Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Department of Computer Science

Extensible Metadata Repository for Information Systems

by Pedro Honrado Rio Pereira

Thesis submitted to Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa, in partial fulfillment of the requirements for the degree of **Master in Computer Science**

Supervisor: PhD João Moura Pires

Lisbon

2009

Resumo

Sistemas de informação são, muitas vezes, sistemas com uma componente de integração de informação muito forte. Alguns desses sistemas recorrem a soluções de integração fazendo uso de metainformação (informação que descreve informação). É necessário lidar com essa metainformação e geri-la do mesmo modo que se faz com informação “normal”, para tal a existência de um repositório de metadados que garanta o armazenamento, integridade, validade e facilite os mecanismo de integração do sistema de informação é uma escolha lógica.

Existem vários repositórios disponíveis no mercado, mas nenhum virado para as exigências dos sistemas de informação, genérico o suficiente e com as características de integração necessárias. No projecto SESS, da agência espacial europeia (ESA), foi desenvolvido um repositório de metadados genérico, baseado em tecnologias XML. Esse repositório proporcionava mecanismos de integridade, validade, armazenamento, partilha, publicação, importação, integração de sistemas e de dados, mas obrigava à utilização de regras sintácticas fixas, colocadas dentro dos documentos XML, o que dificultava a integração de documentos de fontes externas.

Nesta tese desenvolveu-se um repositório de metadados, com base em tecnologias XML, que proporciona os mesmos mecanismos de armazenamento, integridade, validade, etc, mas que tem em atenção a capacidade de integrar, de forma fácil, metainformação estrangeira de qualquer tipo (em formato XML) e que é capaz de proporcionar um ambiente onde o reaproveitamento dos tipos de metadados para a construção de novos tipos de metadados é uma constante, sem ter necessidade de modificar os documentos que armazena.

O repositório armazena documentos XML, denominados de Instâncias, que são instâncias de um Conceito, esse Conceito define uma estrutura XML Schema que valida as Instâncias. Para lidar com o reaproveitamento, foram criadas unidades chamadas Fragmentos, que permitem definir uma estrutura XML Schema (que pode ser criada à custa da composição de outros Fragmentos) que pode ser reutilizada por Conceitos para definir a sua própria estrutura. Os elementos do repositório (Instâncias, Conceitos e Fragmentos) têm um identificador próprio baseado em (e compatível com) URIs, denominado MRI (Metadata Repository Identifier). Esses identificadores assim como informações de relacionamento e de gestão são geridas pelo repositório evitando assim a utilização de regras sintácticas fixas, facilitando a integração.

Um conjunto de testes, utilizando documentos do projecto SESS e da *software-house* ITDS, serviram para a validação bem sucedida do repositório em relação aos objectivos da tese, em termos de integração e reaproveitamento.

Abstract

Information Systems are, usually, systems that have a strong integration component and some of those systems rely on integration solutions that are based on metadata (data that describes data). In that situation, there's a need to deal with metadata as if it were "normal" information. For that matter, the existence of a metadata repository that deals with the integrity, storage, validity and eases the processes of information integration in the information system is a wise choice.

There are several metadata repositories available in the market, but none of them is prepared to deal with the needs of information systems or is generic enough to deal with the multitude of situations/domains of information and with the necessary integration features. In the SESS project (an European Space Agency project), a generic metadata repository was developed, based on XML technologies. This repository provided the tools for information integration, validity, storage, share, import, as well as system and data integration, but it required the use of fix syntactic rules that were stored in the content of the XML files. This situation causes severe problems when trying to import documents from external data sources (sources unaware of these syntactic rules).

In this thesis a metadata repository that provided the same mechanisms of storage, integrity, validity, etc, but is specially focused on easy integration of metadata from any type of external source (in XML format) and provides an environment that simplifies the reuse of already existing types of metadata to build new types of metadata, all this without having to modify the documents it stores was developed. The repository stores XML documents (known as *Instances*), which are instances of a *Concept*, that Concept defines a XML structure that validates its Instances. To deal with reuse, a special unit named *Fragment*, which allows defining a XML structure (which can be created by composing other Fragments) that can be reused by Concepts when defining their own structure. Elements of the repository (Instances, Concepts and Fragment) have an identifier based on (and compatible with) URIs, named Metadata Repository Identifier (MRI). Those identifiers, as well as management information (including relations) are managed by the repository, without the need to use fix syntactic rules, easing integration.

A set of tests using documents from the SESS project and from software-house ITDS was used to successfully validate the repository against the thesis objectives of easy integration and promotion of reuse.

Agradecimentos

Quero agradecer a toda a minha família, em especial ao meu pai José, à minha mãe Isabel e ao meu irmão Tiago, pelo amor e respeito dado e ensinado ao longo da minha vida. Obrigado por sempre me terem apoiado e por me terem dado tudo para eu conseguir chegar onde cheguei hoje. Votos de saúde e felicidade para todos.

Obrigado ao orientador, supervisor e colega Professor João Moura Pires. A sua participação neste projecto foi vital, em primeiro lugar por me ter proposto o desafio e em segundo por ter trabalhado nele comigo. Aprendi imenso consigo e espero voltar a ter a oportunidade de trabalhar novamente consigo.

Obrigado a todos os colegas que partilharam o gabinete 244 durante o tempo que estive lá enquanto fazia o mestrado, a vossa contagiante boa disposição foi uma grande ajuda.

Obrigado a todos os colegas de faculdade que de um modo ou de outro me ajudaram a chegar onde cheguei, particularmente os auto-denominados “Culelos” e em particular o colega que me acompanhou durante todo o curso e parte do mestrado, o Pedro Andrez.

Obrigado Teresa, pelo apoio e amor incondicional durante todo este processo, que em muito ajudou a ultrapassar os momentos mais difíceis.

Index

Chapter 1 Introduction **1**

1.1. Motivation **3**

1.2. Context **4**

1.3. Objective **7**

1.4. Thesis Structure **7**

Chapter 2 State of the Art **9**

2.1. Metadata in Organizations **10**

2.1.1. The Use and Management of Metadata 11

2.2. XML Technologies **12**

2.2.1. Definition of XML Vocabularies and Validation 12

2.2.2. XML Processing 13

2.2.3. Querying and Updating XML 14

2.3. Semantic Web **15**

2.3.1. RDF 18

2.3.2. OWL 20

2.3.3. Simple Knowledge Organization System 21

2.3.4. State of the Semantic Web and its applicability to organizations 21

2.4. Metadata Repositories and Tools **24**

2.4.1. Repository In a Box 24

2.4.2. DSpace 25

2.4.3. Protégé 26

2.4.4. Fedora 29

2.4.5. Extensible Metadata Repository for Information Systems 31

2.5. Metadata Management Technologies and Repositories Appreciation **36**

Chapter 3 Architecture Design **39**

3.1. Requirements **41**

3.2. Architecture **42**

3.3. Information Model **43**

3.4. M0 Layer (External Entities) **43**

3.5. M1 Layer (Instances) **44**

3.5.1. Instance Versions 45

3.5.2. Instance Relations 46

3.6. M2 Layer (Concepts) **49**

3.6.1. Fragments 49

3.6.2. Concepts 50

3.7. M3 Layer – Meta-meta-model **54**

3.7.1. Evaluation 54

Chapter 4 Functional Design **55**

4.1. Metadata Repository Identifiers (MRI) ... 56
4.1.1. Instance Identification ... 57
4.2. Concept & Fragment Definition ... 57
4.2.1. Fragment Definition Language ... 58
4.2.2. Concept Definition Language ... 65
4.3. Metadata Validation and Integrity ... 76
4.3.1. Fragment Validation & Integrity ... 76
4.3.2. Concept Validation & Integrity ... 77
4.3.3. Instance Validation & Integrity ... 78
4.3.4. Instance Relations, Creation & Validation ... 78
4.4. System Concepts and Instances ... 80
4.5. Metadata Querying and Transforming ... 80
4.5.1. XQuery ... 81
4.5.2. Transforms ... 82
4.5.3. Generic Transforms ... 83
Chapter 5 Implementation ... 87
5.1. Technologies ... 88
5.2. Architecture Design Implementation ... 88
5.2.1. High Level Architecture ... 89
5.2.2. Low Level Architecture ... 89
5.3. Choice for the Underlying Database of the Storage Model ... 90
5.3.1. eXist XML Database ... 91
5.3.2. Sedna XML Database ... 91
5.3.3. Berkeley DB XML ... 92
5.3.4. Query Benchmarking ... 94
5.3.5. Storage Benchmarking ... 98
5.3.6. Final Evaluation ... 100
5.4. Storage Model ... 101
5.4.1. Concept Storage ... 101
5.4.2. Fragment Storage Model ... 102
5.4.3. Instance Storage Model ... 103
5.4.4. Additional System Management Information ... 104
5.4.5. Complete Storage Model ... 106
5.4.6. Access Permissions ... 106
5.5. Information Model ... 107
5.5.1. M2 Layer – Meta-model ... 107
5.5.2. M1 Layer - Model ... 109
5.6. Querying and Transforming ... 111
5.6.1. Repository Built-in XQuery Functions ... 115
5.7. Implementation Status ... 115
Chapter 6 Validation ... 119
6.1. Space Environment Support System - SESS ... 120
6.1.1. Standalone Test ... 123

6.1.2. Reusability Test 125
6.2. ITDS - Xeo 129
6.2.1. Evaluation 131
Chapter 7 Conclusions and Future Work 133
7.1. Conclusions 134
7.2. Future Work 136
References 137

Figure Index

Figure 1.1 SESS project architecture, taken from [1] 5
Figure 1.2 MOF model, taken from [1] 6
Figure 2.1 Semantic Web Layer Stack 16
Figure 2.2 RDF Example 19
Figure 2.3 RDF with multiple examples 19
Figure 2.4 RDF Graph 20
Figure 2.5 Semantic Web Layer Stack with Datalog Rules 23
Figure 2.6 Protégé-Frames editor 27
Figure 2.7 Protégé-Frames Query Interface 28
Figure 2.8 Protégé-OWL - Ontolgy Visualization Plugin 29
Figure 2.9 Fedora's online catalog 30
Figure 2.10 Fedora's Object Model 30
Figure 2.11 MOF model in the context of the MDR, taken from [1] 31
Figure 2.12 Instance processing and transforming capabilities, taken from [1] 33
Figure 2.13 Relationship syntax in a SESS Instance 34
Figure 2.15 Instance with outlined rules required by the MDR 35
Figure 2.14 Concept with outlined rules required by the MDR 35
Figure 3.1 Architecture of the Metadata Repository 42
Figure 3.2 MOF layered architecture 43
Figure 3.3 M0 Layer - External Entities 44
Figure 3.4 M1 Layer: Instances 44
Figure 3.5 Temporal evolution of Instance versions 45
Figure 3.6 Instance Version Control Fields and Modification Notation 45
Figure 3.7 Information Model with Instance Versions 46
Figure 3.8 Relation between two Instances 47
Figure 3.9 Instance Relation with a Locking Version 47
Figure 3.10 Instance Relation with "Last Version" 48
Figure 3.11 M1 Layer, Instances with Relations and Versions 49
Figure 3.12 Fragment versions 50
Figure 3.13 Automatic Relation based on identifiers 52
Figure 3.14 Content Relations 52
Figure 3.15 M2 Layer, with Concept and Fragment Versions 53
Figure 3.16 Full Information Model, with M3 Layer 54
Figure 4.1 Fragment main structure 58
Figure 4.2 Fragment definition 59
Figure 4.3 Embedded XML Schema in a Fragment definition 59
Figure 4.4 GlobalComposition element converted to XML Schema 60
Figure 4.5 Global Composition with Attributes 60
Figure 4.6 Structure of a Sequence element 61
Figure 4.7 Structure of the Schema element 62
Figure 4.8 Correspondence of a Fragment definition and XML Schema 63
Figure 4.9 Use of local embedded XML schema in a Fragment definition 63
Figure 4.10 Use of Constant Annotation in Fragment definition 64
Figure 4.11 Structure of a XSL element 64
Figure 4.12 XSL code used in element XSL 65
Figure 4.13 Generic Structure of a Concept 66
Figure 4.14 XML syntax for Instance Identification 66
Figure 4.15 Instance Identification element with Namespace binding 67

Figure 4.16 Concept structure definition referencing a Fragment ... 67
Figure 4.17 Schematron list syntax ... 68
Figure 4.18 Structure of the Schematron element ... 68
Figure 4.19 Schematron element with embedded Schematron code ... 69
Figure 4.20 Schematron element with reference ... 69
Figure 4.21 XSLList element syntax and usage ... 70
Figure 4.22 Syntax of the Relations element ... 70
Figure 4.23 List of valid target Concepts for a relation ... 71
Figure 4.24 Syntax of the Cardinality element ... 71
Figure 4.25 Automatic Relation based on content syntax ... 72
Figure 4.26 Definition of targets for the automatic relation based on content ... 72
Figure 4.27 Usage of the LocalInstanceXPath element ... 73
Figure 4.28 RemoteInstanceXPath element syntax ... 73
Figure 4.29 Syntax of the behavior of a relation ... 73
Figure 4.30 Syntax for the behavior (update) of a relation ... 74
Figure 4.31 Syntax for the creation of automatic annotations of relations ... 74
Figure 4.32 Structure of the AutoRelMRI element ... 75
Figure 4.33 Structure of the AutoRelContent element ... 75
Figure 4.34 Syntax for constant annotations to the Concept ... 76
Figure 4.35 Behaviors of a relation in case of an update/removal of a target Instance ... 79
Figure 4.36 Metadata querying output capabilities ... 80
Figure 4.37 Metadata repository transforming and output capabilities ... 81
Figure 4.38 XQuery functions to access relations in instances ... 81
Figure 4.39 Example XSLT association to a Fragment ... 82
Figure 4.40 XSLT with reuse of Fragment templates ... 83
Figure 4.41 Regular transforming process in the repository ... 84
Figure 4.42 Generic Transform processing in the repository (example for a HTML Generic Transform) ... 84
Figure 5.1 Metadata Repository's High-level architecture ... 88
Figure 5.2 Metadata Repository's low-level architecture ... 89
Figure 5.3 XMark XML structure [2] ... 93
Figure 5.4 Results for Query 2 of the XMark Benchmark ... 96
Figure 5.5 Results of Query 8 of the XMark Benchmark ... 97
Figure 5.6 Concept's Storage Model ... 101
Figure 5.7 Fragments Storage Model ... 102
Figure 5.8 Instances Storage Model ... 103
Figure 5.9 Additional System Management Information ... 104
Figure 5.10 Internal Structure of the IdentifierList.xml ... 105
Figure 5.11 FragmentList.xml structure ... 105
Figure 5.12 ConceptList.xml structure ... 106
Figure 5.13 Metadata Repository Complete Storage Model ... 106
Figure 5.14 Storage Model's access permissions ... 107
Figure 5.15 Sample Concept Management File ... 109
Figure 5.16 Sample Instance Management File ... 111
Figure 5.17 Club Concept XML Schema structure ... 111
Figure 5.18 Instance Manchester United of Concept Club ... 112
Figure 5.19 Instance Arsenal of Concept Club ... 112
Figure 5.20 XQuery example ... 113
Figure 5.21 Instance of the Query System Concept ... 113
Figure 5.22 Result of XQuery execution ... 114
Figure 5.23 XSLT applied to the result of a query ... 114
Figure 6.1 SESS domain concepts relationships, taken from [1] ... 122
Figure 6.2 SESS Concepts used as an example in import ... 122
Figure 6.3 Definition of Concept Groundstation from SESS ... 124

Figure 6.4 Graph of captured relations from Instances of SESS ... 125
Figure 6.5 Relations between Concepts and included Schemas ... 126
Figure 6.6 Fragment definition of the DIM Fragment ... 127
Figure 6.7 Groundstation Concept Definition ... 128
Figure 6.8 Result of converting a Concept definition in XML Schema ... 129
Figure 6.9 xeoModel Concept definition ... 130
Figure 6.10 xeoModel relation definition sample ... 131

Table Index

Table 5.1 Table of document size and number to benchmark 99
Table 5.2 Sedna XML database storage results 99
Table 5.3 Berkeley DBXML database storage results 99
Table 5.4 eXist XML Database storage results 100
Table 5.5 Properties of a relation in an Instance management file 110
Table 5.6 List of XQuery functions provided by the repository 115
Table 5.7 Implementation status of the features of the Repository 116
Table 6.1 List of Concepts from SESS project 120

Chapter 1
Introduction

This chapter presents the motivation, context and objective of this thesis, followed by the thesis structure

1.1 Motivation.. 3
1.2 Context... 4
1.3 Objective.. 7
1.4 Thesis Structure... 7

This chapter presents the motivations for the elaboration of this thesis, the main one being the use of metadata repositories to support the life cycle of Information Systems (IS). The chapter also presents the goal of this thesis and the context in which it was created.

1.1. Motivation

Metadata, usually defined as "data about data" or "information about information", can be described, in the context of IS, as "all physical data and knowledge-containing information about the business and technical processes, and data, used by a corporation" [1]. Metadata, as in any other kind of information, can be grouped in categories. The main categories found in common literature, are "technical metadata" and "domain metadata" [1] [3] [4].

Domain metadata is specified by domain experts and software analysts, while technical metadata is specified by technicians and developers. Both kinds of metadata can further be categorized as "documentation metadata" and "operational metadata". The purpose of "documentation metadata" is, as the name implies, to document functionality (ex. data dictionaries, system diagrams or corporate policies). "Operational metadata" models the behavior of the system itself (or part of it), this means that the behavior of the system is directly tied to the content of the metadata.

The tool that eases management, provides the means to validate, maintains integrity, stores and visualizes metadata within an organization, is designated as **Metadata Repository (MDR)**.

In the context of Information Systems, metadata can be seen as any information required to develop and maintain a system [5]. In this perspective, metadata is the fundamental component of an Information System, as it allows the supervision of all development phases, since initial requirement analysis, through the implementation and maintenance phases.

The use of metadata in an Information System brings an additional startup cost to the development, due to the necessity of agreement among domains experts, analysts, technicians and development teams on what the domain metadata and technical metadata is, how it's obtained, produced and processed. Later on, this cost is rewarded, since the existence of such metadata represents a source of reliable information that describes the behavior of the system and its capabilities, in a precise and explicit way. This promotes a better documentation where choices are thoroughly explained, providing an excellent source of knowledge for users and developers, easing, for example, the development of new features. Storing metadata in a computable format enables an Information System to use that metadata for its normal operation, knowing that the quality and integrity of the information is guaranteed [1]. Gains are most notable in the case of a **metadata-driven** system, that is based on a declarative architecture and its behavior is specified with metadata. One example of a metadata-driven system is one using a SOA architecture [6], where an application is created by specifying a workflow between a composition of already existing services. This

specification is stored as metadata and can be done using several standard Web Service (WS) composition formats, such as BPEL4WS [7], BPML [8], among others (BPSS, DAML-S, WSCI).

The explicit use of metadata in Information Systems has several advantages, as pointed earlier, but its adoption has been slow and limited to some specific areas. There are specialized solutions of information management (which possess internal metadata repositories), although they are part of systems with a considerable dimension, which may have a very high cost for a small or medium enterprise (SME) and, even still, they don't allow for another application to take advantage of that metadata (because it's in an internal repository). The ideal solution would be a Metadata Repository that is capable of validating, storing, transforming, visualizing, importing and exporting metadata, as well as, above all, be extensible and versatile enough to centralize all metadata of a given system or systems [1].

As such, to summarize, the motivation for this thesis was the opportunity to work in an essential field (metadata repository) of information systems that base their integration strategy on a metadata solution.

1.2. Context

The Space Environment Support System (SESS) [9] was a project developed for the European Space Agency (ESA) with the objective of providing the tools and means to analyze and monitor Space Weather (S/W) occurrences. S/W is the combination of conditions in solar wind, magnetosphere, ionosphere and thermosphere that can influence the performance, integrity and reliability of space-borne and ground-based technological systems [1]. Degradation of sensors, or unpredicted changes in the on-board memories are some of the consequences of S/W phenomenon.

The scientific domain of the problem is complex and, to be able to monitor it properly, it was necessary to integrate data from several heterogeneous sources for analysis. The data consists of S/W parameters, Spacecraft (S/C) parameters, S/W events, S/C events, etc. These are domain data, not including all the technical data required for their monitoring. Thus, a considerable amount of metadata existed that needed to be stored, managed, validated, visualized and created.

For the SESS project, an evolution of the SEIS project [10], a metadata repository was designed that attended the previously referred requirements, as well as other requirements necessary to metadata management [3, 11]. The repository was able to deal with such different kinds of information as the ones described in the previous paragraph.

XML [12] technologies were a natural choice for storing metadata in the repository, given their flexibility in the creation of controlled vocabularies, validation of those vocabularies, transformation, visualization and update capabilities, as well as a strong maturity and wide acceptance in the open

source community and the commercial industry. The vast number of tools and libraries that process XML reflects that acceptance.

The architecture of the SESS project is depicted in Figure 1.1. There's a Data Processing Module (DPM), which is responsible for downloading the necessary information available in external Data Providers (through HTTP/FTP) and, after the download, process the data passing it to the Data Integration Module (DIM). The DIM stores, in a series of databases, values of S/W and S/C that will feed the Client Tools Module (CTM), which is a set of graphical tools for users to monitor the necessary parameters, configure alarms for special situations, etc.

All the tools read and store metadata in the repository and their behavior is dependent on the content of metadata stored in the repository. This situation makes the repository the "glue" that connects all parts of the system.

In this project, the metadata repository also stores the documentation, maintains the relations between technical metadata and domain metadata, guarantees their integrity and validates the metadata it stores. Metadata transforming and querying is also used by the components to extract information from the repository.

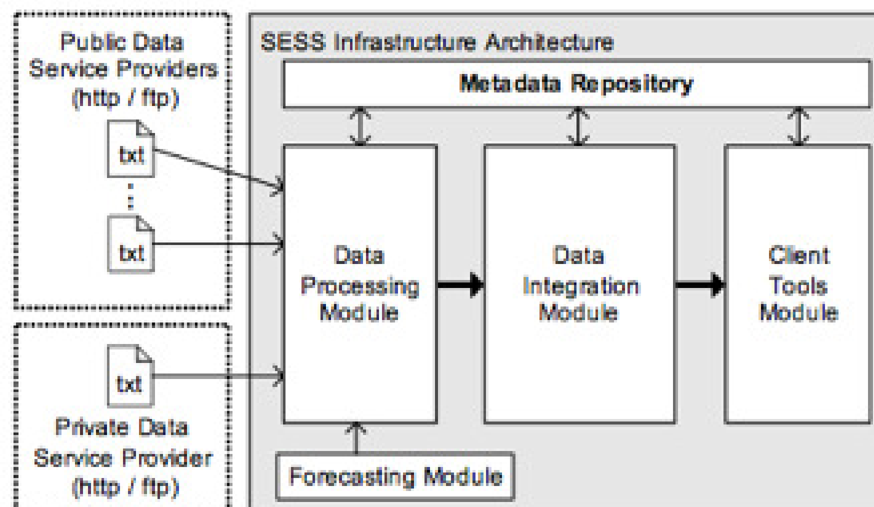

Figure 1.1 SESS project architecture, taken from [1]

The repository is built on top of XML technologies (with a Java [13] engine) and using the MOF [14] information model, an OMG [15] standard. MOF, Figure 1.2, is a layer-based model where each layer describes the layer immediately bellow. At the base of the hierarchy, is the M0 layer that represents "real" objects of a given reality, as such, the M1 layer (Model) describes the objects in the M0 layer. The M2 layer (Meta-model) describes the models of the M1 layer, which means it's the definition of a language, notation or properties of a model. The M3 layer (Meta-meta-model) describes Meta-models, i.e., a set of rules and common structures to all

meta-models. In the context of the SESS metadata repository, the M1 layer is designated as the "Instances" layer (represents metadata), which is described by "Concepts" (that define a controlled vocabulary and represent the M2 layer) that, in turn, are subject to "Rules" (the M3 layer).

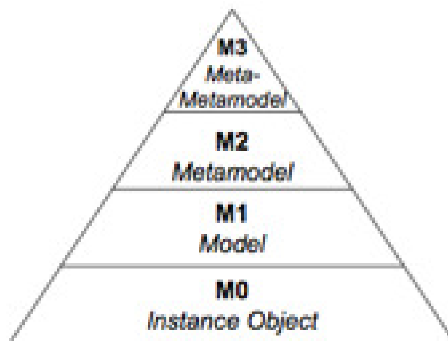

Figure 1.2 MOF model, taken from [1]

An Instance is a XML file that is compliant with the vocabulary defined by a Concept. That vocabulary is specified using XML Schema and, optionally, Schematron [16]. Concepts are subject to restrictions imposed by a set of rules. The rules are composed of XML Schema, Schematron and Java (to ensure some restrictions not possible with the two previous technologies) and they exist to assure that a certain level of "order" is present, both in Instances and Concepts. Rules include, for example, that all Concepts must include a "base" XML Schema (supplied by the repository) or that Concepts must identify their root element with a specific attribute. Java rules guarantee referential integrity in the relations between Instances and assure Concept and Instance uniqueness.

Although successful and well accepted, the MDR has some issues that create difficulties in its adoption as a solution for external data integration in other environments.

The use of fixed syntactic rules, such as the inclusion of a base XML Schema, that imposes certain restrictions to the structure of each Concept, the use of internally generated identifiers which are only known after insertion of an Instance and management information about relations being stored inside the content of Instances, as well as those same relations being dependant on the previous identifiers, implies that every external XML document must be modified to be accepted in the repository. On another hand, the original motivation for the use of syntactic rules was to enable reutilization (of code that processes Instances). However, and because new Concepts in practice mean new XML Schemas, the reutilization rate proved to be very low, because people developing new Meta-models (new XML Schemas) rarely checked on the already existing ones.

In a system that stores a considerable amount of metadata (possibly scattered through several machines or databases), if the cost of importing that metadata to a repository is high and requires non-trivial and non-automatic conversions, it's very likely an organization will not see the benefits in using such a repository (due to this startup cost).

Non-promotion of Concept reuse is a problem that affects an organization after the adoption of the metadata repository. When new definitions need to be created, if there's no mechanism to reuse already existing definitions it will require the creation of the new ones from scratch and code to process those definitions will also have to be created from scratch, making the use of a metadata repository less attractive.

These two aspects: difficult external metadata integration and the non-promotion of Concept reuse (or parts of them), are serious obstacles in the adoption of the MDR as a solution of integration for the design and development of Information Systems.

To address these issues a profound and nontrivial revision of the philosophy of the repository is required. The work on this thesis pretends to present a concrete solution to solve these problems.

1.3. Objective

The objective of this thesis is to propose and implement a Metadata Repository that is able to easily integrate metadata from heterogeneous sources, without having to modify the documents in order to integrate them, and supply a set of features that promote the reuse of previously created metadata fragments.

1.4. Thesis Structure

The content of this thesis is structured in seven chapters and one annex. Follows the listing and description of each chapter:

Chapter one (Introduction) presents the motivation, context and objectives of this thesis, followed by the thesis structure. In chapter two (State of the Art) the state of the art in metadata management, XML technologies, Semantic Web and metadata repositories and tools is presented. Chapter three (Architecture design) describes the architecture of the metadata repository as well as the information model. In Chapter four (Functional design) the design of the features of the repository is presented. Chapter five (Implementation) presents implementation details of the architecture, information model and features of the repository and in chapter six (Validation) a set of validations tests, using real-world files from existing applications are presented. The thesis is concluded in chapter seven (Conclusion and Future Work) where future work is presented and conclusions are drawn on the design and implementation of the metadata repository. The annex chapter is only present in the digital copy of this thesis.

Chapter 2

State of the Art

This chapter presents the state of the art in metadata management, including technologies, tools and repositories to deal with metadata.

2.1 Metadata in Organizations………………………………………………………………………………… 10
2.2 XML Technologies……………………………………………………………………………………… 12
2.3 Semantic Web…………………………………………………………………………………………… 15
2.4 Metadata Repositories…………………………………………………………………………………… 24
2.5 Metadata Management Technologies and Repositories Appreciation……………………………… 36

In this chapter the state of metadata management in organizations is presented, featuring technologies and tools to deal with it. Among those technologies are the XML technologies and the Semantic Web, which will be presented and evaluated. Some metadata repositories and tools will also be presented and evaluated. In this chapter and this thesis, the focus is on metadata related to Information Systems and organizations and not on metadata in general.

2.1. Metadata in Organizations

For several years, information technologies have been focused on processes and data with the aim of building systems and manage their operation [17]. Information and its processing is a vital part of these systems, which, in turn, are the core of modern organizations. The amount of information produced by organizations is increasing and this information is increasingly important (this includes digital and non-digital information). The information produced includes, for example, documents with operative rules of the company, specifications of products, security policies, sales reports, etc. [18]

Some organizations and enterprises have a Knowledge Management (KM) department. KM focuses on the acquisition, maintenance and access to information within an organization. It is a core activity within them, because organizations themselves see their internal information as a resource that can be used to improve productivity, create added value and enhance their competitiveness. It is, therefore, vital to know the answer to five questions within an organization [4].

- What information do we have?
- What is the significance of that information?
- Where is it?
- How did it get there?
- How can I access it?

The second question is particularly important, because information without its associated context is worth little. To this “context” one could call “meta-information” (or metadata) and, therefore, it can be concluded that without metadata, data has no value [17]. A good example is in [17]: If someone in a crowd shouts "42", most people will simply stare. However, people who are fans of writer Douglas Adams and know his work in "The Hitch Hiker's Guide to the Galaxy" certainly know that "42" is the answer for the "*meaning of life, the universe and everything else*". This example shows that the context of any type of data is extremely important for any system and to any attempt of analysis of their content, although every one of the other four questions is equally important in knowledge management.

Metadata helps people to find the information they need, to know what it means and to decide whether that information is useful or not. It allows the reuse of knowledge rather than "reinventing the wheel" each time some information (or procedure) is required.

Metadata eases the search for information, promotes the reuse of already existing knowledge, facilitates access to correct data [17], aids in the integration of systems, increases the confidence of users in the information they are using and is a great tool when there's staff replacement in the organization, which requires to make them aware of how the organization's internal procedures and processes work [11]. Several standards of metadata can be used to enhance interoperability between internal systems, or with external systems [19].

There are several different types of metadata in Information Systems, but they can be grouped in two categories. Business Metadata and Technical Metadata [17]. Technical Metadata helps development teams, programmers and technical users in the tasks of maintaining and developing new features. Business metadata helps nontechnical users (administrators, consultants, etc.) to make better decisions for the organization based on available information or to find the information they need for their tasks.

A classic example of technical metadata is a data dictionary that helps development teams to know, in detail, the structure of a database. In the case of business metadata, one can think of the example of a library catalog in which summaries and descriptions of books are stored, for users to consult in order to find a book that covers a topic in which they may be interested.

2.1.1. The Use and Management of Metadata

The use of metadata in an organization brings added value, but assumes that it's available in an easy, safe and valid way. The problem is that, generally, there's no centralized and controlled source of metadata in organizations [18] and the sources of metadata are spread over various parts of the organization, leading to the repetition of definitions and mismatched versions. Another issue is that, frequently, there is nobody responsible for the production of metadata [18] and much of the metadata remains in the minds its potential producers [17]. The problematic of capturing metadata in an automatic way is also a barrier to its adoption[18].

The solutions in current literature [1, 3, 10, 16, 18], propose the creation of a centralized repository of metadata (that can also be in a distributed environment) that allows the storage of different types of metadata, is designed to ensure the integrity and validity of metadata, enables metadata relationships, eases the search, import and export of metadata.

The use of metadata of various kinds also raises the issue of how it's encoded to be stored in the repository and what is the capacity of the metadata repository to accept that metadata. The next section presents two families of technologies that can be used for the representation of metadata and, after that, a small set of metadata repositories and their features will be analyzed.

2.2. XML Technologies

XML (eXtensible Markup Language) is a meta-markup language that defines a syntax with which other markup languages can be created [20].

XML derives from SGML [21] and was defined by the World Wide Web Consortium (W3C) [22], an international consortium that sets standards for the Internet. XML enables the creation of languages in which the vocabulary (the set of tags) is defined by the designer and not fixed like, for example, in the HTML markup language [23]. XML format is easily readable by people, although restrictive rules regarding the opening and closing of tags and the definition of attributes exist. These features make XML an appropriate technology for the representation of all kinds of information, as along with being a simple textual format, they allow the representation of domain specific languages, make data self-descriptive and allow easy sharing of information between software from different manufacturers or technologies [20].

XML allows the coexistence of vocabularies with tags that have the same name, but are defined by different organizations, using namespaces [24]. Namespaces allow the use of a prefix in elements of a vocabulary, which maps to a URI [25]. This technique allows solving any ambiguities in the resolution and validation of vocabularies.

The easy creation of a vocabulary in a format readable by people and the ability to model any existing domain, made XML a rather well adopted technology for the representation of metadata, as the number of different domain specific (or generic) metadata standards that are XML-based is a proof. Examples of these standards are the Dublin Core Metadata Initiative (DCMI or DC) [26] for any kind of metadata or METS [27] for metadata about objects in a digital library.

In the following sections, some of the XML technologies and how those technologies are suited for representing, processing, validating and querying metadata, will be discussed.

2.2.1. Definition of XML Vocabularies and Validation

XML Schema [28] is a XML-based language to define vocabularies, this means it uses XML syntax to define the language and, therefore, is itself a XML document. XML Schema defines, accurately, the structure of a document, which elements are in it, the attributes of those elements, the order in which they are, they're cardinality, inclusions of external schemas, among others. Elements can be designed in a modular way and be reused throughout the schema. XML Schema supports namespaces, which avoid ambiguities between vocabularies. It's a technology that can be used for validation, documentation, query, data binding and guided editing [29]. By defining a language, XML Schema implicitly documents that language and has support for explicit documentation mechanisms, such as notes and comments; it also has mechanisms to provide documentation for applications that might process the document (through the "appinfo" tag). XML Schema is a technology that is well

spread in the market, as there are multiple tools that can process it, both commercial [30] [31] and open source. There are also libraries such as libxml [32] (C) or Xerces [33](Java) and there is a quite extensive list of tools with support for XML Schema[34].

Relax NG [35] is another technology for the definition of XML vocabularies; it was defined by the Organization for the Advancement of Structured Information Standards (OASIS) and has several similarities with XML Schema, however, it has two ways of expressing the vocabulary. One is through a XML syntax and the other through a simple syntax that is convertible to XML, this possibility is similar to what was possible with DTD [36], one of the first technologies to define vocabularies. Relax NG also has extensive support in tools and libraries [37].

Schematron is a XML-based language to make assertions over the presence or absence of patterns in XML documents [16]. In Schematron, instead of defining the structure of a document, the name of the elements, their attributes or data-type (like in XML Schema), one can define properties that elements should have (or not have) as well as values they must match. Schematron is able to verify several conditions that are not possible with XML Schema, such as, for example, check if any given element has a certain attribute with a certain value. Schematron's nature makes it better suited for validation than for documentation (it would be rather hard to infer the structure of a document from a set of Schematron rules), so the most frequent use is as a companion to other validation technology such as XML Schema or Relax NG. The set of tools that natively supports Schematron is small, but there's a XSLT implementation of Schematron [38] that enables any recent XML processing library to use it.

XML Schema is the ideal choice for Concept definition and Instance validation due to the fact that it's a W3C standard that can be used to create controlled vocabularies, document (implicitly or explicitly) those vocabularies and is very well supported in both commercial and open source tools (also the XML community is of a considerable dimension). Schematron's ability to verify conditions that are not possible with XML Schema and by being usable through the XSLT implementation, make it the best choice for extra-validations on Instances.

2.2.2. XML Processing

XML documents can be seen as a tree of elements beginning in a root node, with several child nodes. To navigate in such a structure the XML Path Language (XPath) [39] was defined by the W3C. XPath is a language used to describe and access parts of a XML document [40] and supplies a set of simple functions to navigate and return the meaningful document nodes (which can be the entire document, or an attribute of a node). It's a standard technology, that can be found in the majority of tools/libraries and it's used by several other technologies, by its ability to locate parts of a document, such as XML Schema, Schematron, XSLT, XQuery or XUpdate.

eXtensible Style-sheet Language Transformation (XSLT) [41], is a declarative XML-based language, for transforming XML documents into other XML documents [42]. It's a language based on patterns (or rules) that identify a part of a document (using XPath) and apply a transformation to that part, using a set of predefined functions to produce the desired result. XSLT can be used to transform XML in XML, but also to produce documents such as HTML, Rich Text Format (RTF), plain text, Javascript, SQL and XSL-Formating Object (XSL-FO) [42].

In the context of metadata, XSLT supplies the necessary mechanism to visualize metadata under various forms (HTML, RTF or other) as well as convert the metadata into different formats. The ability to transform metadata to other formats means that it's possible to convert metadata from one standard to another standard or to be compatible with a given application/repository. Transformations with the purpose of visualizing, enable the explicit creation of documentation, which makes the life of any one trying to understand that information, a lot easier.

2.2.3. Querying and Updating XML

XQuery [43] is a query language designed by the W3C to deal with the issue of selecting from a XML document, or a collection of XML documents stored in a file system or database (relational, or native XML), elements of interest, reorganize them, eventually transform them and return the results in a structure of the interest of the user issuing the query [44].

XQuery is a typed, functional and declarative language that shares the same data model and type system with XML Schema, as do other technologies of the XML family, such as XPath 2.0 and XSLT 2.0. It was designed to work with non-typed XML documents (without an associated schema), typed with a XML Schema or a combination of both [45]. XQuery is a modular language that supports functions and libraries. There are two kinds of functions, the built-in functions and the user defined functions; built-in functions and operators supply the means to deal with the several data types available. Presently XQuery does not provide support for data types found in other schema languages such as DTD, Relax NG or Schematron [44]. The XQuery language is well spread, being implemented in several native XML databases, as well as relational databases [44].

XUpdate [46] is an update language for XML files, developed by the XML:DB Group [47] to update instances (or collections) of XML document stored in native XML databases. It's a declarative language that enables to creation, change or removal of XML fragments from a document, using XPath and specific constructions. Although initial adoption, the definition never really matured and the last known draft is from the year 2000.

XQuery Update Facility (XQUF) is an extension to XQuery that provides expressions that can be used to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model [48]. XQUF

is, as of August 2008, a W3C candidate recommendation and provides the following set of operations over a XML instance:

- Insertion of a node.
- Deletion of a node.
- Modification of a node by changing some of its properties while preserving its node identity.
- Creation of a modified copy of a node with a new node identity.

XQUF was created to address the problem of updating the content of XML documents in a simple way. Despite being a candidate recommendation, there are already some implementations that support it, such as XQilla [49] , a XQuery processor included in the Oracle Berkley DBXML native XML database, and the MonetDB [50] database system.

In the context of Metadata, XQuery provides the means to query and transform XML documents, given its powerful mechanisms and the fact it's widely supported in all major databases that store XML. On the other hand, XQuery Update Facility provides the mechanisms to update the content of documents.

2.3. Semantic Web

The Semantic Web [51] is a vision of Tim Berners-Lee, regarding the current World Wide Web, that can be defined as "*a network of intelligent data that is machine processable*". Intelligent data can be described as data that is independent of an application, can be composed, is classified and is part of a bigger information ecosystem (an ontology) [52]. The concept of Semantic Web requires that the web is filled with metadata that catalogs, relates and identifies the documents present in it and, also, the existence of applications that can understand that information and process it. The W3C established an activity (composed by several groups) dedicated to the implementation of the vision of the Semantic Web.

Another way of defining the Semantic Web is "*a framework to create, maintain, publish and search semantically rich metadata about web resources; annotating web resources with precise information and meaning about conceptual aspects of its content that provides the bases to resolve the existing limitations with current search engines*" [53].

The structure of the Semantic Web is defined as a "Layered Cake". The decision is based on the fact that it's simpler to reach consensus through small steps than trying to achieve all at once. The process of building one layer of the Semantic Web on top of another, should follow two basic principles [54]:

- "*Downward compatibility. Agents fully aware of a layer should also be able to interpret and use information written at lower levels. For example, agents aware of the semantics of OWL can take full advantage of information written in RDF and RDF Schema*".

- "*Upward partial understanding. On the other hand, agents fully aware of a layer should take at least partial advantage of information at higher levels. For example, an agent aware only of the RDF and RDF Schema semantics can interpret knowledge written in OWL partly, by disregarding those elements that go beyond RDF and RDF Schema*".

In figure 2.1 the layer stack of the Semantic Web is depicted; bellow a description according to [54].

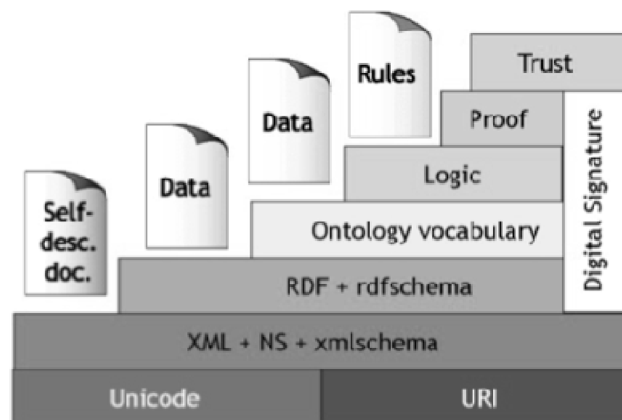

Figure 2.1 Semantic Web Layer Stack

At the bottom of the "cake" is XML that lets one write structured documents using a controlled vocabulary and is particularly suitable to send documents through the web.

The second layer is the Resource Description Framework (RDF), which is the basic data model to write statements about web objects (resources) that are identified by a URI. RDF does not depend on XML, but has a XML serialization, being that the reason it's placed on top of the XML layer.

The third layer, RDF Schema (RDFS), which is based on RDF, supplies the primitives to organize objects in hierarchies, with the major ones being relations of "class", "property", "subclass", "sub-property", domain restrictions and range of values. RDFS can be seen as a simple language to write ontologies.

An ontology is "the set of terms used to describe an area of knowledge" [55]. Since there's the need to express more complex relations between resources, a more expressive language is required and, as such, on top of RDFS there are languages to write complex ontologies.

In the "Logic" layer, lies the ability to extend an ontology and, also, where application-specific declarative knowledge can be declared.

The "Proof" layer is where the deductive process occurs, as well as proof validation, including all the knowledge provided by lower layers.

Finally, the “Trust” layer is used to evaluate if the knowledge deducted in the “Proof” layer can be trusted or not. Applications can have a “Trust” certification and use those certificates to prove to other applications that they’re a reliable source of information.

The Semantic Web vision of transforming the current global web in a web full of machine-processable information, has several applications, such as, intelligent agents that execute actions on behalf of users (for example search for a doctor’s appointment near home for the best price), Business-2-Consumer & Business-2-Business Electronic Commerce, or, in the area of Knowledge Management (KM) [54].

KM will be improved by the Semantic Web in the following fields [54]:

- Knowledge will be organized by areas, according to its meaning.
- Automatic tools would deal with information maintenance, checking for inconsistencies and extracting new knowledge.
- Query based searches will replace the current keyword based ones.
- Support for query answering over several documents.
- Ability to define who may see certain parts of information (event parts of a document) will be possible.

To make the Semantic Web possible, a set of technologies and practices are required. First, information must be annotated with metadata that enables it to be processed. On top of metadata, ontologies that define a vocabulary and its semantic (relations between words and their meaning). The logic layer enables that, starting from ontologies and metadata, new knowledge can be extracted and that that knowledge is valid. Above all these, there are still agents that can receive a set of requests (and knowledge) and will try to satisfy those requests with the knowledge they possess and the one they find throughout their execution [54].

Consider the contribution that the Semantic Web can bring by the value of the following query: “I want all articles about Intel’s competitors” and, given ontologies that define the notion of “competitor” and give semantic aid (that guarantees, for example, that “Palm” and “Palm, Inc.” are considered the same company) and the existence of metadata that relates articles with the companies they refer. The results would be very meaningful and, thus, there’s great added value in this kind of searches [56].

To give support to that vision, several specifications were created, among them, the Resource Description Framework (RDF) and the Web Ontology Language (OWL). Those specifications can be seen as means to represent and deal with metadata and, as such, will be presented in the next section.

2.3.1. RDF

The Resource Description Framework [57] (RDF) is a W3C standard to describe resources through statements. In RDF, a statement (or triple) is composed of three parts: A **resource**, a **property** and a **property value**. In the case where the statement is referred to as a triple, the component names are, respectively, subject, predicate and object.

A **resource** identifies an object through a URI (any object is identified with a URI).

A **property** describes an attribute of a resource identified by a URI.

A **property value**, is the value a given property has. This value can be another resource, if a URI is used.

As an example, to describe the title of the movie based on the book by Dan Brown, "The Da Vinci Code", identifying the movie with the URI http://www.sonypictures.com/…/index.html and the title property by the URI of the Dublin Core element *title*, http://purl.org/dc/elements/1.1/title, the following statement can be defined.

Resource (Subject): http://www.sonypictures.com/homevideo/thedavincicode/index.html

Property (Predicate): http://purl.org/dc/elements/1.1/title

Property Value (Object): The Da Vinci Code

A RDF document contains a set of RDF statements. To define RDF documents, several syntaxes are available [58], one of them being the RDF/XML syntax [59]. This syntax defines the structure of a RDF document as the following:

A root element "rdf" that must include the namespace declaration with the "rdf" prefix, mapped to the address "http://w3.org/TR/1999/PR-rdf-syntax-19990105#". Children of this node can only be elements of type "**Description**".

As an example, the previous statement is depicted in Figure 2.2, in a RDF/XML serialization.

In RDF, each predicate **must** be associated to a namespace [60], to be possible to resolve ambiguities between predicates with the same name, but defined by different entities and to know where to search for the meaning of any given predicate.

```
<rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <rdf:Description rdf:about="http://www.sonypictures.com/homevideo/thedavincicode/index.html">
        <dc:title>Da Vinci Code</dc:title>
        <dc:creator rdf:resource="http://www.imdb.com/name/nm0000165/"></dc:creator>
        <dc:contributor rdf:resource="http://www.imdb.com/name/nm0326040/"></dc:contributor>
    </rdf:Description>

</rdf:RDF>
```

Figure 2.2 RDF Example

The next example (Figure 2.3), adds more information to the movie, including the movie director with the predicate "**DC Creator**" and a film contributor with the predicate "**DC Contributor**" and defining them as references to other elements (using the attribute **rdf:resource**) inside the same document.

```
<rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:id="http://www.example.com/identifier">

    <rdf:Description rdf:about="http://www.sonypictures.com/homevideo/thedavincicode/index.html">
        <dc:title>Da Vinci Code</dc:title>
        <dc:creator rdf:resource="http://www.imdb.com/name/nm0000165/"></dc:creator>
        <dc:contributor rdf:resource="http://www.imdb.com/name/nm0326040/"></dc:contributor>
    </rdf:Description>

    <rdf:Description rdf:about="http://www.imdb.com/name/nm0000165/">
        <id:Name>Ron Howard</id:Name>
    </rdf:Description>

    <rdf:Description rdf:about="http://www.imdb.com/name/nm0326040/">
        <id:Name>Akiva Goldsman</id:Name>
        <dc:description>WGA Writer</dc:description>
    </rdf:Description>

</rdf:RDF>
```

Figure 2.3 RDF with multiple examples

From the previous example, the W3C RDF validation service [61] can be used to create a graph to visualize each predicate, as can be seen in Figure 2.4.

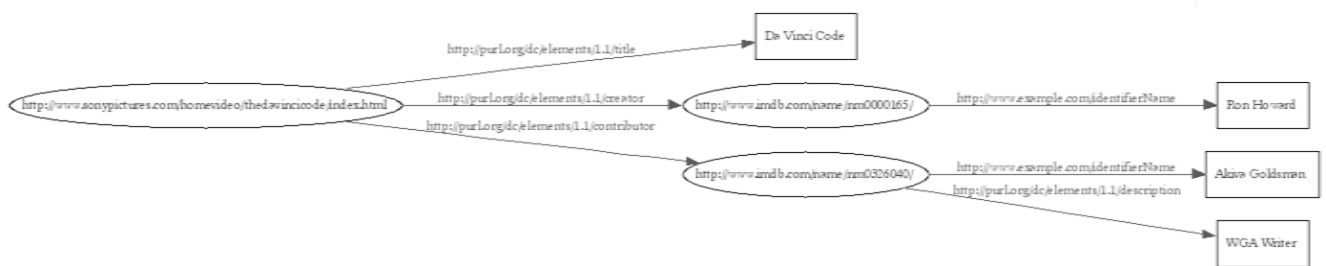

Figure 2.4 RDF Graph

RDF is a technology that enables the annotation of existing content with a well defined semantic, and is the base for the Semantic Web. In the RDF family there are still some relevant technologies that will be briefly presented in the following paragraphs.

RDF Schema [62] is a W3C recommendation for the construction of RDF based vocabularies. Essentially, it allows the definition of classes, subclasses and properties of those classes, in a very similar way to most Object Oriented (OO) programming languages, such as Java. As such, it allows the creation of hierarchies of classes for the description of “objects”.

SPARQL [63] is a RDF document query language designed by the W3C, that since January 2008 is a W3C Recommendation. SPARQL allows querying multiple sources of information, whether that information is natively in RDF or is supplied by some middleware. The results of a query can be a RDF graph or a set of statements.

2.3.2. OWL

The Web Ontology Language (OWL) [64] is a family of knowledge-representation languages to create ontologies. It was designed for usage by applications that need to process the content of information, instead of just presenting it to people. OWL promotes a better interoperability of web content, between machines, than what is supported by formats such as XML, RDF and RDF Schema, supplying an extended vocabulary as well as a formal semantic, allowing to develop formal ontologies.

OWL is composed by three sub-languages, progressively more expressive: OWL Lite, OW DL and OWL Full [1, 55].

1– OWL Lite supports class hierarchies and simple restrictions. It also supports limited cardinality restrictions (0 or 1), equality restrictions between classes, all features for all

properties (transitivity, symmetry), restrictions to the property values (for example values from a list of values, or a subset of values from a list), class intersection, versions and annotations.

2– OWL DL includes support for all functionalities, allowing full expressiveness, having some restrictions at the computability and decidability level (i.e. all conclusions are computable and are guaranteed to finish).

3– OWL Full allows for full expressiveness and syntactic freedom of RDF, but gives no guarantees about the end of the computation.

2.3.3. Simple Knowledge Organization System

The Simple Knowledge Organization System (SKOS) is a RDFS designed to represent and share controlled vocabularies, such as taxonomies, glossaries or thesauri in a simple way within the structure of the semantic web [65]. An ontology language such as OWL, adds a layer of greater expressiveness to RDF, which is used to create statements about resources. However, to develop a complete ontology, a precise modeling effort (a time consuming process) is required and that process demands qualified people. In several cases there's no need of such a formal effort, thus, there was the need to develop a language to write vocabularies to use with semantically rich metadata, powerful enough to support semantic querying, but simple enough so that is doesn't require a vast amount of resources in its production [53].

The core vocabulary of SKOS allows the definition of several types of controlled vocabularies normally used in IS, such as, for example a glossary, taxonomy, a thesaurus of a classification schema. SKOS can be combined with other vocabularies and the semantic of its relating properties, extended [53].

Currently, SKOS is a W3C working draft and a great amount of work still needs to be done, as such, it cannot be used as a standard, but the basis is solid enough to consider that in the future, with the availability of tools (that are being created) for the production, maintenance, management and sharing of these vocabularies, this system can be a viable choice for integrating metadata in information systems.

2.3.4. State of the Semantic Web and its applicability to organizations

The Semantic Web promises a set of technologies and features that will resolve some of today's greatest issues at the information management level, as well as the search level. Since this vision includes annotating data with meta-information, having mechanisms to provide that data with semantic/context and above that level to have tools to process (and extract new knowledge from) that knowledge, this would make the Semantic Web technologies the best choice to deal with

Knowledge Management within organizations. In spite of this situation there are several factors that don't favor the Semantic Web (RDF/OWL) at its current development stage.

RDF and OWL are standard technologies, although they are not so widely available and implemented as, for example, XML technologies. XML is found in nearly everywhere, and most people know XML (or, at least, have heard of it). Literature and support at the open source (and commercial) level is very high and web services (currently very popular) use XML for transmitting information. Also, XML is widely used since controlled vocabularies can be created with it and there are several standard XML vocabularies that model numerous domains of knowledge [66] and technologies to validate, transform and store XML are also very wide-spread [27, 34, 38, 39, 41, 44, 60].

For some time now, most open source and commercial databases include support for XML in several ways while support for RDF was introduced as an add-on only to the latest version of Oracle 11G Enterprise Edition [67] of the popular Oracle Relational database, but it's not found in other popular databases such as MySQL, PostgreSQL or MS SQLServer, however, all of them include support for XML [68-70]. Also, several native XML databases exist, but the choice for RDF native storage is much smaller and less mature.

As an example, a simple search for "XML" in one of the major online book store, returns more than 18.000 results. A search for "RDF" returns about 6.000 results and a "Semantic Web" search, about 4.000 results. It strengthens the idea that XML knowledge is more widespread and known; on the other hand Semantic Web technologies (and there several of them) would force technicians, users and development teams to a long learning curve, given the quantity of new content to master and its complexity (particularly at the inference level).

The nature of the Semantic Web is such that users and organizations must build applications and add content to make use of the content. The problem is that users cannot wait for the vision of the Semantic Web to fully materialize, because that can take over ten years (taking into account the current vision) [54]. In that scenario, the adoption of XML for the representation of metadata and its management in an organization, presents a far lower cost.

The Semantic Web does not have issues only with maturity, technological support and documentation support for its adoption at a large scale. It has its own problems to solve, even inside its community, namely the Open Word Assumption (OWA) versus the Closed World Assumption (CWA) issue and the problematic of ontology integration.

The Semantic Web, in particular the OWL language, makes use of OWA [71]. In OWA, if a statement cannot be proven true with the current knowledge, then it cannot be proven false. The inverse model is CWA, where if something cannot be proven true, it's considered false.

OWA models well several normal inference situations, but can be inadequate for problems that need full knowledge about the "world" (the context of the problem). Consider the problem of a train schedule; if on the schedule there is no information about the train at 12.47, one would assume that that particular train does not exist. In OWA, that cannot be concluded, only in CWA [72]. The problem is, there are real-world problems that require CWA, or at least CWA applied to part of the problem. There's some research regarding this topic [72, 73], that propose the extension of OWL to include non-monotonic constructors, although there are some who defend that the problem can be solved recurring only to first order logic, a statement that the authors of [72] argue that only partially resolves the issue. OWL is also criticized for having a rich set of constructors for classes, but not as rich in dealing with properties. In particular, it's not possible to capture a relation between a composed property and another property (regardless of it being composed or not). The classic example is the relation between the composition of "father" and "brother" and the property "uncle" [74].

Another issue with the Semantic Web is that its structure is not fully stable. Since the beginning of the Semantic Web the layer stack proposed by Tim Berners-Lee was accepted. Recently, with the stabilization of the lower layers, some work has been done in the logic layer and the problem of the exclusive OWA usage versus the usage of local CWA on some situations, has risen. This situation led to some alternative propositions of the well-known layer stack model, for example in [75]. Figure 2.5 depicts the proposed model.

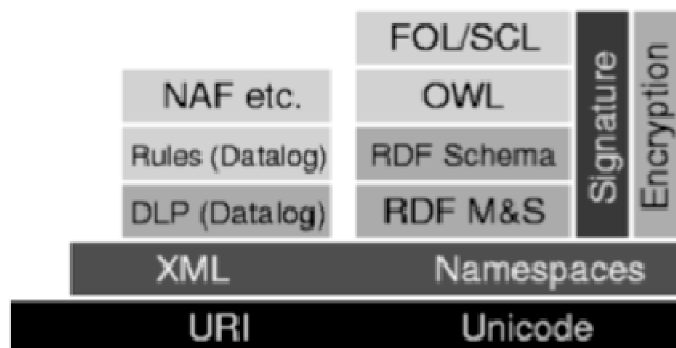

Figure 2.5 Semantic Web Layer Stack with Datalog Rules

This structure has the obvious disadvantage of taking the Semantic Web in two separate ways, which the author of [75] considers harmful (because it promoted the division of the Semantic Web), and proposes another alternative in [75].

A real Semantic Web would allow for applications that process information based on ontologies, at any given time to integrate a new ontology about the same subject, or just part of the subject, as long as it's related somehow. In an ideal situation, a program would be capable of integrating the

ontology, process the data taking into account the new knowledge it possesses and extract new knowledge. Reality, however, shows that ontology integration is a difficult process. There are several challenges in ontology integration and a large number of investigators assume this is one of the biggest challenges of the Semantic Web [76]. The existing challenges for ontology integration according to [77] are the following:

- Discovery of similarities and differences between ontologies in an automatic or semiautomatic way;
- Definition of mappings between ontologies;
- Development of an architecture for the integration of ontologies;
- Composition of mappings between different ontologies;
- Representation of uncertainties and inaccuracies in mappings.

The requirements derive from problems found, at various levels, when ontologies need to be combined. A list of these problems can be found in [78] and can be divided in problems related to the used language (in the cases where the language used to define the ontology was different), such as syntax problems, logical representation problems and semantic disparities with the operators of the language. The other class of problems is related with ontologies themselves. For ontologies that model overlapped domains (partially overlapped or fully overlapped) there are, potentially, scope problems when two classes that represent the same concept have different scope in each ontology or when ontologies model the domain with different levels of granularity or do not cover the same extension of the domain. Essentially the problem is: different people use different terms for the same things and, as such, there's the need for mappings and translations between different ontologies [79]. Another fact is that ontology integration is a subject of great interest in the academic world, but not as much in the commercial one (with no tool support for this).

2.4. Metadata Repositories and Tools

This section presents a set of tools and repositories that provide the means to manage metadata for end users and systems. There are two kinds of metadata management tools, the specific tools (that only allow a certain type of metadata) and the generic tools that allow any kind of metadata. There are several software programs that are based on metadata; the most well known examples are Data Warehousing and ETL (Extraction, Transformation and Loading) tools. No commercial tool will be presented in this section, only open source tools that make use of the previously mentioned technologies.

2.4.1. Repository In a Box

Repository in a Box (RIB) [80] is a software that enables the creation of metadata repositories for the web, developed by the Innovative Computing Laboratory at the University of Tennessee. Metadata, from RIB's perspective, is information that describes reusable objects, such as software [80]. The repository supports the Basic Interoperability Model (BIDM) [81] to store metadata about

objects (resources) and can share the information with other repositories that support the same model (RIB repositories, or not). The Basic Interoperability Data Model (BIDM) is a standard to define the minimal set of information for assets (generic items of interest) in reusable libraries. BIDM results of a collaboration between the Reuse Library Interoperability Group (RIG) and the Software Engineering Standards Committee (SESC) of the IEEE Computer Society and was developed in 1995. It's composed of five classes, each of them having a fix set of attributes and relations [81]. It's possible to alter the BIDM model used by the repository, although it's not advisable because it can limit its interoperability features with other BIDM repositories, despite that, a data model editor is supplied so that the model can be altered.

RIB can create an online HTML, customizable, catalog from its stored metadata and it's possible to make simple searches in the catalog. Data input is done in a web-based interface (with web forms) so that users can add, update and maintain metadata about objects. The repository offers a web-based administration interface for each of the three kinds of existing users. The "general" user, which can add and edit metadata as well as browser the catalog. The "repository administrator", which can alter the data model used by the repository and the "RIB administrators" that can create or delete repositories. At the security level, a password for browsing a given repository/catalog can be set.

RIB is built entirely with Java technology, with data persistency being assured by the open source relational database MySQL. Server side components are Java Servlets, while on the client side Java Applets are used. It's possible to communicate with the repository thorough the RIBAPI, an API supplied by the authors, so that other applications can use RIB and access its data. The API usually returns the result as a XML document, with a well-defined structure (described in the manual of the API).

Although it supports a standard for metadata, makes some use of XML technologies and has an interface for other applications to communicate with it, RIB is very specific and not adequate for a generic information system; also the last update to RIB was in October 2006.

2.4.2. DSpace

DSpace is an open source digital repository that "*captures, stores, indexes, preserves and shares digital research material*" [82]. It was developed by the Massachusetts Institute of Technology (MIT) in partnership with Hewlett-Packard laboratories. It's a repository that answers the needs of teaching institutions (as well as other types of institution) that wish to have digital repositories of several kinds of files, to archive and preserve those files. It's capable of indexing elements such as scientific articles, technical reports, thesis, images, audio and even video using the Dublin Core standard to add metadata to every file. It is, however, the only standard which DSpace is compatible with [82]. DSpace provides a customizable web interface so that users can search for the available items, authors can submit their documents (with the necessary metadata) and administrators can organize

these documents in collections. DSpace is implemented in Java, runs in a web server such as Apache Tomcat and uses a relational database to store the data. The database that can be open source (such as PostgreSQL) or commercial (such as Oracle) or a database that is not part of a DBMS, such as McKoi [83]. DSpace specifically runs in UNIX systems, but it can run on Windows also [84] and has an API that allows for external applications to use the repository and have access to its content.

DSpace has a fixed data model [85] that allows the definition of a hierarchy of "Communities" each them being a set of "Collections" that contain "Items" composed by "Resources" (Files) and "Metadata" (compatible with Dublin Core). DSpace is considerably versatile and there are several Institutions/Organizations and Universities that use it [86] and it's considered a good solution for sharing digital assets, but the fact that it only supports the Dublin Core format, makes it an inadequate solution for a generic information system that does not require (or need) support for the format.

2.4.3. Protégé

Protégé [87] is an open source tool for creating domain models and knowledge-oriented applications through ontologies, developed by the Stanford Center for Biomedical Informatics Research of the Stanford University School of Medicine. The core of Protégé, implements a rich set of knowledge representation structures and actions that allow creating, managing and visualizing ontologies in several formats. Although not a metadata repository, protégé was chosen as a representative tool of the Semantic Web world for illustration purposes and to illustrate some possibilities of the semantic web techonologies.

It has a customizable interface, so that a simple graphic application can be produced for the creation of ontologies (and populating them with information). Protégé can be extended with plugins and a Java API exists so that applications on top of (or connected to) protégé can be built [88].

The protégé platform is available as two products. **Protégé-Frames** and **Protégé-OWL**, explained in the following sections.

Protégé-Frames

The Protégé-Frames editor provides the tools to easily create an ontology for any domain, as well as its maintenance and data input. Protégé implements a knowledge model that's compatible with the Open Knowledge Base Connectivity protocol (OKBC) [89]. In this model, an ontology is seen as a set of classes that can be organized in a hierarchy (to represent domain concepts in a very similar way an object-oriented programming language does) and a number of "slots" can be associated to classes in order to describe its properties and relations. It's possible, after that step, to create

instances of the ontology that are individual examples of classes that have specific (and distinct) property values.

The ontology creation process is eased by a simple graphic interface, as depicted in Figure 2.6. After creating non-abstract classes in an ontology, it's possible to create instances of those classes, an action Protégé eases by generating specific forms to create those instances (the forms can be further

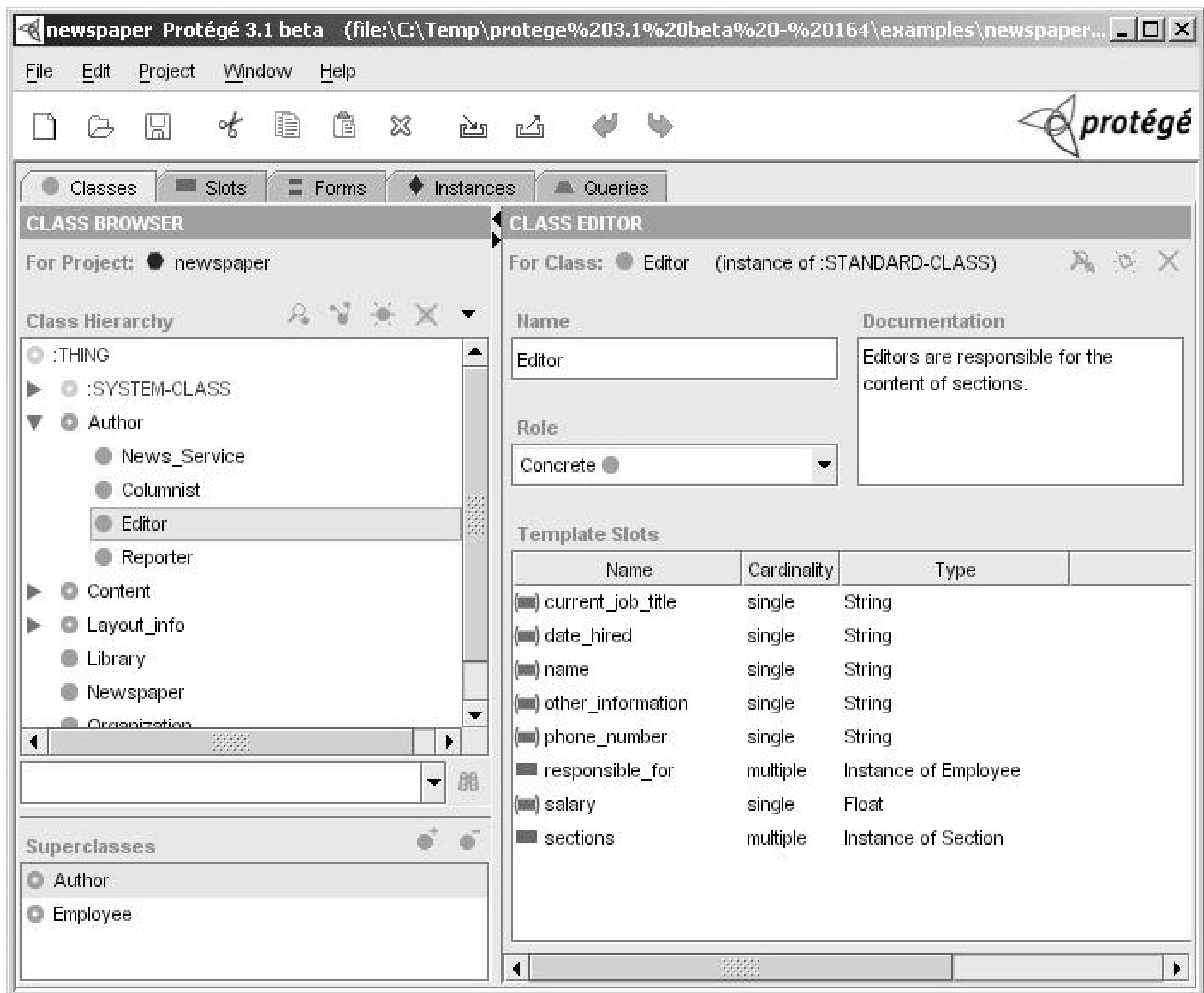

Figure 2.6 Protégé-Frames editor

customized to make it even easier for users to interact with them). During the creation phase of ontologies (or of class instances) all rules are verified, including cardinality rules, relationship rules or restrictions to values of properties and the generated interfaces already have those restrictions into account [90].

Protégé-Frames supplies a graphical interface to query instances of the ontology, and allows making selections based on criteria over the values of the slots of a class. In figure 2.7 is depicted the interface for querying, using "salary" as a criteria for the search.

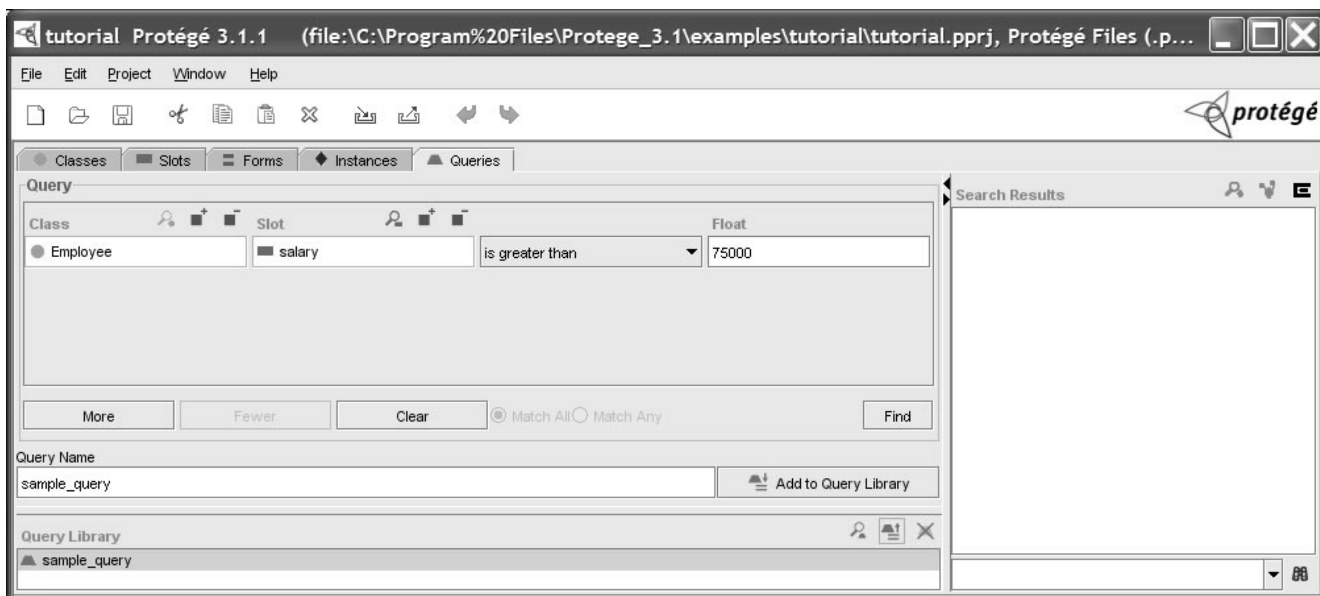

Figure 2.7 Protégé-Frames Query Interface

Protégé-OWL

Protégé-OWL is an extension of Protégé that enables the creation of ontologies based on the Web Ontology Language [91]. Beyond the possibility of creating classes (and restrictions, instances, etc.) offered by Protégé-Frames, Protégé-OWL allows users to create, or load, their ontologies in the RDF/OWL format, allows defining the properties of classes (specific of OWL) as well as use inference engines to extract new knowledge not explicitly in instances, but achievable through the semantic rules of the ontology. It also supplies an equally simple interface (supported by several plugins) that eases the graphical inspection of an ontology (among many other actions) that is depicted in Figure 2.8.

Protégé-OWL is closely tied to the Jena framework [92]. Jena is a Java based framework that supplies a standard environment for the creation of Semantic Web applications that supports RDF, RDFS, OWL and SPARQL, featuring an inference engine that can extract new knowledge from existing ontologies. There's a Java API to which applications can connect to take advantage of the features available in Protégé-OWL. Protégé-Frames and Protégé-OWL are tools in the domain of the Semantic Web and, thus, are not of interest to this thesis, which will be using XML technologies, but both are generic tools to deal with metadata in an Information System.

2.4.4. Fedora

The Fedora project is an open source software that supplies institutions and organizations with flexible tools to manage their digital content [93]. Fedora (acronym for Flexible Extensible Digital Object Repository Architecture) features an object model that supports abstraction (an object is always “seen” in the same way, regardless of its type). The content of the object can be located in a remote address or be managed locally by Fedora and there’s support for object relations and, to each object, several visualizers can be associated [94].

Objects, and all operations regarding them, are exposed through a Web Service API (SOAP and REST) and all the operations can be protected with an Access Control List (ACL) [95]. A distributed environment with several repositories is supported to enable data separation, workload balance and there’s the possibility of several physical repositories being seen as one logical repository [94].

At the storage level, Fedora keeps its objects in a XML format and possesses versioning mechanisms for all of its content, as well as keeping a history of all updates done to an object [94].

Figure 2.8 Protégé-OWL - Ontolgy Visualization Plugin

Relations between objects are done using RDF and that information is kept in a specialized database that can be queried using a language to query relations in graphs. Relations may be derived from any ontology, including the basic relationship ontology supplied by Fedora itself [96]. The collection of objects is available through a web interface depicted in the Figure 2.9.

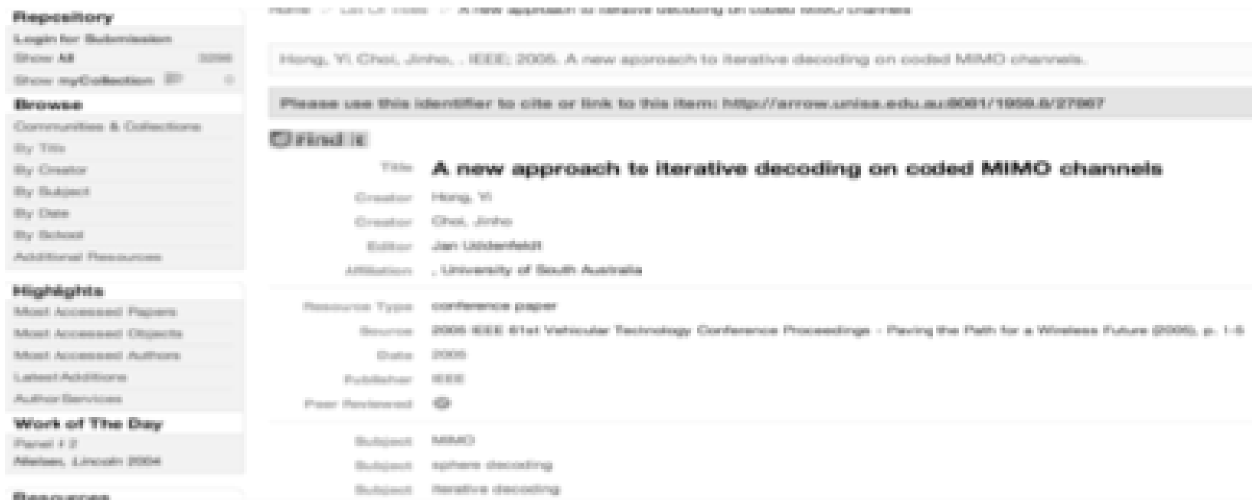

Figure 2.9 Fedora's online catalog

Each object is accessible through the API, using an identifier (a URL) and to each object several contents can be associated (an object is a collection of contents); to access to them, URLs derived from the main identifier are used. Each object may also have several “disseminators”. A disseminator is a service that receives as input a data stream (one of the contents of an object) and returns a visual representation of that content [96].

Internally, each object is represented with the model depicted in the Figure 2.10.

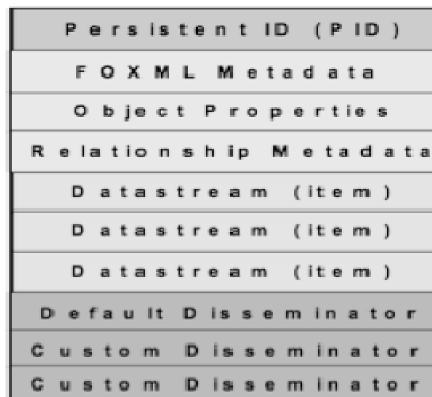

Figure 2.10 Fedora's Object Model

Each object has a PID (Persistent ID) that is used as an identifier inside the repository. There’s metadata to help managing the object, which is required to exist (Fedora Object XML data, or FOXML). “Object Properties” stores the values for the type of object, its state, update date, etc. Relations with other objects are stored in “Relationship Metadata”. The several sources of information of the object are stored as “Data streams”. The model also features the disseminators.

Fedora is primarily a document repository and its use as a metadata repository in Information Systems has its advantages, but the FOXML metadata in the object (which is required) is specific to the management of the repository, any other metadata of an object must be added as a datastream, but it's still part of the object. Fedora has several features that make it very useful as a metadata repository, but does not promote (or allow) reusability, because it's concerned in storing and managing documents and not reusable knowledge and thus, objects cannot be reused to create new objects, although relations between objects are supported. This fails to comply with the objectives defined for this thesis.

2.4.5. Extensible Metadata Repository for Information Systems

The Extensible Metadata Repository for Information Systems (MDR) [1], was developed in the context of the SESS project. The repository was built to manage all the metadata of the project (easing and promoting its use) and was available to every application in the system. The repository was built with the following requirements in mind.

- Extensibility
- Usability
- Integration
- Security

To assure extensibility, an information model based on XML technologies and the MOF meta-architecture was chosen. XML is used to store metadata and XML Schema is used for validation (with the option to use Schematron to verify additional restrictions). The information model is based on MOF and maps to the repository like it's depicted in the Figure 2.11.

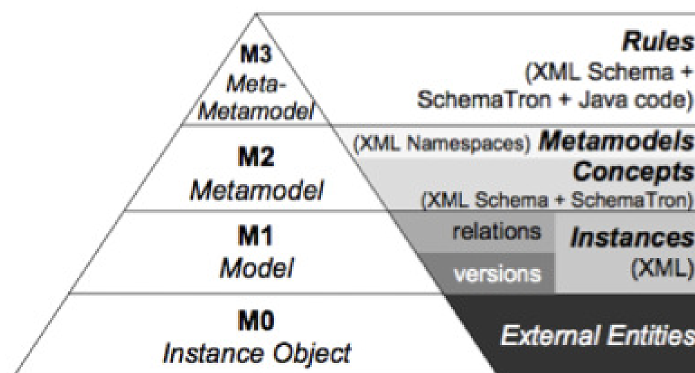

Figure 2.11 MOF model in the context of the MDR, taken from [1]

As stated in chapter one, metadata is stored in XML files and, in the context of the repository, these files are designed as "Instances". The vocabularies to which these XML files abide are created

using XML Schema (with optional use of Schematron) and are named “Concepts”. The set of rules and restrictions that apply to all Concepts is also defined using XML Schema, Schematron and, additionally, Java code.

A Concept in the repository is more than a XML Schema. In the Concept definition, one can define how Instances of that Concept can relate with Instances of other Concepts and what restrictions to apply to those relations (such as cardinality control). It’s also possible to define Schematron restrictions that Instances of that Concept must comply with. The definition of all these properties is done with a specific syntax that is available in the base schema that all Concepts must include. For a Concept to be stored in the repository, it must be valid against a set of rules: It must include (and use elements from) a base XML Schema and comply with a set of global schematron rules. Some rules are verified by Java code, because the proper functioning of the repository depends on that.

Regarding integration, the repository features a very simple, but extensive, web service API that allows any application to communicate with the repository and request anything (validate an Instance, update an Instance, add a new Concept, etc.). It has a notification mechanism that allows the repository manager to select a set of Instances (according to several criteria) and create a trigger that invokes a remote web service (with any number of parameters) when there’s a change in any of those Instances. Metadata integration is assured through importing and exporting mechanisms (from and to repositories of the same type, respectively). There’s also a subscription feature from another repository to ensure that an updated local copy of a given set of Instances document, always exists. Specific importers (small applications that can transform external documents to valid Instances) can be connected to the repository.

An ACID transaction system on top of the database is present, as well as a replication mechanism. Regarding user security, the repository features permissions and an authentication mechanism.

Usability is assured by using open source and platform independent technologies, such as Java, the eXist native XML database, etc.). The repository’s management console and external metadata editors (connected to the repository) further extend the usability of the repository.

The repository allows users to execute XQuery over Instances and transform them. The result of a XQuery can be passed to a single XSLT, or a XSLT pipeline, as depicted in Figure 2.12.

The repository supports Instance relations assuring referential integrity between them and cardinality restrictions. It’s possible, for example, to define that an Instance of the Concept “Master of Science Thesis” is related with at least one Instance of the Concept “Author” and if there’s an attempt to remove an Instance that’s related to another, the repository blocks that attempt. It will only allow the removal when no relations exist.

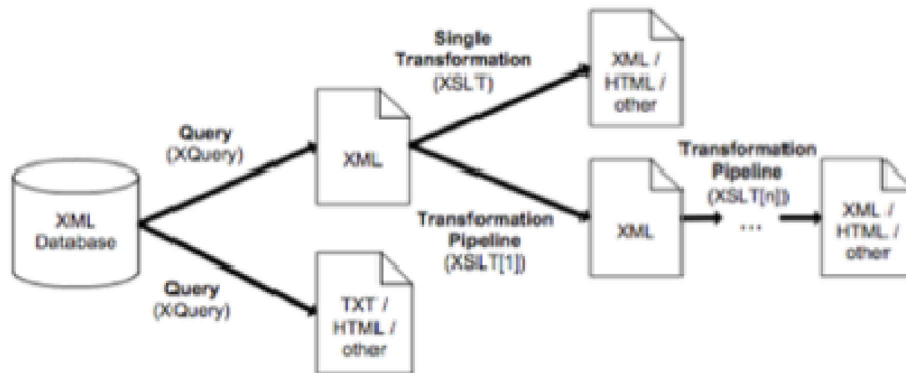

Figure 2.12 Instance processing and transforming capabilities, taken from [1]

The repository has, however, some issues that are major drawbacks for its adoption in a generic information system. Those issues cause the following problems:

- Difficult integration of external documents
- Reduced extensibility and reusability

The integration of external documents is a problem, mainly because of the following situations in the repository:

- Instance Identification
- Instance Relations
- Schema Rules

Instance identification in the MDR is done using an internal identifier, that is the concatenation of a “Server identifier” with a “Database identifier” and a sequence number. As an example, the identifier “1.1.4” means Instance number four, located in database one of server one. These identifiers are generated by the repository on the moment of Instance insertion, and are stored inside the XML content of each Instance. Instances may have a “Named” identifier, but an internal identifier is always required as the name may change over time. Relations between Instances are based on the previous identifiers and the declaration of a relation is done using a fix and rigid syntax, which is stored inside the content of an Instance, as is depicted in the *DataServiceProviderRelation* element of Figure 2.13.

Since relations are based on the previous identifiers, it is impossible to add cross-related Instances simultaneously, because Instances are only given an identifier after they are inserted, thus they must be previously inserted and relations created afterward. The use of named identifiers is possible to overcome this situation, but it requires that someone has to create those named identifiers.

```
<ParameterType>Other</ParameterType>
<Unit> [4 lines]
<OOL> [17 lines]
<DataOrigin>Real</DataOrigin>
<DataServiceProviderRelation type="relation">
    <mdr:Relation globalId="1.5.26" lockVersion="false"
        name="ESOC - European Space Operations Centre (Simulation)"
        namespace="http://sess.uninova.pt" semantic="provided by" version="last"/>
</DataServiceProviderRelation>
<SpacecraftSubsystemSensorRelation type="relation">
    <mdr:Relation globalId="1.5.5" lockVersion="false"
        name="INTEGRAL - International Gamma-Ray Astrophysics Laboratory"
        namespace="http://www.esa.int" semantic="measured by" version="last"/>
</SpacecraftSubsystemSensorRelation>
<DataGenerationModelRelation type="relation"/>
<CalibrationRelation type="relation">
    <mdr:Relation globalId="1.5.34" lockVersion="false" name="HSL Traffic Indicator 8019"
        namespace="http://sess.uninova.pt" semantic="calibrated by" version="last"/>
</CalibrationRelation>
<GrainLevel>Second</GrainLevel>
```

Figure 2.13 Relationship syntax in a SESS Instance

The set of schema rules that apply to all Concepts, forces that if a XML Schema was not created for the MDR it has to be updated to be compatible with the rules. This situation imposes that every XML file that was valid against the previous schema, has to be updated to conform to the new schema. At the schema level, the rules force that a base XML Schema, supplied by the repository must be imported and that the root element of the schema must be identified with an “id” attribute, which must have the value “root”, among other restrictions. Most of the rules are contained in a

Schematron file (and thus if the Schematron would be removed they would no longer be applied, but if they're not enforced the repository engine will not work as expected). As a visual example in figures 2.14 and 2.15 are depicted the rules imposed by the repository to an Instance and a Concept, respectively. Outlined in red are the rules that the Java code expects to see checked, outlined in green are the rules that are part of the Schematron rule file, but could be removed without affecting the repository.

```xml
<Component xmlns:mdr="http://di.fct.unl.pt/MetadataRepository"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sch="http://www.ascc.net/xml/schematron" xmlns="http://sess.uninova.pt/Management"
    globalId="1.4.11" version="last"
    xsi:schemaLocation="http://sess.uninova.pt/Management Component.xsd">
    <mdr:Name>Alarm Engine</mdr:Name>
    <mdr:ShortName>Alarm Engine</mdr:ShortName>
    <mdr:Description>Inference engine that triggers alarms according to received data</mdr:Description>
    <mdr:AuthoringInfo>
        <mdr:Author>Ricardo Raminhos</mdr:Author>
        <mdr:CreationDate>2007-05-22T13:39:02</mdr:CreationDate>
        <mdr:ModificationDate>2007-05-22T13:39:02</mdr:ModificationDate>
        <mdr:Status>Proposal</mdr:Status>
        <mdr:Comments/>
    </mdr:AuthoringInfo>
    <Type>Application</Type>
    <SubComponents/>
    <ResponsibleEntityRelation type="relation">
        <mdr:Relation globalId="1.4.9" lockVersion="false"
            name="UNINOVA - Instituto de Desenvolvimento de Novas Tecnologias"
            namespace="http://sess.uninova.pt/Management"
            semantic="Responsible entity for the component" version="last"/>
    </ResponsibleEntityRelation>
    <RequirementStartNumber>2400</RequirementStartNumber>
</Component>
```

Figure 2.14 Instance with outlined rules required by the MDR

```xml
<xs:schema xmlns:mdr="http://di.fct.unl.pt/MetadataRepository"
    xmlns:sch="http://www.ascc.net/xml/schematron"
    xmlns:cd="http://di.fct.unl.pt/MetadataRepository/ConceptDefinition"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.esa.int"
    attributeFormDefault="unqualified" elementFormDefault="qualified"
    targetNamespace="http://www.esa.int">
    <xs:import namespace="http://di.fct.unl.pt/MetadataRepository" schemaLocation="base.xsd"/>
    <xs:include schemaLocation="UIM_base.xsd"/>
    <xs:element id="root" name="GroundBase">
        <xs:annotation>
            <xs:documentation>definition of a ground base</xs:documentation>
            <xs:appinfo source="ConceptInfo">
                <cd:ConceptInfo>
                    <cd:Name>Ground Base</cd:Name>
                    <cd:Type>Domain</cd:Type>
                    <cd:DocumentationURL/>
                </cd:ConceptInfo>
            </xs:appinfo>
            <xs:appinfo source="ConceptAuthoring">
                <cd:ConceptAuthoring>
                    <cd:Author>Marta Pantoquilho</cd:Author>
                    <cd:CreationDate>2007-05-22T14:29:53</cd:CreationDate>
                    <cd:ModificationDate>2007-05-22T14:29:53</cd:ModificationDate>
                    <cd:Version>1.2</cd:Version>
                    <cd:Status>Accepted</cd:Status>
                    <cd:Comments/>
                </cd:ConceptAuthoring>
            </xs:appinfo>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:group ref="mdr:identificationElementsGroup"/>
                <xs:group ref="mdr:documentationElementsGroup"/>
                <xs:element minOccurs="0" name="HostingInstituteIdentification"
                    type="generalIdentificationType">
                    <xs:annotation>
                        <xs:documentation>The hosting institution identification</xs:documentation>
```

Figure 2.15 Concept with outlined rules required by the MDR

The MDR has some extensibility issues, linked to reusability issues. The major issue with reusability is that the basic unit in the repository is the XML Schema (as a Concept) and there are no tools or mechanisms to reuse a Concept (or part of it) to create a new Concept. This means that, in practice, people designing new Concepts will not try to reuse existing parts, because they will not gain anything from it and this leads to creating Concepts that have similar meaning but different structure because no one checked that a similar Concept already existed. In addition, the MDR does not feature Concept versions; only Instances can have versions, which means there is no explicit support for temporal evolution of Concepts.

The MDR is a good solution for the management of metadata in the SESS project and provides several features identified as essential in metadata management earlier in this chapter and chapter one. However, for a generic information system that needs to integrate external metadata that may be in very different formats, it is not suitable to the task. The cost of transforming XML documents and XML Schemas into Instances and Concepts, respectively, is very high. The lack of support for the use of Concepts (or parts of) to create new Concepts does not promote reutilization and, as such, people may find themselves redefining Concepts that are very similar (or that **should** have part of the same structure) which leads to several conflicting definitions in the same repository.

2.5. Metadata Management Technologies and Repositories Appreciation

There are several technologies to deal with metadata and two of the most representative ones are XML and the Semantic Web. Semantic Web would be the most promising technology, but it has a greater learning curve and some issues are still unresolved and subject to research. XML, although not delivering some of the very interesting features of the Semantic Web is very stable and widespread and there is much knowledge available about it, thus, it suits a current metadata repository better than the Semantic Web. On another hand, current metadata repositories and tools are limited because they either don't support different kinds of metadata or, as seen in the SESS repository, have issues with integration of foreign metadata.

Commercial solutions were left out of the evaluation since the use of metadata is not widespread and many organizations don't even grasp the benefits of using it; yet alone pay for a commercial solution to manage their metadata. In the case of a SME, even if there's interest in adopting a solution for metadata management, the cost of a commercial one may simply be prohibitive. There are some commercial solutions [97, 98] for metadata management, but they are directed to enterprises of considerable dimension that can afford the high price of such a solution. These solutions are very complete and provide enterprise wide management of metadata but usually are deployed in level above the Information System (rather than supporting it) providing document management, but not having a special focus on technical metadata and its relations with other metadata. Computer-Aided Software Engineering (CASE) tools are, as the name implies, tools to

assists in the several phases (analysis, design and programming) of software development. They provide assistance in modeling the problem and generating code based on the model and typically they use an internal metadata repository to store model definitions, user requirements and templates for code generation. CASE tools are very specific tools that have a closed model and don't provide access to their metadata repository, thus, they are not a good solution for metadata management outside of their purpose.

Although in this work there is no capacity to reproduce a project like SESS and all of its features, there is enough interest and value to invest in a new metadata repository that has some of the features of the SESS one and deals with the limitations described earlier, promoting the reuse of information and easily integrating external metadata. This is the work developed in this thesis.

Chapter 3

Architecture Design

This chapter presents the architecture design of the metadata repository, identifying requirements, discussing the high-level architecture and information model

3.1 Requirements... 41
3.2 Architecture.. 42
3.3 Information Model.. 43
3.4 M0 Layer (External Entities) 43
3.5 M1 Layer (Instances) 44
3.6 M2 Layer (Concepts) 49
3.7 M3 Layer – Meta-meta-model.. 54

This chapter introduces the design of the metadata repository starting with a list of general requirements for metadata repositories. With those requirements in mind, the high-level architecture of the repository is presented, followed by the information model that defines what types of information the repository must store.

3.1. Requirements

The list of requirements that guided the design of this repository (and were identified in the previous chapters) is present next. Most of the requirements are described in [3, 11] and in the previous chapters, although not all of them were considered as priorities (given the time restrictions) and the following list reflects that.

Ability to handle several types of metadata (1) – It is important to be flexible enough to support standard and non-standard types of metadata, due to much of the existing metadata in Information Systems not being compatible with any standard.

Metadata Storage (2) – The repository must store metadata in efficient and flexible databases, enabling querying and transforming of metadata. The repository shall support several databases.

Metadata Relationships (3) – To represent relations between objects that metadata describes or to make the connection between domain and technical metadata. In Information Systems this property is very important as it represents data dependency between different items.

Metadata Integrity and Validity (4) – Data integrity is crucial in any system thus, the metadata repository must assure that the metadata it stores is always valid as well as relations between them.

Metadata Change Management (5) – Metadata has a dynamic nature, because it describes objects that have themselves a dynamic nature. Therefore the repository must support temporal evolution of the metadata, in the form of a versioning system.

Import (6) – The metadata repository must support the import of metadata from external sources, such as databases or other information systems regardless of the metadata being in a standard, or non-standard format. By importing and centralizing metadata in the repository, the repository becomes the central source of information in the system, helping to create new systems and ensuring consistency in all processes.

Export (7) – To support reuse of metadata, the repository must supply the means to query, retrieve, transform and share metadata with other systems or tools. This promotes the reuse of already existing (and valid) definitions, a crucial factor in creating a coherent and consistent set of systems.

Reusability (8) – The repository shall provide the means to easily reuse definitions and processing capabilities to promote the reuse of information. Users shall gain benefit from reusing and, therefore, make use of those features. This promotes consistency of information throughout the set of systems that use the metadata repository.

Concurrency (9) – Multiple-user access for read and write purposes must be supported by the repository so that availability is high, but data is not damaged by concurrent access.

3.2. Architecture

The metadata repository design takes into account the previously mentioned requirements. The diagram of the architecture is depicted in Figure 3.1, which is detailed in the next paragraphs.

In the center of the figure is the Metadata Repository Engine. This engine implements and

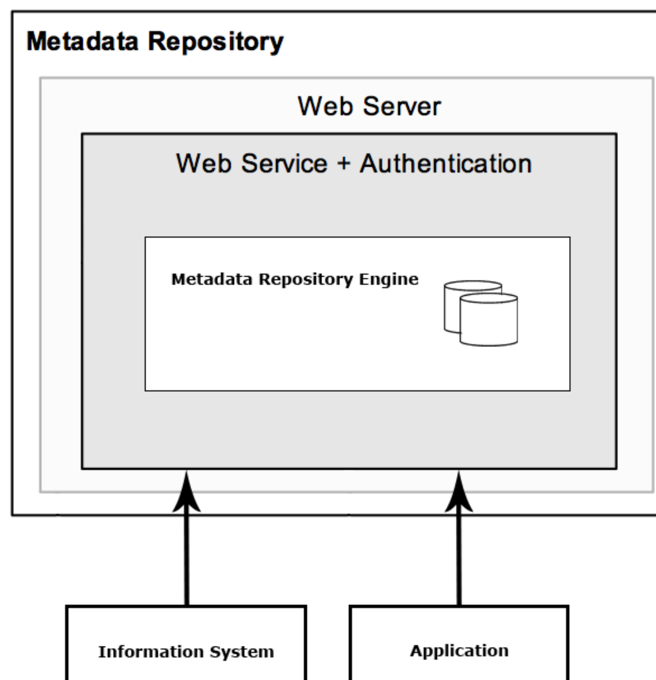

Figure 3.1 Architecture of the Metadata Repository

provides all the features of the repository, storing the metadata in a database, or series of databases (to comply with the requirement of metadata storage). The engine is deployed in a web server and exposes its functionalities through a Web services interface. The Web services interface allows any application, developed in a modern language, to communicate with the repository. The popularity of Web services makes them available in most recent programming languages and the fact that is platform-independent, using XML as a vehicle to pass information, is the reason behind the choice of exposing the functionality of the repository as Restful Web Service Interface (REST) [99]. REST philosophy was chosen for its simplicity, it’s very light-weight as no extra XML markup is required in

communications, and is very easy to build and consume. The only point of entry in the repository is through the Web service interface and, as such, the security control (user login) is done here.

Using a Web Service interface the repository can be in a distributed environment, with several systems remotely communicating with the repositories, requesting metadata, executing queries and requesting transformations over the results of those queries.

3.3. Information Model

The Metadata Information Model of the repository represents the organization of the information the repository must store and the kind of information it supports. The Metadata Information Model is based on the Meta-Object Facility (MOF) depicted in Figure 3.2. It's designed to meet the requirements listed in 3.1.

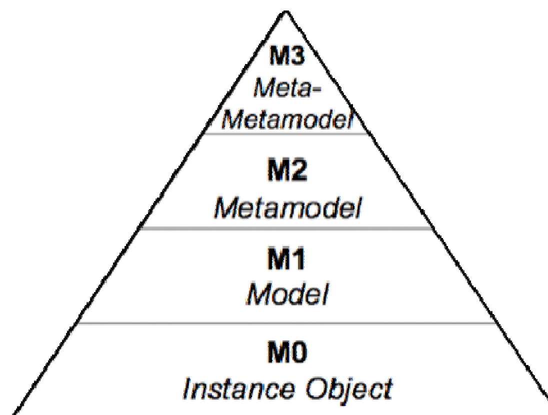

Figure 3.2 MOF layered architecture

3.4. M0 Layer (External Entities)

The M0 layer (Instance Object) represents objects in a given reality. Any entity, physical or not, can be considered for this layer. Examples of these objects can be data dictionaries, reports from an organization or persons. These objects are out of the scope of the metadata repository and are considered external entities, which are represented in the Information Model as depicted in Figure 3.3.

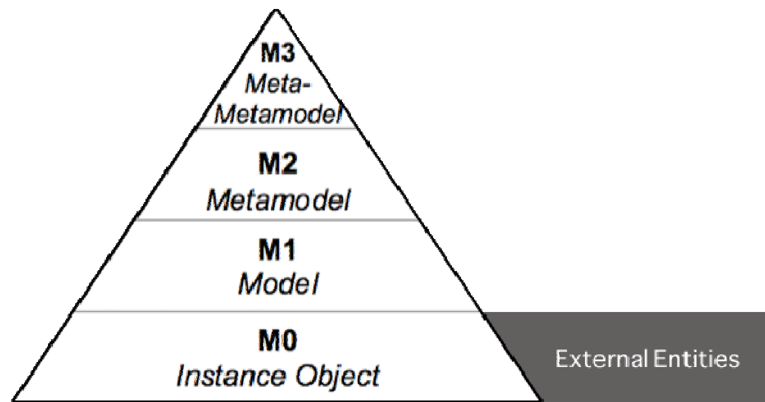

Figure 3.3 M0 Layer - External Entities

3.5. M1 Layer (Instances)

The M1 (Model) layer describes objects or entities in the M0 layer. A model is a pattern, plan, representation or description, designed to describe a main object or workings of an object, system, or concept. The model of a database can be its name, location, system username and password. Models are "metadata" about the M0 objects and, as such, are stored and managed by the Metadata Repository. In the context of the Metadata Repository, Models are designed as "Instances" and are represented by XML documents, as depicted in Figure 3.4.

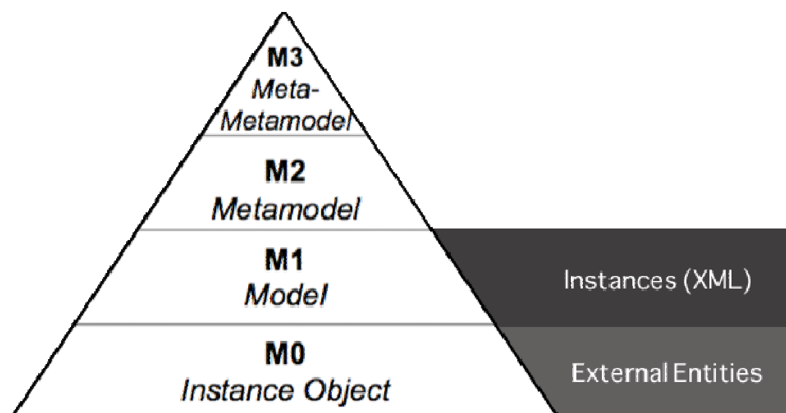

Figure 3.4 M1 Layer: Instances

XML, as presented in chapter 2, is the most natural choice to represent metadata, due to its ability to represent any type of information while being platform-independent. The extensive list of tools, libraries and databases that support it, combined with the rest of the technologies that allow validating, querying and transforming further confirm XML's ability for representing metadata and handle the requirements for metadata management. The following subsections detail the features of Instances to comply with the requirements of metadata change management and metadata relationships.

3.5.1. Instance Versions

Instances feature a versioning system, to deal with the change management requirement. Each Instance has an associated **version number**, a **creation date** and an **update date**. The version number is a positive integer that uniquely identifies that version from other existing versions. Version "1" represents the first version of that Instance, up to the Nth version. Creation Date and Update Date are, respectively, the dates where the Instance was created and the date where the Instance was last updated.

Each time an Instance is to be inserted in the repository, if a previous version already exists one can chose if the previous Instance is to be replaced (overwrite) or if a new version is to be created. This leads to the possibility of, after ten consecutive inserts in the repository, the result being ten different documents (ten versions) or only one document (one insert and nine replacements). The following figures (Figure 3.6 and Figure 3.5) depict the possibilities. The first exemplifies the fields, and the second shows a temporal evolution.

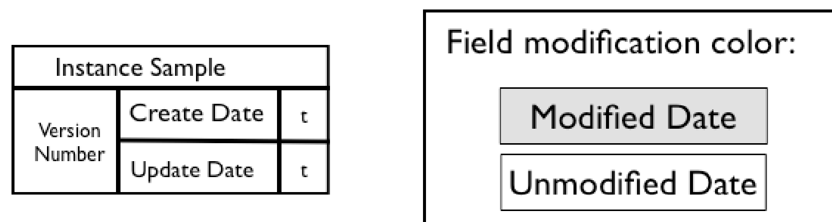

Figure 3.6 Instance Version Control Fields and Modification Notation

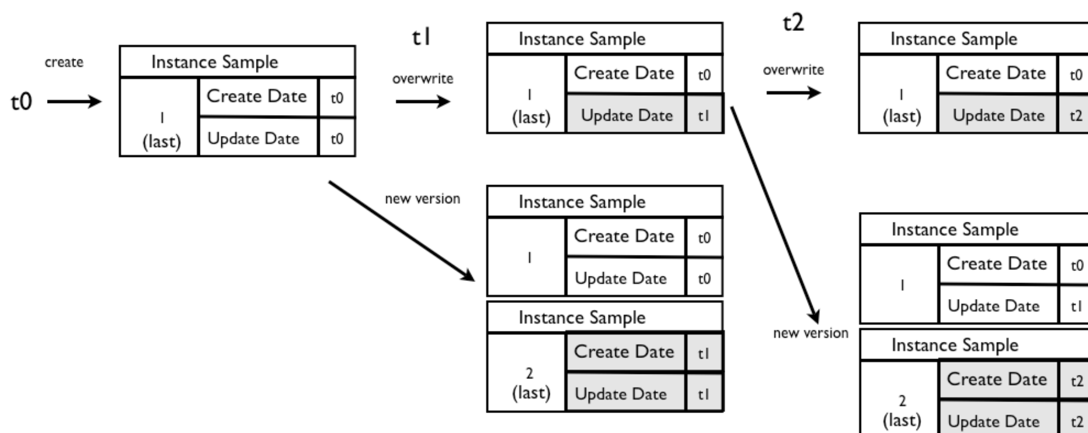

Figure 3.5 Temporal evolution of Instance versions

Starting with time t0 (in Figure 3.6), a sample Instance is created (with version 1, create and update date, t0). In a future, t1, time a new Instance version is to be added and one can choose to overwrite the previous one, or create a new version. Overwriting only changes the "update date", while creating a new version increases the version number and set the two dates, as depicted in figure 3.6. There's the notion of "last" Instance version, as seen in the previous figure. It's a notion that is used in the update of Instance relations; in a situation where instance A is related to Instance B and a new version of Instance B is created, the relation can be migrated to the new version of B. Version number, create and update dates, among others, are metadata about Instances and are stored in a separate file, as such, external XML files can be imported "as-is" to the repository. Instance versions are considered a feature of Instance and the updated Information Model is depicted in Figure 3.7.

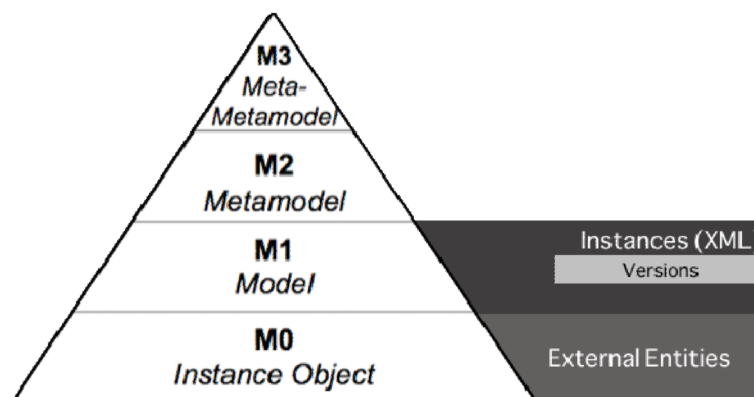

Figure 3.7 Information Model with Instance Versions

3.5.2. Instance Relations

Relationships in metadata represent dependencies between the objects they describe and, as such, are a crucial part of the Metadata Repository and its Information Model. Relation management implementation is dependent on validation and integrity constraints over Instances, because if a relation exists between two Instances, none of them can be removed before that relation is broken.

Relations are only established between Instances and are stored separately from Instances themselves, in internal management files. There are several mechanisms for creating relations, which will be presented in the next chapter. A relation is as a binary association between two Instances and can be described as an arc between two nodes of a graph, where each node represents an Instance. Figure 3.8 provides an illustration (Instance A1 is related to Instance B1).

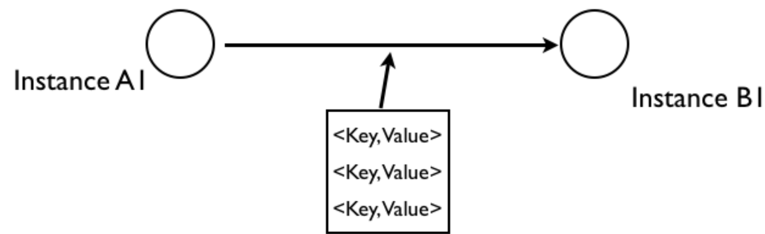

Figure 3.8 Relation between two Instances

Relations can have associated metadata, in the form of key-value pairs. This feature allows the documentation of relations as well as a method to distinguish between two relations with the same target Instance.

A relation between two Instances can be one of two types of targets. The first type is designated as “locked version” and the second as “last version”. “Locked version” is a kind of relation where two Instance versions are related and that relation is locked, i.e. even if someone created a new version of the target Instance, the relation is maintained with the previous version. The “last version” means that, if there is a relation of Instance A with Instance B, when a new Instance of B is inserted in the repository, A will no longer be related to the first version of B and will “migrate” its relation to the new Instance of B and so on every time a new Instance of B is added. When creating a relation, if an Instance version number is provided, the relation is “locked version”, if no version number is supplied, the relation is “last version”. In Figure 3.9 the “lock mechanism” is depicted.

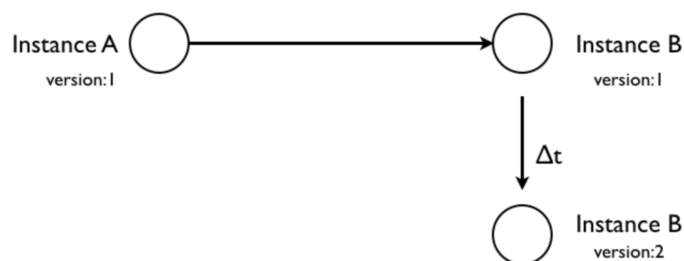

Figure 3.9 Instance Relation with a Locking Version

In the previous figure a “locking relation” between Instance A, version one (A1), and Instance B, version one (B1), exists. At a given time, a new version (version two) of Instance B is added; the relation is maintained between A1 and B1. The opposite situation is depicted in Figure 3.10.

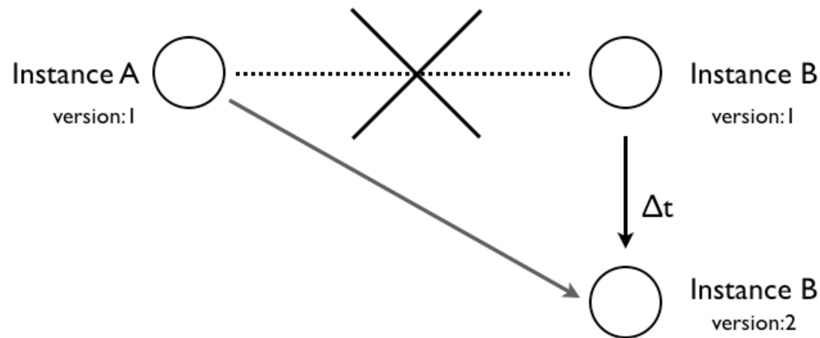

Figure 3.10 Instance Relation with "Last Version"

In the previous figure, a “last version” relation between Instance A and Instance B (both version one, A1 and B1, respectively) exists. At a given time, a new version of Instance B is added, and the repository automatically updated the relations, removing the relation between A1 and B1 and creating a new relation between A1 and B2. Any associated metadata with the previous relation is also “migrated” to the new relation.

A relation between two Instances can also have a finer granularity. It’s possible to specify a XPath that identifies part of the content an Instance and, if that part exists, a relation is established with that specific content. The behavior is equivalent to that of a relation with a complete Instance, the only difference is that a certain part of the target Instance must match a XPath.

Cardinality restrictions can also be set, so if a certain Instance “A” of Concept “Book” exists, it can be set that “A” can only have a relation with one Instance of Concept “Author”, meaning in this example that the maximum cardinality is one. The maximum cardinality can be any positive integer, making it possible to have relations like “one-to-one”, “one-to-many” or “many-to-many”. Minimum cardinality can also be set.

Relations are established between Instances, but the target of a relation is always an Instance of a Concept. As such, one can define acceptable “target Concepts” for a relation. For example, one may want that Instances of the Concept “Thesis” can only be related with Instances of the Concept “Thesis Author” and Instances of Concept “Thesis Supervisor”. In this situation, the Metadata Repository will deny the attempt to relate an Instance of “Thesis” with an Instance of “Book”. The repository will only allow for relations explicitly accepted in the Concept. The definition of relations is a subject that will be introduced jointly with Concepts in the next section and in chapter four.

Relations are considered a sub-feature of the Instances layer and, as such, the final M1 layer model in the Information Model is like the one depicted in Figure 3.11.

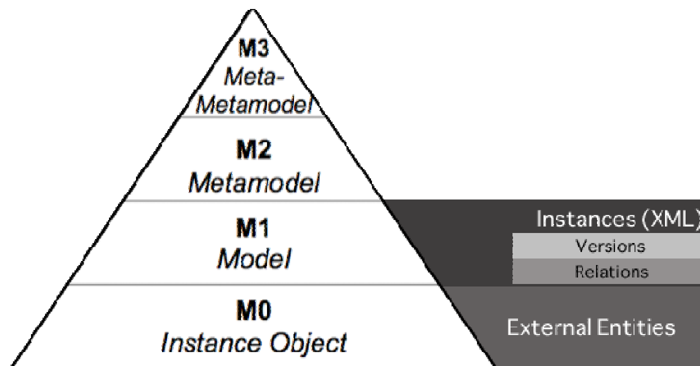

Figure 3.11 M1 Layer, Instances with Relations and Versions

3.6. M2 Layer (Concepts)

The M2 layer (meta-model) describes the models in the M1 layer. A meta-model is the definition of a language, notation or properties of a model, as seen in chapter one and two. In the context of the Metadata Repository a meta-model is defined as a **Concept**; since models are XML files, the meta-models are represented as XML files, which include a XML Schema definition inside them. The reason for this is to separate the definition of the vocabulary, from the other features of Concepts. To comply with the reusability requirement, the notion of **Fragment** will be presented in the next section.

3.6.1. Fragments

A Fragment is, in the context of the Metadata Repository, a reusable piece of structure and processing. A Fragment represents a XML Schema structure and can have metadata about itself and can hold a list of XSLT that know how to process that structure (although there are restrictions to the kinds of XSLT used in Fragments). The structure of a Fragment can be reused by other Fragments to create a more complex structure and reused in Concepts. The best analogy would be with the popular Lego ® pieces, were one can build a structure using a set of small pieces and, afterward, reuse those pieces to build something larger. The purpose of this feature is to explicitly enable, and promote, the reuse of already existing information; if there's a definition for how addresses must be written, an "Address" Fragment can be used to define that structure and all Fragments or Concepts that require the use of addresses can reuse the definition in the "Address" Fragment. Furthermore, if an address has a pre-defined way of being presented in HTML, a XSLT that can process the structure of an address can be associated to the "Address" Fragment and be reused by Concepts. A Fragment is not related directly to an Instance; only a Concept can have Instances.

The repository supports versions of Fragments, so that temporal evolution of the vocabularies is allowed. However, and contrary to the Instances situation, there isn't the notion of "last" version when other Fragments and Concepts reuse the structure of a Fragment. This is required, because the

Fragments can be used by Concepts to define a vocabulary that Instances of that Concept must be valid against. If the structure of the Concept were not static, then it would not be possible to guarantee that Instances would always be valid. When a Fragment or Concept wishes to reuse a Fragment it must explicitly choose a version and if a version number is not supplied, the repository will automatically search for the last version and always use that version. Figure 3.12 depicts this situation.

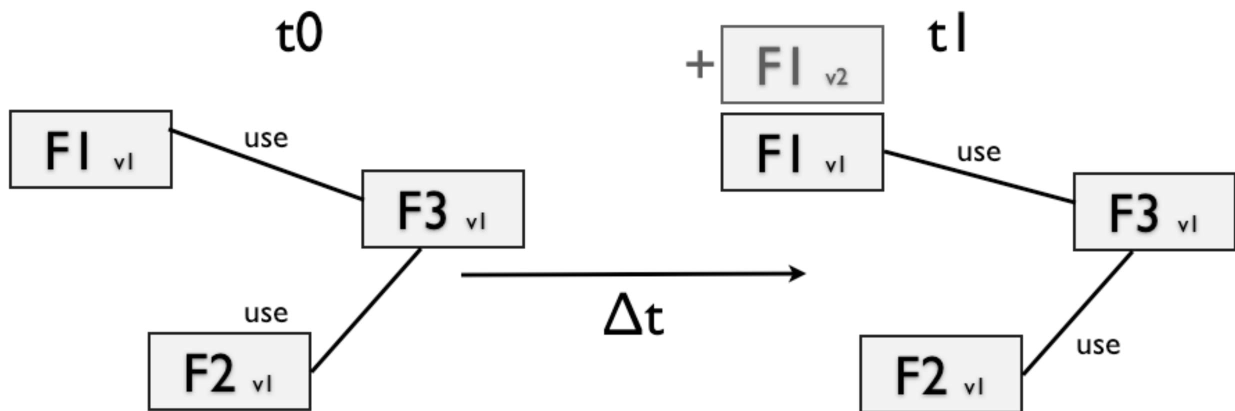

Figure 3.12 Fragment versions

In the previous figure at time “t0”, there are three Fragments (F1, F2 and F3), each of them only has one version (version one) and Fragment F3 reuses both Fragment F1 and F2. At time “t1” a new version (version two) of Fragment F1 is added, but Fragment F3 will not reuse the structure of F1 version two, because it’s locked to version one, for the reasons previously explained. This behavior is also valid for Concepts reusing Fragments.

3.6.2. Concepts

A Concept represents an item that has a series of properties. The most important property is the definition of a vocabulary in XML Schema that Instances of that Concept will be valid against. XML Schema was chosen because Instances are represented as XML documents and, currently, XML schema is the most popular and widely available language to define and validate XML documents. A Concept can, additionally, make use of Schematron to enforce more restrictions to Instances, since Schematron is capable of verifying restrictions that XML Schema isn’t able. An Instance document is an instance of only **one** Concept and a Concept can have zero or more Instances.

A Concept is an item that can have the following properties (in bold, the required properties):

- **Instance Identification method**
- **XML Schema vocabulary**
- Schematron validations

- XSLT
- Relations
- Metadata about the Concept
- Rules to create Metadata about Instances

All the properties of a Concept are defined using the language provided by the M3 layer, that ensures a level of "standardization" in the definition of a Concept and assures that some restrictions exist in the definition of Concepts.

The *Instance Identification method* defines how Instances of this Concept are identified and will be explained in later. The *XML Schema vocabulary* is created by using a set of Fragments (using a method called "Composition" that will be detailed in later sections) or by using XML Schema code freely. Schematron validations can also be used in the Concept to assure additional validations. A list of XSLT can be associated to a Concept and these XSLT are to be used **exclusively** in Instances of the Concept to perform transformations.

Instances can have relations with each other, as previously described. The set of valid targets for Instance relations, as well as the cardinality of those relations, is defined in their respective Concept. Relations can be manually created, or a set of rules can be used to create them automatically upon Instance insertion (the behavior of automatically created relations, can be defined, in case a target Instance is removed or updated). There are two kinds of automatic relations: *Identifier Relations* and *Content Relations*.

Identifier Relations are based on the fact that Instances have a unique identifier within the Metadata Repository (identifiers will be explained in chapter four). That identifier has a particular syntax that allows distinguishing it from regular text, thus, if an identifier is found in the content of an Instance and that identifier represents an existing Instance in the repository, the repository will create the relation between those two Instances automatically. The situation is depicted in Figure 3.13.

In Figure 3.13, Concept A enables relations with Concept B, which has one Instance that is identified with the string "ID" (the identifier is not part of the content of the Instance). On insertion of the Instance of Concept A its content is parsed and the "ID" identifier is found, thus, since Concept A allows *Identifier Relations* with Concept B, the relation between those two Instances is automatically created.

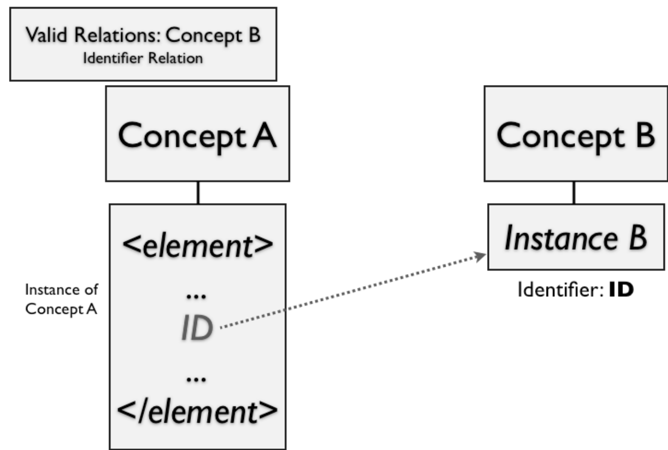

Figure 3.13 Automatic Relation based on identifiers

Content Relations are, as the name implies, based on the content of Instances. This mechanism assumes that there are valid targets defined in the Concept of the Instance that is being inserted. Since relations are content-based, the target Instance has to be "searched". The mechanism works as follows: At the Concept level, two rules to search for specific content are declared. One rule to search in the origin Instance and other to search in candidate target Instances. When an Instance of that Concept is inserted, one rule is applied to the origin Instance and a specific content is retrieved; afterwards, all Instances of valid targets for a relation will have the second rule applied to them and content will also be retrieved. If the content matches with the one from the origin Instance, then a relation is automatically created. The situation is depicted in Figure 3.14.

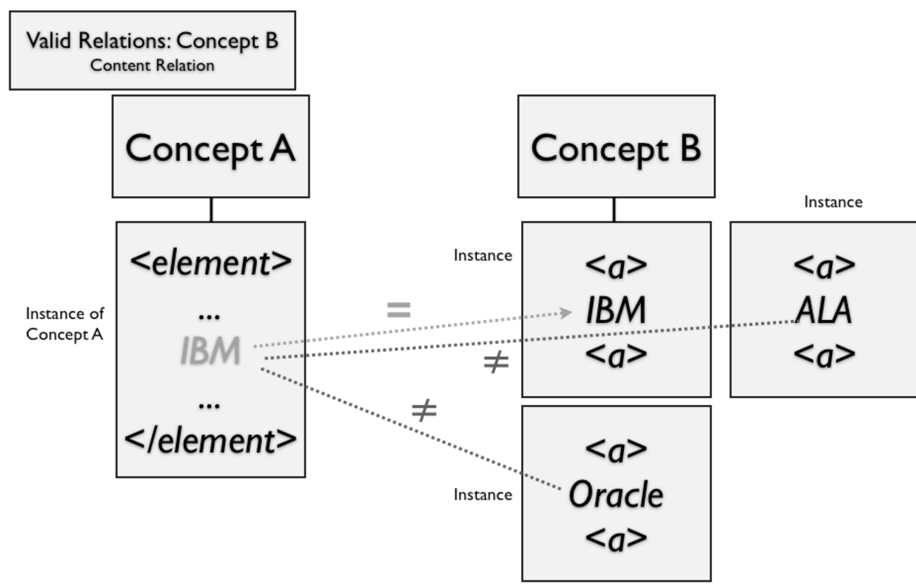

Figure 3.14 Content Relations

In Figure 3.14, Concept A allows relations with Concept B, which has three Instances. The rule for the *Content Relations*, of Concept A has retrieved the string “IBM” from the document, that string will be compared with the result of the rule for target Instances, applied to those same Instances. If any of them matches, the relation is created. In figure 3.14, a relation is found between one Instance of Concept B (the one with the “IBM” string inside it) and the original Instance of Concept A, being inserted. *Identifier Relations* and *Content Relations* are grouped in what the Repository considers *Automatic Relations*. *Automatic Relations* are useful to extract relations without human assistance and provide a mechanism to help users with metadata relationship.

A supported feature by the repository, to comply with the metadata update requirement, is versioning of Concepts. Each Concept has a version number (a positive integer), in a very similar way to Fragments; this means that each Instance version, is always linked (and locked) with its parent Concept version. There isn’t the notion of “last” Concept version, regarding the update of valid relation targets or to change the parent Concept of an Instance. If an Instance is inserted in the repository and it does not explicitly state which version of what Concept it belongs to, the repository will automatically choose the last version of that Concept and, if the Instance is valid, it becomes locked to that Concept version. The same happens if the target Concepts for a relation do not state which version of each Concept they will relate to. The reason for this choice is that Concept versions may be very different from the previous versions, as such, Instances of an older version may not be valid with the newer version or the rules to create automatic relations may not apply if the structure of a Concept changes significantly.

The Information Model, including Concepts and Fragments is depicted in Figure 3.15.

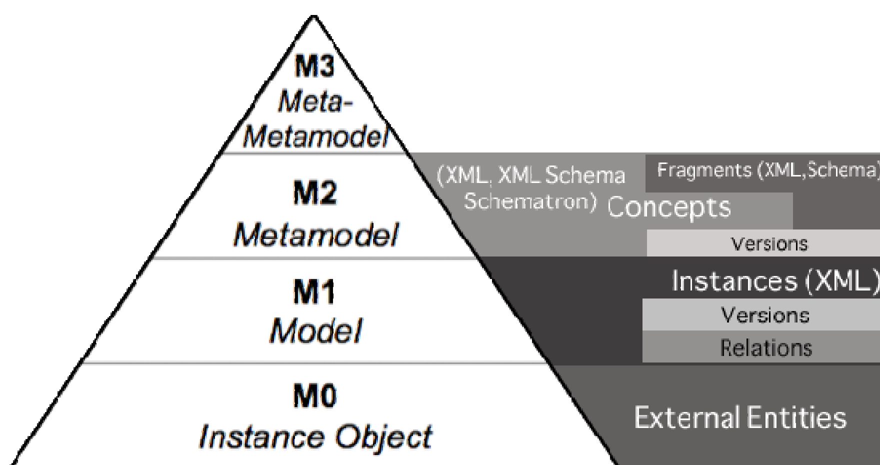

Figure 3.15 M2 Layer, with Concept and Fragment Versions

The Information Model, in Figure 3.15, features “versions” as a sub-feature of both Fragments and Concepts. Concepts, on another hand are on top of Fragments (because they can use them to define their XML Schema vocabulary), but can also exist independently.

3.7. M3 Layer – Meta-meta-model

The M3 layer defines a set of rules and properties that all Concepts must have. The M3 layer supplies a XML language for the definition of Concepts and Fragments that will be verified with XML Schema and Java code to assure several properties are valid; the XML language will be described in chapter four, with examples of usage. The XML Schema is used, at this level, to ensure that the definition of Concept/Fragment respects the vocabulary defined to create them, whereas the Java code is used to ensure that the values chosen for several of the properties (identifier of the Concepts, identification of Instances, targets for relations, rules for automatic creation of relations or for automatic creation of metadata) are valid. This layer is very important as it establishes the basis for the storage model, validation of Concepts, Fragments and Instances as will be described in chapter four. The full diagram of the Information Model is depicted in Figure 3.16.

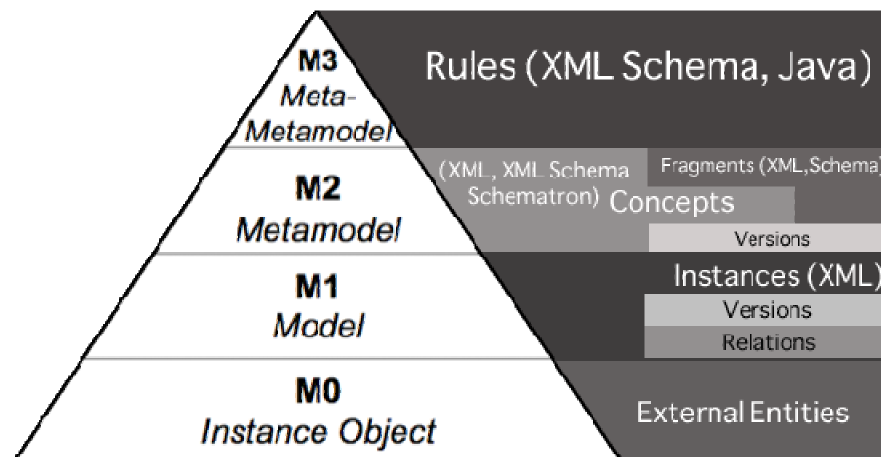

Figure 3.16 Full Information Model, with M3 Layer

3.7.1. Evaluation

The Information Model based on MOF allows a clear distinction between abstraction levels and a special attention is devoted to models and meta-models. In this Information Model, Models are denoted as Instances, metamodels are Concepts. For a metadata repository implementing this Information Model, these are the most important components as they are responsible for metadata itself and its validation, as well as being the ones the repository will store and manage. XML technologies were chosen over Semantic Web technologies, because of their stability, availability and user familiarity (including the author's personal experience) as well as the other reasons stated in chapter two.

Chapter 4
Functional Design

This chapter presents and discusses the various metadata repository features and functionalities, such as validation, integrity, querying and transformation

4.1 Metadata Repository Identifiers (MRI)........ 56
4.2 Concept & Fragment Definition........ 57
4.3 Metadata Validation and Integrity........ 76
4.4 System Concepts and Instances........ 80
4.5 Metadata Querying and Transforming........ 80

This chapter presents the design of several of the functionalities presented as requirements, in chapter three, as well as other functional properties of the repository. This includes the definition of a Metadata Repository Identifier (MRI), which is the base for several of the features of the repository. Validation of Fragments, Concepts and Instances is also detailed, as well as a comprehensive description (with usage examples) of the Concept & Fragment definition language. Metadata Querying and Transforming mechanisms are also explained. A subchapter is dedicated to meta-metadata (metadata about the metadata in the repository) and searching.

4.1. Metadata Repository Identifiers (MRI)

Each item of the Information Model (except the M3 layer) has a public Identifier in the repository, which is defined as its Metadata Repository Identifier (MRI). This means that every Fragment, Concept and Instance has its own unique MRI. The MRI is a URI-compatible identifier, meaning that every MRI is a URI, but not every URI is a MRI. Being able to have a standard syntax enables for example the usage of standard libraries to process URIs and enables foreign applications that deal with URIs (namely, Semantic Web technologies, RDF, RDFS) to use the information of the Metadata Repository as input. The abstract syntax of a MRI is the following:

mdr://namespace/Name[&Version]/InstanceName[&InstanceVersion][?Query]

The first part of the MRI is the URI-equivalent of "scheme" and the *mdr* prefix is used. This was chosen in order to have the possibility to distinguish normal URIs from MRIs. That feature will allow the creation of relations based on identifiers found in the content of Instances, as described in chapter 3.6.

The MRI **namespace** item is equivalent to the URI "authority", but only using the host component, not including the "userinfo" or "port" parts of the URI "authority" item.

The **Name** item, is a string that is equivalent to a "path-segment" in URIs (excluding the use of "." and ".."). The concatenation of the **prefix**, with the **namespace** and the **Name**, creates an identifier of a Concept, or a Fragment. As an example, if for the namespace the string "*www.di.fct.unl.pt*" is chosen, and the **Name** for the Concept is "DataModel", the MRI of this Concept would be:

mdr://www.di.fct.unl.pt/DataModel

A Concept and a Fragment cannot have an equal MRI identifier, although there's no risk of collision due to the fact that Fragment and Concept MRIs are used in different contexts, but because the MRI is a unique identifier, it's not possible for Fragments and Concepts to share the same MRI. A Concept/Fragment MRI may optionally have a **Version** item and, as the name implies, it's a positive integer that explicitly specifies the version of the Concept/Fragment.

Using the previous MRI example, version two of the "DataModel" Concept would have the following MRI.

mdr://www.di.fct.unl.pt/DataModel&2

4.1.1. Instance Identification

The identifier of an Instance is a MRI based on the MRI of its parent Concept, requiring the use of another "path-segment" that identifies the Instance (this is the **InstanceName** item of the MRI syntax). As an example, using the previous Concept MRI, if the Instance would have the string "SRTA"[1] as its name, the final MRI identifying that Instance would be:

*mdr://www.di.fct.unl.pt/DataModel&2/**SRTA***

Instances have versions and when one needs to identify a particular Instance version, the **InstanceVersion** item of the MRI syntax is used (again, a positive integer), so the MRI of version one of the previous example is like the following:

*mdr://www.di.fct.unl.pt/DataModel&2/SRTA**&1***

The last item in the MRI syntax is the **Query**. This element consists of a XPath that can locate a part of the content of an Instance; not every XPath can be used, because since we're aiming to maintain URI syntax compatibility, only a XPath that does not break that compatibility can be used. This item enables to create relations with parts of Instances and if the Instance with the previous MRI would have a root element named "srta" and a child element with name "identifier", a MRI could be built like this:

*mdr://www.di.fct.un.pt/DataModel&2/SRTA&1**?srta/identifier/text()***

That MRI identifies the Instance and points to a specific part of its content. The use of both **Version** and **InstanceVersion** is required, but a "virtual" MRI of an Instance without these items can be supplied to the repository, which in turn will look for the last version of each of those items and assign their values to the MRI, completing it. This is the mechanism that allows referring to the last version of a given Instance, Concept or Fragment.

4.2. Concept & Fragment Definition

Concepts and Fragments are part of the M2 layer of the Information Model, as such they define the structure of Instances in the M1 layer and, since XML was chosen as the mechanism to represent

[1] SRTA is an acronym for *Sistema de Recepção de Trabalhos dos Alunos*, a project developed by the author of this thesis.

Instances, XML Schema was the choice to define a vocabulary. However, Concepts and Fragments have several properties besides defining the vocabulary for Instances. To deal with this situation a XML language was specially created that allows making the specification of the vocabulary for a Concept/Fragment, as well other features such as Schematron validations, relations, etc. This section will present the XML language and provide examples.

The XML language for defining Fragments includes a sub-set of the language used to define Concepts and, as such, will be presented first; the parts of the language that are common will be noted.

4.2.1. Fragment Definition Language

A Fragment, as described in chapter three, in the Information Model, is a reusable item that defines a XML Schema structure, along with XSLT and metadata about itself. A Fragment definition also includes the values required to build its identifier. The main structure of a Fragment is depicted in Figure 4.1.

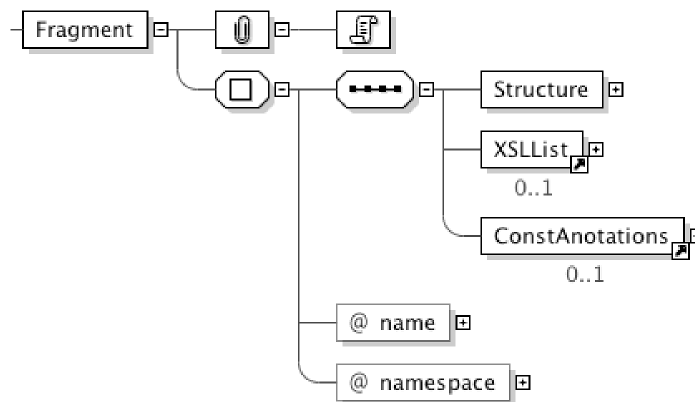

Figure 4.1 Fragment main structure

A Fragment is defined with a root element named *Fragment* and has three child elements; a *Structure* element (where the XML Schema vocabulary will be defined), a *XSLList* element, which is where XSLT templates can be added, a *ConstAnotations* element where metadata about the Fragment can be stored. Of all these elements, only the *Structure* element is required. The Fragment definition is required to have two attributes, the *namespace* attribute and the *name* attribute. These two attributes will be used to build the Fragment's MRI. The *namespace* attribute is equivalent to the "namespace" element in the MRI syntax, seen in the previous subchapter, as well as the *name* attribute, which is equivalent to the "Name" element in the MRI syntax. With those two attributes, the MRI of the Fragment is built as *mdr://namespace/name*. An example XML definition of a Fragment is depicted in Figure 4.2.

```
<Fragment name="Name" namespace="example.namespace.com">

    <Structure> [36 lines]

    <XSList> [2 lines]

    <ConstAnotations> [2 lines]

</Fragment>
```

Figure 4.2 Fragment definition

The structure of the XML vocabulary can be defined using one of two methods:

- Embedded XML Schema (Embedded Mode)
- Composition of existing Fragments (Composition Mode)

Embedding XML Schema code in the Fragment definition is a simple way to reuse a piece of already existing XML Schema element (for example, a file that is included by other schemas) so that other Fragments/Concepts can reuse it. There are no restrictions to the type of XML Schema used in Embedded Mode, so even includes and imports can be used, as is depicted in Figure 4.3 (the embedded schema is placed inside the *GlobalEmbeddedSchema* element, a child of the *Structure* element).

```
<Structure>
    <GlobalEmbeddedSchema>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.sapo.pt" xmlns:ex="http://www.sapo.pt/">

            <xs:include schemaLocation="IncludeA1.xsd"></xs:include>
            <xs:import schemaLocation="ImportA1.xsd"
                namespace="http://www.sapo.pt/"></xs:import>
            <xs:element name="table"> [11 lines]
            <xs:element name="fields"> [6 lines]
            <xs:element name="field"> [7 lines]

        </xs:schema>
    </GlobalEmbeddedSchema>
</Structure>
```

Figure 4.3 Embedded XML Schema in a Fragment definition

The second, and last, method for defining the XML Structure is the *Composition Mode*. The *Composition Mode* is a way of composing (hence, the name) a XML Schema using already existing parts. The result of a composition is always a XML Schema element (complex type) with a certain name, which holds a sequence/choice of XML Schema elements (that element can also have attributes, which may be references to already existing attributes). A Composition is created as child element of the *Structure* element with the name *GlobalComposition*. It has an attribute *rootElement*,

which is required, that it will wrap its content inside a XML element with the name of the value of *rootElement*. If, for example, the *GlobalComposition* element has the *rootElement* attribute with value “Database” the resulting XML Schema, is depicted in Figure 4.4.

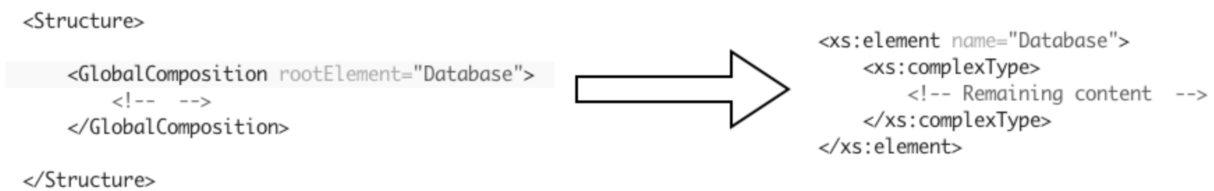

Figure 4.4 *GlobalComposition* element converted to XML Schema

A *GlobalComposition* element can optionally have two attributes, the *targetNamespace* and the *targetNamespacePrefix* attribute. These two attributes make it possible for the resulting XML schema of the composition, to be associated with a target namespace (as every XML schema document can); the use of these attributes will make the repository know that this Fragment has a target namespace and, as such, when other Fragment wants to reuse this Fragment the repository will make the appropriate import (if these attributes are not used, then it’ll make an include) when converting that Fragment’s structure to XML Schema. Since a composition will be converted to a XML Schema element, there’s the possibility to add XML Schema attributes to that element; an attribute for a *GlobalComposition* can be reused from other Fragments, or be created in the definition using actual XML Schema code, as seen in Figure 4.5.

```
<GlobalComposition rootElement="Calibration"
    targetNamespace="http://sess.uninova.pt" targetNamespacePrefix="pt">

    <Sequence> [71 lines]

    <Attributes>
        <!-- Attribute created using XML Schema code -->
  A     <xs:attribute name="id" type="xs:string"></xs:attribute>

        <!-- Attribute that is a reference to an already
             existing attribute GROUP of a Fragment -->
        <attributeFragGroup
  B             refFrag="mdr://example.fct.unl.pt/baseMdr"
                name="identifierAttributesGroup"></attributeFragGroup>

        <!-- Attribute that is a reference to an already existing
             SINGLE attribute of a Fragment -->
        <attributeFrag
  C             refFrag="mdr://example.fct.unl.pt/baseMdr"
                name="locationAttribute"/>

     </Attributes>

</GlobalComposition>
```

Figure 4.5 Global Composition with Attributes

In the previous figure, there are three different ways of adding attributes to the *GlobalComposition* element; this is done creating an *Attributes* child element and defining the attributes as children of that node. One way is to use XML Schema to define an attribute, as seen in Figure 4.5, situation "A", this will be converted into a XML Schema attribute when the Fragment generates its structure's XML Schema representation. If there's a Fragment that provides a set of reusable attributes, they can be referenced with an *attributeFrag* or *attributeFragGroup* element (referring to an individual attribute or an attribute group, respectively) using the *refFrag* attribute to indicate the MRI of the Fragment, and the *name* attribute to select which attribute (or attribute group) from that Fragment should be used (as seen in Figure 4.5, situation "B" and "C").

To build the content of the *GlobalComposition* element, there are two possible child elements, the *Sequence* element and the *Choice* element. Each of them having the same meaning as their XML Schema counterparts, the *Sequence* defines a sequence of elements and the *Choice* a choice between a set of elements. Each of these elements can hold additional *Sequence* and *Choice* elements inside them (as do their XML Schema counterparts) or they can hold a *Composition* element or a *Schema* element. The *Composition* element behaves exactly the same as the *GlobalComposition*, but only happens inside another Composition (it also has the same structure). The *Schema* element is the element that enables the use of already existing Fragments, or to make local use of XML Schema. In Figure 4.6 is depicted the structure of a *Sequence* element (which is the same for the *Choice* element).

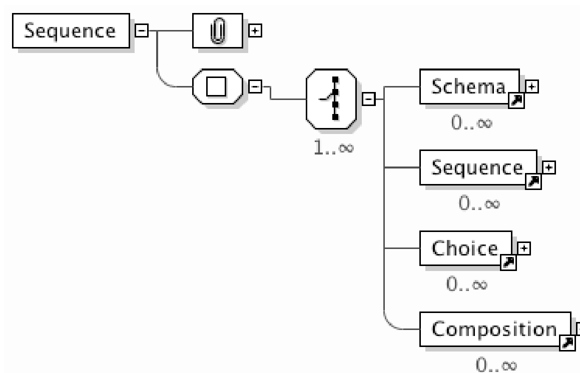

Figure 4.6 Structure of a *Sequence* element

A Schema element can reference an already existing Fragment, or make a local inclusion of XML Schema code and use **one** XML schema element (or group) of that Fragment/code. The structure of the Schema element is depicted in the Figure 4.7.

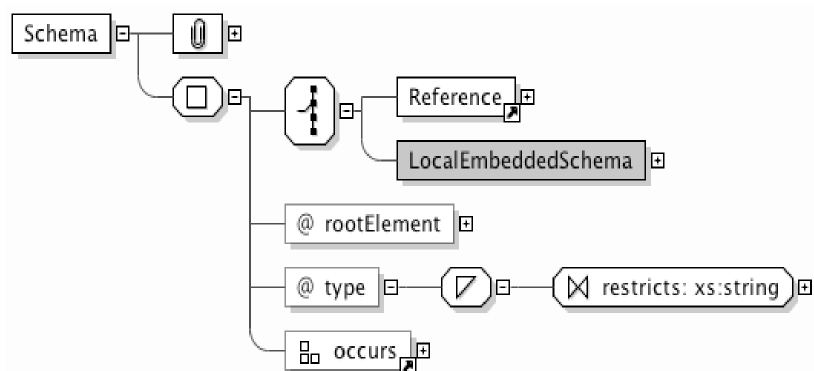

Figure 4.7 Structure of the *Schema* element

If the user defining the Fragment structure whishes to use one element from an existing Fragment then it must do the following: Create the *Reference* element under the *Schema* element and fill the content with a valid MRI of an existing Fragment, then it must use the *rootElement* attribute and fill its value with the name of an element that exists in the target Fragment and set the *type* attribute to "element"; if the user wants to use a XML Schema group element, then it must set the name of the group and set the *type* attribute to "group". The *occurs* attribute group is the same found in XML Schema, and encloses the *minOccurs* and *maxOccurs* attribute, which control the cardinality of that element in a Sequence/Choice. Figure 4.8 depicts the correspondence between a Fragment definition that makes references to other Fragments, using the previously described *GlobalComposition* element and the final XML schema that is generated by the repository from the definition. In light blue, in the definition, the *rootElement* attribute value, will be the name of the element in the XML Schema; since the definition uses the *targetNamespace* (green) attribute, the final XML schema has the target namespace declaration also. The use of the *Sequence* element in the definition (purple) generates a XML Schema sequence element, and the *Schema* elements in the definition refers to group elements in a Fragment with MRI – mdr://lol.fct.unl.pt/baseMdr, which in turn are converted into group references in the XML schema. It can be concluded that the Fragment that is referenced has a target namespace because the repository automatically inserted the *import* XML Schema element with the corresponding namespace and the references to the group element have the "mdr:" prefix (which is the prefix that Fragment uses for its target namespace)

```
<GlobalComposition rootElement="Calibration"
    targetNamespace="http://sess.uninova.pt" targetNamespacePrefix="pt">

    <Sequence>
        <Schema rootElement="identificationElementsGroup" type="group">
            <Reference>mdr://lol.fct.unl.pt/baseMdr</Reference>
        </Schema>
        <Schema rootElement="documentationElementsGroup" type="group">
            <Reference>mdr://lol.fct.unl.pt/baseMdr</Reference>
        </Schema>

    </Sequence>

    <Attributes> [16 lines]

</GlobalComposition>
```

Fragment definition

```
<xs:schema xmlns:mdr="http://di.fct.unl.pt/MetadataRepository"
           attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="http://sess.uninova.pt">

    <xs:import namespace="http://di.fct.unl.pt/MetadataRepository"
               schemaLocation="base.xsd"/>

    <xs:element name="Calibration">
        <xs:complexType>
            <xs:sequence>
                <xs:group ref="mdr:identificationElementsGroup"/>
                <xs:group ref="mdr:documentationElementsGroup"/>
            </xs:sequence>
            <xs:attributeGroup ref="mdr:identifierAttributesGroup"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

XML Schema

Figure 4.8 Correspondence of a Fragment definition and XML Schema

It's possible to make local use of XML Schema, in the reference element, when there is no Fragment that has the needed structure. Instead of using the *Reference* element, as a child of the *Schema* element, a *LocalEmbeddedSchema* element can be used. As the content of that element, XML Schema code can be used, although some restrictions apply, such as the impossibility to use import/includes or to use target namespaces. Figure 4.9 provides an example of this situation.

```
<Schema rootElement="calibGroup" type="group">
    <LocalEmbeddedSchema>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:mdr="http://di.fct.unl.pt/MetadataRepository">
            <xs:group name="calibGroup">
                <xs:sequence>
                    <xs:element name="CalibrationType"> [10 lines]
                    <xs:element name="CalibratedExpressionDescription"
                    <xs:element name="CalibrationValues"> [36 lines]
                </xs:sequence>
            </xs:group>
        </xs:schema>
    </LocalEmbeddedSchema>
</Schema>
```

Figure 4.9 Use of local embedded XML schema in a Fragment definition

A Fragment can have metadata about itself embedded in its definition in the form of Key-Value pairs. If a description of a Fragment is desired, a pair of Key (with the content, "Description") and Value (with the content of the description) may be created. In the Fragment definition, these pairs are created under the *ConstAnnotations* element (a child of the root *Fragment* element) as elements of name *Pair*, which have two children: An element *Key* and an element *Value*. Figure 4.10 depicts the use of these elements.

```
<ConstAnotations>
    <Pair>
        <Key> Name </Key>
        <Value> Calibration Fragment </Value>
    </Pair>
    <Pair>
        <Key> Description </Key>
        <Value> Definition of a Calibration set of values </Value>
    </Pair>

</ConstAnotations>
```

Figure 4.10 Use of Constant Annotation in Fragment definition

The last item in a Fragment definition is *XSLList*. Fragment can have XSL templates associated to them (for Concepts to reuse, which will be described further in subchapter 4.5) and that association is done through this element. The *XSLList* element can have a sequence of *XSL* elements with each of them having the structure depicted in Figure 4.11.

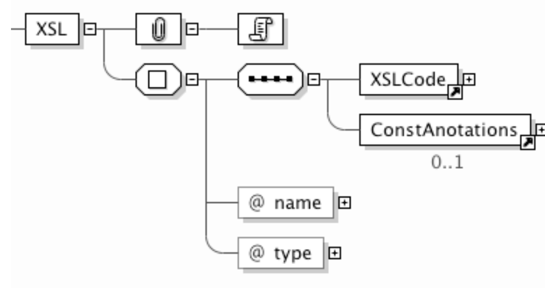

Figure 4.11 Structure of a XSL element

Each element must have a *name* attribute and a *type* attribute, the reason for this will be explained in chapter 4.5, but essentially has to do with reusability. The *name* attribute is an identifier for the XSL inside the Fragment and the *type* attribute defines a category for the XSLT templates. The *ConstAnnotations* element is a reuse of the previously described element, and is used to add metadata about the specific template. Finally, the *XSLCode* element can hold XSLT code, but that code must follow a set of rules. The design of every *xsl:template* element must be made bearing in mind they are designed to process elements of that Fragment and since Fragments can be reused in a composition by other Fragments or Concepts, the XSLT templates cannot be bound to a specific structure because in each of the Fragments/Concepts that reuse this Fragment may be reused in different locations, therefore no template can have a *match* attribute that starts with '/' or with '//' (i.e the *match* attribute must always be relative). In Figure 4.12 an example of the *XSLCode* element use is depicted (featuring real XSLT code to generate HTML).

```
<XSLCode>
 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:template match="/mylist">
   <html>
    <head>
     <title>My contact list </title>
    </head>
    <body>
     <h1>Here is my contact list </h1>
     <p>
      <xsl:apply-templates/>
     </p>
    </body>
   </html>
  </xsl:template>
  <xsl:template match="contact">
   <b><xsl:value-of select="name"/></b>
   Email is: <xsl:value-of select="email"/>
  </xsl:template>
 </xsl:stylesheet>
</XSLCode>
```

Figure 4.12 XSL code used in element XSL

The use of XSLT templates in Fragment will be further explained in chapter 4.5, where Generic Transforms are introduced.

4.2.2. Concept Definition Language

A Concept is the item where the definition of the structure of XML files in the repository (known as Instances) is declared, as well as several other properties regarding features provided by the repository. A Concept can define the following list of properties (bold items are required).

- Instance Identification
- **Instance Structure**
- Schematron validations
- XSLT
- Relations
- Metadata about the Concept
- Metadata about the Concept's Instances

The Concept's definition language has a root element *Concept* that has two **required** attributes, the *name* attribute and the *namespace* attribute, which have the same meaning as in the Fragment definition; to identify the Concept and are used to build the MRI of the Concept. All of the properties of the Concept are defined with child elements of the *Concept* element. In Figure 4.13 it's depicted the structure of a Concept.

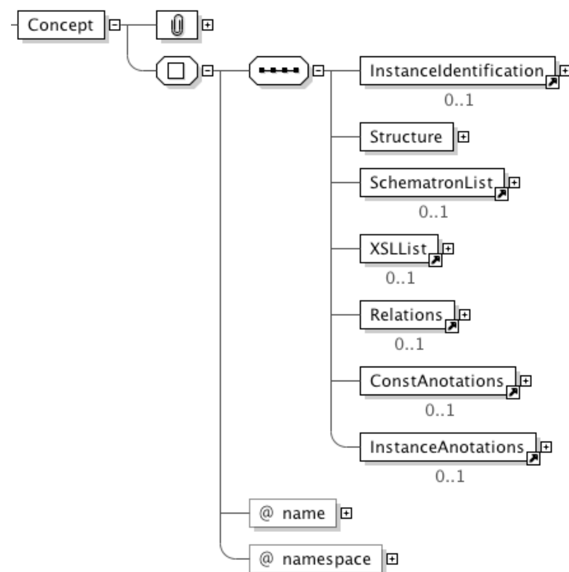

Figure 4.13 Generic Structure of a Concept

Each of the elements in the previous figure allows defining the properties listed before, and will be described in the following paragraphs.

Instance Identification

In chapter 4.1, the Metadata Repository Identifier syntax featured an item named “InstanceName”. That item is the identifier of that Instance, among the other Instances of the same Concept, i.e. every Instance of that Concept has a different “InstanceName” component in their MRI. There are two ways of retrieving that identifier: one is using a XPath that retrieves a small part of the content of the Instance and uses it as the “InstanceName”, the other is letting the Concept generate a unique identifier for each Instance (using a sequential counter). For example, if a particular Instance described a certain automobile brand, the name of the brand could be used as an identifier (as there are no two brands with the same name). The *InstanceIdentification* element allows users to define a XPath that will be applied to Instances to retrieve a name/expression, that will be used as an identifier. Absence of this element in a Concept definition means that the repository, using the sequential counter, will generate identifiers for Instances automatically. The XML syntax is depicted in Figure 4.14.

```
<InstanceIdentification>
    <Xpath>/Calibration/mdr:Name/text()</Xpath>
</InstanceIdentification>
```

Figure 4.14 XML syntax for Instance Identification

The XPath can contain namespace prefixes and the namespace mapping only needs to be declared on the Instance where this XPath will be evaluated, so that the XPath can be correctly executed and return the expected value. However, if the Instance document makes use of a default namespace, any XPath that needs to access elements in the default namespace will fail if there's no mapping to that namespace, hence the *InstanceIdentification* element supports a namespace binding for such situations with the syntax depicted in Figure 4.15.

```
<InstanceIdentification>
    <Xpath>/ex:Calibration/mdr:Name/text()</Xpath>
    <NamespaceBinding prefix="ex" uri="http://sess.uninova.pt"></NamespaceBinding>
</InstanceIdentification>
```

Figure 4.15 Instance Identification element with Namespace binding

The *NamespaceBinding* element has two attributes, a *prefix* attribute and a *uri* attribute, which are the components of any namespace (a prefix and a URI to map that prefix) and, as such, will be used to declare a namespace for the XPath execution. The XPath must return a **single** text value, and that value may not contain characters that are invalid in a URI, because the retrieved value will be part of the MRI of the Instance and, as described earlier, MRIs are URI-compatible.

Instance Structure

A Concept is the only entity that can have Instances and it dictates the XML structure of those Instances. The XML language to define the structure of Instances is a super-set of the one used by Fragments to define their structure. The only difference in the language is that there's a third way of defining the structure (besides the Composition and the Embedded Schema ones) of Instances. The third method is called "Reference" and it allows choosing the structure of Instances reusing the structure already defined by a single Fragment and the syntax is depicted in Figure 4.16.

```
<InstanceStructure>
    <Reference>mdr://sess.uninova.pt/FragSpaceCraft</Reference>
</InstanceStructure>
```

Figure 4.16 Concept structure definition referencing a Fragment

The *Reference* element content is a single MRI of an existing Fragment in the repository, if the Fragment version is not specified the repository will find the last version and lock the Concept's structure to that version.

Schematron Validations

Schematron is a very useful technology to make validations over XML documents, because it allows verifying certain restrictions that XML Schema cannot guarantee. A Concept can also have Schematron applied to Instances to assure extra validations, creating a child of the root element named *SchematronList*. A *SchematronList* element can have a sequence of children named *Schematron*, which in turn will contain the Schematron code, as depicted in Figure 4.17.

```
<SchematronList>
    <Schematron> [22 lines]
    <Schematron> [2 lines]
    <Schematron> [2 lines]
</SchematronList>
```

Figure 4.17 Schematron list syntax

The structure of a Schematron element is depicted in Figure 4.18.

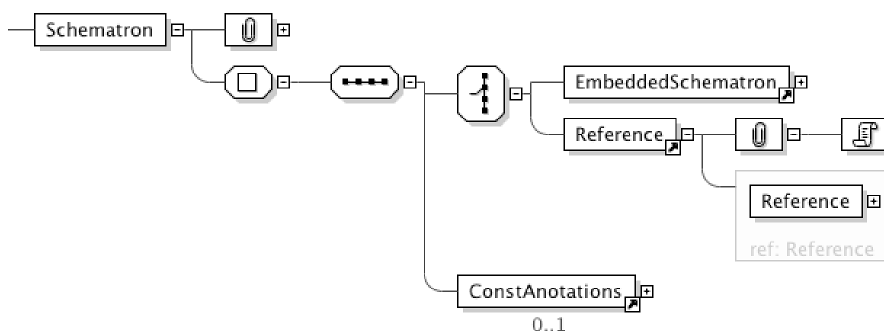

Figure 4.18 Structure of the Schematron element

A *Schematron* element, may contain actual schematron code in the content of the *EmbeddedSchematron* element or it can contain the MRI of a Schematron “System Instance” as the content of the *Reference* element. Each *Schematron* may also have metadata about itself using the already described *ConstAnnotations* element. The repository supports a notion of “System Instances” for some kinds of “special” Instances that can be used for specific purposes. One kind of those “System Instances” are Schematron documents, which can be reused by Concepts to assure certain restrictions without having to include in every Concept the embedded schematron code. System Instances will be described in 4.5. As an example, in Figure 4.20 and Figure 4.19 is depicted the use of both *Reference* and elements *EmbeddedSchematron*, respectively.

```
<Schematron>
    <Reference>mdr://system.di.fct.unl.pt/Schematron/addressing</Reference>
</Schematron>
```

Figure 4.20 Schematron element with reference

```
<Schematron>
   <EmbeddedSchematron>
        <sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
            <sch:pattern name="Test of element">
                <sch:rule context="/SCParameter">
                    <sch:assert test="@version">The version is not set</sch:assert>
                </sch:rule>
            </sch:pattern>
        </sch:schema>
   </EmbeddedSchematron>
```

Figure 4.19 Schematron element with embedded Schematron code

XSLT

Instances in the repository are XML files that obey the structure defined by their Concept and XSLT is one of the most popular technologies to perform transformations over XML. As such, it makes sense to associate XSLTs that can process Instances of that Concept, or to reuse already existing XSLTs (in the form of "System Instances") to make that processing. The *XSLList* element provides the means to associate XSLTs to Concepts. It has a similar structure to the *SchematronList* element, having multiple *XSL* child elements where the XSLT can be defined (using embedded XSLT code, or a reference to an already existing XSLT), and define annotations for each XSLT, using the already known *ConstAnnotations* element. Since several XSLTs can be associated to a Concept, every XSLT has a *name* attribute, that is required and will be used to invoke the XSLT for processing. Figure 4.21 depicts the use of the *XSLList* element.

```
<XSLList>
    <XSL name="Html">
        <Structure>
            <EmbeddedXSL>
            <xsl:template match="/"> [9 lines]
            <xsl:template match="/Baltimore"> [2 lines]
            <xsl:template match="/Baltimore/Understress"> [2 lines]
            <xsl:template match="mylist"> [6 lines]
            </EmbeddedXSL>
        </Structure>
        <ConstAnnotation>
            <Pair>
                <Key>Description</Key>
                <Value>This XSLT displays the content in HTML</Value>
            </Pair>
        </ConstAnnotation>
    </XSL>

    <XSL name="Javascript"> [4 lines]

</XSLList>
```

Figure 4.21 *XSLList* element syntax and usage

Relations

Instances can have relations with other Instances and the definition of which Instances can relate to another is done at the Concept level, using the XML definition language. Each Concept can define a set of allowed relations, choosing the valid targets, the cardinality of the relations and can create rules for automatic relations between Instances. This is done using the *Relations* element, which can have a set of *Relation* child elements where each relation will actually be specified.

```
<Relations>
   <Relation>
       <Targets> [4 lines]
       <Cardinality max="1"/>
       <AutoRelContent> [16 lines]
       <AutoRelMRI> [2 lines]

   </Relation>
    <Relation> [2 lines]
    <Relation> [2 lines]
```

Figure 4.22 Syntax of the Relations element

Each relation has the following properties (each property translates to a child element of *Relation*)

- **Targets (required)**
- Cardinality (optional)
- Automatic Relations through Content (optional)
- Automatic Relations through MRI (optional)

The general syntax of a *Relations* element is depicted in Figure 4.22.

The *Targets* element is where the valid Concept targets are defined, the presence of a Concept in the list of valid targets means that an Instance of the Concept where the relation is being defined, can relate to that Concept. Relations have a white-list approach; Instances of a Concept can only relate to a Concept that is explicitly declared as a valid target. For the definition of the valid targets a list of MRIs can be enumerated, like in Figure 4.23.

```
<Relation>
    <Targets>
        <Enumeration>
            <target>mdr://sess.uninova.pt/Spacecraft</target>
            <target>mdr://sess.uninova.pt/Spaceweather</target>
            <target>mdr://sess.uninova.pt/Calibration</target>
        </Enumeration>
    </Targets>
```

Figure 4.23 List of valid target Concepts for a relation

The *Targets* element can also have a child element named *XQuery*, which selects a number of Concepts based on the result of the XQuery.

The *Cardinality* element allows controlling the cardinality of the relation by establishing a minimum and maximum number of relations with Instances of those Concepts declared in the *Targets* element. If the cardinality is set with a maximum value of one, any Instance can only have one relation with an Instance of each of the valid target Concepts. The syntax for the Cardinality element is depicted Figure 4.24.

```
<Cardinality min="1" max="2" />
```

Figure 4.24 Syntax of the Cardinality element

Both attributes are optional and their absence means that there is no maximum or minimum cardinality. If the *max* attribute is used, then the maximum cardinality restriction comes in effect, the same is true if the *min* attribute is used (for the minimum cardinality).

For each Relation defined, “automatic” relations based on the content of Instances can be configured, as stated in chapter 3.3. These automatic relations based on content rely on XPath to find the content in two Instances, that content will be compared and if it’s equal, a relation is created

without user intervention. There are also automatic relations based on MRIs found inside the content of Instances, which is an approach useful in a context where the production of metadata in an organization knows the repository and can take advantage of this feature; for situations where external metadata must be imported from a context that does not contemplate the repository, the content based relations are more appropriate. The first kind of automatic relation to be described is the content based one. This kind of relations, has five properties (that are converted to child elements of the *AutoRelContent* element) that can be specified:

- **Target Match** (required)
- **LocalInstanceXPath** (required)
- **RemoteInstanceXPath** (required)
- **Behavior** (required)
- **Annotations** (optional)

The automatic content relation is declared using the *AutoRelContent* element (several can be declared for each relation) and each of the previously mentioned properties represent a child element of that element, as depicted Figure 4.25.

```
<Relations>
   <Relation>
       <Targets> [4 lines]
       <Cardinality max="1"/>
       <AutoRelContent>
           <TargetMatch> [4 lines]
           <LocalInstanceXPath> [2 lines]
           <RemoteInstanceXPath> [3 lines]
           <Behavior> [2 lines]
           <InstanceAnotations> [5 lines]

       </AutoRelContent>
```

Figure 4.25 Automatic Relation based on content syntax

```
<AutoRelContent>
    <TargetMatch>
        <List>
            <target>mdr://sess.uninova.pt/Spacecraft</target>
            <target>mdr://sess.uninova.pt/Calibration</target>
        </List>
    </TargetMatch>
```

Figure 4.26 Definition of targets for the automatic relation based on content

The *TargetMatch* element is an element that allows choosing the list of target Concepts with whom this Concept will have these automatic relations (the list of Concepts must be a sub-set of the list of targets defined in the *Targets* element of the relation). The list constitution is made by enumeration, as depicted Figure 4.26.

The *LocalInstanceXPath* element is where the XPath (XPath 2.0) to be applied to an Instance of the Concept being defined (hereafter referred to as "local Instance"), in search for content that is a key in other Concepts Instances. If the local Instance has a default namespace, the *LocalXPath* element has support for the definition of a namespace mapping, to enable the XPath to return results. The usage of the *LocalInstanceXPath* element is depicted in Figure 4.27 (using the optional namespace binding).

```
<LocalInstanceXPath>
    <Xpath>/pt:shiporder/pt:shipto/pt:country/text()</Xpath>
    <Namespace prefix="pt" uri="www.frat.com"/>
</LocalInstanceXPath>
```

Figure 4.27 Usage of the *LocalInstanceXPath* element

The *RemoteInstanceXPath* element is used to define the XPath (XPath 2.0) to be applied to Instances of the Concepts in the list of targets for the automatic relations. All Instances of each valid target Concept will have this XPath executed over their content and if the result is equal to the local Instance XPath result, the relation is created. The syntax is very similar to the *LocalInstanceXPath* element and is depicted in Figure 4.28.

```
<RemoteInstanceXPath>
    <Xpath>/ex:Calibration/mdr:Name/text()</Xpath>
    <Namespace prefix="ex" uri="http://sess.uninova.pt"/>
</RemoteInstanceXPath>
```

Figure 4.28 *RemoteInstanceXPath* element syntax

The behavior of these automatically created relations (when there's a change in the target Instance of the relation, the relation can behave in a predefined way) is controlled in the *Behavior* element, where one can choose if the relation is permanent, i.e. the relation is always maintained even if there's some change in the target Instance. This is done using the syntax in Figure 4.29.

```
<Behavior>
    <GenerateArc>maintain</GenerateArc>
</Behavior>
```

Figure 4.29 Syntax of the behavior of a relation

In case the user wants a dynamic relation, then he can choose the *Update* element and then choose if changes in the target Instance break the relation, or if changes in target Instance must be prevented (and that change blocked) or if the content on the original Instance, must be updated. The content of the original Instance can only be updated if the *LocalInstanceXPath* is such that it's possible to find the content it represents. With a XPath like "*/root/child/text()*" it's possible to find the elements position and update with the content that was changed in the target Instance and that was initially the responsible for creating the relation. Any other kind of XPath makes this situation impossible and the repository will not support the update of the local Instance. The choice of each behavior is done using the syntax in Figure 4.30 (choosing from one of three possibilities in the figure)

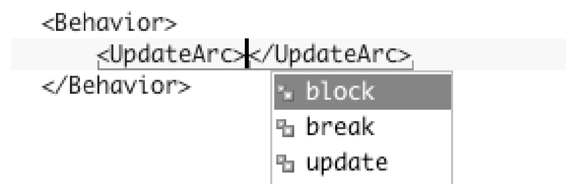

Figure 4.30 Syntax for the behavior (update) of a relation

Each automatic relation defined, can be configured to generate automatic annotations (metadata) about the relation based on constant values and/or content found on Instances. It's similar to the way annotations are defined for other elements, but since these can be based on content, now both *Key* and *Value* elements must have a *type* attribute, which can have the value "constant" or "xpath", meaning it's a constant value (a string) or a XPath to be applied over the Instance and return a value. These values will be stored in the repositories management structures, so that they can be queried. To perform these actions the *InstanceAnnotations* element is used, with the syntax depicted in Figure 4.31.

```
<InstanceAnotations>

    <Pair>
        <Key type="constant">Description</Key>
        <Value type="constant">Automatic Relation</Value>
    </Pair>
    <Pair>
        <Key type="constant">Semantic</Key>
        <Value type="xpath">/Spacecraft/SpaceAgencyRelation/mdr:Relation/data(@semantic)</Value>
    </Pair>
    <Pair>
        <Key type="xpath">/Spacecraft/SpaceAgencyRelation/mdr:Relation/data(@name)</Key>
        <Value type="xpath">/Spacecraft/SpaceAgencyRelation/mdr:Relation/data(@version)</Value>
    </Pair>

</InstanceAnotations>
```

Figure 4.31 Syntax for the creation of automatic annotations of relations

Automatic relations based on MRIs have an equal syntax, but don't require the use of the *LocalInstanceXPath* and *RemoteInstanceXPath* element. The element to define these relations is named *AutoRelMRI* and has the structure depicted Figure 4.32.

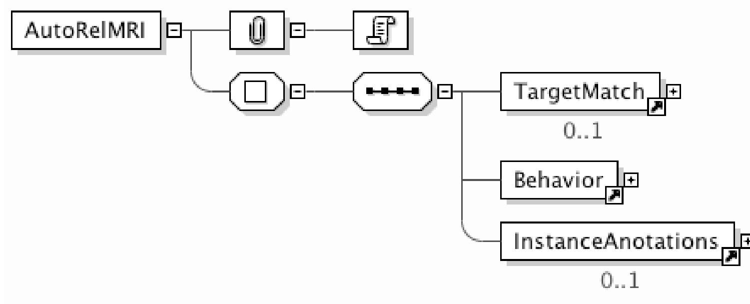

Figure 4.32 Structure of the AutoRelMRI element

The structure is equal to the *AutoRelContent* element, except in the elements dealing with XPath, as can be concluded by comparing with the structure of the *AutoRelContent* in Figure 4.33.

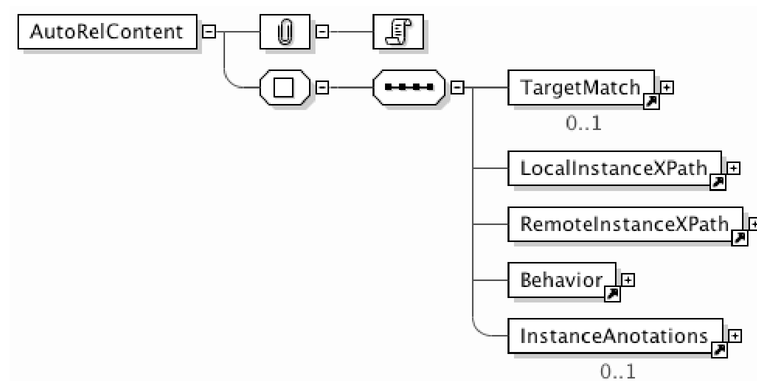

Figure 4.33 Structure of the AutoRelContent element

Metadata About the Concept

The Concept definition can contain metadata about the Concept itself, such as descriptions or any other kind of information. That metadata is declared with the previously described element *ConstAnnotations*, declared as child of the root element *Concept*; the metadata is in the form of key-value pairs and both of them have a content that's a static value. An example of the syntax is depicted in Figure 4.34, in which the annotations are used to add a name and a description to the Concept.

```
<ConstAnnotations>
    <Pair>
        <Key>Name</Key>
        <Value>Example Concept</Value>
    </Pair>
    <Pair>
        <Key>Description</Key>
        <Value>This is the description of this Concept</Value>
    </Pair>
</ConstAnnotations>
```

Figure 4.34 Syntax for constant annotations to the Concept

Metadata about the Concept's Instances

Whenever a new Instance of the Concept is inserted in the repository, metadata about that Instance can be created in an automatic way, using the *InstanceAnnotations* element. This element was already described in the automatic relations paragraph and it basically creates key/value pairs where the content of each key/value can be a static value, or the result of a XPath, in this case the XPath is also applied over the content of the Instance. For an example of the syntax, please refer to Figure 4.31.

For a full Fragment and Concept definition example, as well as the full XML Schema for the Fragment and Concept definition language, please check the annexes.

4.3. Metadata Validation and Integrity

Validation and Integrity are two of the major concerns in metadata management and, as such, they are present in both the metadata repository design and in most of its features. Validation is performed before insertion of any item and in case the validation fails, the insertion is aborted so that no invalid metadata is stored in the repository. Once the storage process is completed for any given item, the repository ensures that that item maintains its validity and integrity through time. This chapter describes the validation and integrity checks made upon Concepts, Fragments and Instances.

4.3.1. Fragment Validation & Integrity

As discussed in the metadata Information Model, the M2 layer (Concepts & Fragments) is subject to rules defined in the M3 layer and a Fragment is defined using a special XML language that imposes a set of restrictions, such as the use of a *namespace* attribute and a *name* attribute to build the identifier (MRI) of the Fragment. The first step in validating a Fragment is processing its XML definition, validating it against the XML Schema that defines the Fragment definition language. The next step is verifying if the value of the *namespace* attribute is a namespace present in the repository's list of valid namespaces and if the *name* provided is a unique name within that namespace (unless a new version of the Fragment is being created, or if the current one is to be

replaced). If the namespace exists and the name is different from other Fragments in that namespace, the repository will analyze the structure of the Fragment and check if all of the embedded XML Schema code is valid and if the referenced Fragments in the definition exist and the element inside them also exists. If there are included/imported schemas by the Fragment they are also checked to guarantee they are valid XML Schemas. If XSLT templates are declared in the definition of the Fragment, the repository will check if any template has a *match* attribute whose content has any of the forbidden values (as described in the Fragment definition language), if no problem is found the repository creates a management file with information about the Fragment and stores both of the files in their respective place (the storage model will be detailed in chapter five) and also updating the management file of every Fragment that is reused, this eases the process of removing/replacing a Fragment because if a Fragment is referenced by another, it cannot be removed/replaced until that relation is broken. This ensures integrity of Fragments, because if a Fragment A depends on Fragment B, B will never be removed.

4.3.2. Concept Validation & Integrity

Concepts are included in the M2 layer and, like Fragments, they are subject to the rules imposed by the definition language. Concepts must also declare a namespace and a name, that will be used to generate the Concept's MRI, this means that the namespace must be present in the list of namespaces of the repository and the name must be unique within that namespace (unless a new version is being created or the Concept is being replaced). The next step is verifying if there's a XPath declared to identify Instances and, if it is, the XPath is analyzed to see if it returns a single textual value as is required by the repository, in the case it does not conform with that requirement the insertion is aborted. Afterwards the structure of the Concept will be analyzed and if it only uses embedded XML schema, that schema will be checked against the XML Schema's schema; if it uses a composition of Fragment's structure and local embedded schema, the repository will check if the references to Fragments represent Fragments that are stored in the repository and will validate the embedded schema (if any of these steps fail, the insertion operation will be aborted). The Concept definition can include Schematron references (or embedded Schematron code) as well as XSLT references (or XLST code) and if any of these elements are present they will be validated against their respective XML Schemas. Validating the definition of relations includes verifying if all the chosen targets are Concepts stored in the repository, if the cardinality values are coherent (i.e. if the minimum value is smaller or equal than the maximum value) and if the XPaths declared in automatic relations are valid. All the rules for automatic creation of metadata about Instances that use XPath will also be validated. If all these steps are well succeeded, a management file is created and stored (along with the definition of the Concept) as well as a "compiled" XML Schema from the definition (i.e. a XML Schema is generated based on the structure declared in the definition language), to ease the validation of Instances. Every Fragment being used in the structure of the Concept will be updated to know this Concept is using it; this is used to ensure integrity, described in the next paragraph.

To ensure Concept integrity each of the Fragments reused in its structure cannot be removed; to remove such a Fragment, every Concept and Fragment that reuses its structure would have to be removed first. The same is true for the Concepts that are targets of relations they to cannot be removed and to remove a Concept it's required that it does not have any Instance (or that every Instance will be removed with the removal of the Concept, a situation only possible if the Concept's Instances do not have Instances of other Concepts related with them).

4.3.3. Instance Validation & Integrity

An Instance, to be stored in the repository must be validated by its Concept's XML Schema structure. The first step when storing an Instance is to check if its parent Concept exists (Instances must always provide the MRI of their parent Concept since there's no possible way to extract that information from the XML document representing the Instance) and if it does, a compiled XML Schema of the Concept's structure is generated (if it was never done before), featuring all the included/imported XML Schemas and every Fragment, and placed in a local directory to validate the Instance; if the parent Concept uses Schematron, the Instance is also validated against the Schematron file(s). Upon successful validation, a management file is created for the Instance and both of them are stored in the repository; if the parent Concept of the Instance declares automatic relations, the Instance's content will be scanned and the relations may be created if matching values are found (either MRIs or content). If a MRI is found in the content of an Instance but it does not exist in the repository, the relation will not be created. To deal with cyclic references the "batch add" method must be used (batch add, deals with cyclic references by first adding all Instances in a first step and scanning for all relations in a second step when all Instances are already stored).

4.3.4. Instance Relations, Creation & Validation

Relations between Instances can be created in an automatic way, or by a manual procedure. A relation that is created by user intervention (from now on, referred to as a *manual relation*) is a persistent relation, i.e. the relation is maintained until it's manually removed; even if the content of both the related Instances changes, or even their MRIs change, the relation is still valid. The other kind of relations is the automatic one, created in the act of Instance insertion (or if the definition of a Concept is updated with a new automatic relation and Instances are rescanned for relations). Every relation is kept in the management file of the Instance that relates to another, with several metadata about the relation, such as its behavior, the identifier of the target. On the other hand, every Instance that has an Instance that relates to it, has a reference on its management file as well, this enables to find out if any given Instance has Instances related to it and, as such, removal of an Instance can be prevented, preserving integrity of the relations. Automatic relations, however, may have distinct behaviors, as described earlier in chapter 4.2. There are four possible behaviors for automatically generated relations (in case the target Instance is updated/removed/replaced):

- **Generation of Arcs**

- **Block Update**
- **Break Relation**
- **Update content**

The **Generation of Arcs** behavior basically means that the relation is created and can only be removed by manual intervention (or by removing the origin of the relation). If an Instance A is related to Instance B with a "Generation of Arcs" behavior, Instance B cannot be removed, until the relation is broken manually. If Instance A would be removed, the relation would also be removed. This behavior is essentially the same as creating a manual relation.

The **Block Update** behavior is a way to assure that if there's an attempt to remove/replace an Instance that's the target of a relation, that attempt will be blocked. To update the target of such a relation, the relation must first be removed and only then the target Instance can be updated/removed.

The **Break Relation** behavior is a behavior that, in the situation where a target Instance of a relation is to be updated or removed, the update or remove operation can be executed and the relation will immediately be broken. If the user wants to recreate it, he will have to do it manually.

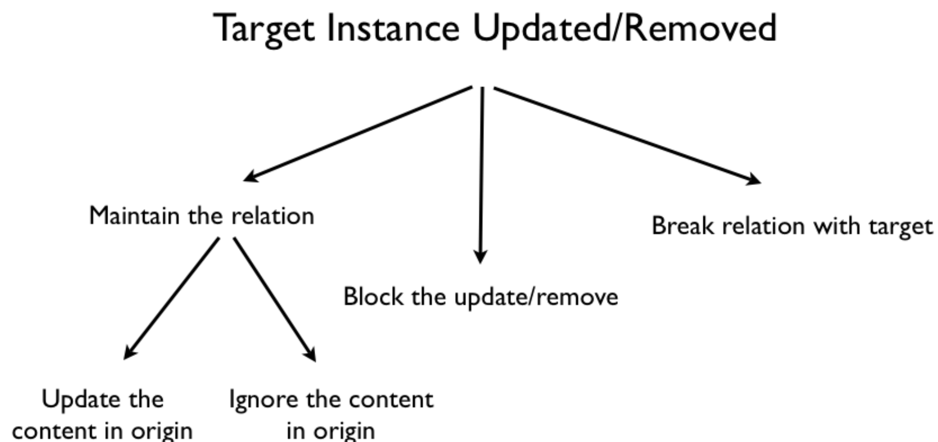

Figure 4.35 Behaviors of a relation in case of an update/removal of a target Instance

The **Update Content** behavior is a behavior that is not fully supported by the repository. This behavior would allow for an automatic update of the content of the Instance that is the origin of the relation. If a relation would be generated because the string "Scott" was present in Instance A and Instance B and at some point in time, Instance B would be updated and the "Scott" string is replaced with the "Tiger" string, the purpose of this behavior would be to replace the "Scott" in Instance A with "Tiger". Since the way to find these values is with XPath, this behavior can only be assured (and it's not encouraged) if the XPaths used are extremely simple to analyze and from them a path can be determined, so that an update can be made on the Instance.

Figure 4.35 summarizes the behaviors of a relation, in case there's a change in a target Instance (update or remove).

4.4. System Concepts and Instances

The Metadata Repository needs to deal with additional metadata information to control its operations and functionalities. Instead of using specific internal structures to deal with management information for functionalities such as querying, transforming or validations the repository's storage model is used. The flexibility provided by the Information Model, combined with the features provided by the metadata validation and integration mechanisms allows saving this kind of technical metadata in the repository as **System Concepts** and **System Instances** that are stored in the repository and each MRI uses the "*system.di.fct.unl.pt*" namespace as a way of separating these Concepts and Instances from regular ones. These are further presented in section 4.5.

4.5. Metadata Querying and Transforming

Metadata querying allows the execution of XQuery expressions over the content of metadata stored in the database (Instances and Concepts), allowing to retrieve these resources, or part of them, using XPath and control structures to select the desired content and using the richness of

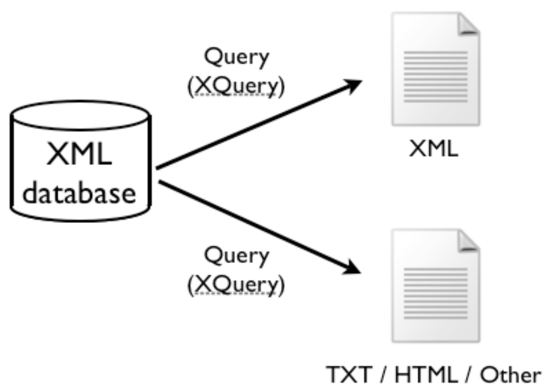

Figure 4.36 Metadata querying output capabilities

XQuery to produce a result that can be a structured XML document or other formats (such as plain text, HTML or SQL). The metadata querying output capabilities are depicted in Figure 4.36.

Metadata Transforming allows the transformation of XML content, for example, the content of an Instance, or the result of a query, into other formats such as XML, HTML or others, using XSLT style sheets. Since XSLT only allows XML content as input, they can only be applied to queries that return XML or to Instances stored in the repository. Transformations generally are used to output HTML as visualization or documentation, but can also be used to transform metadata from one standard to another. There are two types of transformations, single transformations and transformation pipelines; every transformation can be a single isolated XSLT, a XSLT that reuses templates from Fragments or a Generic XSLT that reuses templates from Fragments, each of them will be described in the following sections. Figure 4.37 depicts the repository's transforming output capabilities.

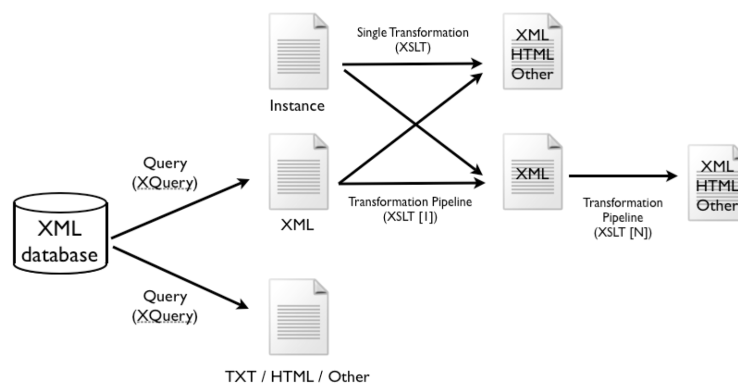

Figure 4.37 Metadata repository transforming and output capabilities

4.5.1. XQuery

Queries are executed in the context of the database environment and, to provide an abstraction of the storage model, several XQuery functions are provided in the form of a module that user-defined queries include to have access to some features, such as the relations of an Instance. As an example, the headers of the functions that provide access to relations of Instances are depicted in Figure 4.38.

```
declare namespace mdr="http://mdr.di.fc.unl.pt";

declare function mdr:getRelationsInstance($id as xs:string)
{ [9 lines]

declare function mdr:getInverseRelationsInstance($id as xs:string)
{ [9 lines]
```

Figure 4.38 XQuery functions to access relations in instances

Using the repository's web service interface any application can execute queries, by calling a specific method and using as a parameter a XQuery expression to execute. This, however, forces that every application must know how to build XQuery expressions; to allow for better separation of concerns and because the use of XQuery was a design choice, the repository can store XQuery expressions to be later used by external applications. These XQuery expressions are stored using the repository storage model, as Instances of a System Concept that external applications can access by invoking a specific method and using as a parameter the name of the query, instead of having to supply a XQuery expression. The result of a query can then be used to feed a transformation or a transformation pipeline, or be outputted as it is. To provide an extra degree of organization, queries can be grouped in categories; each category can have an unlimited number of queries and a category is saved in the repository as an Instance of a System Concept.

4.5.2. Transforms

Transforms can be executed over the content of an Instance or over the results of a query. As described in the Concept definition language, Concepts can have XSLT style-sheets embedded in their definition that can be applied to the content of their Instances (and only to Instances of that Concept), these are stored along side Concept management information. XSLT style-sheets can also be stored as Instances of a System Concept and be used by external applications using the web service interface. Transforms can be associated to queries, so that the result of query can be directly passed to a transform (or a transform pipeline). Transform Pipelines are also stored as Instances of a System Concept and can also be associated to queries. The integrity and validation are assured by the Repository's integrity and validation mechanisms (since it's dealing with Concepts and Instances).

To promote reuse, Transforms can make use of templates associated to Fragments. These templates are meant to process only the structure of a Fragment (that may be reused by Concepts), but if a user is designing a XSLT to output a visualization in HTML of a given type of Instances, and all of them have a parent Concept that reuses a given Fragment, the user can include the templates associated to that Fragment to make an HTML visualization of that structure. As an example, consider there's a Fragment with MRI *mdr://example.com/F1*, and it has a XSLT associated like the one depicted Figure 4.39.

```
<XSLList>
    <XSL name="Documentation" type="HTML">
        <XSLCode>
            <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"> [4
        </XSLCode>
    </XSL>
```

Figure 4.39 Example XSLT association to a Fragment

A user defining a XSLT to process Instances that use this Fragments structure (because their parent Concepts reuses the Fragment structure) could include these templates in the XSLT, by using the following processing instruction in the XSLT code:

<?fragmentXSL mri=”mdr://example.com/F1” name=”Documentation”?>

When a Transform definition is processed, its content is scanned for processing instruction that follows this pattern. To be valid, the processing instruction must have the *mri* attribute whose value must be a valid Fragment MRI and the *name* attribute that must have as value, the name of a XSLT associated to a Fragment (it must match the *name* attribute of a *XSL* element in the Fragment definition, of the Fragment referenced in the processing instruction). This would make an example XSLT look like the code in Figure 4.40.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="html" version="1.0" encoding="UTF-8" indent="yes" omit-xml-declaration="yes"/>
    <xsl:template match="/"> [9 lines]
    <?fragmentXSL mri="mdr://example.com/F1" name="Documentation" ?>
    <xsl:template match="*"> [4 lines]
```

Figure 4.40 XSLT with reuse of Fragment templates

It’s up to the user to build the XSLT in a way that the XSLT processor can reach the templates included with this method. If the templates are included by the XSLT and it does not have a set of templates that can make the processor reach the elements of the Fragment structure, it will not use those templates.

4.5.3. Generic Transforms

The Generic Transform is a notion in the repository to promote even further the reuse of existing code. If a user wants to build a generic visualization in HTML for all kinds of Instances he can make a generic XLST that matches any element and output a HTML representation of that element. However, if the Instance to which the Generic Transform would be applied has a parent Concept that reuses Fragments and those Fragments have associated to them a set of XSLT templates that know how to create a HTML visualization for the structure of their Fragment, it would be interesting to be able to “override” the Generic Transform generic templates with the specialized ones from the Fragments. The problem with this, is that Fragments reused in Concepts can be included inside compositions (described in the Concept & Fragment definition language) and those compositions create new elements that are not accounted for in the Fragment templates, causing the generic templates of the Generic Transform to output the content without any processing (the standard behavior of a XSLT processor). Since the repository is aware of the structure of each Concept and can generate a set of

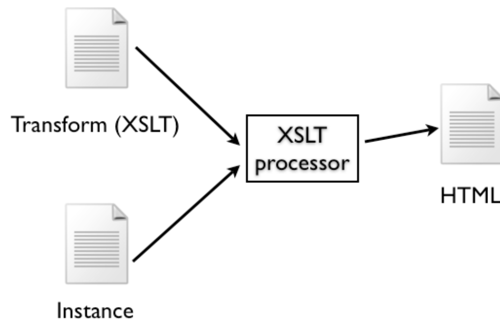

Figure 4.41 Regular transforming process in the repository

XSLT templates that can reach the composition elements and tell the processor to advance further in the XML tree to reach the parts of the document that are processable by the Fragment templates, it's possible to dynamically build a XSLT that's based on the Generic Transform's templates, including the Fragment templates and the templates generated based on the Concept's structure. Figure 4.41 depicts the normal transforming process for a Transform operation (in this example a transformation to HTML is performed).

In the case of a Generic Transform the process is depicted in Figure 4.42.

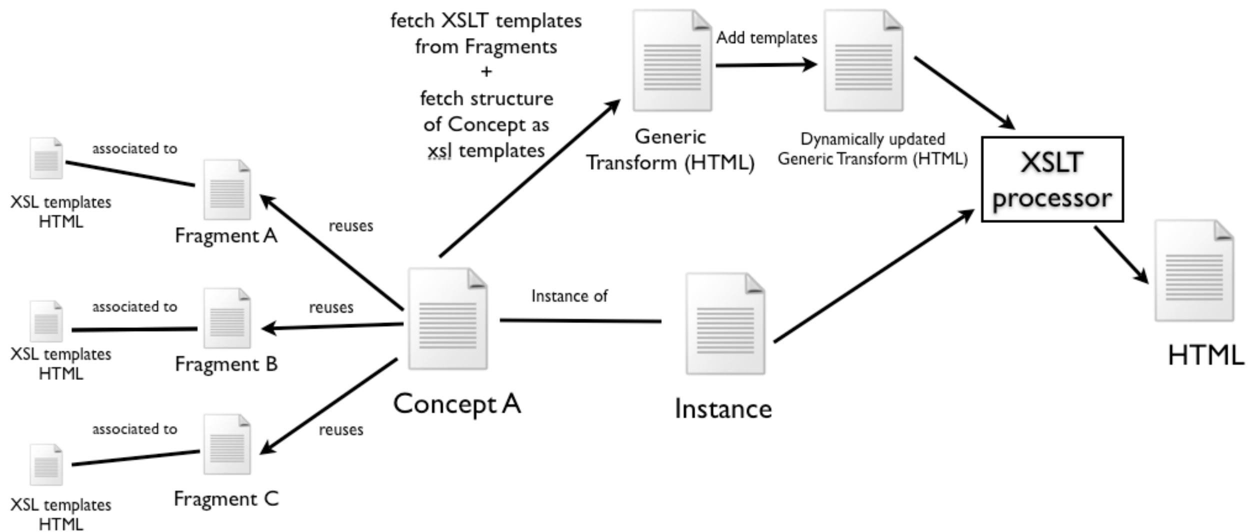

Figure 4.42 Generic Transform processing in the repository (example for a HTML Generic Transform)

The process of building the Generic Transform is done in two steps, first retrieving the Concepts structure as a set of XSLT templates and second retrieving the set of XSLT templates, from Fragments, that apply. In the Fragment definition language, each XSLT associated to a Fragment had an attribute named *type*; this attribute is what defines the category (or group) of that XSLT. When executing a Generic Transform, a *type* parameter must be passed so that the Repository can choose the correct templates from each Fragment, as such, the repository will choose the templates whose *type* attribute matches the *type* parameter of the Generic Transform operation.

A Generic Transform is stored as an Instance of System Concept and it basically consists of XSLT code, but that code must be valid according to two restrictions:

- It must provide a template that matches the root element ('/') and uses the *apply-templates* primitive.
- It must provide a template that matches any element ('*') and uses the *apply-templates* primitive.

These two rules ensure that any element will be matched, starting by the root element, proceeding down the XML tree and applying the Generic Transform templates. Since the XSLT templates from the Fragments are included in the style-sheet, each time the processor reaches the elements that belong to a Fragment structure those templates will process them. If the Concept structure used compositions, the templates generated from the structure will make sure those elements are skipped, this is due to compositions generating wrapper elements around what it's being composed and as such the Fragment XSLT templates don't account for those elements.

XSLT templates can have a priority attribute to ensure that a specific XSLT template is processed instead of another in a situation where both templates could be executed, as is the case with Generic Transforms where's a template that matches any element and the templates from the Fragments. However, there's no need to explicitly manipulate the priority attributes since the standard behavior of a XSLT processor is to apply the template that's more specific (in terms of what it's matching) and in the case of Generic Transforms the match any element template always has the lowest priority, so there's no need to change priorities.

Chapter 5
Implementation

This chapter presents implementation details of the architecture, functionalities and information model of the repository

5.1 Technologies…… 88
5.2 Architecture Design Implementation…… 88
5.3 Choice for the Underlying Database of the Storage Model…… 90
5.4 Storage Model…… 101
5.5 Information Model…… 107
5.6 Querying and Transforming…… 111
5.7 Implementation Status…… 115

This chapter describes the implementation of the metadata repository architecture and functional designs discussed in the previous chapters, starting by listing the technologies used for the implementation, followed by a description of the architecture and functional design implementations.

5.1. Technologies

To address the requirements of multi-platform and portability, the metadata repository is developed using open source technologies. The metadata repository engine is developed using Java 2 Enterprise Edition (J2EE) 6.0 [13] running as a web application in the Java 6 embedded Web Server. The engine includes support for multiple databases of the open source native XML database (that will be described in this chapter) each one supporting e Metadata Repository instance. XML technologies are used extensively by the repository engine namely XML, XML Schema, Schematron, XSLT and XQuery. The repository features a single web service interface for any external application to connect and request services. In this way it's possible to deploy the repository in any java-enabled platform (Windows and Mac OSX platforms were tested successfully)

5.2. Architecture Design Implementation

This section describes how the architecture of the Metadata Repository (chapter 3) is implemented. Starting with a high-level view of the architecture and finalizing with the low-level architecture. The implementation details of the Information Model will be presented in section 5.4, preceded by a presentation of the reasons that led to the choice of the underlying XML database and the repository's storage model.

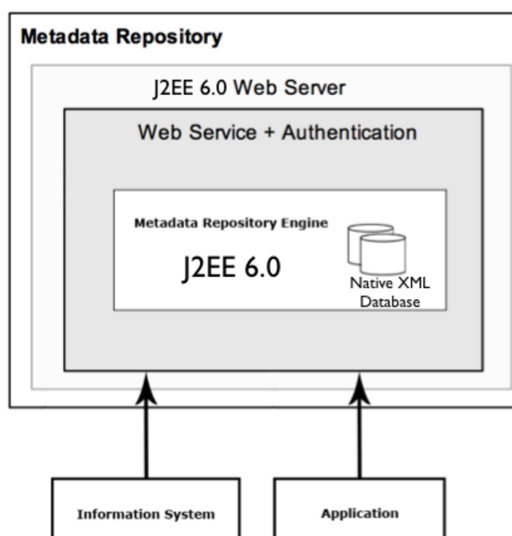

Figure 5.1 Metadata Repository's High-level architecture

5.2.1. High Level Architecture

The repository's high-level architecture proposed in chapter three and implemented using the above technologies is depicted in Figure 5.1.

5.2.2. Low Level Architecture

The repository is internally implemented by layers, this means that a top layer relies on the services provided by a lower layer. This architecture is depicted in Figure 5.2.

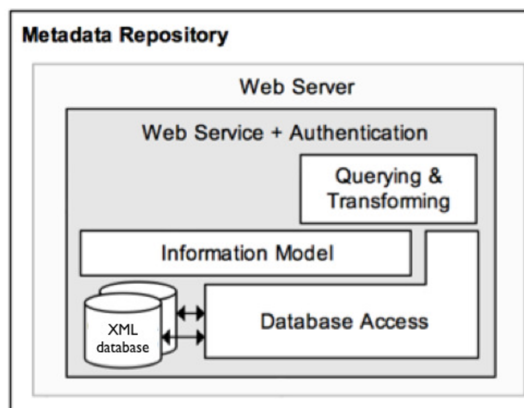

Figure 5.2 Metadata Repository's low-level architecture

The Database Access layer is responsible for managing the connections to every database and providing access to its content to the upper layers. This layer provides an interface to manage resources and collections in the XML database (the database will be presented in the following section), as well as executing queries. The Information Model layer implements the Metadata Repository's Information Model (section 3.3), including all metadata validation and integrity checks. This layer interacts with the Database Layer, to retrieve, query and store resources in the database and provides high-level functionalities such as adding an Instance version, a Fragment, a Concept or checking integrity constraints in Instance relations. The Querying and Transforming layer is responsible for all XQuery execution in the repository and for every Transformation (Single, with Template reuse, Generic or Pipeline) in the repository. Queries are executed directly over the database and, as such, require the services of the Database Layer, but on the other hand, every Query and Transform is stored as an Instance of a System Concept and, thus, requires the services of the Information Model layer. Separating the various repository functionalities in layers is an advantage in terms of source code modularity, easing the maintenance and the development of new features. For example, if a different database were to be chosen to support the storage model, only the Database Layer would need to be modified, as long as the interface to the upper layers remains the same.

5.3. Choice for the Underlying Database of the Storage Model

To support the storage model, a database is required. In order to choose the database that best suits the MDR a comparison between several databases was made.

The metadata repository (MDR) is a document repository, as such, it will have to store documents (in this case Instances, Fragments and Concepts) and will have to be able to retrieve/query/update them.

There are several alternatives, which provide such core functionality. The various databases that are of interest to this project can be divided in two categories: the XML enabled-databases and XML native databases [100].

XML Enabled Databases (Relational Databases)

XML enabled databases are databases that derive from relational databases where XML support was latter added. Relational databases are very popular for storing application data, and have proved their value over the years in terms of design, scalability, querying and update capabilities [56]. Relational databases have a record-centric data model, meaning that the fundamental unit of information are records stored inside tables (each record is a set of data-typed values). Several databases (both open source and commercial) have some/full support for XML. Examples are Oracle 11g [101] and MSSQLServer [102] (commercial) or MySQL [103] and PostgreSQL [104] (open source).

Native XML Databases

Native XML databases were built specifically to deal with XML data and their data model uses the XML document, as its fundamental unit. These databases feature indexing mechanisms off all fragments of the XML documents, on optimized structures to provide fast querying and updating. These databases rely on XML technologies for providing most of the querying/validation/updating. Storage-wise, documents are usually grouped in collections and resources inside the database, similarly to directories and files in a conventional file system.

The MDR extensively uses XML technologies and, as such, the use of a native XML database to support persistency and querying is a choice that brings advantages because these databases were specifically built to deal with situations like this. Some of the features required for the MDR are the following (they can be built-in in the database or be provided by some third party):

- Support for multiple databases
- XQuery [43] compliant
- Open Source
- Fault Tolerance

- Multi-platform
- Provide an Update Language
- User definable Transactions
- Indexing of documents
- Efficient Storage and Querying
- Java API with all the important primitives

In the following sections the list of analyzed databases is introduced:

5.3.1. eXist XML Database

The eXist XML database [105] is a candidate, since it proved that it can be used as the basis for a metadata repository [1] (although with its limitations, for example, eXist does not provide user-definable transactions, which had to be implemented on top of it, in that project). eXist provides, however, a great set of features with support for major XML standards (XML Schema, XSLT, XPath, XQuery, XQuery Update Facility) and enables users to write web applications entirely using XQuery extensions to present the content in (X)HTML with XSLT.

eXist features collection-based storage of XML documents and it provides security mechanisms, such as users and permissions. The storage mechanism is based on b+-tree and pages [106]. It has an automatic index, based on a numeric index scheme, to quickly identify node relationship and features an optimized XQuery engine that uses this schema to provide efficient querying, as described in [107].

eXist provides backup and recovery functionalities and has basic document-level transaction, although (as stated before) they're not visible to the user. eXist is developed in Java and is available in all major platforms (Windows, Linux, OSX).

The deployment of eXist can be within a web server (Such as JBoss [108]/Tomcat [109]) it can be run as standalone application or embedded in a Java application and is able to control XQuery access with XACML [110]. eXist is one of the most widely used XML databases and has wide community support.

5.3.2. Sedna XML Database

The Sedna XML database [111], is an open source native XML database produced at the Institute for System Programming at the Russian Academy of Sciences, since 2006. It's developed in Scheme and C/C++ (Scheme is used for static query analysis and optimization, C/C++ is used to implement the parser, executor, memory manager and transaction manager), from scratch. It was designed having two goals in mind: To be a full featured database system and to provide a run-time environment to XML-intensive applications [112].

Sedna's storage of XML documents, uses a descriptive schema approach [112]. A descriptive schema is a concise and accurate structural summary of a XML document [113], generated from the XML document and maintained through the existence of that document in the database. Contrary to prescriptive schema which dictate the possible structure of the document (DTD, XML Schema), this approach enables multiple, efficient, optimizations for the storage and querying of documents and collections, as described in [113].

Sedna highly supports the XQuery standard for Querying documents (98.8% on the XQuery Test Suit [114]) and supports a declarative node-update language. The update language is based on the XQuery update proposal by Patrick Lehti [115]. Sedna was developed with the data model of XQuery in mind and offers a number of optimization techniques around that model [113].

Sedna is deployed as a standalone application (with a simple command line interface, there is no GUI administration provided, but third party ones, exist) and features a range of built-in API's (featuring Java, C, Scheme) and a number of third-party produced API's (.NET, Pyhton, PHP) are available.

Sedna supports database users, permissions, roles and it provides recovery and backup mechanisms (including "hot-backup" done while the database is still running and performing requests). Concurrency-control mechanisms exist and user-definable transactions are supported.

Sedna is in active development, although, since it's a new database, the community support is somewhat small. The developers provide extensive documentation and a mailing list is available to anyone. Even though Sedna does not support neither XQuery Update Facility nor, for example, XUpdate, it's still an interesting choice, because it features everything else that is required and, in the MDR, direct updates over the database will not be possible, so it stands as a candidate.

5.3.3. Berkeley DB XML

Oracle Berkeley DB XML is an embeddable XML database engine that provides support for XQuery access [116]. Berkeley DBXML is developed on top of the well-known Berkeley DB and inherits its features, such as concurrency control, efficient storage and retrieval, transactions, backup, recovery and replication. Oracle Berkeley DB XML adds a document parser, XML indexer and XQuery engine on top of Oracle Berkeley DB to enable fast and efficient retrieval of data [116].

XML Documents are stored in "containers" (a collection of XML documents) and each container maintains the indexes created for each document. Being an embeddable database, means that it does not provide for features such as users, permissions or roles (the application using DBXML must deal with this) but enables operating the database with zero-administration and reduces hardware costs (the memory footprint is small). As such, there are no administration utilities, only a command line console to enable interactive sessions.

Berkeley DBXML uses several optimization techniques, such as partial document re-indexing, intelligent cost-based query processing and iterator-based processing instead of tree-based processing [116].

It supports the major XML standards such as XML Schema (for validation of documents in a container and, contrary to most database systems, each container may validate XML documents associated with different XML schemas), XQuery, XPath and XQuery Update Facility. One feature of Berkeley DBXML is the possibility to associate individual metadata to a document (and query that metadata).

Berkeley DBXML is a product of Oracle [101] and, as such, has extensive support via online forums, mail and several resources on the internet are available.

Evaluation

All three databases presented, have the features that would make them a good choice for the MDR, although, eXist would require implementing a transaction layer and Berkeley DBXML would require implementing a user/permission layer. The one factor that has not been considered is the performance of each database.

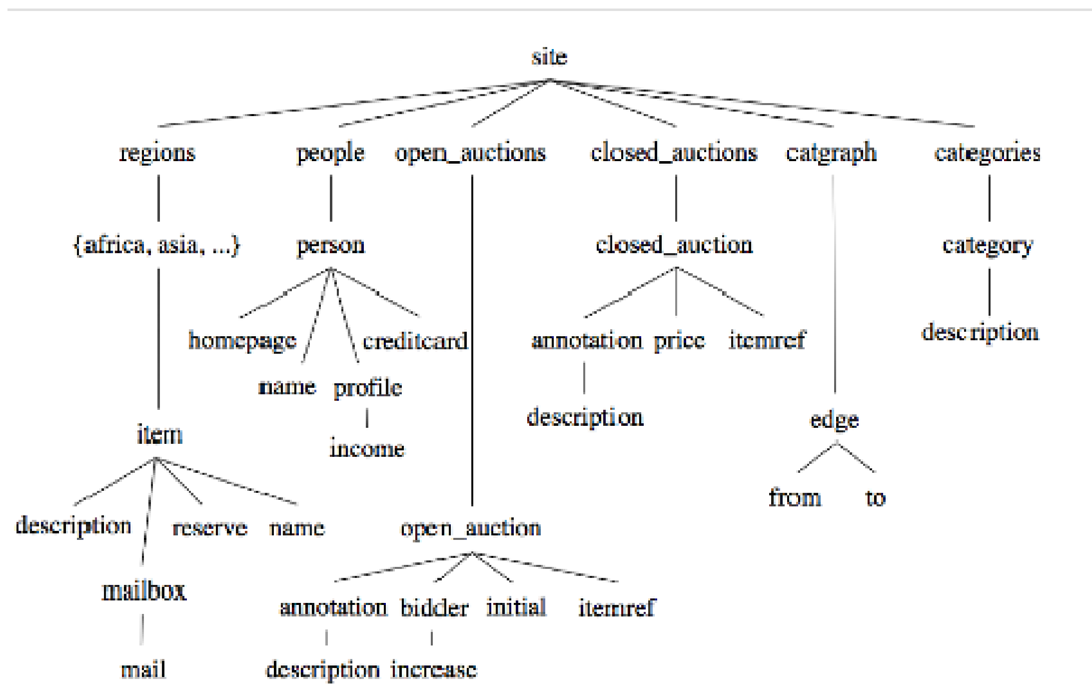

Figure 5.3 XMark XML structure [2]

Several studies around the performance of XML databases are available, however none of them include all three databases and some only measure storage performance [117, 118]. In order to have more reliable data, a benchmark on all three databases was performed.

5.3.4. Query Benchmarking

For the benchmarking of the XML databases, the benchmark framework X-Mark [2] was chosen. X-Mark is designed to test the performance of XML databases with a broad range of typical queries found in real world scenarios. This set of queries challenge the XQuery processor in several important primitives of the XQuery language. The structure of the data used by the X-Mark framework is based on an Internet auction site and is presented in Figure 5.3.

There are relationships between elements. Some relationships are based on references (person, open_auction, closed_auction, item and category) and some using natural text (annotation and description).

The X-Mark framework comes bundled with a data generator, which can generate documents in a scalable way, maintaining the structure presented before and populated with meaningful data. This means scalability can be tested, since documents as small as 36 kilobytes or as big as several gigabytes can be produced.

There are 20 different queries in X-Mark. This set of queries, explores several of XQuery's capabilities and can be grouped in these categories:

- **Exact Match (Query 1)**
- **Ordered Access (Query 2,3,4)**
- **Casting (Query 5)**
- **Regular Path Expressions (Query 6,7)**
- **Chasing References (Query 8,9)**
- **Construction of Complex Results (Query 10)**
- **Join on Values (Query 11,12)**
- **Reconstruction (Query 13)**
- **Full-Text (Query 14)**
- **Path Traversals (Query 15,16)**
- **Missing Elements (Query 17)**
- **Function Application (Query 18)**
- **Sorting (Query 19)**
- **Aggregation (Query 20)**

For further information about X-Mark and its queries, see [2].

Evaluation

The benchmark was run on a Pentium Dual Core (2.6 GHz per core) with 4GB of RAM and a Serial-ATA disk with 500GB running Windows XP Professional (Service Pack 3). Every undesirable running process was terminated (including anti-virus software and such) so that the benchmark was as little disturbed as possible.

The data generator was used to produce a set of six files, starting from 36KB, including a 100KB one, a 1MB one, a 11MB one, a 111MB one and a 1GB file.

For each database, Windows XP binaries were downloaded and installed (no compilation from source code was made) and the databases were used “out-of-the-box”, i.e. no indexes were created or optimizations were made. The latest stable versions for each database were used, meaning:

- **eXist XML Database version 1.2.4**
- **Sedna XML Database version 3.1**
- **Oracle Berkeley DB XML version 2.4.13**

For each database system, six databases were created and one collection inside each of the six databases was created. Each collection was populated with one of the six files generated. Each of the twenty queries was run ten times in a row against each of the files stored in the collections, for every database. Three small Java applications were responsible for connecting to each database system, selecting the appropriate database (and collection) executing the queries and measuring the time taken by each one. The times obtained reflect the query execution only (excluding the result serialization). Time was measured issuing a (Java) call to System.getTimeMillis() before and after executing the query, and the difference was stored in an array for calculation of the average result.

This test provides a performance evaluation over one file, which is representative for queries made against a specific file, but does not cover a query over an entire collection. In order to assess the collection-querying capabilities of the XML databases, another test was run. The test consisted in loading several documents (with random content) to a collection of a database and one document from the X-Mark set (the 100KB document), creating a large collection to be queried. Two collections were created, the first collection was loaded with one hundred equal documents of random XML (each document’s total size was 100KB) and one 100KB document generated for the previous test; the total size of the database was 11 Megabytes (hereafter described as “Test1”). The second collection was loaded with 3700 equal documents of random XML (36 Kilobytes, each document) and one 100 Kilobytes generated from the previous test (hereafter described as “Test2”). The total size of the previous collection was 111 Megabytes.

For this test, the X-Mark queries were updated in order to query the collection and return the results. From the results of the twenty queries, two were chosen because they illustrate the general trend in the query results. The chosen queries were Query 2 and Query 8. Results presented include the average of ten runs of a query and the worst results of the ten runs. This is to show that the databases apparently use some sort of caching mechanisms, although it does not seem to be always used. Some cases are very clear of caching being used, and in some cases the worst result is very close to the average one. The results were the following (presented Figure 5.4 and Figure 5.5, results are presented in milliseconds).

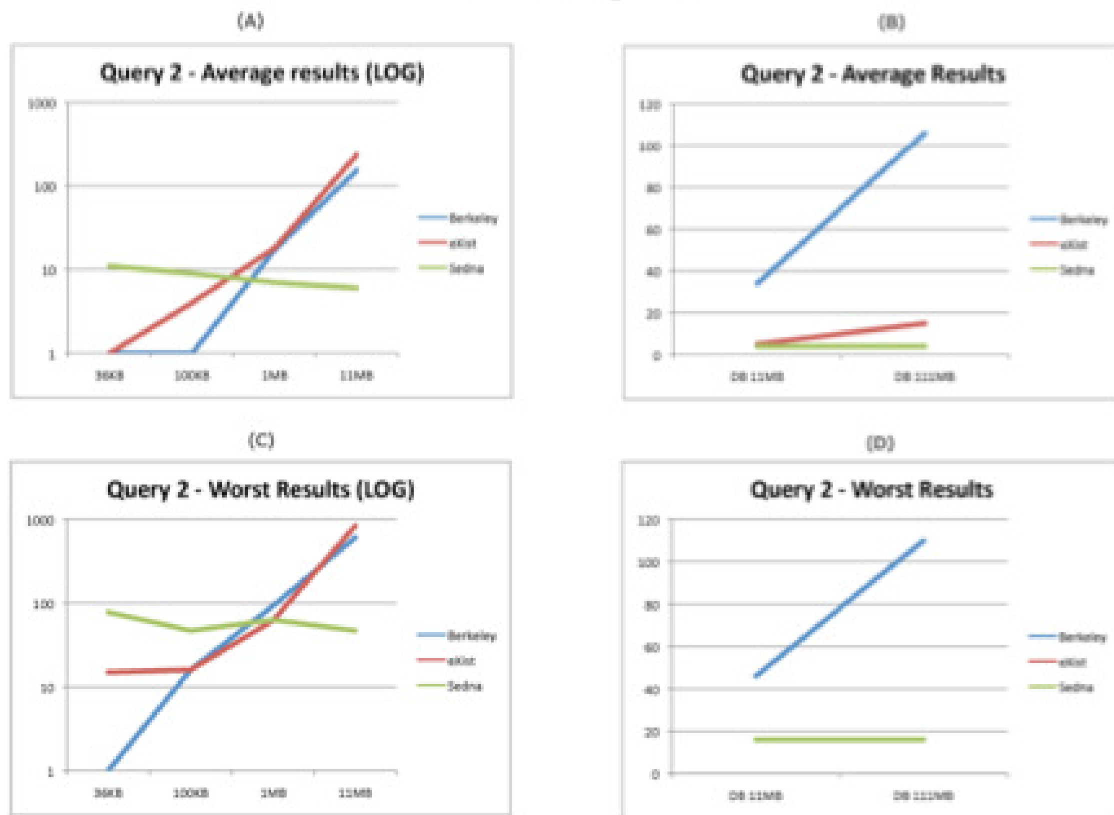

Figure 5.4 Results for Query 2 of the XMark Benchmark

Note: Figures with (LOG) in the their label, present the results in logarithmic scale for better understanding. Charts (A) and (C) in the figure represent the average and worst results, respectively, for “Test1” while charts (B) and (D) represent the average and worst results, respectively, for “Test2”.

Query 2 - Analysis

Query 2 is a query that “*evaluates the cost of array lookups. Note that it may actually be harder to evaluate than it looks; especially relational back-ends may have to struggle with rather complex*

aggregations to select the bidder element with index 1." [2]. This query was chosen to show the apparent caching mechanisms present in the database systems. Looking at chart (A) and (C), from Figure 5.4 and comparing the results from the worst (C) with the average (A), in Sedna's case, the worst result is roughly 10 times slower than the average result. eXist and Berlekey DBXML also show some signs of caching in this query. The clear winner of this query is Sedna, as it can scale very well, while the other systems have difficulty with larger files.

Querying a collection (even if bigger in size, than file to be queried), proved to easier for every system, and both eXist and Sedna, have a good performance in this query. Berkley DB XML does not scale so well with the size of the database.

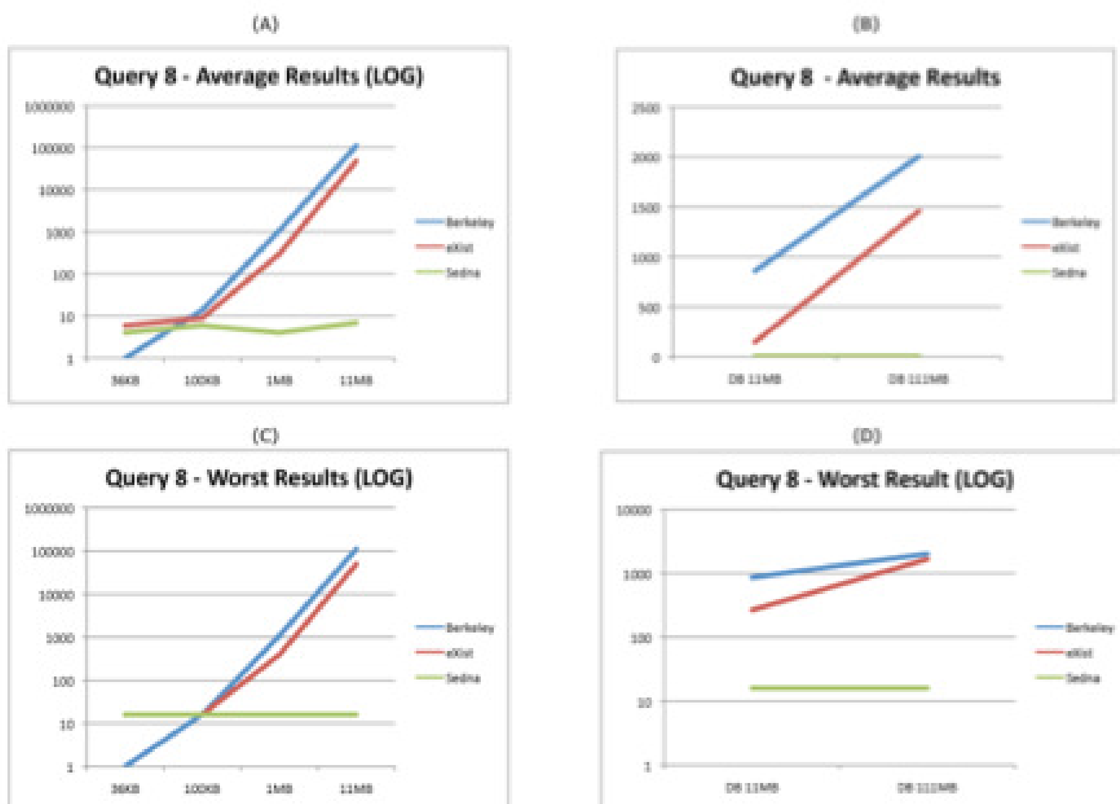

Figure 5.5 Results of Query 8 of the XMark Benchmark

Note: Figures with (LOG) in their label, present the results in logarithmic scale for better understanding. Charts (A) and (C) in the figure represent the average and worst results, respectively, for "Test1" while charts (B) and (D) represent the average and worst results, respectively, for "Test2"

Query 8 - Analysis

Query 8 is a query that "*List the names of persons and the number of items they bought. (joins person, closed auction). References are an integral part of XML as they allow richer relationships than just hierarchical element structures. These queries define horizontal traversals with increasing complexity. A good query optimizer should take advantage of the cardinalities of the operands to be joined*." [2]. Analyzing the results of all queries, queries 8 through 12 are the hardest to evaluate (i.e. those who take longer to provide the result) and, as such, query 8 was chosen to show the capabilities of the databases. In Query 8, there are no signs of caching mechanisms (depicted in a comparison between charts (A) and (C) and between (B) and (D)) as the values are very close to one another. One conclusion that can be drawn is that eXist and Berkeley DBXML have great trouble with larger files, while Sedna provides good performance. In the collection-querying situation Sedna is still a clear winner, but the difference to the other two systems is not so big (although it is still ten times better than eXist and eighty times better than Berkeley for the 11 MB collection (B)).

Querying Evaluation

For small files (less than 1MB) every database produces fast results, all bellow 50 ms. However, when the size of the files starts growing (1MB/11MB) there's a clear difference between Sedna and the other two, especially on queries that involve joins (Q8 through Q12). In Q8 of the 11MB test, Sedna average result outperforms eXist by approximately 3000 times, and Berkeley DBXML by 6900 times. It's also clear that Sedna uses some kind of caching mechanism, as the first result is, usually, slower than the average result (up to 10 times), but still faster (especially in the larger files) than the other two systems. Sedna is the only database able to deal efficiently with an 111MB and 1GB example (in all queries). Querying entire collections, is still an advantage for Sedna, but eXist and Berkeley perform fairly well also (although several times slower that Sedna in most queries). Querying a collection with a total size of several megabytes is more expectable to happen than querying a file of the same size. Collection benchmark is more interesting in terms of real-world benchmark and Sedna is the one that provides better results.

Sedna consumes more resources than the other two systems. A freshly created database requires, by default, over 200 megabytes in the file system and each running instance of a database, by default, has a footprint of 100 megabytes of RAM memory, although it's possible to configure these values. Berkeley DBXML has the smallest footprint in memory and the file system.

5.3.5. Storage Benchmarking

Query performance of a XML database is extremely important, but other factor is also important: The storage performance. Applications that use a XML database with intensive insert/update operations will require that these operations are quick and efficient. In order to assess the capabilities of the databases in this situation, a loading test was performed.

The test consisted in loading a set of equal documents to each database; the sets are as follows:

Table 5.1 Table of document size and number to benchmark

Number of Documents	Size of Documents
10000	36 Kilobytes
1000	100 Kilobytes
100	1 Megabyte
1	111 Megabytes

Note: The files used, were the same ones used in the querying benchmark.

For each set of documents, a collection in each database was created and a small Java application was developed to load the entire set into the collection. Times were measured in the same way as before, issuing a call to System.getTimeMillis(). Times presented here are a mean of five tests.

Sedna XML results

Table 5.2 Sedna XML database storage results

Number of Documents	Size of Documents	Times (ms)
10000	36 Kilobytes	101.606
1000	100 Kilobytes	41.850
100	1 Megabyte	68.081
1	111 Megabytes	36.278

Berkeley DBXML results

Table 5.3 Berkeley DBXML database storage results

Number of Documents	Size of Documents	Times (ms)
10000	36 Kilobytes	765
1000	100 Kilobytes	90
100	1 Megabyte	68
1	111 Megabytes	7.062

Note: Berkeley DBXML is very quick for small documents, but, for example, for the 111 Megabytes document, the first run took 35.313 ms, and the following ones 0, so it means that file was probably in cache and insertion is a process that simply stores the file in the container.

eXist results

Table 5.4 eXist XML Database storage results

Number of Documents	Size of Documents	Times (ms)
10000	36 Kilobytes	2.615.059
1000	100 Kilobytes	2.401.752
100	1 Megabyte	2.866.520
1	111 Megabytes	3.212.509

Storage Evaluation

Anayzing pure storage performance, Berkeley DBXML is the clear winner for small files. Even for the 111 MB file, it had a great performance, but as stated, the first result was equivalent of Sedna's mean result, so it must mean that storage is a process of simply storing the documents in the containers, after the first run. eXist takes a huge amount of time, for big files and for a large number of file. After this test, eXist can't be considered for the underlying database. Berkley DB XML storage performance is further confirmed by [118]. Sedna's performance is quite acceptable, considering the number of documents (and size) tested. eXist is very slow for a large number of documents or for large documents, thus, it's best suited for small collections of small documents.

The metadata repository will hold metadata and, usually, metadata is smaller in size (and number) of documents compared to the data it describes by several orders of magnitude, but, