, numeric values, date and time comparison, node and Qname manipulation, sequence manipulation, Boolean values and more.

When XPath is used to circumnavigate an XML document, the document is treated as a tree of nodes. Through the application of XPath expressions, one is able to address nodes and retrieve their values. In XPath, the so-called nodes come in different varieties; element, attribute, text, namespace, processing-instruction, comment and document (root).

2.7 eXtensible Stylesheet Language Transformations

eXtensible Stylesheet Language Transformations (XSLT) [17] is a powerful and flexible language for transforming XML documents into other kinds of documents and it is also a W3C recommendation. It belongs to the eXtensible Stylesheet Language (XSL) family, which also includes XPath and XSL Formatting Objects. The reader should however notice right away that XSLT, although mainly associated with XML, is capable of transforming XML into a wide variety of documents. These include XML, HTML, XHTML, XSL, DTD, CSS, text, source code in a programming language of your choice, indeed almost anything you want. The most important use of XSLT is the ability to specify styling of XML, and it is in that particular area of application it will be used in this project. It came about as an alternative and more flexible way to control the appearance and transform XML documents as opposed to Cascading Style Sheets (CSS). CSS is a useful way to define various appearances of markup, but it lacks some vital properties which is needed in this project such as the ability to organise elements in a totally different fashion concerning hierarchy and attributes, combine multiple documents, do computations and performing branching to name a few. XSLT is a quite robust language for the above mentioned requirements as it is able to revamp the look and hierarchy of an XML document, provide control structures for branching of actions, variables for storing retrieved or computed values and functions or templates for the reuse of code. The relationship between XSLT and CSS is therefore in many ways analogous to the relationship between XSD and DTD. Both CSS and DTD provide the simpler way of doing things, but since they in many cases are too simple, XSLT and XSD were designed to have more advanced options available at your fingertips.

XSLT is different from other programming languages in some aspects. First of all, the XSLT stylesheet is an XML document in itself and it bases its transformation on pattern matching. Although it is possible to have functions in XSLT, they do not comprise the main part of the transformation procedure. XSLT is also free of side effects, meaning that many different stylesheets could be applied simultaneously without affecting the data they are working with. Lastly, instead of making loops in order to process many elements at a time, the methods iteration and recursion are

used to iterate over the elements in the selection. You are able to run through a set of elements according to which search criteria you specify and define what you want to do with each of the elements. To get back to the aspect of pattern matching, this means that it searches for patterns or rather element names that match the names of the templates. Once the stylesheet finds an element in the XML document for which there is a corresponding template match, the code of the template will be executed. Information contained in the original XML element will be transformed according to what the template dictates. Doug Tidwell[18] in his book 'XSLT' explains neatly how templates 'talk' to the parser : «When you see part of a document that looks like this, here's how you convert it into something else.»
Within templates, XPath is used to navigate, find and retrieve paths and elements. After having found them, you are able to access the information contained within and organise it freely.

Thus, the introductory theory is concluded. In the following chapter, design criteria for designing a metadata format for bibliographic records with the use of XML will be presented.

3 Design Criteria

In this chapter, some important XML design criteria for bibliographic records will be presented. The choice to use XML to describe bibliographic records is based to a great degree on the aspects covered in this chapter. The aspects which will be covered are:

- XML as a metadata format
- Choosing the root element
- Expressing relationships in XML
- Choices regarding use of ID/IDREF
- Including records outside of hierarchy tree
- Exchange of bibliographic records
- Validation
- Readability of XML
- Expressing metadata fields

3.1 XML as a metadata format

Before a language such as XML came along, it was difficult to use programming logic on MARC records. There was no standard way of accessing a MARC record in order to store, retrieve or manipulate the content of a record and it was common for software developers to define own formats or languages to share between data programs. This of course also meant that they had to define the structure of the schema by themselves and also design tailor-made parsers for their specific format. With the advent of SGML and XML, any data at all could be represented in a hierarchical fashion and with language-based tags. Already from the beginning of XML, tools for maintaining and manipulating XML were present. This was one important reason for XML's quick ascent into widespread use.

XML is currently being used for storage of metadata in a vast variety of formats. The main reasons for the application of XML to store metadata or indeed any kind of information is due to the flexible options that are available. Metadata schemes such as MODS and Dublin Core offer implementation with the use of XML and have namespaces designated for xml element names. When using XML, one has an array of tools at hand, such as different schema definition languages:

- XML Schema
- DTDs
- RELAX NG

Thus, a user has several choices when it comes to defining how a format is too look and what is allowed. This validation of structure is necessary when dealing with a format for storing metadata records. Metadata formats are to be used by other types of software which expect a format to look a certain way, therefore it is imperative that there also is mechanisms for securing the properties and structure of a format. Indeed, with these methods of defining your own format, you are not forced to choose a specific format in particular. Much like has been done in this thesis, one could choose features from different formats which have been in use before and been proven to work well and combine these features into your own schema definition.

Further advantages include the eXtensible Stylesheet Language Transformation (XSLT). With this tool, XML is able to be transformed into virtually anything the user wishes. This enables features

such as:
- Transformation to other metadata formats
- Transformation to other file formats (Text, Comma-Separated Values, HTML, Java source code etc.)
- Extraction of a selection of records, selection of fields

XML's readability is something which is discussed in more detail in chapter 3.8, but for starters what can be said about XML is that one other key point to using it is readability. XML uses language-based tags which makes it readable for virtually anyone, both computers and humans. Instead of code-based tags which presuppose that a user knows what they mean, XML can spell right out what the property is.
This readability issue also has a down-side, namely that when hierarchical structures become too big, they sometimes become less readable. Sometimes related information can be spread apart by large nested elements, causing users to have trouble regaining a full overview of information that actually belongs together.

When using XML, one can express relationships by storing a record as the child element of the record is has a relationship to. There is also the option to use ID/IDREF which does not store the entire record as a child element, but has a reference to the ID of the record being referenced. These options will be explained in more detail in chapter 3.3.

Not everyone is lauding the use of XML to store metadata. XML documents have the potentiality of becoming too large and therefore require more resources. This is due to XML's verbosity, the very thing that causes legible tags. When maintaining, transforming or retrieving information from XML, an entire XML tree needs to be parsed into memory before the actual processing can occur. These kinds of operations could be experienced as too heavy-set and slow. Another problem with XML is that designing an XML data structure should be easier.

Despite negative sides of XML, it has numerous tools and features and it has become such a widely-used technology and the advantages of cooperating through the use of the same standard is definitely worth it.

3.2 Choosing the root element

When the entities in the FRBR model and their relationships are to be expressed using XML there has to be a certain entity which is the root element of a record. XML bases itself upon hierarchy and one of the entities will have to be the parent element.

Upon looking at the FRBR Product model, the hierarchy is Work -- > Expression -- > Manifestation --> Item. Upon first glance, an obvious choice would be Work as root element. But could any of the other entities also be a root element? What about considering also Person or Corporate body or even Concept or Place?

First considering the entities of the FRBR Product model, Work, Expression, Manifestation or Item could be the root element. Work is an abstract entity which contains conceptual information and ideas as well as relationships to Persons or Corporate bodies. The same goes for Expression. Manifestation and Item are concrete entities and contain information pertaining directly to physical books. A format which is to contain bibliographic metadata might benefit the most from using an entity which describes a physical entity as the root element. This discussion will be taken further in chapter 4.1.

If the entity Person is chosen as the root element and relationships are expressed using nested/hierarchical storing, all the belonging Works, Expressions, Manifestations and Items will be stored as children and descendants beneath the Person. The same goes for Corporate body. An advantage of this design is that one has an immediate overview over all the Person's writings and accomplishments. A disadvantage is that a specific Work needs to be retrieved in the context of a Person. One cannot retrieve a Work or a Manifestation without first retrieving the Person which is responsible for the entity. This is a disadvantage in terms of fast search and retrieval of relevant information. One does not necessarily want information about the Person in every circumstance.

Using Concept or Place as the root element is not something which is a smart design solution. A Concept might be related to a vast amount of records and it is not reasonable to store Works according to the related Concept. Keeping all the Works related to the Concept 'World War II' stored together is not a wise choice for many of the same reasons as storing Works under a Person is not a wise choice. It is not efficient or relevant to store information based on the Concept as this is secondary information.

If one does not desire to have any of the FRBR entities functioning as the root element of a record, being the parent element of the others, one must express every relationship with IDREF. This is the only way to avoid any parent and child relationships between the FRBR entities. The advantages and disadvantages of this approach will be explained in chapter 3.3.

3.3 Expressing relationships in XML

Bibliographic metadata organised in XML which is based upon FRBR will have relationships. An XML format which is to describe bibliographic records therefore must have ways to express many kinds of relationships/relations between records. These relationships occur between the elements of the FRBR Product model as well as to Persons, Corporate bodies, Concepts and Places. In this chapter we will take an extensive look at ways of expressing relationships and when it is best to use each method.

There are basically two ways of expressing relationships:

1. Hierarchical storing (Nested storing)
2. Using ID/IDREF

3.3.1 Hierarchical storing

Hierarchical storing or nested storing means that one stores the related entity underneath the entity from whence it came. If Work has been used as the root element of a record, it could have its connected Expressions as child elements. So between the typical entities which are found within the FRBR Product Model, this storing technique could ideally be employed. But any kind of relationship could also be expressed in this manner. What is important to remember, is that one cannot continue hierarchically storing forever, as this would lead to potentially infinite trees. Relationships occur in such a fashion that they are nearly always pointing to another entity which again has relationships. Therefore, when using XML to express any relational data model, there must either occur a combination of hierarchical storing and IDREF or using IDREF in all instances.

Hierarchical storing has therefore the following advantages:
- Fast retrieval
- Increased proximity of related records, increased readability from an FRBR perspective

and the following disadvantages:
- Duplicated records lead to larger collection

3.3.2 Using ID/IDREF

XML-documents are structured in a hierarchical fashion and can contain vast amounts of data. But what happens when redundance of data occurs within an XML-document? This is not necessarily a problem, but it can be. Redundance of data will certainly lead to larger documents, as well as the cost of having to enter the same data a given number of times. What about editing data which exists several places? For example, if you have a Person who is responsible for many Works and add him or her as an entry in the sub tree of each Work, you first have the task of adding the name a given number of times. If you have to make corrections to this name afterwards, you will have to retrace your steps and correct it the same number of times as before. Luckily, since we have efficient search mechanisms, it is not too hard finding the occurrences and editing them, but that all depends on how many occurrences there are. However, if you are unlucky enough to misspell the name in one of the entries, you face a risk of not finding it again, and thus you could be left with an erroneous as well as an obsolete entry. The way to solve this problem, is by using ID/IDREF[9].

However, there are things against using ID/IDREF also. Using references to connect pieces of information will provide you with the aforementioned advantages, but when you use a language like XPath to retrieve information from an XML-document, the subject of cost comes into consideration. During certain retrieval operations in XPath, it could be more efficient to have the data organised in a tree-structure. Although this will lead to duplication of data, the lookup will be faster, because the costly join operation (depending on data-size) will be avoided.

In terms of readability, records that are related will not be stored in proximity of each other. From an FRBR perspective, this leads to fragmented data and less readable records.A Work, Expression, Manifestation and Item which belong together semantically, should ideally be stored together.

Using IDREF (Referencing) has therefore the following advantages:
- No duplication; no increased data size
- Updates are easily performed since data is stored one place only

and the following disadvantages:
- More costly search operations
- Related records and related data is stored separately and closely connected information will appear fragmented. This yields decreased readability from an FRBR perspective.

3.4 Choices regarding use of ID/IDREF

As described previously, relationships or relations may be handled in different ways, either through hierarchical storing or through the use of ID and IDREF. When is it most prudent to use either technique? Could we always use hierarchical storing or could we always use ID and IDREF? Certainly choosing one and sticking to that design throughout all relationships could be a solution, but as previously mentioned there are negative side effects of both. Hierarchical storing will yield larger XML documents and readability can be diminished depending on who the user is. A casual user would probably prefer that information is segmented according to which entity they belong to whilst a librarian might prefer to have the entire Actor record stored within the belonging Work record. Using IDREF everywhere will lead to efficiently smaller XML documents but related information will be very fragmented, leading to more searching and look-up operations. There must be an evaluation of which relationships should be expressed using an entire record and which ones it will suffice to express using IDREF.

3.4.1 Use according to need

What choice to make regarding the use of IDREF depends upon what the need is. There could be several types of needs such as the need for:

- No duplication
- Full editability
- Storage optimization
- Efficient querying
- Readability

No duplication

No duplication refers to the storage where no data is duplicated at all. If one wants a design with no duplication whatsoever, it is necessary to use IDREF in all circumstances regarding expressing relationships. This means that all relations are expressed with IDREF to other records and each record is stored directly under the root element of the collection. With this design, there is no hierarchical storage between the FRBR entities.

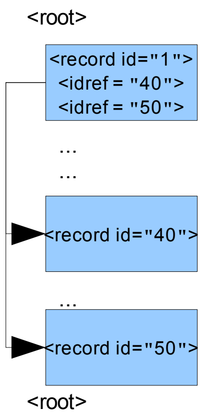


Figure 16: Using only IDREF to express relationships

The advantage of this design is of course that no data is duplicated. Every record is stored only once beneath the root element. Since every piece of data exists only one place, it is also much easier or much less taxing to perform updates. If a certain Actor changes his/her name, the update will only have to be executed once. Had it been stored hierarchically, a search and replace algorithm would have to be performed on several records although it actually concerns the same data.

If storage space is an issue, this approach would be preferable. Disadvantages of the design is that lookup operations will take longer. Also records belonging to each other will not be stored together, they will only be remotely linked through the IDREF. They will appear as very fragmented and information contained within connected FRBR-entities will be more difficult to gather.

Full editability

If one needs a database where updates can be executed with the least effort, one wants full editability. This need goes hand in hand with the need for no duplication in a database as the least taxing edit operations can only be performed when data only exists once. The only way to avoid any duplication is as before mentioned to use IDREF when expressing relationships between any entity.

Storage optimization

Storage optimization is a design where you mainly try to keep a hierarchical scheme of things but use IDREF in those situations where there is most to gain in terms of storage space. The advantages are therefore that you are able to make use of hierarchical storing when it is appropriate and since you are concerned about storage space, you can implement relationships with high possibility of duplication as IDREFs. For instance if hierarchical storing is implemented between Work and Expression this will probably not yield much duplicated data. If you however implement hierarchical storing between Work and Actor, it seems likely that the data connected to Actor will be duplicated under each Work he/she has created. More detailed statistics regarding these matters will be presented in chapter 6.6.

Efficient querying

Efficient querying means that everything is implemented hierarchically to the greatest extent possible and will lead to no cost at all through the use of retrieving records which are pointed to through IDREFs. In order not to have infinitely large relationship trees, IDREF will have to be used at some point, but to a certain extent hierarchical storage can be employed. Drawbacks of this design is the obvious increase of data size, as much data will be duplicated.

Readability

Readability concerns being easily able to read a specific record and not have trouble recognizing where data belongs categorically. But this subject could depend on who the user is. If you are a regular user you might prefer to have the Work and Actor records stored separately, not to confuse information about them. A more experienced user such as a librarian would probably prefer that such data was stored together.

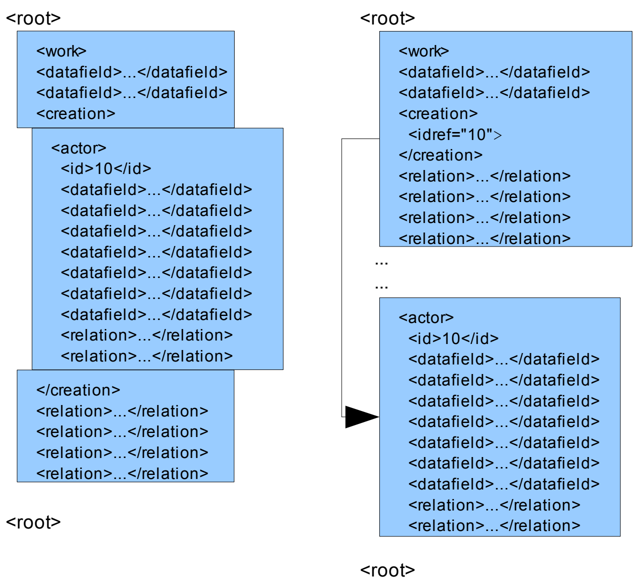


Figure 17: Maintaining readability in a Work record by storing Actor separately

In the figure above, the first alternative shows a hierarchically stored <actor> element within a <work> element. Depending on the size of the <actor> element and of the placement of the <creation> element, the <actor> element will make a considerable gap between the pieces of data connected more closely to the Work itself. Depending on the user, this may be considered as a reduction of readability or even a nuisance.

In chapter 4, more specific decisions regarding use of IDREF will be taken.

3.5 Including records outside of hierarchy tree

When using XML to store metadata, you have the option to store data hierarchically and also express relationships to a great extent through hierarchical/nested storage. You may also choose to include each record directly beneath the root element in addition to storing records inside the hierarchy tree below the chosen top element of the record. These records who are directly beneath root level, might have relationships expressed using IDREF to avoid further duplication underneath them.

Advantages of this design are:
- One has both the hierarchical view with Actors and Subjects connected and the simpler separate view.
- It is easy to retrieve a single record if one does not wish to have any additional data connected to it.
- One can exchange records segmented by type if this is desirable
- Separate storage of records will also list those records who do not have any relationships. Through XSLT processing records which perhaps mistakenly lack relationships may not be included in the new file.

Disadvantages:
- Storing records separately as well as nested will lead to even more duplicate entries.

3.6 Exchange of bibliographic records

When exchanging bibliographic records one has several options.
- One record
- Batches of records
- Entire record collections

Most commonly, you can exchange entries one by one, many at the time or even an entire database. An XSD which is to contain bibliographic entry metadata has to support these ways of exchange. Common ways of exchanging records are across FTP, Email or even CD/DVD-ROM for bigger datasets or even complete databases. When it comes to exchanging one entry, it is quite trivial. All that is required is that you are able to navigate and search for one specific record, through for instance XPath. Considering this in terms of XML Design, one would only have to keep one record for each XML file. But exchanging 'chunks' of records requires the possibility of keeping many entries in a single XML file. The flexibility of XML allows you to do this by storing many records underneath the root element of the file.

An advantage of exchanging records using the format which will be discussed later, is that it splits up records into Work, Expression, Manifestation and Item respectively. As we will see later on, the format presented here, has the potential to contain all these records in one. Therefore, you will get the segmentation between the elements of the FRBR product model, but in the neat package of one record. The benefit of this design is that you are able to retrieve a record, a Manifestation, and in the same go also retrieve all its connected Expressions and Works. There is no need to reconnect them or use IDREF to locate the referenced records. There are no extra resources being used to perform queries either.

What exactly is a record in an FRBR sense? As mentioned in the paragraph above, a record can contain all of the entities in the FRBR model. But Manifestations, Expressions and Works contain relationships to other entities such as Persons, Corporate bodies, Concepts and Places. If these entities then also are included in one record, a record may become quite large. In an exchange context, it can be superfluous to include the entire record of a Person with every Work, Expression or Manifestation he/she has made. Imagine attaching the complete Person record of William

Shakespeare to every Work, Expression and Manifestation he has written. It is both a waste of storage space and bandwidth if records are transferred electronically.
Therefore, when exchanging records both parties to the exchange must have an agreement of a record should comprise of entities. Relationships with Persons could be expressed using IDREF instead.

As Aalberg *et al.*[2] organised their FRBRized XML of a MARC record, they could split it up into several pieces. This means that one MARC record could exist as several FRBRized records which are linked to each other. Using either IDREF for interconnected records within the same file and HREF for interconnected documents in separate files, one is able to split files in any size one wants. This is a huge advantage when exchanging over the Internet, either by FTP, E-Mail or other, because you are able to exchange in intervals. Using the format which will be presented later, you can organise the files in which ever manner that you prefer using XSLT. This way you can adjust the size of each file, the structure of its records or even its internal consistency. Internal consistency means whether or not a file contains records which are linked internally in the same file or if the records have a lot of external references. A high internal consistency would mean that files contain records that are mostly internally referenced. When you have files with high internal consistency, these can be delivered to a recipient and be of rapid use independently of other files since they do not have many external references.

The use of XML opens up for use of Unicode. Unicode is a text format which allows for potentially all kinds of characters. Instead of storing each symbol as one byte, it uses two or more bytes per symbol and thereby is able to address a vast number of symbols. Previously the ASCII character set was used and had limited capacity, forcing those with other character sets to conform and latinize their character data. XML can make use of the advantages of Unicode and thereby users can employ their native character sets. Exchanging records using Unicode will hinder problems at the recipient end such as lacking support for certain characters.

3.7 Validation

XML documents which store information that many people are going to use and edit need to have a set of rules as to how they are allowed to be structured. When there many different users of a common format implemented in XML, there needs to be a schema that sets the boundaries of how the format is allowed to be. Users exchanging XML must know what to expect when receiving metadata and so must the software which uses the XML documents.
Databases which are implemented using XML as the format for storage, must have some kind of mechanism to ensure that documents are conforming to a certain format. Other software which uses the database as a fundament are dependent on data being stored correctly. In order to validate XML documents, they must have a schema attached. This schema must be prepared for all possible ways an XML document can be structured.

Rigidity and flexibility

When designing an XML Schema for the maintenance and validation of XML documents, one must remember that the design will be closely tied to how you want to validate your XML. Not only must the schema accommodate rigidity in terms of the general hierarchy, order of elements and cardinality, it must also accommodate some flexibility. An XML Schema within an FRBR context is rarely based on a finite set of elements, datafields and relationships. A schema must therefore be extendible and make room for new types of datafields or relationships. One way of solving this problem could be to include a general element which accommodates for lesser used data and also for new types.

Validating IDREF

One disadvantage of schema validation is that when ID/IDREF is used in contrast to storing the entire entity hierarchically, one does not have easily available methods of checking that the entity being pointed to actually is the correct entity. One solution could however be to ensure that the <id> elements of certain types have different properties. Since XML Schema supports checking that an id field has a certain number of characters, or even that an id field has certain 'identifying' characters within itself, an IDREF field could be validated. If different FRBR entities were given id fields where one could tell by the appearance or content of the id which entity it pointed to, one could also validate the use of IDREF. This would of course presuppose that this segmentation feature of the id field was implemented.

3.8 Readability of XML

The readability of XML is one of the key reasons for using it. With its application of language based elements and hierarchical structure, it makes records legible to anyone who has had some experience with HTML. Previously used formats such as MARC are on the opposite side of the scale concerning this matter. The readability of a MARC record is quite low, but it was of course never meant to be read by humans. Any given MARC record is normally stored on one single line, making it very difficult to read to the human eye. However, many providers of MARC metadata present records with line breaks, easing the job of interpreting the information given. When searching in BIBSYS-ASK [10], one is able to get the BIBSYS-MARC format displayed with line breaks for any given bibliographic entry. This also goes for Z39.50-clients such as Mercury Z39.50 Client[11]. Despite it being easier to read, unless you have intimate knowledge with each of the possible tags used in MARC, you will only be able to draw out the most intuitive information of a record. As presented in section 2.4, the most basic tags are easily identified, such as author, title, genre and so forth. Besides these, MARC offers a plethora of other tags, with their corresponding indicators and subfield codes.

What XML offers in contrast to other formats, is the ability to organise fields (elements) hierarchically, with legible and immediately intelligible names, due to the fact that the designer is able to decide for himself or herself what elements are to be called.

A problem with XML files getting too large, is that it actually decreases readability and the aforementioned advantages of easier understanding the elements. When a large bibliographic record is expressed in XML and the full hierarchy is present, it is difficult to maintain the overview of the fields. XML is often indented to represent the hierarchical structure embodied within and if the hierarchy becomes large and the number of elements in each record grow to a large number, it can seem rather chaotic. Newer XML editors equip techniques for showing which layer of the hierarchy a certain element belongs to, like showing a different colour for a layer or clearly indicating the number of indents that has been made. Someone experienced with MARC or derivatives however, would probably prefer its short and concise way of expressing a record. This again presupposes that you have learnt what tags mean. It could be contended that the initial threshold for learning how to use an XML-based format is lower than that of other more minimalistic formats, due to its legible and intelligible element names. Still, XML's verbosity is something which can cause problems and which is one of the most commonly used arguments of those opposed to XML.

3.9 Expressing metadata fields

An XML format for the preservation, exchange and maintenance of bibliographic records, has to have a specific structure and certain metadata fields which contain the information. This information may of course be expressed in XML either by an element or by an attribute belonging to an element.

Most of the bibliographic entries nowadays are based on MARC 21 or variants of MARC, like the previously mentioned NORMARC or BIBSYS-MARC which is used in Norway. Then the pressing question is whether it is prudent to alter the metadata set which already is an existing and well established standard.

In this thesis, a proposed XSD for an FRBRized bibliographic record structure will be thoroughly described. However, a brief look at some of the previous studies in this field is required.
There are other suggested XML-based implementations of the FRBR model, such as in Aalberg *et al.* [2] and Sirris and Strømsodd [7]. The paper 'FRBR i bibliotekkataloger'[2] is based on an XML format which directly reflects the FRBR model, due to the fact that it was easier to develop prototypes and to study the differences between conventional catalogues and an FRBRized catalogue. The set of metadata elements was structured entirely according to the datafield and subfield-based MARC structure. This also goes for [7]. The advantage in this approach is that you don't need to come up with something new in terms of storing metadata pertaining to a single record, and as previously mentioned, users who are familiar with MARC don't need to be reeducated. But is using MARC the only alternative to expressing data of a record?

Describing attributes of a record could be done in any way at all. When this is to be done in XML there are many attribute sets which are supported in XML, such as MARCXML, MODS, Dublin Core and several others. This will be further demonstrated in chapter 4.6.

4 Application and examples of XML design criteria

This chapter contains several examples of XML which are loosely based on an XSD which will be presented in the next chapter. Rather than presenting the XSD up front, the XML design criteria will be presented first accompanied with examples in XML to clarify the design issues. The idea behind this is to give the reader a gradual introduction to the concepts which are implemented in the XSD by guiding him/her through its various areas of application. This is also to show that the general principles of these XML design criteria are ideas that do not necessarily have a specific implementation. The implementation of a metadata format will always depend on which need is the most pressing to the users. Keeping that in mind, the code examples might vary from actual implementation which is shown later on. Reading these criteria, the reader should gain an understanding on a conceptual level and should not be biased by the implemented XSD which is shown later when reading the design criteria. Of course, examples will also be shown to illustrate potential usage and to clarify the concepts. The examples will be brief, and elements not necessary in the examples will be omitted.

4.1 Choosing the root element

When XML is used to describe metadata records, it is inevitable that some element will have to be the root element of a record. XML is based on a hierarchical structure with elements acting as parent and child to each other.

The FRBR Product Model splits up a product in four entities, where Work is the top element and Expression, Manifestation and Item follow underneath each other.

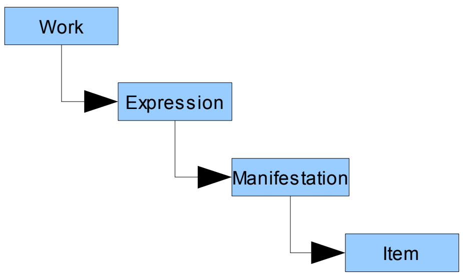


Figure 18: The FRBR Product Model

How is this organisation from a design point of view? If Work is the top element, what are then the advantages and properties of such a structure?

Certainly, as Work signifies the top element concerning point of origin for any specific book, one would think it logical for this also to be the top element implementation-wise. The nature and cardinality of the elements of the FRBR Product model is that a Work can have several Expressions, an Expression can have several Manifestations and lastly a Manifestation can have several Items.
Let us examine further what happens through an XML example where the end tags have been left out for saving space:

```
<work>
       <expression>
              <manifestation>
              <manifestation>
       <expression>
              <manifestation>
       <expression>
              <manifestation>
              <manifestation>
              <manifestation>
       <expression>
              <manifestation>
              <manifestation>
```

Figure 19: FRBR Product Model Implementation in XML

We might end up with a structure similar to this. Notice that this is just one way of implementing the design. A <work> element can have several <expression> elements underneath itself and an <expression> can have several <manifestation> elements underneath itself. The advantage of such a design is that you get a top-down hierarchical view which concurs with the FRBR Product Model.
The problems surrounding this design is that:

- Libraries do not contain conceptual Works, they contain Manifestations. If someone requests a specific book, it is a Manifestation they are after. The imperative thing to search for in those circumstances, are the Manifestations. Your primary information retrieved would in this case be Work, which in reality is secondary information.
- Librarians are not accustomed to thinking in the way of searching for a Work to retrieve a Manifestation. It is the Manifestation which is the focal point also for a librarian.
- It increases the effort to retrieve a specific Expression or Manifestation when this is stored under a Work whose name might not be representative of the name of the Manifestation. Unless you know that the Expression called 'My Fair Lady' would be stored under the Work 'Pygmalion', there could be retrieval problems.
- In those cases where you wish to find the originating Work of a Manifestation, it will be less readable when you potentially have several Manifestations within several Expressions belonging to the same Work.
- Relationships to author occur at Manifestation level, causing important relationships to be less apparent. This point is of course strongly related to the one above.

For these reasons, the conventional approach was abandoned for an implementation which had Manifestation as the top element. This automatically puts the focus on Manifestation being the element of interest both for librarian and user.
In this case, the implementation-look of things are turned around in comparison to the original FRBR Product Model:

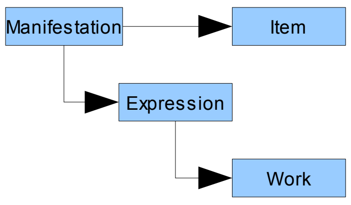


Figure 20: Manifestation as top element

With Manifestation as the top element, its belonging Expression and Work will be stored nested below. Item is also stored beneath Manifestation, but can be optional. The reason for Item being in less of a focus is that it does not contain the most important parts of information.
Let us take another look at a simple XML example demonstrating how this could be implemented:

```
<manifestation>
      <expression>
            <work>
      <expression>
            <work>
```

Figure 21: FRBR Product Model Implementation in XML

The advantages of this design are:
- The primary information sought after is put first, rather than the previous design where Work is top element.
- Secondary information such as Expression and Work is stored beneath Manifestation, thus placing it after Manifestation according to importance.
- The nested hierarchy here is on average easier to read than the previous design.
- Relationships to author are stored directly underneath a Manifestation and these important relationships become clearer/easier to retrieve.

After considering these advantages, reorganizing the FRBR Product Model slightly to conform to this design would bring out a better practical use of the model. The examples used from here on out use Manifestation as the top element unless anything else is specified.

4.2 Expressing relationships in XML

The FRBR Product model presupposes relationships between entities. An example of the hierarchical storing design between the entities of the Product Model is given below:

```
<manifestation>
  <id>c069184ddd1acbf03639434fc8e96dac</id>
  <datafield>...</datafield>
  ...
  <carries>
    <expression>
      <id>78319241a1a876bb25506d5d4f5375ce</id>
      <datafield>...</datafield>
      ...
      <realises>
        <work>
          <id>264ac95b53196df9c5a9e4462424f217</id>
          <datafield>...</datafield>
          ...
        </work>
      </realises>
    </expression>
  </carries>
</manifestation>
```

Figure 22: Using hierarchical storing

A Manifestation, as well as Expressions and Works, have a given number of datafields, but as these are not relevant they are left blank. What is more important in this example, is the way a Manifestation carries an Expression hierarchically below itself, and Expression realising a Work. The <carries> and <realises> elements, belonging to Manifestation and Expression respectively, contain the entire Expression or Work record underneath them. This approach makes it easy to retrieve information concerning either of the three FRBR-entities, as they are all located under the same Manifestation record. There is no cost involved in searching for the belonging records, which makes this a fast method of finding information on all levels of the FRBR model. However, when you for instance have several Manifestations and Expressions that derive from the same Work, this information will be duplicated. As for the example with 'Pygmalion' and 'My Fair Lady', the information on the Work 'Pygmalion' would be fully present under both the Manifestations 'Pygmalion' (the theatrical play) and 'My Fair Lady' (the musical). If such duplicate relations occur with high frequency, it will increase the size of the xml document considerably.

4.2.1 Hierarchical storing between Manifestation, Expression and Work

One of the most important relationships are between Manifestation, Expression and Work. Hierarchical storing could ideally be used to express this relationship. The reasons for this is that:

- There is relatively little duplication
- These entities are semantically closely related and this is elegantly illustrated using hierarchical storing

To reiterate some of the principles from the FRBR model, there must be a way to differentiate between Work, Expression, Manifestation and Item and to express relations between them. If we consider the Work 'Pygmalion', this has further led to the musical called 'My Fair Lady', so both the theatrical play 'Pygmalion' and the musical 'My Fair Lady' must be linked to the original Work 'Pygmalion'. If we express this with FRBR terminology, 'My Fair Lady' is an alternative Expression of the Work 'Pygmalion'.

Some of these functional requirements which have arisen from the FRBR model are poorly represented in older formats, such as MARC. Below follow examples on both of the above mentioned relationships and how they could be expressed using FRBR as a conceptual backbone and XML as the physical format of implementation.

```
<collection>
  <record>
    <manifestation>
      <datafield tag="200">
        <subfield code="a">My Fair Lady</subfield>
      </datafield>
      <carries>
        <expression>
          <datafield tag="200">
            <subfield code="a">My Fair Lady</subfield>
          </datafield>
          <realises>
            <work>
              <datafield tag="200">
                <subfield code="a">Pygmalion</subfield>
              </datafield>
              ...
            </work>
          </realises>
          ...
        </expression>
      </carries>
      ...
    </manifestation>
    ...
  </record>
</collection>
```

Figure 23: Relation from the Manifestation 'My Fair Lady' to its work 'Pygmalion'

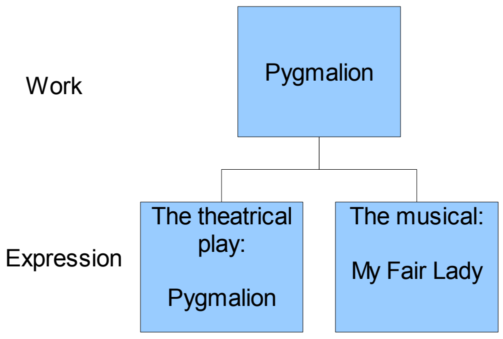


Figure 24: The FRBR relations between 'Pygmalion' and 'My Fair Lady'

The figure above shows a snippet of XML which gives an example of how 'My Fair Lady' could be linked up to its original Work 'Pygmalion'. The next figure displays how the two Expressions relate to each other in an FRBR sense as an auxiliary explanation. The XML example contains Manifestation, Expression and Work from the FRBR terminology. With the Manifestation and Expression both being 'My Fair Lady', the Expression *realises* the concepts and ideas of the Work 'Pygmalion'.

Relationships between a Work and its parts

Let us delve into the relationship called 'Work has part' and pursue it as an example. It must be possible to express a relation to a Work which has parts or related Works. This is only one important relationship as there are many other relationships which need to be expressed such as 'Work has a subject' or 'Work is created by' or 'Person has created'. In previous formats, these types of relations have been difficult to express. When you have a Work which is a part of a bigger Work, such as 'The Fellowship of the Ring', where this bibliographic entry is one of the three parts of 'The Lord of the Rings', this must be expressed somehow. Ideally, this relation must be expressed both in the Work having parts and the Work being the part.

Shown after this paragraph, the figure illustrates a way to make relations to a Work that has parts. In the example below, the elements which are called <has_part> and <is_part_of> are used to express the relationship between these Works. The Work which has parts is 'Lord of the Rings' which has in this example been given the simplified ID of '100'. The subsequent Works are given the ID of '101' and '102'. The three digit ID-number is given for simplifying purposes only, whilst in reality this would be either an IDREF to a record in the same document or an HREF to a record in another document. As the reader will notice, this particular relationship is represented by an element exclusive for its use, and it would be a good decision to give the most used relationships their own element. It is a better practice to give frequently occurring types of information their own element, than to for instance have a general element with an attribute that indicates its specific purpose each time. This is because it will make it easier to emphasize and take notice of elements that have the highest frequency of use.

With the simplification of the ID, the example below shows deviance from the actual implementation, but the intention is that the reader understand the concept of relating different records together and under which circumstances this is prudent, which is the same in both the example and reality.

A Work which has a part is a typical example of a relationship which, depending on other implementation, could be wise to express using IDREF. A Work is an entity having a lot of relationships and has several relationships pointing to other Works. In order to avoid infinite trees one must use IDREF somewhere and this is a typical relationship to employ IDREF.

The diagram below displays the interconnectedness of the three Works 'The Lord of the Rings', 'The Fellowship of the Ring', and 'The Two Towers'. The last book of the trilogy, 'The Return of the King' has been left out since it is not relevant to the example. What must be stated however, is that 'The Lord of the Rings' is sometimes published as one, three or even seven books[14]. Therefore one is able to have 'The Lord of the Rings' as a Work by itself, or merely as a 'pseudo Work', functioning as a container for the other three or seven books, and there must exist a way of making these relations.

```
<collection>
  <record>
    <work>
      <id>100</id>
      <title>The Lord of the Rings</title>
      <has_part>101</has_part>
      <has_part>102</has_part>
       ...
    </work>
    ...
  </record>

  <record>
    <work>
      <id>101</id>
      <title>The Fellowship of the Ring</title>
      <is_part_of>100</is_part_of>
       ...
    </work>
    ...
  </record>

  <record>
    <work>
      <id>102</id>
      <title>The Two Towers</title>
      <is_part_of>100</is_part_of>
      ...
    </work>
    ...
  </record>
</collection>
```

Figure 25: A Work which has parts

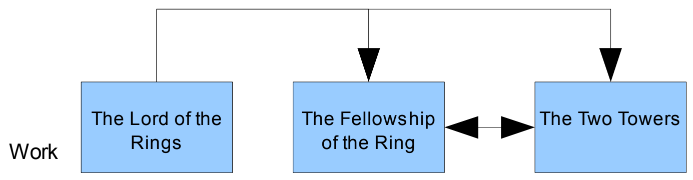


Figure 26: The FRBR relationships between books contained within The Lord of the Rings

4.2.2 Using IDREF between Manifestation, Expression and Work

In consideration to the three FRBR entities Manifestation, Expression and Work, we are faced with three options as to how we can use IDREF referencing:

1. Reference Work
2. Reference Expression with nested Work
3. Referencing between all three entities

Note that the element Item is left out of these options. This is due to its low frequency of use and therefore the insignificant effect it would make to reference it instead of hierarchically storing it when in a large collection.

Considering the elements of the FRBR Product model, the first alternative is to store the Expression record nested in Manifestation and reference Work in Expression, The second alternative is to reference Expression in Manifestation and store Work nested in Expression. The third alternative is to use IDREF between all the three entities. The figure below will illustrate this.

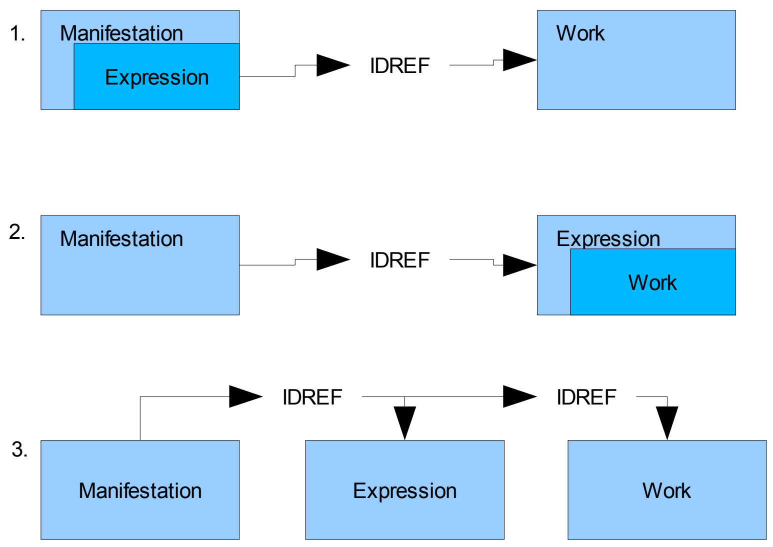


Figure 27: The three possible designs for referencing between entities in the FRBR Product model

Alternative 1 : Reference Work

As we see illustrated in the previous figure, the first alternative is to contain the entire Expression record within the Manifestation record. In the Expression record there is a field called <realises> which either can contain an entire Work record or reference to it. The referenced <work> element is located somewhere else in the same or a different document. This alternative would be wise if statistically there is on average more Expressions per Work than Manifestations per Expression. That is to say, that a Work generally has several Expressions and on average the number of Manifestations an Expression has is low.

An example of this in XML is found below:

```
<manifestation>
  <id>c069184ddd1acbf03639434fc8e96dac</id>
  <datafield>...</datafield>
  ...
  <carries>
    <expression>
      <id>78319241a1a876bb25506d5d4f5375ce</id>
      <datafield>...</datafield>
      ...
      <realises>
        <idref>264ac95b53196df9c5a9e4462424f217</idref>
      </realises>
    </expression>
  </carries>
</manifestation>

<!-- The referenced Work is located somewhere else in the same or a
different document-->

<work>
  <id>264ac95b53196df9c5a9e4462424f217</id>
  <datafield>...<datafield>
  ...
</work>
```

Figure 28: Referencing between entities in the FRBR Product model; alternative 1

Here the Manifestation first has its id and datafields before the <carries> element occurs. This contains an Expression. The Expression is carried in a normal fashion concerning hierarchy. Further, the Expression has its belonging id and datafields before the <realises> element occurs. The <realises> element however, does not contain the entire Work record, but only an <idref> element which contains an id. This id functions as a pointer to a Work record which is located somewhere else in the same or in a different document.

Alternative 2: Reference Expression with nested Work

In the second alternative the Manifestation record stands alone and references an Expression record by the use of its <carries> element which again contains an <idref> element. The Expression record will in this case contain an entire Work record. This would then be the right choice if there generally are several Manifestations per Expression and the relationship between Work and Expression is close to one to one.

```
<manifestation>
  <id>c069184ddd1acbf03639434fc8e96dac</id>
  <datafield>...</datafield>
  ...
  <carries>
    <idref>78319241a1a876bb25506d5d4f5375ce</idref>
  </carries>
</manifestation>

<!-- The referenced Expression is located somewhere else in the same or a
different document-->

<expression>
  <id>78319241a1a876bb25506d5d4f5375ce</id>
  <datafield>...</datafield>
  ...
  <realises>
    <work>
    <id>264ac95b53196df9c5a9e4462424f217</id>
    <datafield>...</datafield>
    ...
    </work>
  </realises>
</expression>
```

Figure 29: Referencing between entities in the FRBR Product model; alternative 2

Like the previous example, the Manifestation starts out with its regular fields; id and datafield. When we reach the <carries> element, it this time does not contain an Expression but rather an <idref> element with an id. The id functions as a pointer to a complete Expression record which first contains its id and datafields. Its <realises> element contains an entire Work record.

Alternative 3: Referencing between all three entities

The third alternative is to use referencing between all the three entities. If the general tendency of the collection is that there are several Expressions per Work and also several Manifestations per Expression, this could be a solution to save some storage space.

```
<manifestation>
  <id>c069184ddd1acbf03639434fc8e96dac</id>
  <datafield>...</datafield>
  ...
  <carries>
    <idref>78319241a1a876bb25506d5d4f5375ce</idref>
  </carries>
</manifestation>

<!-- The referenced Expression is located somewhere else in the same or a
different document-->

<expression>
  <id>78319241a1a876bb25506d5d4f5375ce</id>
  <datafield>...</datafield>
  ...
  <realises>
    <idref>264ac95b53196df9c5a9e4462424f217</idref>
  </realises>
</expression>

<!-- The referenced Work is located somewhere else in the same or a
different document-->

<work>
  <id>264ac95b53196df9c5a9e4462424f217</id>
  <datafield>...</datafield>
  ...
</work>
```

Figure 30: Referencing between entities in the FRBR Product model; alternative 3

The first record in the document is a Manifestation with its datafields, and it carries an Expression which in turn realises a Work. The difference here from the previous example, is that neither the <expression> element nor the <work> element is in its full state underneath the <manifestation> element. The only information to be found there is the identificator which points to the respective records which are stored at another place in either the same or a different document. The advantage with this design approach is that we avoid the problem of duplication. With information retrieval concerns however, it might cost more to retrieve records this way, as we first have to search for them.

Which choice to make regarding these alternatives depends on how much you will save by not duplicating the records that occur the most and have the most relationships to other records. If there are not that many relationships of this kind, there might not be a lot to save on it. This subject will be further addressed later on in the Analysis and Discussion chapter.

4.2.3 Relationships seldom used

There are relationships existing which are seldom used and it could be avoided giving these their own elements, as that would lead to considerable sporadic use of them. One way of expressing this with XML could be to have a general element called *relation*, which has an attribute expressing what kind of relation this is and an identifier pointing to the Work being referenced.

```
<collection>
  <person>
    <id>500</id>
    <datafield>...</datafield>
    ...
    <relation type="is subject of" idref="100"/>
  </person>
</collection>
```

Figure 31: A Work which has parts

This example demonstrates how a Person could be expressed to be subject of the entity with ID 100. The attribute 'type' is used to differentiate between which kind of relationship this is.

4.3 Choices regarding use of ID/IDREF

If a combination of hierarchical storing and usage of IDREF is to be done, every relationship needs to be evaluated according to which need is the most prevalent.

Relationships from Work

Work is normally the entity which has the most relationships as it is the conceptual origin of any book and therefore all connotations to Expressions, Concepts, Places, Persons, Corporate bodies and other Works lead from here. The obvious relationship to consider first, is the one going to Expression. It has been the aim of this thesis to maintain relationships between Work, Expression and Manifestation using hierarchical storing, but the exact implementation will be dependent on which need the users might have. Also, statistics regarding these relationships will be given later in chapter 6. They will give notable numbers regarding the most intelligent design choice when it comes to saving space. Ideally, relationships between these entities will always be expressed using hierarchical storing unless there is compelling evidence that using IDREF will avoid much duplication and therefore yield much smaller files.
Some of Work's relationships could be more complex when it comes to storing them hierarchically. These relationships are:

Forward relationship	*Inverse relationship*
Work.is_supplement_of	Work.has_supplement
Work.is_part_of	Work.has_part
Work.is_subject_of	Work.has_subject

The thing these relationships have in common is that they are all self-relationships in term of referencing the same type. The two pairs point back to each other's entity types. If we then use hierarchical storing, we need some mechanisms preventing that a Work includes another Work infinitely. In this case, there needs to be some programming logic which only stores hierarchically when the Work record possessing the relationship already is the top element in this potential relationship tree.
A simpler implementation would in this case definitely be to only use ID/IDREF. This avoids all problems concerning infinite inclusion. It is also important to consider whether it is useful information to include the entire record of a Work when it is merely a supplement, member or subject of. This information is normally regarded as too distant for inclusion as child record beneath Work and IDREF will suffice in this occasion.

Any given Work has a relationship to the originating Actor, being either a Person or a Corporate body. Including the Actor record beneath Work could give a less readable Work record as well as some duplicated data, all depending on the size of the Actor record. Is it interesting for a user to have all the information about an Actor stored inside every Work record he/she has made? It is probably too excessive for a casual user and it will be enough using IDREF. It depends on the need of the user though, as a librarian would probably prefer to include the Actor. This problem is quite similar in the inverse relationship which is explained below in the 'Actor and belonging Works' subsection.

Relationships from Expression

Those relationships from Expression which need to be considered are those which point to the originating Work, the embodying Manifestation and the Actor being responsible. As has been mentioned earlier, relationships to Work and Manifestation will ideally be implemented hierarchically unless it will prevent much duplication to use IDREF. Therefore, these relationships will also be assumed to be hierarchical for now. Furthermore, Expression has the same issues regarding relationships to Actor as does Work.

Relationships from Manifestation

Likewise as Work and Expression, the relationship between these entities are ideally expressed using hierarchical storing. The same issues concerning relationships to Actor are present also here.

Actor and belonging Works

Concerning metadata about a Person/Corporate body, an Actor, there are datafields which accompany it and there are relationships to those Works he/she has written, those Expressions he/she has realised and the Manifestations he/she has produced. How would a Actor record look if his entire repertoire is included hierarchically as full records? Certainly in cases where someone has only one Work affiliated with himself, it would not affect the record dramatically (depending on the size of the Work record). However, many of the Actors will have several Works, Expressions and Manifestations, thus making the Actor record excessively large and removes focus from the data about the Actor itself. Using IDREF instead of hierarchical storing will lead to more readable records and less duplicated information.

4.4 Including records outside of hierarchy tree

Whether or not to include an additional copy of a record outside the hierarchy is something which might be useful at some point. If one wants to examine a record which normally exists within the hierarchy tree but without the surrounding parent and child records, it can be stored separately and directly underneath the root element of the document.

The entities Work, Expression, Manifestation and Item are very closely related and there are not strong reasons to split these up in order to have them separate view of them. Organizing them in a hierarchical fashion which is a standard way of describing relationships in XML, is prudent both in terms of readability and search algorithms.

What could be more interesting to have as separate records in addition to having them in the tree is the Actor and Subject entities.

To illustrate this point, both Subject and Actor will be examined more closely.

Subject metadata

In the library world today, we have the Dewey Decimal Classification to classify Works into different categories and to further express the subject contained within it. There must be mechanisms and means to express this also in XML. One method of doing this is to have a separate record covering the aspects within classification and subject metadata. This data mostly pertains to a particular Concept, either abstract or concrete and even confined to a particular historic period. Some of this subject metadata could also include specific geographical locations and place names. It is important to include also this type of metadata to classify a Work within a certain genre or type. For this example, Concept will be used, although in the XSD presented in this thesis, Concept and Place have been joined into one entity, Subject.

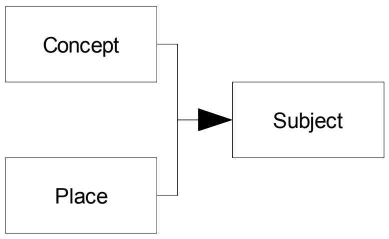


Figure 32: The entities within Subject

```
<work>
  <id>
    264ac95b53196df9c5a9e4462424f217
  </id>
  <datafield>...</datafield>
  ...
  <has_subject>
    <idref type="concept">
      00ab0bd2d78e28ab945244d96c2c6486
    </idref>
  </has_subject>
</work>

<!-- The referenced Concept is located somewhere else in the same or a
different document-->

<concept>
  <id>
    00ab0bd2d78e28ab945244d96c2c6486
  </id>
  <datafield c="1" tag="675">
    <subfield code="v">do 4. izd.</subfield>
    <subfield code="a">094.1=863"1913":886.3-32</subfield>
      </datafield>
</concept>
```

Figure 33: Referencing between Work and Concept

This example exhibits how one could reference between Work and Concept. The relationship between Work and Concept is that a Work might have a specific Concept as a subject. The exact same goes for the relationship between Work and Place. Therefore, the example below is quite similar to how a relationship between Work and Place could be described. Here the <work> element has its basic elements such as <id> and potentially many <datafield> elements. Then we can express that a Work has a subject through the element <has_subject>. Like previously, this relation could be expressed either through hierarchical storing or referencing. The example below employs referencing.
Considering this type of relationship, a particular Concept may be quite wide and general and might be applicable to a large quantity of Works. Therefore, if hierarchical storing is used, a lot of information is potentially duplicated if a Concept occurs within many Works. Only using one's own judgement, one would most likely arrive at the conclusion that referencing would be worth it in this case. The exact numbers concerning duplicates in this relationship will be addressed in the 'Analysis and discussion' chapter and therein will also lie the recommended implementation.

Another view of this matter, is to question whether it is necessary to have Concept as a separate record at all. Could it just be stored directly within a Work record without noticeably superseding the space an IDREF field would take up? Will referencing a Concept record yield a significant amount of saved space or will it merely cause minor differences? In order to answer this, we have to examine a typical Concept record.

```
<concept>
  <id>
    00ab0bd2d78e28ab945244d96c2c6486
  </id>
  <datafield c="1" tag="675">
    <subfield code="v">do 4. izd.</subfield>
    <subfield code="a">094.1=863"1913":886.3-32</subfield>
      </datafield>
</concept>
```

Figure 34: Concept as a separate record using MARC datafields

Had this been a record stored directly beneath a <work> element, the <id> element could have been avoided and thereby reducing the size of the record. In this instance, the 675 tag is used to contain subject metadata. This tag could strictly speaking also be removed to give even smaller <concept> elements. Even though a <concept> record as found in the test collection in this thesis (which is further described in chapter 6.3) is segmented in MARC tags such as 606, 610, 675, these could potentially be joined as they all pertain to subject data. With these observations in mind, we are left with two lines of information, here contained within the <subfield> elements. These do not necessarily have to be expressed using the MARC-based subfield and indicator structure. They could also be expressed using Dublin Core:

```
<work>
  <id>
    264ac95b53196df9c5a9e4462424f217
  </id>
  <datafield>...</datafield>
  ...
  <has_subject>
    <concept>
      <dc:subject>do 4. izd.</dc:subject>
      <dc:subject>094.1=863"1913":886.3-32</dc:subject>
    </concept>
  </has_subject>
</work>
```

Figure 35: Concept as a part of Work using Dublin Core datafields

Now, instead of the Concept record taking up 217 characters, it takes up 123 characters when stored hierarchically and with the <subject> elements from the Dublin Core Metadata Element Set. Even though Concepts are statistically highly duplicated, it will not be a vast increase due to the small average size of the record. This is certainly only one way of doing it, and it differs from the design made in this thesis. In the XSD presented here later, the MARC datafields have been kept.

The main conclusion to draw from this is that Subject does not need to be stored separately as there is little need for it. The records are small and there is little or no interest in retrieving a specific Concept record to examine its potential relations further.

Actor Metadata

Actor encompasses both Persons and Corporate bodies.

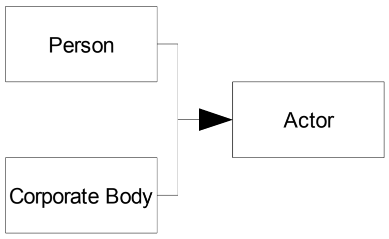


Figure 36: The entities within Actor

The reason for joining Person and Corporate body is that they share a great deal of properties. They share the same properties regarding name and other qualities pertaining to the Person or Corporate body. They are both equipped with relationships in the sense that they are creators, realisers, producers and publishers of Works, Expressions and Manifestations. There are few factors differentiating them in a metadata perspective and therefore they are easily joined as one.

It is beneficial to have two-way-relationships between Actor and his/her respective Works, Expressions and Manifestations. What this means is that a Work will have a relationship to the Actor who has conceived it. The inverse relationship, namely that a Actor has a relationship to the Work he/she has conceived, is also present. This is in order to have an overview of their accomplishments. When searching for metadata about Actors, one will almost certainly be interested in which writings they have conceived, realised or produced. Therefore, this must also be included as metadata about an Actor. It would take an extra effort to search for and retrieve this information explicitly for every occurrence of an Actor. The question however, is whether it would generate a great amount of duplicated data if records are stored nested under each respective Actor. Most probably, this is the case, but more specific numbers regarding this matter will be presented in chapter 6.

The next example is regarding relationships between an Actor and a Work. More precisely, it deals with an Actor and the Work he/she has conceived, realised or produced. In the example below, an Actor has conceived a Work.

```
<actor>
  <id>
    add4deaac43c94e4f2d0b24b4908f93
  </id>
  <datafield>...</datafield>
  ...
  <has_conceived>
    264ac95b53196df9c5a9e4462424f217
  </has_conceived>
</actor>

<!-- The referenced Work is located somewhere else in the same or a different
document-->

<work>
  <id>
    264ac95b53196df9c5a9e4462424f217
  </id>
  <datafield>...</datafield>
  ...
</work>
```

Figure 37: Referencing a Work through Actor

This way of referencing Works through Actor will maintain relationships to all the creations he/she has made without making the Actor record seem too large or lose readability. One way of differentiating between which entity is being referenced is to use reserved terms like:

- <has_conceived> for Works
- <has_realised> for Expressions
- <has_produced> or <has_published> for a Manifestation

To illustrate this point further, the figure below shows schematically how a record would look if there were no referencing used. As seen in the figure below, the three Works are stored directly underneath the Actor record, thereby spacing up the Actor record considerably.

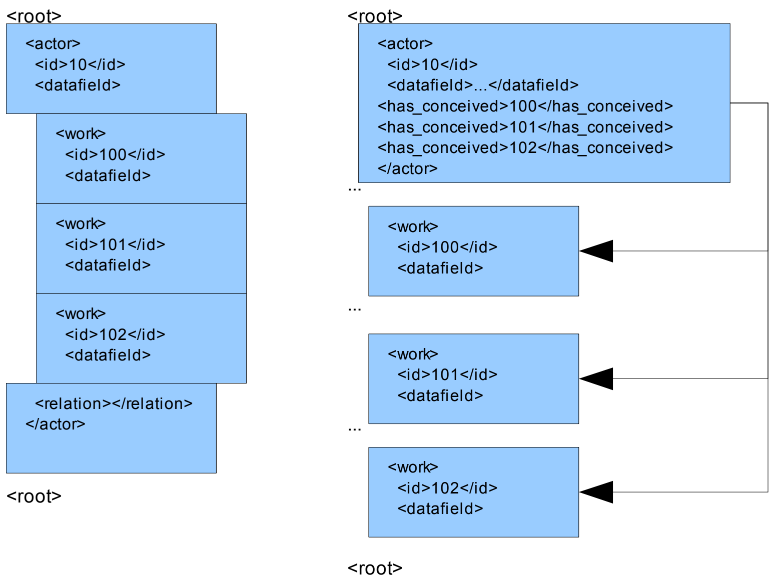


Figure 38: Maintaining readability in an Actor record

The leftmost alternative has all the Actor's Works stored hierarchically within itself. Depending on how many Works he/she has created, the record could become very long. A single Work record could be long itself with all its relationships. Then, further decisions regarding how many of each of Work's relationships should be included as children need to be taken. A simpler and better choice is to express such relationships from Actor to Work with IDREF.

Providing that an Actor's relationships is expressed with IDREF, it could prove a good idea to include Actor as a separate record. This way, a user has easy access to retrieving a particular Actor without the surrounding wrapping of an FRBR entity.

4.5 Readability of XML

To examine the readability of XML in terms of metadata storage, let us examine it in comparison with a MARC record.

```
100 1  $a Rand, Ayn.
245 10 $a Atlas shrugged.
260    $a New York, $b Random House $c [1957]
300    $a 1168 p. $c 23 cm.
655  7 $a Science fiction. $2 gsafd
```

Figure 39: 'Atlas Shrugged' in MARC

In the figure above, we see an excerpt of Ayn Rand's 'Atlas Shrugged' in the MARC format. Knowing in advance that this is a library entry, these fields in particular are quite intuitive and it is not too difficult to deduce the information presented to us here. The first two tags obviously contain the authors name followed by the title of the publication. Both of these fields have indicators and a single subfield, but they leave behind no doubt to what they actually contain. The third field (260) is slightly more diffuse without knowing the subfield codes. $a , $b and $c signify respectively 'place', 'name of publisher' and 'date' of publication. The 300 field contains the information regarding physical description, more precisely $a which is 'extent' and $c which is 'dimensions'. Lastly, 655 contains genre/form of which $a is 'genre/form data or focus term'. The $2 subfield contains 'source of term', but this piece of information will be ignored in this example, as it is used internally in the Library of Congress. After having this explained, the basic MARC fields become apparent to a novice reader. Now let us take a look at the same information, but this time organised in XML.

```
<collection>
  <record>
    <id>randayn#atlasshrugged</id>
    <manifestation>
      <title>Atlas Shrugged</title>
      <publication_event>
        <actor>Random House</actor>
        <date>1957</date>
        <place>New York</place>
      </publication_event>
      <extent>1168p.</extent>
      <dimensions>23cm</dimensions>
      <carries>
        <expression>
          <form>Science Fiction</form>
        </expression>
      </carries>
      ...
    </manifestation>
    <actor>
      <name>Ayn Rand</name>
      ...
    </actor>
  </record>
</collection>
```

Figure 40: 'Atlas Shrugged' in XML

The figure displays the same information as in the previous figure but with the use of XML. Note that this is not any implemented translation of the MARC format, this is merely a tag-by-tag translation of the record excerpt shown above used to state the difference between them. Three dots (...) are used to show that there could be elements left out in order to simplify the example. The organization of the elements is based on the FRBR model. Therefore, such elements as <manifestation> and <expression> occur in this example. If we consider this XML presentation in terms of understanding, a novice reader would be able to understand this format better than MARC, and for someone who has a basic knowledge of HTML this is highly understandable. However, the usage of the elements <manifestation> and <expression> to distinguish the different aspects of a bibliographic entry will require knowledge of FRBR. A reader familiar with FRBR will notice that in this example <manifestation> is the parent of <expression> and not vice versa as the FRBR model suggests. In this particular format, a Manifestation is said to *carry* an Expression*,* although other terms could be used, such as *embodies* or *concretises*. This is one way of illustrating that the FRBR model is conceptual and that the implementation of it will depend on the user's needs. In this case, the XSD is centered around <manifestation> being the parent element and <expression> and <*work*> being child and descendant element respectively. The reason for this is that libraries are centered around the Manifestations of Works and these are the entries which are registered in their databases. The logical relationship and integrity of FRBR is maintained however, as Manifestation still is an embodiment of an Expression and Expression is a further specification of the ideas contained within a Work.

Comparing the size of both these examples, one sees that XML takes up 504 characters (also counting blank characters) whilst MARC takes up 152 characters for essentially the same information. Therefore, with the increase of readability follows often the increase of size. To avoid XML growing too large, research has been conducted on compressing XML files, such as XPRESS[12]. XPRESS is an XML compressor which allows a user to perform queries on compressed XML data. Employing Huffman encoding[13], this technique could be a significant step towards improving query performance with reasonable compression ratios.

4.6 Expressing metadata fields

As previously stated, virtually any set of attributes could be used to describe a metadata record. In this chapter, examples with the use of MODS and Dublin Core will be given.

4.6.1 Using MODS

When there are several XML-based formats out there, such as MODS or Dublin Core, these could be implemented as the method to describe the attributes of a bibliographic record. First of all, MODS uses language based tags, replacing many of the numerical MARC tags. This is definitely an improvement concerning readability and user friendliness. It also maintains the ideas which derive from MARC, so users accustomed to MARC would have a smoother transition to a MODS-based record.

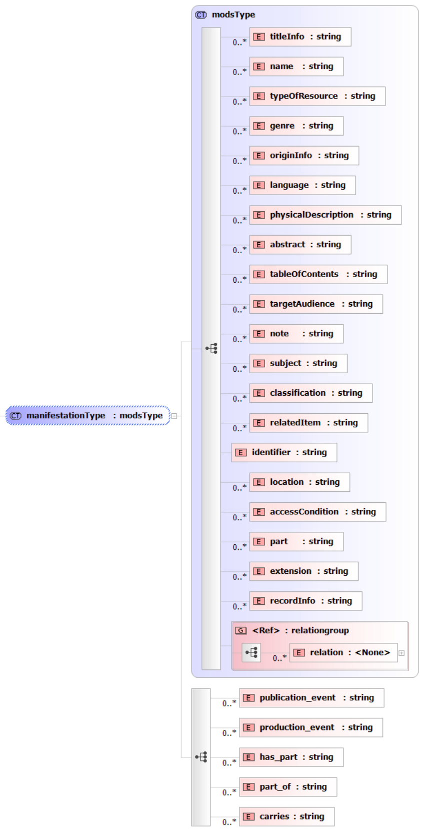


Figure 41: Using MODS to express attributes

This record has an extension base which is called MODStype. It implements the originally 20 top elements of MODS. These have all been set to String in this example, but would in an actual implementation most likely include other elements as well as perhaps text. It would be fully applicable to use these elements to describe the record's data and use FRBR-based elements to describe relationships. If there are any other data present in the record which is not covered by MODS' elements, there is an element called <extension> which covers this purpose. MODS also has elements which covers some of the relationships which the FRBR model promotes, such as <relatedItem> or <part>. If there are certain types of relationships occurring with high frequency, they could be added as separate elements apart from the general <relation> element found in the MODStype. The example does not take any stand as to how to implement the FRBR Product model, although the segmentation of elements between Work, Expression, Manifestation and Item would be fully realisable. This would be done by segmenting the elements which belong to the separate FRBR entities and relating them. The relating of the FRBR entities could of course be done as described in 4.2.2. This particular example has implemented only the Manifestation element of the FRBR Product Model. It has also included such elements as <publication_event>, <production_event>, <has_part>, <part_of> and <carries> which provide functionality that is normally found within Manifestation. Publication and Production is clearly connected to a specific Manifestation and the <carries> element is to contain the Expression which is the origin of this Manifestation.

```
100 1  $a Rand, Ayn.
245 10 $a Atlas shrugged.
260    $a New York, $b Random House $c [1957]
300    $a 1168 p. $c 23 cm.
655  7 $a Science fiction. $2 gsafd

<!-- This MARC record is expressed using MODS' datafields  -->

<record>
  <manifestation>
    <titleInfo>Atlas Shrugged</titleInfo>
    <name>
      <namePart type="family">Rand</namePart>
      <namePart type="given">Ayn</namePart>
    </name>
    <genre>Science Fiction</genre>
    <originInfo>
      <dateIssued>1957</dateIssued>
      <publisher>Random House</publisher>
      <place>New York</place>
    </originInfo>
    <physicalDescription>
      <extent unit="page">1168</extent>
      <extent unit="cm">23</extent>
    </physicalDescription>
    <identifier>randayn#atlasshrugged</identifier>
  </manifestation>
</record>
```

Figure 42: Using MODS to express attributes in XML

The example in XML uses the schema from above. It contains the same information as the MARC record shown at the top, but now organised using MODS datafields. Under elements such as <name>, <originInfo> and <physicalDescription>, there are other elements which further refine and segment data. This segmentation adheres more closely to the actual MODS format.

The decision not to choose MODS was due to the already prominent usage of MARC in the library world and that it would require less of a transition for users. Additionally, the implementation of language-based tags for every element seems as an unnecessary and tedious task when the data already is available in a MARC structure.

4.6.2 Using Dublin Core

A structure very similar to the one above, but instead implementing Dublin Core elements, could be applied. The various DC elements would replace the ones from MODS, but the relationgroup would be intact to maintain the relationship properties. Similarly to the MODS implementation, a Qualified Dublin Core implementation would successfully be able to handle all the data of a record. The same disadvantages as mentioned previously, such as forcing users over to a potentially new format would also be present. Accordingly, the idea of using the DCMES as a basis of metadata field expression was abandoned.

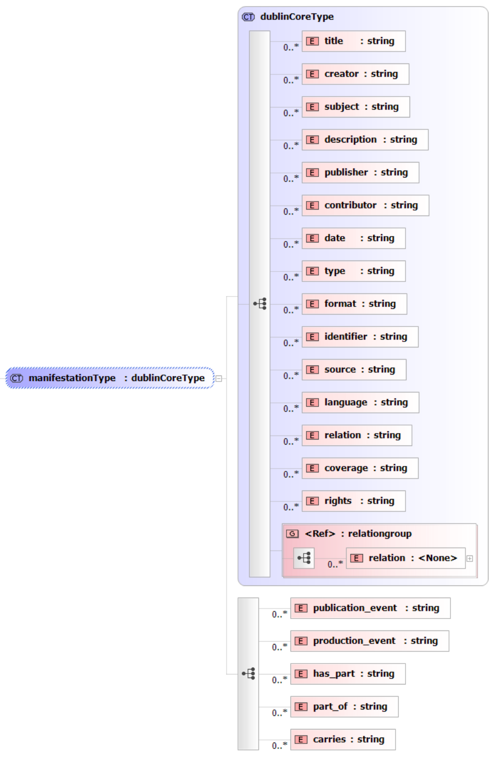


Figure 43: Using Dublin Core to express attributes

Here is a suggested implementation of the Manifestation element using the Dublin Core Metadata Element Set. Elements such as <relation> and the other elements not contained within dublinCoreType are similarly as with the MODS-example included to express relational data and data concerning Manifestations. All elements are optional and repeatable, therefore the cardinality 0..* is put on each element, including the <identifier>. Each element is set to be of type String as according to the Dublin Core definition that every value is a literal string. This could be altered so that the elements contain other Complex Types nested below themselves. However, that would exclude the use of the 'dc' namespace which is used in the following example.

The example below contains the same information as used from the excerpt of a MARC record shown on the top. Here the namespace 'dc' has been used to validate the elements. This example is different from the MODS one in that it contains occurrences of the Dublin Core elements and values of type String. Fields such as <publisher> and <format> have been repeated in order to contain all the data. The <publisher> element contains both of the items contained within the 260-tag (containing Publication, Distribution, Etc.) and the <format> element contains both of the items contained within the 300-tag (containing Physical Description).

```
100 1  $a Rand, Ayn.
245 10 $a Atlas shrugged.
260    $a New York, $b Random House $c [1957]
300    $a 1168 p. $c 23 cm.
655  7 $a Science fiction. $2 gsafd

<!-- This MARC record is expressed using Dublin Core datafields  -->

<record xmlns:dc="http://purl.org/dc/elements/1.1">
  <manifestation>
    <dc:identifier>randayn#atlasshrugged</dc:identifier>
    <dc:title>Atlas Shrugged</dc:title>
    <dc:creator>Rand, Ayn</dc:creator>
    <dc:date>1957</dc:date>
    <dc:type>Science Fiction</dc:type>
    <dc:publisher>Random House</dc:publisher>
    <dc:publisher>New York</dc:publisher>
    <dc:format>1168 pages</dc:format>
    <dc:format>23 cm</dc:format>
  </manifestation>
</record>
```

Figure 44: Using Dublin Core to express attributes in XML

5 The XML Schema Definition

After having looked into different criteria for the design of a bibliographic format, the time has come to approach the more detailed levels of design. The aspects covered in the previous chapter have been an attempt to establish a foundation and give an introduction to the format which will now be presented. Screenshots from Liquid XML Studio[22] will be given.

5.1 Overview of the XML Schema Definition

The XSD which is the basis of this thesis is a proposed FRBRized metadata format supporting FRBR relationships and uses MARC datafields to express metadata fields. It is designed with the intention of reorganizing the MARC format into a way that conforms to the conceptual FRBR framework.

The Record

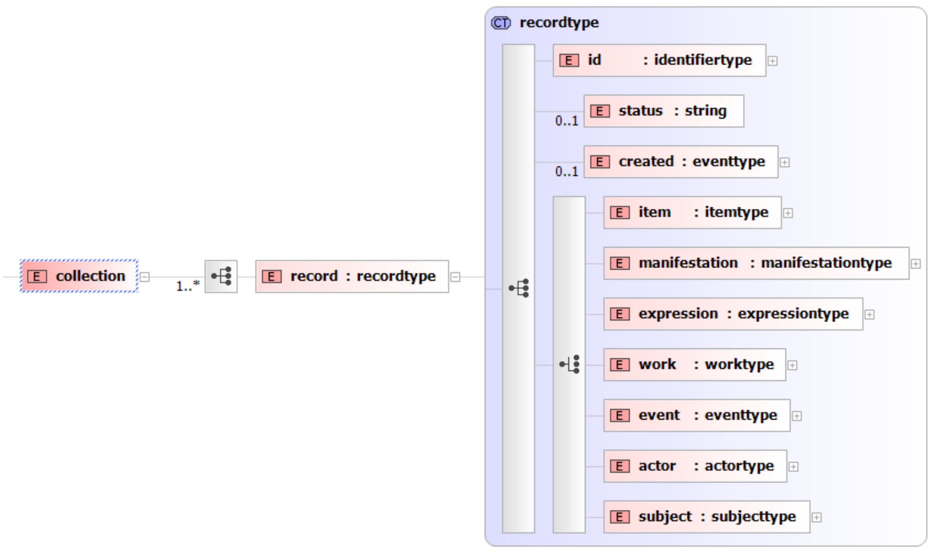


Figure 45: The record element

This figure is a graphical representation of the XSD, which displays the hierarchical relations and cardinality of the elements. The symbols used for notation are E for Element and CT for Complex Type. A Complex Type is a definition for a collection of elements, and a Complex Type can be used as the datatype of an element. This way of thinking can be related to object-oriented programming, in which the Complex Type called recordtype could be regarded as a class definition, whilst the record element could be regarded as an instance of recordtype.

The root element is called <collection>, which can contain 1 or more occurrences of the <record> element. The <record> element contains the attributes <id>, <status>, <created>, <item>, <manifestation>, <expression>, <work>, <event>, <actor> and <subject>. The element <id> identifies the record, <status> contains the status of the record, whilst <created> is an Event element, containing <responsible_part>, <date> and <place>, but this will be further described later. Then we encounter four elements whose names are directly transferred from the FRBR model, which are <item>, <manifestation>, <expression> and <work>. After that follows <event>. The element depicted next in the model, is <actor>, a Person or Corporate body responsible for the record. The last element is <subject>, which is either a Concept or a Place.

The four elements in the FRBR product model are organised in a slightly different manner than what the FRBR model suggests, as Manifestation is here the top element. Before the further description of Manifestation's sub elements begins, one should notice Manifestation's sibling elements <work>, <expression>, <item>, <actor>, <event> and <subject>. They also exist on this layer of the hierarchy because there might be a need to register an own record for a particular entity such as these. An XML Schema Definition such as this will have to make room for all the possibilities there are. In the figure, there are two grey vertical bars within the recordType element. The first one indicates a sequence of elements, whilst the second one indicates a choice between Item, Manifestation, Expression, Work or Actor. That doesn't mean that every record will contain all of these elements, which the above figure might suggest. What it actually indicates are the possibilities that lay at hand.

The Basic Hierarchy

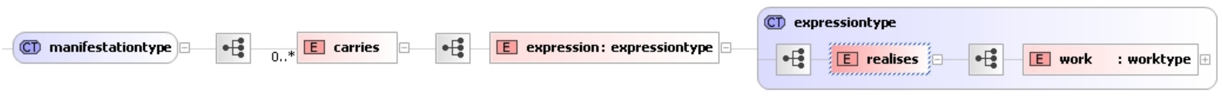


Figure 46: The hierarchy of Manifestation, Expression and Work

This figure shows a selection of Manifestation's child elements, explaining the relationship between Manifestation, Expression and Work. A Manifestation carries an Expression, which in turn realises a Work. Manifestation has been made the super element because a bibliographic entry centers itself around the Manifestation of a Work. As mentioned in chapter 3.2, the library world is mainly preoccupied with Manifestations, as they are the physically palpable entities which librarians observe and handle on a daily basis.

Fundamental Elements

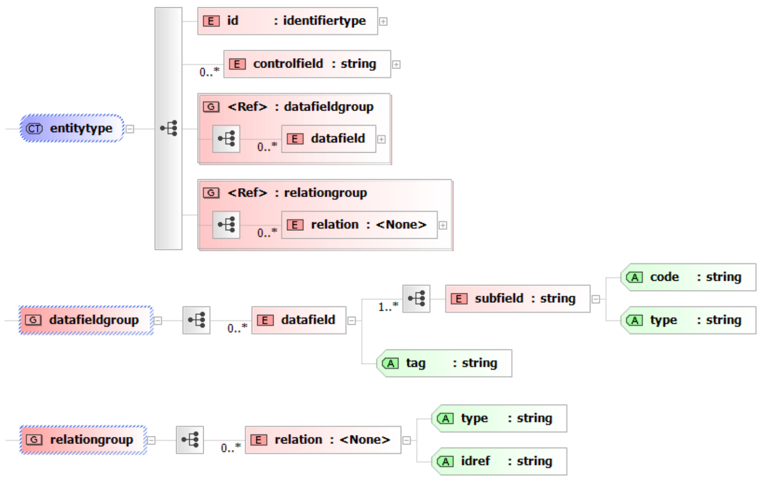


Figure 47: Some fundamental elements in the XSD

The figure above shows some important and fundamental elements of this XSD which are important to know. The element which is called <entitytype> contains the most common elements which are used in Work, Expression, Manifestation, Item, Actor and Event. These commonly used elements have been put in a type of their own, a separate container if you will, which the other types extend from. This is done to ensure that the same set of elements form the basis of each main type.

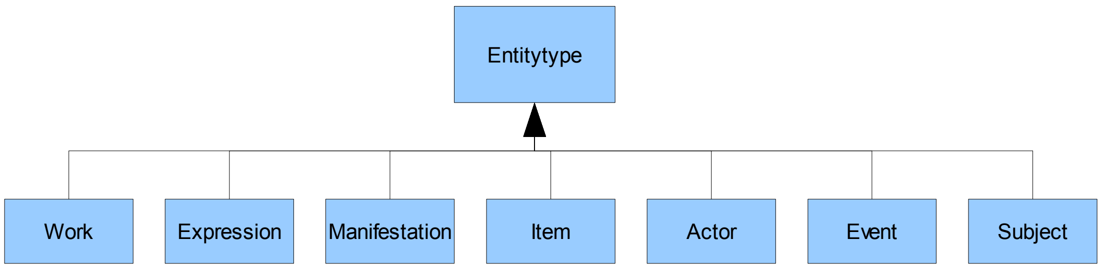


Figure 48: Elements which extend from Entitytype

First, there is the compulsory identification field, simply called id.
Then there is the datafieldgroup which maintains the MARC structure with its datafield, tag, subfield and subfield code. The tag, code and type fields have been implemented as attributes since they act in such close correlation to their respective elements.

This datafieldgroup which describes the attributes of a record could just as well have been implemented with MODS, Dublin Core or indeed any metadata element set which is available. The choice to finally use the MARC datafield structure was taken because it is the most widespread metadata element set and is familiar to most librarians.

Lastly, there is the relationgroup which is not intended to have any content within its element <relation>. Its two attributes however are to store the type of relationship this is and to which other element it is referencing. This element was designed in order to express any kind of relation, but is intended to store the less frequently used relationships. In the next paragraph the actual application of the entitytype is shown.

Manifestation

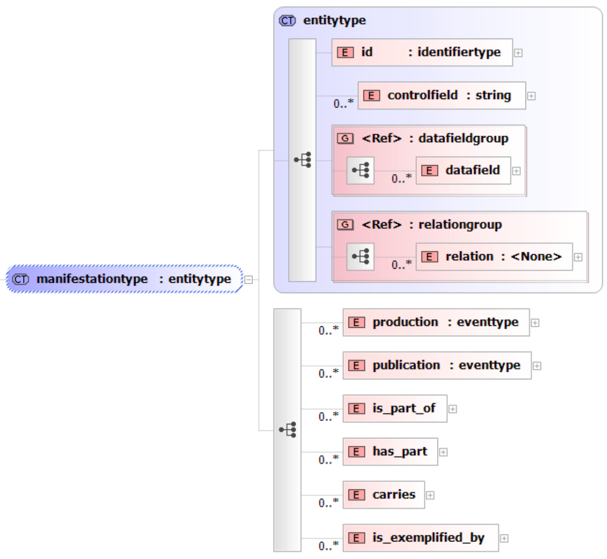


Figure 49: Manifestation

This is the Manifestation element. The extension base of this element is entitytype, which secures that it contains these basic elements. The elements <production> and <publication> contain information pertaining to the publication and production of the Manifestation. The <has_part> and <is_part_of> elements are used in cases where a Manifestation belongs to a bigger Manifestation, such as in The Lord Of The Rings. Here you have the choice between referencing another Manifestation or simply containing it underneath this element. The <carries> element either references or contains an Expression record. The last element called <is_exemplified_by> is used to contain or reference an Item which is connected to this particular Manifestation.

Expression

Note that the extensionbase of Expression, Work, Item, Actor and Event is the same as of Manifestation. The extensionbase Entitytype has only been left out in the examples below to save space.

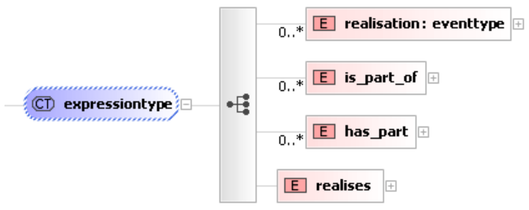


Figure 50: Expression

Expression has all the inherent fields from entitytype, including the elements above. <realisation> deals with the circumstances around the creation of the Expression. <is_part_of> and <has_part>

are used if this Expression is included in other Expressions. <realises> is the element which stores the reference to or the entire <work> element which it realises beneath it.

Work

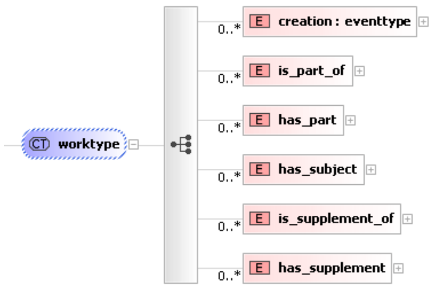


Figure 51: Work

Work has all the inherent fields from entitytype, including the elements above. The three first elements are quite analogous to the ones of Expression. <creation> contains information about the conception of the Work with the belonging Actor being referenced or stored beneath here. <is_part_of> and <has_part> contains either a reference or an element of another Work which is connected to this specific Work. <has_subject> contains a reference or an entire element of something which is a subject of this Work. It can contain any other type found in this XSD. <is_supplement_of> and <has_supplement> are used when the Work in question acts as a supplement or an introduction to another Work.

Item

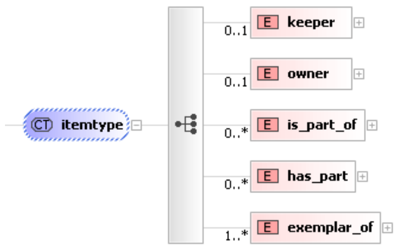


Figure 52: Item

Item has all the inherent fields from entitytype, including the elements above. <keeper> and <owner> are elements regarding who is keeping and who owns this exemplar. <is_part_of> and <has_part> is either a reference to or an entire Item record for which this Item has as part or is part of. <exemplar_of> is a reference or an element of the Manifestation from whence this Item came.

Actor

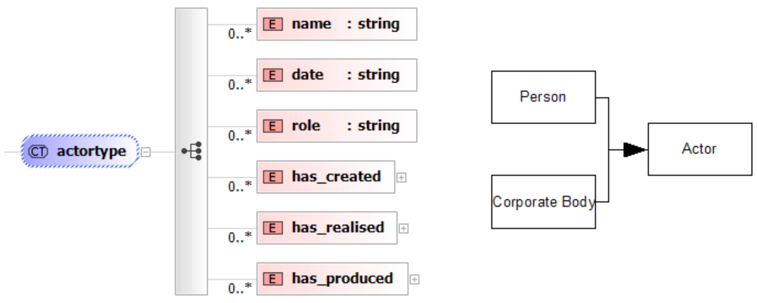


Figure 53: Actor

Actor has the inherent fields from entitytype, including the elements above. It is meant to describe either a Person or a Corporate body. <name>, <date> and <role> are elements which contain a text string and they store the name of the Actor, birth date and the Actor's role. The element <has_created> contains a reference to a Work, <has_realised> contains a reference to an Expression, and <has_produced> contains a reference to a Manifestation. These three last elements thereby contain all the Works, Expressions and Manifestations this Actor has made.

Event

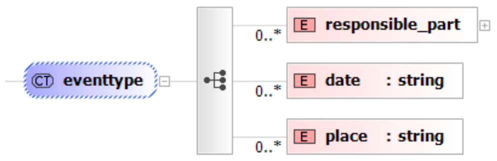


Figure 54: Event

Event has the inherent fields from entitytype, including the elements above. <responsible_part> is an Actor who has been involved in the event and is stored either as a referece or an element. <date> is the date it took place and <place> is the geographical area or region where it happened. Event is used in conjunction with the creation, realisation, production and publication of Work, Expression and Manifestation respectively.

Subject

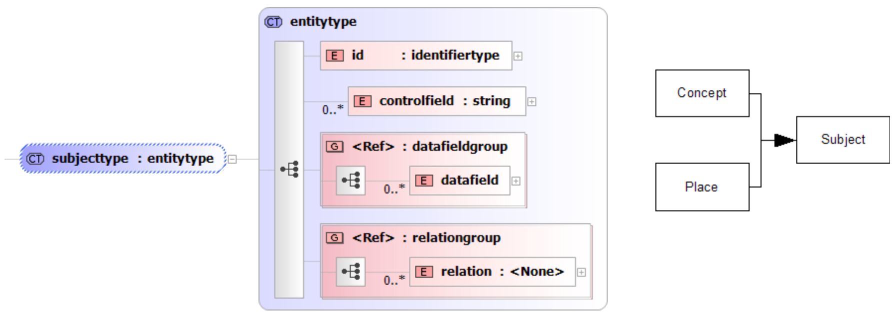


Figure 55: Subject

Subject does not have any elements which are unique for its entity. It merely contains the elements derived from entitytype. All the data which is to be expressed by this entity can be expressed using the datafield elements. Subject is intended to keep data both regarding Concepts and Places. No relational elements have been given to this entity as Concepts and Places mostly only have relationships pointing towards them, as is the case in the test collection used in this thesis, described in chapter 6.3.

5.2 Relation between elements

The relations are the essential connectors between entities in this schema. The relations between elements have been expressed using a mix of hierarchical storing and IDREF.

Flexible referencing

A flexible property of the XSD is that it permits any relationship to be expressed either by hierarchical storing or IDREF. This is done in XML Schema by using groups which contain both an option for IDREF as well as the record being referenced. An example of this is the 'workreferencegroup':

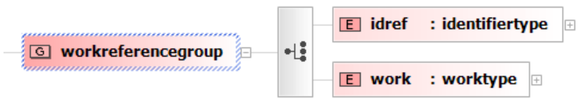


Figure 56: Work reference group

This way of grouping the option for IDREF or the entire record is done with all the other elements which can be referenced as well. To pursue this example further, the 'workreferencegroup' is used in the relationship 'Expression realises Work', expressed with the <realises> element in Expression.

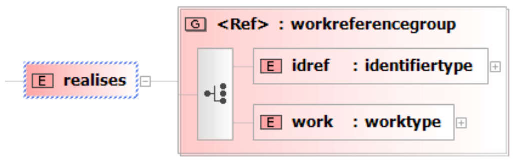


Figure 57: Expression's <realises> element

Let us see how this looks in XML:

```
<expression>
  ...
  <realises>
    <idref>8a657c33e8a7fc423f9fdc2f84b763eb</idref>
  </realises>
</expression>

<!--Both of these implementations are valid according to the XSD -->

<expression>
  ...
  <realises>
    <work>
      <id>8a657c33e8a7fc423f9fdc2f84b763eb</id>
      <datafield>...</datafield>
      ...
    </work>
  </realises>
</expression>
```

Figure 58: Referencing a Work through Actor

Either using IDREF to reference the Work or including the entire Work record hierarchically is valid according to this design.

Relationships seen on a larger scale

As previously mentioned, the hierarchy has been turned around, making Manifestation the top element.

The different entities have relationships to other entities in this way:
A Manifestation is the root element, containing Expression which again contains a Work. An Item can be contained underneath Manifestation. A particular Subject is stored under Work. An Actor is however possessed with relationships to Work, Expression and Manifestation. Relations from Work to Work, Expression to Expression and Manifestation to Manifestation are also present, and are recommended to be expressed using IDREF between them.

In the figure below, a schematic overview of the relationships between the FRBR entities Manifestation, Expression, Work, Item, Subject and Actor is given. The box around the entities Manifestation, Expression, Work and Item signify that all of these entities have relationships to the outer entities Actor and Subject. An Actor as well as a Subject may be 'shared' between several records, in that a unique Actor or Subject may have relationships to many different entities.

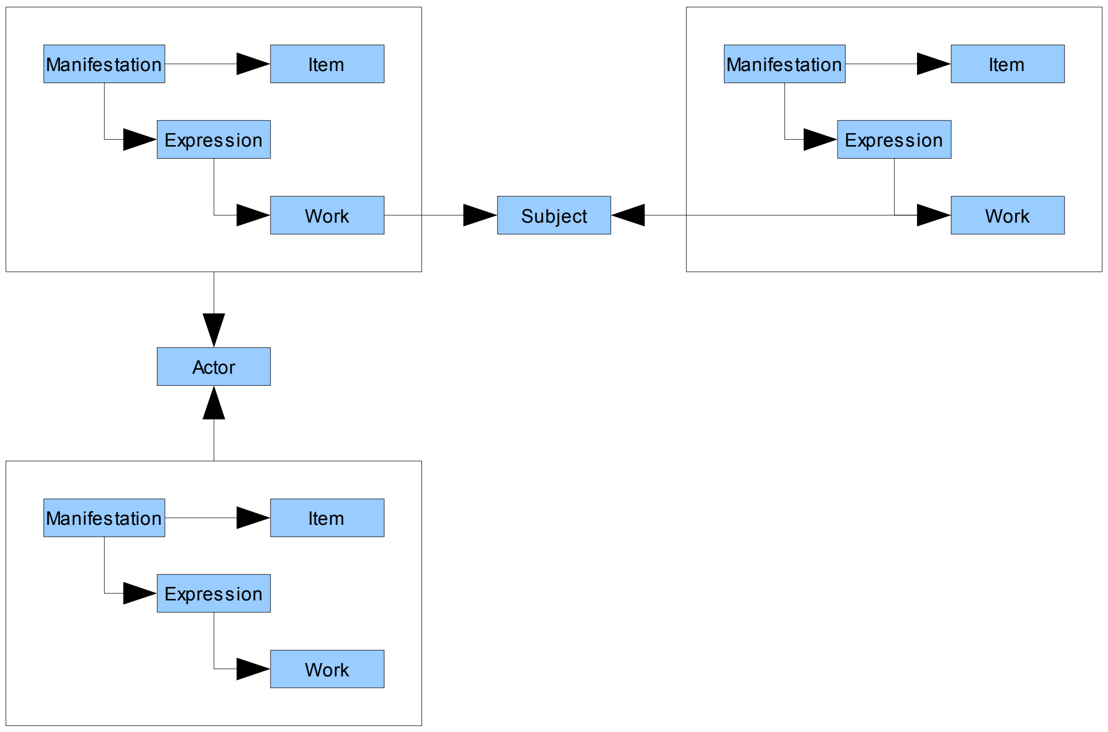


Figure 59: Relation between elements

6 Analysis and Discussion

In this chapter, the choices made in implementation will be analysed and discussed in consideration of the design criteria.

6.1 Application of the FRBR model

The FRBR model is the conceptual basis for the XSD described in this thesis and has of course greatly influenced its design. The product model, containing *Work, Expression, Manifestation* and *Item* make the fundament of the XSD. There are however some differences in comparison to the FRBR model as known from IFLA.

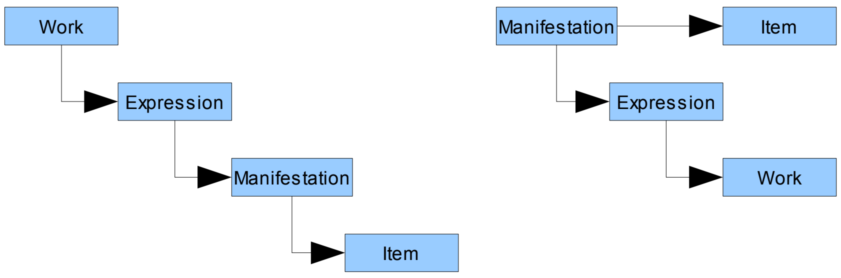


Figure 60: The FRBR Product Model; Regular hierarchy vs. hierarchy with Manifestation as top element

As the reader observes, the model to the left in the figure above is the same as the standard FRBR model described earlier in section 2.2. This is the structure of the product model in which a Work is realised by an Expression, an Expression is embodied in a Manifestation, and a Manifestation is exemplified by an Item. The structure of the product model used in this project, shown in to the right in the figure above, is slightly different as it is turned upside down. Manifestation has here been made the top element, Expression lies beneath Manifestation and subsequently Work beneath Expression. Furthermore, information about Item is optional in this model and could be stored beneath Manifestation. The reason for the reverse design of this hierarchy is that Manifestation is the entity which librarians are most closely in connection with. Although the libraries are filled with Items according to FRBR terminology, all Items pertain to a Manifestation and it is the Manifestation which is the focal point concerning metadata storage. Items might have individual differences over time and information about condition, location and other physical properties would be stored under Item. However, this information is not the focal point of this model, but it is fully possible to include it. As we shall see later, the frequency of the element Item is quite low when examining an actual XML database. Thus, the design choice to handle Item as an optional element is prudent. The information contained in the Manifestation layer concerns the edition of a book and it is here the vital metadata begins.

6.2 Application of MARC

The XSD previously presented was not developed specifically with basis in MARC. Initially, it was considered whether or not to implement an element structure more focused on fully legible and intelligible element names. Such a structure, had it been implemented, would retrieve each MARC-tag and transfer it to a corresponding element with a fully descriptive name. Thus it would have been a totally disparate structure from that of MARC-based formats. This would have resulted in a more complex conversion process in which every potential MARC tag would have to have a designated element. Alternatively, the tags used most frequently could have been given their own elements, whilst lesser used ones could have been placed in a datafield-like structure, much like the one in the figure below. Had such an approach been chosen, it would have to adhere to statistical information as to which tags occur the most. A solution like that using both translated and MARC elements would probably be more confusing than helpful. Users would have to learn which tags that were translated to separate elements and which ones that kept their original MARC structure. Eventually, these ideas was abandoned for simply implementing the already existing and well established MARC-based datafield structure. Not only is this type of implementation easier to do than to translate every single tag, but it will be less of a transition for the users of the previous format. The library world is accustomed to thinking in a 'MARC' way and changing that could possibly lead to more problems than solutions.

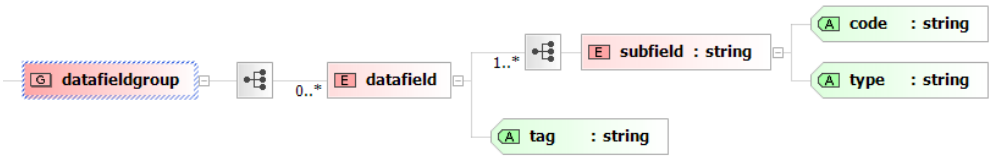


Figure 61: The MARC datafield structure

6.3 Testing

This section describes the structure of an existing and implemented XML database which has been used as a test collection both in respects to transforming the collection to the new format and as statistical data. The database contains metadata for the entire Slovenian National Bibliography. It has been FRBRized, so that it conforms to the conceptual FRBR model. It is an internal and intermediate FRBRized format which is not intended to be used as any final implementation.

This test collection is presented now because the subsequent section use information and statistics gathered from this collection.

6.3.1 Area of use

Using the collection from the Slovenian Nation Bibliography there is to be developed a format in conjunction with the FRBR model and which to a greater extent offers possibility for exchange. Initially, all the original records had to be translated and split up into all the entities in the FRBR Product Model. Data which has been present originally in the MARC datafields have been segmented to their belonging entities. Relationships have been maintained by placing <relationship> elements between the appropriate entities. Certain other attributes have been added for internal maintenance, but these will be left undiscussed.
This FRBRized database is thus the end result of a translation from the original database. It is solely an intermediate format used for internal investigation and test purposes.

The reasons for using this collection as test data are:
- It is a complete collection
- It is fairly small
- It has many relationships
- It is a representative collection
- It is also used in a different project connected to NTNU

The main purpose of using this database as a test collection, is to ensure that the new format is able to accommodate all of the data and functionality which is present here. It is not ultimately intended to be used for this purpose, although it might be in the future. The database is equipped with labels for explanation purposes and has all records stored sequentially directly beneath the root element <collection> and therefore is easily searchable and accommodates the use of XSLT for transformation.

6.3.2 General Structure

The general structure of this database is that it consists of a root element, <collection>, which contains <record> elements. These records are differentiated into 7 different types and are distributed over a total of 407805 records in the collection:

Label	*Frequency*
Work	102267
Expression	100908
Manifestation	69961
Person	49637
Corporate body	6219
Concept	75507
Place	3264
Total	407805

The first three types are exactly the same as those that have been discussed previously and which are directly derived from the FRBR Product Model. The Item entity is missing from this overview, simply because it has too few entries in this collection to be of any significance. When looking upon the cardinality between the types, we see that an Expression can only realize one Work, but a Work can be realized by several Expressions. An Expression can be embodied by several Manifestations, and the same goes for the inverse relationship, namely a Manifestation can be the embodiment of several Expressions. This is due to the fact that a Manifestation can act as an introduction to an Expression which is different from the one it originally is embodying.

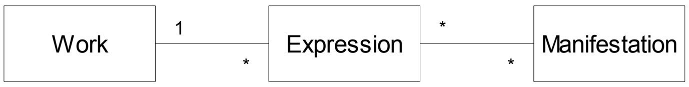


Figure 62: Cardinalities between elements

Person and Corporate body are entities which can be responsible for the creation, realisation or production of either a Work, an Expression or a Manifestation.
Concept and Place can act as the subject of either a Work, an Expression or a Manifestation.

There are however some differences from the paradigm of this thesis. Person and Corporate body are two entities whose properties are so alike that the information has been put in the <actor> element in this thesis' XSD. Also Concept and Place have been merged into the <subject> element as they both are subject data and actually differentiate very little in terms of relations and use of datafields. Another reason for both of these mergers is that Corporate body and Place both have very few occurrences in comparison to their counterparts.

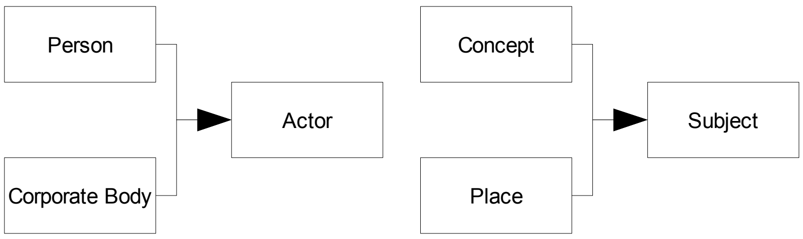


Figure 63: Unification of elements into Actor and Subject

Each <record> element implements a datafield structure which is identical to the one in this thesis' XSD. It uses MARC datafields, subfields and subfield indicators.

There are 25 different types of relationships between the above mentioned entities.

RelType	*Label*	*Frequency*
5.2.1.1.F	Is realized through	101181
5.2.1.1.R	Is a realization of	101181
5.2.1.2.F	Is embodied in	122813
5.2.1.2.R	Is the embodiment of	122813
5.2.2.1.F	Has created	104781
5.2.2.1.R	Is created by	104781
5.2.2.2.F	Has created	9273
5.2.2.2.R	Is created by	9273
5.2.2.3.F	Has realized	67958
5.2.2.3.R	Is realized by	67958
5.2.2.4.F	Has realized	44
5.2.2.4.R	Is realized by	44
5.2.2.5.F	Has produced	22108
5.2.2.5.R	Is produced by	22108
5.2.3.1.F	Is subject of	374
5.2.3.1.R	Has as subject	362
5.2.3.10.R	Has as subject	5351
5.2.3.5.F	Is subject of	6569
5.2.3.6.F	Is subject of	3118
5.2.3.6.R	Has as subject	3118
5.2.3.7.R	Has as subject	149454
5.3.1.6.F	Supplements	42529
5.3.1.6.R	Has as supplement	42529
5.3.1.8.F	Is part of	21263
5.3.1.8.R	Has part	21263

These 25 relationship types are the ones used in the database for our statistical data purposes. Notice that there are two-way relations between elements as for instance '5.2.1.1.F=is realized through' and '5.2.1.1.R=is a realization of'' which is the inverse of the first one. That also explains why these two relationships have exactly the same frequency.
The labels of these types give are pretty self explanatory, except for possibly 'Supplements' which means an introduction to a Work. Other than that, the word 'realized' is used between Work and Expression and 'embodied' between Expression and Manifestation.
Noticeable is it also that some of these relationships have the same label, such has 'has realized' and 'is realized by' which occurs twice, 'is subject of' which occurs three times and 'has as subject' which occurs four times. This is due to them being relationships of the same property but between different entities. For example 'Work has as subject --> Actor' is a different relationship than 'Work has as subject --> Work'.

Below is shown an example record from this test collection. It shows the Work record of 'A Midsummer Night's Dream' by William Shakespeare.

```
<frbr:record xmlns:frbr="http://bibsys.no/frbrized"
                 id="63f25a64995868f79f1808da58f82f63"
                 type="4.2"
                 label="Work">
  <datafield c="1" tag="500">
    <subfield code="a" type="4.2.1" label="Title of the work">Midsummer night's
        dream
    </subfield>
  </datafield>
  <frbr:relationship type="5.2.1.1.F" label="is realized through" target_type="4.3"
                        target_type_label="Expression"
                        href="20d4c5a41d112691dea2f4aba1e52246"/>
  <frbr:relationship type="5.2.3.7.R" label="has as subject" target_type="4.8"
                        target_type_label="Concept"
                        href="54bece3fe4f182a75d5ebabfdbc7af5a"/>
  <frbr:relationship type="5.3.1.6.R" label="has a supplement" target_type="4.2"
                        target_type_label="Work"
                        href="34e5614ad6e627bb394825b1551d134f"/>
  <frbr:relationship type="5.3.1.8.F" label="is part of" target_type="4.2"
                        target_type_label="Work"
                        href="f59426664ca150d645939eb01c326862"/>
</frbr:record>
```

Figure 64: A record from the test collection

Here we see that a typical record consists of the <record> element which has <datafield> and <relationship> as children. The <datafield> elements are in exact concurrence with the ones in this thesis' format and follow the MARC structure. The <relationship> elements all use referencing through the 'href' attribute.

6.3.3 Disadvantages

This collection is fully FRBRized and each record contains MARC datafields and elements expression relationships. It has all the basic requirements of a metadata format and could have been used as the format of choice. However, it is appropriate to consider any disadvantages it might have.

As this is a format which is FRBRized, it is not especially suited for exchange. If files are segmented, there is a great risk that records being referenced are not in the same file. Connected metadata becomes spread apart.

This segmentation also leads to readability issues as it is not that easy for a human to read the XML file and determine which Works, Expressions and Manifestations are related. Therefore, other mechanisms and tools for this retrieval need to be present.

Furthermore, the record structure itself is less readable due to plentiful usage of attributes as explanative labels. Instead of using labels and attributes and relationships defined by a number, the format in this thesis has used language based elements to express relationships.

6.3.4 Application in this thesis

In this thesis, the test collection has been used as a collection to draw experiences and knowledge from. In order to extract the necessary data from this database, several programming techniques have been used:

- Java with JDOM
- XSLT
- Java with MySQL

At first, Java was used with JDOM[23] (Java Document Object Model), which is a 'Java-based solution for accessing, manipulating, and outputting XML data from Java code.' Through the use of JDOM to access the XML-based collection, the different types and relationships could be identified. This made the properties of the collection well known and ensured that every piece of data in the collection was accommodated in the XSD.
Statistics regarding numbers was extracted, accumulated and calculated in Java. This concerns both accumulating the number of unique types and relationships and also the calculation of duplicated records in selected relationships.

XSLT was used to manipulate the test collection so that it was valid according to the XSD developed in this thesis. Through the uncovering of properties in the collection, each datafield and relationship was handled in XSLT and converted over to an XML structure which conformed to the XSD.

Lastly, Java was used together with MySQL in order to extract a final piece of statistic which was duplication in the full hierarchy. The files in the collection were divided into 5000 records in each file. The computer which the Java software was run on, only had 512 megabytes of RAM and was in any case not able to parse XML trees that were much larger than this. In order to calculate the full duplication, it required having every record with every belonging relationship in a searchable scope. It was not possible to accomplish this with simply parsing every file and storing in memory when the collection itself was over 600 megabytes, so another solution had to be thought up. The solution was to insert every record as a row in MySQL with three fields; ID, type and relationships. This meant having every record's ID and relationships available in the searchable scope and the statistics could be extracted.

6.4 Expressing Relationships

Relationships/relations are a vital part of any bibliographical database. In this thesis, relations were previously discussed in section 3.3, and a solution based on giving separate elements to frequently used relations was presented. Lesser used relations were to be given a general relation element, functioning as a 'catch-all'.

In this and subsequent sections, some statistics which indicate frequency of relationships and cardinality between elements of the FRBR Product Model will be given to back up the design choices that have been made. This statistical material comes from a database containing the entire national bibliography of Slovenia. An explanation of this database was conducted in chapter 6.3.

Between the 7 types and 25 relationships existing in this database, there exists 27 combinations. When deciding how to express these relationships, the most frequent combinations of relationships were to be given their own exclusive element. The top 1 to 21 combinations range from 149454 to 3118 in frequency and form the group of combinations high in frequency. The remaining 6 combinations of relationships range from 374 to 5 in frequency and are therefore considered to appear rarely. According to these statistics, the relationships in the high-frequency group should be given their own element in their respective category, whilst the ones in the low-frequency group can be placed in a general purpose element covering all types of relationships.

Two-way relationships

However, two-way relationships are not to be maintained. A two-way relationship has been the paradigm in the FRBRized format which has been used for statistical purposes here, but this also requires more storage space. The two-way relationship paradigm has been included in this format only as a coadjutant but not because of necessity. Certainly are they helpful in means of knowing which other records that are in some manner affiliated with a particular record, as all these relationships are expressly listed for each occurrence. But the information pertaining to relationships are retrievable using XPath or XQuery, so the two-way relationships are strictly speaking superfluous. Consequently, there are reverse relationships which are not necessary in the new format. Only forward relationships are required to implement. But what exactly is a reverse relationship?

Reverse relationship vs forward relationship

A reverse relationship must be seen in conjunction with the already implemented XSD. The natural order of things as it is implemented in the XSD where Manifestation is the embodiment of an Expression is not a reverse relationship, it is a forward relationship. Expression being the realisation of a Work is also a forward relationship in this XSD, although in the conceptual FRBR model, both of these would have been a reverse relationship. Therefore we can conclude that something referring from a lower part in the hierarchy to something further up in the hierarchy is a reverse relationship. This also goes for the Event and Actor elements, which are sub elements of several other elements. For instance the relationship '5.2.3.1.R=has as subject' which goes from Work to Concept, is a forward relationship, whilst had it gone from Concept to Work, it would have been reverse. Self-relationships such as '5.3.1.8.F=is part of' and '5.3.1.8.R=has part', will both be regarded as forward relationships here.

Below is a table displaying all combinations. The ones with high frequency are the combinations worth spending an exclusive element on. The column on the far right displays whether or not the combination ought to be expressed using an exclusive element.

Type	*RelType*	*Target*	*Freq.*	*Own element in new XSD*
Work	5.2.3.7.R=has as subject	Concept	149454	Work.has_subject
Manifestation	5.2.1.2.R=is the embodiment of	Expression	122814	Manifestation.carries
Expression	5.2.1.2.F=is embodied in	Manifestation	122813	*None, maintained hierarchically*
Work	5.2.2.1.R=is created by	Person	104781	Work.creation
Person	5.2.2.1.F=has created	Work	104781	Actor.has_created
Work	5.2.1.1.F=is realized through	Expression	101181	*None, maintained hierarchically*
Expression	5.2.1.1.R=is a realization of	Work	101181	Expression.realises
Expression	5.2.2.3.R=is realized by	Person	67958	Expression.realisation
Person	5.2.2.3.F=has realized	Expression	67958	Actor.has_realised
Work	5.3.1.6.F=supplements	Work	42529	Work.is_supplement_of
Work	5.3.1.6.R=has as supplement	Work	42529	Work.has_supplement
Manifestation	5.2.2.5.R=is produced by	Person	22103	Manifestation.production

Person	5.2.2.5.F=has produced	Manifestation	22103	Actor.has_produced
Work	5.3.1.8.F=is part of	Work	21263	Work.is_part_of
Work	5.3.1.8.R=has part	Work	21263	Work.has_part
Corp. body	5.2.2.2.F=has created	Work	9273	Actor..has_created
Work	5.2.2.2.R=is created by	Corporate body	9273	Work.creation
Person	5.2.3.5.F=is subject of	Work	6569	Actor.relation (General element)
Work	5.2.3.10.R=has as subject	Place	5351	Work.has_subject
Corp. body	5.2.3.6.F=is subject of	Work	3118	Actor.relation (General element)
Work	5.2.3.6.R=has as subject	Corporate body	3118	Work.has_subject
Work	5.2.3.1.F=is subject of	Work	374	Work.relation (General element)
Work	5.2.3.1.R=has as subject	Work	362	Work.has_subject
Expression	5.2.2.4.R=is realized by	Corporate body	44	Expression.realisation
Corporate body	5.2.2.4.F=has realized	Expression	44	Actor.has_realised
Manifestation	5.2.2.5.R=is produced by	Corporate body	5	Manifestation.production
Corporate body	5.2.2.5.F=has produced	Manifestation	5	Actor.has_produced

Although most of these relationships are expressed using a separate element, a few of them have been designated to be expressed using the general element <relation>. Some of these choices require more explanation. Since Person and Corporate body have been joined into one element, <actor>, both of these combinations can be handled by the same element. The relationship 'Person/ Corporate body is subject of' is to be expressed by the general element <relation> because it is rarely used. Other relationships have relatively low frequency (less than 10.000) and still are treated in a separate element. That is due to the fact that these elements already exist expressing relationships with greater frequencies.

6.5 Choices regarding use of ID/IDREF

In the section above, we considered which relationships should get their own elements according to their frequency. Now, there has to be made a decision as to whether this element should have an entire record as a child (nested storage), or if it should merely contain an IDREF pointing to the actual record.

The reader should keep in mind that the XSD supports both hierarchical/nested storing and IDREF in all the relationships which are to follow, so the decisions being made are not fixed but will give a recommendation. What is also very important to remember, is that it is not possible to use nested storing forever, as the nested tree structure would potentially be infinitely large. At some point one has to use IDREF to stop the inclusion of entities. Typical candidates for IDREF are self-relationships and Actor relationships which point to FRBR entities. These relationships will be addressed and explained below.

The recommendations will be given in tables with columns *Relationship name* and *Nested or IDREF* to indicate which recommendation is given.

Event and its relationships

Event is the first entity to get its relationships analysed due to the fact that it is contained within both Work, Expression and Manifestation. Therefore, one particular relationship within all of them will already be explained here. This is:

Relationship name	*Nested or IDREF*
responsible_part	Nested

This relationship belongs to the Event and points to an Actor who was responsible for it. Whether or not to use IDREF here depends a bit on the user, but hierarchical storing is recommended for the most part. An Event which includes Actor as an entire record can become very long depending on how long the Actor record is. On average, the Actor record is short and it will not lead to severe readability problems although there are a few Actors which have many datafields and also many relationships. These are seldom encountered as most Actors have less data connected to them. If however storage is of the essence, it would be best to implement this as an IDREF due to the fact that Actor is one of the most duplicated entities. To make a final decision regarding this matter, hierarchical storing is recommended due to the fact that it draws information about the Actor closer to the entity for which he/she/it is responsible and does not cause a severe increase in size.

Work and its relationships

Firstly, let us look at the elements covering the relationships belonging to Work:

Relationship name	*Nested or IDREF*
creation	Nested
is_part_of	IDREF
has_part	IDREF
has_subject	-
is_supplement_of	IDREF
has_supplement	IDREF

Firstly, there is the <creation> element which is an Event.
The relationships is_part_of/has_part and is_supplement_of/has_supplement are each other's inverted relationships. As mentioned in the Design Criteria, this causes issues if they are to be implemented as hierarchically stored. Programming logic which prevents these relationships from infinitely including each other has to be devised in order to only include these relationships once. It is definitely more easy implementing the use of IDREF in an XSLT stylesheet than maintaining hierarchical storing to a certain extent. It is neither necessary to store these particular records nested as they are not that closely connected to the Work. Both of these relationship pairs will therefore be recommended to be implemented with IDREF.

The relationship has_subject is the most versatile relationship in that it can relate to any kind of entity. Therefore, it will depend on the referenced entity whether hierarchical storing or IDREF is the smartest technique to use. From the test collection, we know that a Work will most certainly have as subject:

Work has as subject	
Entity name	*Nested or IDREF*
Subject (Concept/Place)	Nested
Actor (Person/Corporate body)	Nested
Work	IDREF

A Subject is normally a short record, and even though it will lead to numerous duplications (which will be shown in the next section), this increase in size will not be an immense one. The choice has therefore been made to recommend Subject as a hierarchically nested record.

As far as Actor is concerned, its records are normally a bit larger than of a Subject. However, an Actor is not as duplicated as a Concept, although the actual size concerning storage might be somewhat equal. The main disadvantage of keeping an Actor stored under Work is worsened readability. As mentioned above in the Event discussion, the way that the Actor element is implemented here, with datafields and relationships to other Works, Expressions and Manifestations, the particular long Actor records cause the Work record to lose its main focus. Again, there are things against this view as librarians might prefer to have Actor stored within Work so that related information is stored closely together. The recommendation is however to express this relation using hierarchical/nested storing.

A Work being subject of a Work is a relation which again is touched by the problem of infinite inclusion which was previously explained. The decision to use IDREF is the simplest and most efficient solution also in this case.

Expression and its relationships

Relationship name	*Nested or IDREF*
realisation	Nested
is_part_of	IDREF
has_part	IDREF
realises	Nested

The <realisation> element is an Event and meets the same issues as explained previously for Event. The two self-relationships is_part_of/has_part encounters the same difficulties as for the ones in Work. The same recommendation to use IDREF is given. The relationship expressed through the element <realises> points to the Work which the Expression realises. It has been the aim of this thesis to contend the use of hierarchical storing for this relationship and such is also the recommendation. This relationship functions well in terms of readability and bringing related records close to one another. Also, as is given in chapter 6.6, statistics show that there is not much duplication either, so the increase in data size is not a problem.

Manifestation and its relationships

Relationship name	*Nested or IDREF*
production	Nested
publication	Nested
is_part_of	IDREF
has_part	IDREF
carries	Nested
is_exemplified_by	Nested

The first two elements, <production> and <publication> are Events which deal with the production and publication of this particular Manifestation. These issues regarding Event have previously been discussed, so we move on to the self-relationships <is_part_of>/<has_part>. Again the recommendation will be to use IDREF as we have to draw the line somewhere concerning nested storage. The <carries> element which points to the belonging Expression is also a relationship which is recommended to be nested. The same arguments as in the Expression discussion above also apply here. That also goes for the <is_exemplified_by> which points to an Item.

Item and its relationships

Relationship name	*Nested or IDREF*
keeper	Nested
owner	Nested
is_part_of	IDREF
has_part	IDREF
exemplar_of	*Nested under Manifestation/IDREF*

The elements <keeper> and <owner> are references to an Actor who either is currently keeping or currently owning the Item in question. These relationships are easily implemented as nested. Then we come to the two self-relationships of Item, namely <is_part_of> and <has_part> which should best be implemented as IDREF relationships. The element <exemplar_of> should store the Manifestation from which the Item came, but since the Item itself is normally stored underneath Manifestation, this is strictly not necessary. Should however Item be stored by itself underneath a <record>, this relationships would best be implemented as an IDREF.

Actor and its relationships

Relationship name	*Nested or IDREF*
is_subject_of	IDREF (General element)
has_created	IDREF
has_realised	IDREF
has_produced	IDREF

The relationship 'is_subject_of' is not a separate element. From the previous section we decided to place seldom used relationships in the general <relation> element. The <relation> element uses IDREF which is fine in this circumstance. The three next elements <has_created>, <has_realised> and <has_produced> should be implemented as IDREFs. This is due to two reasons. Firstly, including Work, Expression and Manifestation would lead to the problem regarding infinite inclusion. We have to stop storing hierarchically somewhere, and Actor is a suitable place to stop. The second reason is that depending on how many FRBR entities they have made, it would lead to very large Actor records.

This leads us to the following recap of all relationships:

Type	*RelType*	*Target*	*Freq.*	*Nested or IDREF*
Work	5.2.3.7.R=has as subject	Concept	149454	Nested
Manifestation	5.2.1.2.R=is the embodiment of	Expression	122814	Nested
Expression	5.2.1.2.F=is embodied in	Manifestation	122813	*Nested under Manifestation*
Work	5.2.2.1.R=is created by	Person	104781	Nested
Person	5.2.2.1.F=has created	Work	104781	IDREF
Work	5.2.1.1.F=is realized through	Expression	101181	*Nested under Expression*
Expression	5.2.1.1.R=is a realization of	Work	101181	Nested
Expression	5.2.2.3.R=is realized by	Person	67958	Nested
Person	5.2.2.3.F=has realized	Expression	67958	IDREF
Work	5.3.1.6.F=supplements	Work	42529	IDREF
Work	5.3.1.6.R=has as supplement	Work	42529	IDREF
Manifestation	5.2.2.5.R=is produced by	Person	22103	Nested
Person	5.2.2.5.F=has produced	Manifestation	22103	IDREF
Work	5.3.1.8.F=is part of	Work	21263	IDREF
Work	5.3.1.8.R=has part	Work	21263	IDREF
Corporate body	5.2.2.2.F=has created	Work	9273	IDREF
Work	5.2.2.2.R=is created by	Corporate body	9273	Nested
Person	5.2.3.5.F=is subject of	Work	6569	IDREF
Work	5.2.3.10.R=has as subject	Place	5351	Nested
Corporate body	5.2.3.6.F=is subject of	Work	3118	IDREF
Work	5.2.3.6.R=has as subject	Corporate body	3118	Nested
Work	5.2.3.1.F=is subject of	Work	374	IDREF
Work	5.2.3.1.R=has as subject	Work	362	IDREF
Expression	5.2.2.4.R=is realized by	Corporate body	44	Nested
Corporate body	5.2.2.4.F=has realized	Expression	44	IDREF
Manifestation	5.2.2.5.R= is produced by	Corporate body	5	Nested
Corporate body	5.2.2.5.F=has produced	Manifestation	5	IDREF

6.6 Storage optimization using ID/IDREF

One of the downsides with XML documents is that they might grow very large. The benefit of XML's verbosity leads to a considerable increase in size. As we have seen previously, a machine-readable format such as the MARC format is short and concise but not intended to be readable for a human. As a result, machine-readable formats will generate smaller files.

The possibility to store records either by hierarchical storing or by using ID and IDREF/HREF, brings up the question of storage optimization. The hierarchical storing scheme is a convenient solution to easily finding the relations between Manifestations, Expressions and Works, but it is certain that this duplication of data will lead to bigger files. When you have these three elements which are linked together, what could be the most profitable way to organise them storage-wise? There are four options available, of which the first one has already been explained and has been called hierarchical storing. This method simply includes the records beneath one another in a parent/child relationship, which makes it easy to locate the originating Work from a Manifestation. When employing referencing, the record is not stored nested under another one, but the IDREF is. The IDREF points to a record which is stored somewhere else in the same or a different document. The three different ways to use referencing between Manifestation, Expression and Work has already been explained in section 4.2.2.

Statistics regarding duplication

As in the previous section, the statistical information from the Slovenian database used as our test collection will be used to support which design choice is the wisest concerning storage optimization. From our statistical material, we will therefore look more closely at numbers regarding the frequency of relationships between

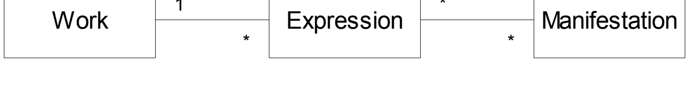
- Work and Expression
- Expression and Manifestation
- Work and Concept
- Person and Work

Finally, we will also check duplication in the full hierarchy.

More precisely the statistics aim to show how relationships are distributed. Each table indicates for example how many Works which have a certain amount of relationships to Expression. The vast majority of Works will therefore have at least one relation to an Expression, and subsequent relation frequencies decline rapidly. The same type of statistics will be given for relationships between Expression and Manifestation, between Person and Work and Work and Concept.

In this context, the number of relationships to another entity could be regarded as the **duplication rate**, as it indicates how many times an entity would be potentially duplicated under another entity if no referencing is used.

If we consider cardinality in the test collection once again, we remember that it is like so:



A Work is embodied by one or several Expressions but one Expression can only realise one Work. An Expression can be embodied by several Manifestations and a Manifestation can be the embodiment of several Expressions because it can act as an introduction to a different Expression than the one it is originally embodying. The relationship '5.2.1.2.R=is the embodiment of' from Manifestation to Expression is therefore not a relationship relevant to consider when it comes to these types of statistics because it contains duplicates inherently.

6.6.1 Statistics : Work is realized through Expression

Firstly, the relationship '5.2.1.1.F=is realized through' with the combination from Work to Expression will be more closely examined.

Work records total	*Relationships total*
102267	*101181*
Number of relationships to Expressions	*Frequency*
0	4376
1	95424
2	1939
3	337
...	...
35	1
Total number of records duplicated	3303

This shows the distribution of the Work records according to how many Expressions they have a relationship to. The statistics above indicate that most of the Works are only realized by one Expression. However, there are 1939 which, according to our XSD, could be reside under two Expressions, making them appear once in surplus. Accordingly, 337 Works could be duplicated under three Expressions, making them appear twice in surplus.

The general formula for calculating duplicated records is:

$$\sum_{i=1}^{n} (d_i - 1) \cdot f$$

d = 'Duplication rate' = 'Number of relationships'
f = 'Frequency'
i = row number

When we apply this formula accumulatively to all duplication rates and their corresponding frequencies, we get that there will be 3303 duplicated records if no referencing is used whatsoever. When the entire corpus of Work records is at 102267, these duplicates comprise a 3,23% increase and therefore hardly make any difference at all.

6.6.2 Statistics : Expression is embodied in Manifestation

The relationship '5.2.1.2.F=is embodied in' from Expression to Manifestation has the following statistics connected to it:

Expression records total	*Relationships total*
100908	*122813*
Number of relationships to Manifestations	*Frequency*
1	90743
2	6175
3	1715
...	...
103	1
Total number of records duplicated	21905

Here too, much like the distribution between Works and relationships to Expressions, most of the totally 100908 records only have one reference. But from these numbers we see that there is a greater number of records which have 2 or more relationships than the previous statistics.
Adding frequencies and duplication rates again we find that there will be 21905 records duplicated with no referencing applied. This represents a 21,71% increase from the original amount of Expression records and must definitely be regarded as a significant augmentation.

6.6.3 Statistics : Work has as subject Concept

Next, the relationship '5.2.3.7.R=has as subject' between Work and Concept:

Concept records total	*Relationships total*
75507	*149454*
Number of relationships to Work	*Frequency*
0	28907
1	35180
2	5259
3	2056
...	...
5556	1
Total number of records duplicated	61207

The 75507 Concept records have the distribution indicated above. The relationships are distributed differently here, as there are 61207 duplicated records. This comprises an 81,06% increase compared to the original corpus of 75507 Concept records.

6.6.4 Statistics : Person has created Work

The relationship '5.2.2.1.F=has created' between Person and Work:

Person records total	*Relationships total*
49637	104781
Number of relationships to Work	*Frequency*
0	12324
1	22360
2	6023
3	2587
...	...
382	1
Total number of records duplicated	67468

The Person records, which number 49637, have relationships to a Work distributed accordingly. If no referencing is used, there will be potentially 67468 Work records duplicated, which is a 135,92% increase. This number clearly shows that if storage optimization is needed, this relationship could be implemented as an IDREF to save some space.

6.6.5 Statistics : Duplication in full hierarchy

Finally, statistics regarding the full hierarchy. These numbers are determined on the basis that all relationships are implemented as hierarchically stored. In this case, the calculations are based on accumulating the number of records being stored throughout the following relationships:

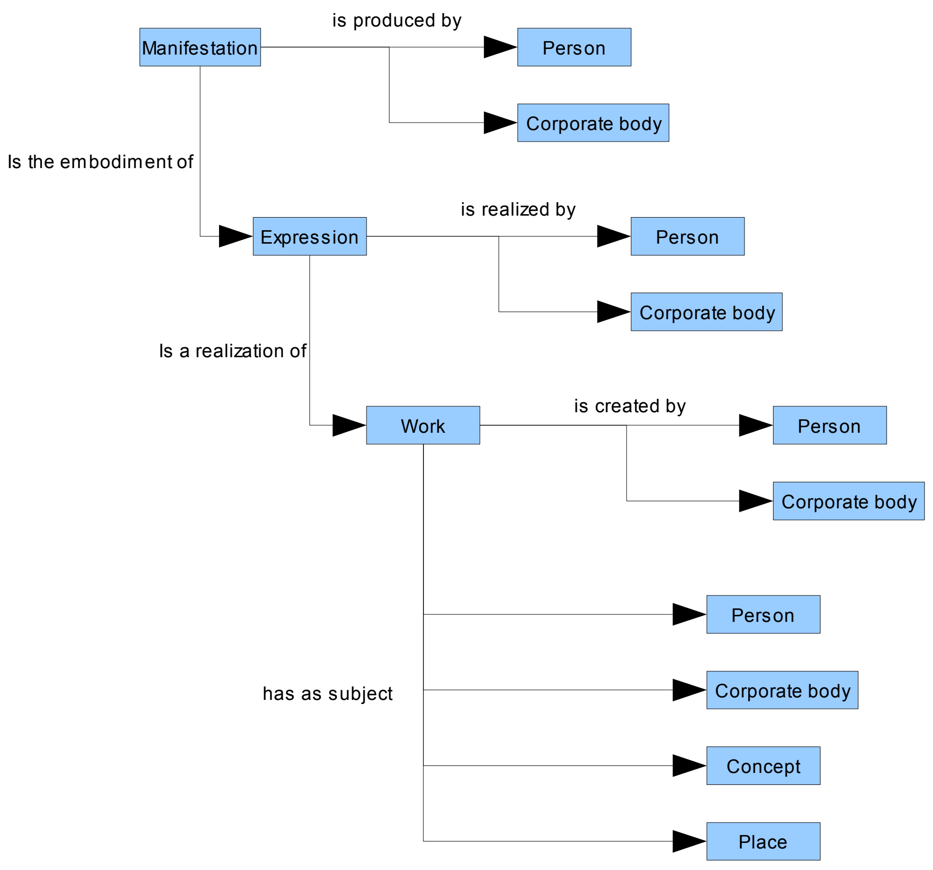


Figure 65: Relationships considered when calculating duplication in the full hierarchy tree

Type	*Unique records in hierarchy tree*	*Records in hierarchy tree*	*Increase in %*
Work	97878	123362	26.04
Expression	100908	122813	21,71
Manifestation	69961	69961	0
Person	45958	243412 (*)	429,64
Corporate body	6211	14067	126,49
Concept	75484	349508	363,02
Place	3264	7423	127,42
Total	399664	930546	132,83

(*)Discovered quite late in this thesis, the relation 'Person is subject of Work' was missing its counterpart, 'Work has as subject Person'. This relation has not been accounted for in the duplication calculation and the number above regarding Person is therefore lower than the actual number. This relation only has a frequency of 6569, and would not have made a considerable increase.

As we see in the table above, the only entity without any duplication is of course the top element, Manifestation. The two entities Expression and Work are moderately duplicated. The elements which normally occur at the bottom of the hierarchy such as Person and Concept are the entities most heavily duplicated. In total, the number of records are more than doubled. Observant readers will also notice that the total number unique of records in the hierarchy tree, 399664, is lower than the number of unique records in the entire database which is 407805. This is because not every relationship has been accounted for, although the most important ones are. There are certain Works that for instance only appear as 'is subject of' for another Work and have no connected Expression or Manifestation.

Concerning storage space, this collection would then be approximately more than double the size it is today, which is an aspect one must consider in implementation. With the storage possibilities available today, this increase should not be a problematic issue.

6.7 Size comparison: MARC vs. FRBRized

In this section, a closer look will be taken on how much data size has been increased from original MARC data to the data which is contained within the test collection. The FRBRized test collection is originally based on MARC data which has been split up into the FRBR entities. This split up has lead to a much larger number of records, as one MARC record has potentially been divided into 8 FRBRized records.

	MARC collection	*FRBRized collection*
Number of records	69961	407805
Size (MB)	74.695	659.333
Average size per file (MB)	1.07	1.62
Size in zip-format (MB)	20.575	94.057

We can see from these numbers that although one MARC record has been split into several FRBRized records, the average size of one FRBRized record is still greater than a MARC record. This is much caused by the XML format itself, with its verbose and space consuming tag-based structure. FRBR also plays a great part in this, as FRBR relationships are expressed in this collection and thereby taking up more space.

7 Conclusion

7.1 Summary

This thesis set out to identify and account for XML design criteria and provide XML examples to clarify and demonstrate the importance of these criteria. When using the FRBR conceptual model as a framework and XML as the physical format for implementation, several design criteria have been discussed. As a result of these design criteria an XML Schema Definition was to be made.

Before the main work could be conducted, introductory theory regarding previous used formats needed to be examined. Afterwards, the decision to use XML was taken, due to the fact that many tools are available that make XML a flexible implementation choice. A brief look was taken to investigate already existing XML-based formats as well as tools for XML. A test collection was used to make sure every piece of data had a designated place in the XML Schema Definition.

7.2 Results

The results from this work has been a rundown of some important XML design criteria in light of the FRBR model concerning subjects like:

- Exchange of metadata
- XML as a metadata format
- Validation
- Applying Manifestation as the top element
- Maintaining readability of XML
- Expressing metadata fields
- Expressing relationships
- Where to use IDREF depending on need

Furthermore, an XML Schema Definition has been designed to concur with the above mentioned criteria. Statistics regarding the duplication of records have been collected from the test collection and have been used as the basis to determine best practices regarding use of IDREF.

7.3 Evaluation

To recap some of the analysis and discussion which has taken place in this thesis, it is appropriate to start with the physical format of implementation, XML, and afterwards look at the conceptual framework which put restrictions on the XML Schema Definition.

XML

The choice to use XML as the format for storage leads to more readable records to also leads to a bigger collection of data. Using both opening and closing tags as well as the overhead of each

record's tags are properties of XML which ultimately causes bigger documents.
XML offers very good flexibility with the tools that are available, such as XML Schema for structuring and validation, XQuery and XPath for data retrieval and XSLT for manipulation. This makes XML a good choice even though increased data size is a consequence.

Use of FRBR and Manifestation as top element

The FRBR model has been the conceptual framework of the work done and the way of approaching the problem has always been executed with the FRBR framework in mind. What is apparent, is that using FRBR increases a MARC-based collection size considerably. The split up of information into the entities within the FRBR Product model not only may lead to internally duplicated data within related entities, but an overhead per unique record will further increase size.
Immediately from the start, different ways of implementing FRBR in XML have been thought of and the most dramatic divergence from the original model is the use of Manifestation as top element. This design choice may initially be perceived as controversial, but after having used this structure for a while it is perceived as an ideal way to implement the FRBR Product model. It is a logic choice to regard Manifestation as the main point of interest for both librarians and casual users, as Manifestation contains information about the physical copy of the entity and Expression and Work are far more abstract entities. As statistics have shown, there is not much increase in size in terms of duplicated entries.

Use of IDREF

Through XML one is able to express relationships with IDREF. Where to use IDREF strongly depends on what a user's need is. In this thesis, the following needs were identified:
- No duplication
- Full editability
- Storage optimization
- Efficient querying
- Readability

The first two needs involve using IDREF exclusively, whilst the other ones involve a combination of hierarchical storing and using IDREF. Hierarchical storing is recommended to be used extensively, due to its advantages of fast search and proximity storage of related entities. IDREF is better used in those cases where an entity has self-relationships or very many relationships which diminish readability.

The need for storage optimization requires IDREF to be used. As statistics have shown, there is little duplication between Manifestation, Expression and Work. The heaviest duplication occurs in the Actor and Subject entities. Ultimately, if storage space is a problem there is most to gain in referencing Actor and Subject. However, it is unlikely that lack of storage space is an issue from this time on and into the future and hierarchically stored relationships are preferable.
To put it more generally, little use of IDREF leads to bigger data collections because of duplication. The opposite, to use IDREF extensively, will lead to less duplication and also a smaller data collection.

When using XML with a bibliographic metadata database where FRBR relationships are normally expressed using hierarchical storing, there has to be used IDREF sooner or later. If not, the hierarchy tree could potentially be infinitely long.

Overall evaluation

How to organise a format for storing bibliographic metadata depends entirely on which data you have and which needs are most important. It could virtually be organised in any thinkable way, so why is the XML Schema Definition presented here a good solution?

- It uses the FRBR model as a framework
- It highlights the most relevant information (Manifestation)
- Important relationships are preferably expressed using hierarchical storing and entities become physically closer
- Relationships expressed with hierarchical storing cost less to retrieve
- It accommodates newly defined relationships through a general <relation> element.
- It has language-based tags

For these reasons, and the previous argumentation and discussion provided in this thesis, the format presented has a great deal to offer in terms of storing bibliographic metadata. It has presented XML design criteria and different alternatives of implementation and their consequences. The design criteria are manifested through the XSD and together they offer important material to consider when designing metadata in XML.

7.4 Future Work

Future work within this theme would be to use the aforementioned XML Schema Definition as a template for an XML format to be implemented in an XML database. This would show how well it could perform in terms of query efficiency and storage space.

Another aspect which would be interesting to investigate is query-optimization. With the hierarchical model that has been presented in this thesis, how much faster is it than implementing every relationship as an IDREF, or indeed is it that much faster? This will depend a upon which type of XML database has been used. What would be interesting to peer into, is methods of improving query-time when IDREF has to be used.

The test collection did not contain any relationships from Concept or Place and therefore did neither the XSD in this thesis implement any specific relationship elements for Subject, although the general <relation> element is present. However, if certain Subjects are indeed Actors, like for example Astrid Lindgren or William Shakespeare, it would be wise to also consider implementing more specific relationships from these entities. This would be valid for those Actors who have such an abundant literature corpus attached to themselves that they become Subjects themselves.

Bibliography

[1] The Library of Congress, *MARC Standards*, http://www.loc.gov/marc/, Last visited 12.10.07

[2] Trond Aalberg, Ole Husby, Frank Berg Haugen og Christian-Emil Ore, *FRBR i bibliotekkataloger*, BIBSYS, 2005

[3] W3C, *Extensible Markup Language (XML) 1.0 (Fourth Edition),* http://www.w3.org/TR/xml/, Last visited: 09.10.07

[4] The Library of Congress, *MARC XML Design Considerations,* http://www.loc.gov/standards/marcxml/marcxmldesign.html, Last visited 11.10.07

[5] National Information Standards Organization (NISO), *Information and documentation – MarcXchange,* www.niso.org/international/SC4/n577.pdf, Last visited 11.10.07

[6] IFLA Study Group on the Functional Requirements for Bibliographic Records. *Functional Requirements for Bibliographic Records - Final Report*. UBCIM Publications - New Series Vol 19. K . G. Saur München 1998. http://www.ifla.org/VII/s13/frbr/frbr.pdf, Last visited: 24.09.07

[7] Berit Eleni Sirris, Aase Margrete Skogvang Strømsodd, *XSL Conversion of Bibliographic Records from BIBSYS-MARC to FRBR*, NTNU, IME faculty, IDI institute Master's thesis, 2006

[8] David Gulbransen, *Using XML Schema*, Que Corporation, 2001

[9] XML Journal, *Eliminating Redundancy in XML Using ID/IDREF*, http://xml.sys-con.com/read/40075.htm , Last visited: 02.10.07

[10]BIBSYS, http://www.bibsys.no/wps/wcm/connect/BIBSYS+Nettsted, Last visited: 23.10.07

[11]Mercury Z39.50 Client | Basedow Information Systems, http://www.basedowinfosys.com/projects/mzc, Last visited: 23.10.07

[12]Jun-Ki Min, Myung-Jae Park, Chin-Wan Chung, *XPRESS: A Queriable Compression for XML Data*, ACM Press, 2003.

[13]D..A. Huffman, *A Method for the Construction of Minimum Redundancy Codes*, Proceedings of the Institute of Radio Engineers 40, pages 1098-1101, September 1952

[14]HarperCollins Publishers Ltd., *Tolkien*, http://www.tolkien.co.uk/, Last visited: 02.11.07

[15]Library of Congress, *Metadata Object Description Schema (MODS)*, http://www.loc.gov/standards/mods/ , Last visited 08.11.2007

[16]W3C, *XML Path Language (XPath)*, http://www.w3.org/TR/xpath, Last visited: 20.11.2007

[17]W3C, *XSL Transformations (XSLT)*, http://www.w3.org/TR/xslt, Last visited: 23.11.2007

[18]Doug Tidwell, *XSLT*, O'Reilly Media, Inc., 2001

[19]Dublin Core Metadata Initiative (DCMI), *Dublin Core Metadata Initiative (DCMI),* http://dublincore.org/, Last visited 29.01.2008

[20]NISO, *Understanding Metadata*, http://www.niso.org/standards/resources/UnderstandingMetadata.pdf , Last visited: 29.01.2008

[21]Relax NG, *Relax NG home page,* http://relaxng.org/, Last visited 11.02.2008

[22]Liquid Technologies, *Liquid XML Studio*, http://www.liquid-technologies.com/, Last visited: 16.05.2008

[23]JDOM, *JDOM*, http://www.jdom.org/, Last visited: 28.05.2008

Appendix

XML Schema Definition code

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com)
-->
<xs:schema xmlns:frbrxml="http://www.example.com/IPO"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="collection">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="record" type="recordtype" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="identifiertype">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="name_space" type="xs:anyURI" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="entitytype">
    <xs:sequence>
      <xs:element name="id" type="identifiertype" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="controlfield">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attribute name="tag" type="xs:string" />
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:group ref="datafieldgroup" />
      <xs:group ref="relationgroup" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="worktype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="creation"
type="eventtype" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="is_part_of">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_part">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_subject">
            <xs:complexType>
              <xs:choice>
```

```
                <xs:element name="idref">
                  <xs:complexType>
                    <xs:complexContent mixed="false">
                      <xs:extension base="identifiertype">
                        <xs:attribute name="type" type="xs:string" />
                      </xs:extension>
                    </xs:complexContent>
                  </xs:complexType>
                </xs:element>
                <xs:element name="work" type="worktype" />
                <xs:element name="expression" type="expressiontype" />
                <xs:element name="manifestation" type="manifestationtype" />
                <xs:element name="item" type="itemtype" />
                <xs:element name="actor" type="actortype" />
                <xs:element name="subject" type="subjecttype" />
                <xs:element name="event" type="eventtype" />
                <xs:element name="any" type="entitytype" />
              </xs:choice>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded"
name="is_supplement_of">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_supplement">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="expressiontype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="realisation"
type="eventtype" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="is_part_of">
            <xs:complexType>
              <xs:group ref="expressionreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_part">
            <xs:complexType>
              <xs:group ref="expressionreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element name="realises">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="manifestationtype">
```

```
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="production"
type="eventtype" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="publication"
type="eventtype" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="is_part_of">
            <xs:complexType>
              <xs:group ref="manifestationreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_part">
            <xs:complexType>
              <xs:group ref="manifestationreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="carries">
            <xs:complexType>
              <xs:group ref="expressionreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded"
name="is_exemplified_by">
            <xs:complexType>
              <xs:group ref="itemreferencegroup" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="itemtype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" name="keeper">
            <xs:complexType>
              <xs:group ref="actorreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="owner">
            <xs:complexType>
              <xs:group ref="actorreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="is_part_of">
            <xs:complexType>
              <xs:group ref="itemreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_part">
            <xs:complexType>
              <xs:group ref="itemreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element maxOccurs="unbounded" name="exemplar_of">
            <xs:complexType>
              <xs:group ref="manifestationreferencegroup" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="eventtype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
name="responsible_part">
            <xs:complexType>
              <xs:group ref="actorreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="date"
type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="place"
type="xs:string" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="actortype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="name"
type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="date"
type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="role"
type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_created">
            <xs:complexType>
              <xs:group ref="workreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_realised">
            <xs:complexType>
              <xs:group ref="expressionreferencegroup" />
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="has_produced">
            <xs:complexType>
              <xs:group ref="manifestationreferencegroup" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="subjecttype">
    <xs:complexContent mixed="false">
      <xs:extension base="entitytype" />
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="recordtype">
    <xs:sequence>
      <xs:element name="id" type="identifiertype" />
      <xs:element minOccurs="0" name="status" type="xs:string" />
      <xs:element minOccurs="0" name="created" type="eventtype" />
      <xs:choice>
        <xs:element name="item" type="itemtype" />
        <xs:element name="manifestation" type="manifestationtype" />
        <xs:element name="expression" type="expressiontype" />
        <xs:element name="work" type="worktype" />
        <xs:element name="event" type="eventtype" />
        <xs:element name="actor" type="actortype" />
        <xs:element name="subject" type="subjecttype" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  <xs:group name="datafieldgroup">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="datafield">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
            <xs:element name="subfield">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="code" type="xs:string" />
                    <xs:attribute name="type" type="xs:string" />
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="tag" type="xs:string" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:group>
  <xs:group name="relationgroup">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="relation">
        <xs:complexType>
          <xs:attribute name="type" type="xs:string" />
          <xs:attribute name="idref" type="xs:string" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:group>
  <xs:group name="workreferencegroup">
    <xs:choice>
      <xs:element name="idref" type="identifiertype" />
      <xs:element name="work" type="worktype" />
    </xs:choice>
  </xs:group>
  <xs:group name="expressionreferencegroup">
    <xs:choice>
      <xs:element name="idref" type="identifiertype" />
      <xs:element name="expression" type="expressiontype" />
    </xs:choice>
  </xs:group>
  <xs:group name="manifestationreferencegroup">
    <xs:choice>
      <xs:element name="idref" type="identifiertype" />
      <xs:element name="manifestation" type="manifestationtype" />
    </xs:choice>
  </xs:group>
  <xs:group name="itemreferencegroup">
```

```
      <xs:choice>
        <xs:element name="idref" type="identifiertype" />
        <xs:element name="item" type="itemtype" />
      </xs:choice>
    </xs:group>
    <xs:group name="actorreferencegroup">
      <xs:choice>
        <xs:element name="idref" type="identifiertype" />
        <xs:element name="actor" type="actortype" />
      </xs:choice>
    </xs:group>
    <xs:group name="eventreferencegroup">
      <xs:choice>
        <xs:element name="idref" type="identifiertype" />
        <xs:element name="event" type="eventtype" />
      </xs:choice>
    </xs:group>
  </xs:schema>
```

Legend for Liquid XML

Symbol	*Explanation*
E Elements	The element symbol denotes the use of an element.
A Attributes	The attribute symbol denotes an attribute of an element. This could for instance be the 'code' attribute of the <subfield> element. For example: <subfield code="a"/>.
CT Complex Types	Complex type is used to construct an XML instance, which may contain a more complex structure of elements. It supports nested element types and mixed content of text and elements. This enables reuse of functionality.
G Groups	Group is used to describe either a an <all>, <sequence> or <choice> element and the set of elements which exist underneath it. Groups can be re-used in complex types.
Sequence	The **sequence** bar is depicted with each node interconnected. Sequence forces elements to appear in a certain order. Elements appearing in any different order will not be valid according to the XML Schema.
Choice	The **choice** bar is depicted with the parent node only having a connection to one child node. Choice only allows one type of child node to occur below this bar. If sibling elements of different types appear below the choice bar, this is not valid according to the XML Schema.

Glossary and abbreviations

CSS	Cascading Style Sheets
DCMES	Dublin Core Metadata Element Set
DCMI	Dublin Core Metadata Initiative
DTD	Document Type Definition
FRBR	Functional Requirements for Bibliographic Records
FRBRize	To transform a specific format (for instance MARC) so that it conforms to the FRBR conceptual model
HREF	Hypertext Reference
HTML	HyperText Markup Language
IDREF	ID Reference
IFLA	International Federation of Library Associations and Institutions
Lossless Roundtrip	When one is able to convert back and forth between formats without the loss of any data in a conversion operation.
MARC	MAchine Readable Cataloging Record
MODS	Metadata Object Description Schema
NISO	National Information Standards Organization
OAI-PMH	Open Archives Initiative – Protocol for Metadata Harvesting
PCDATA	Parsed Character Data
SGML	Standard Generalized Markup Language
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations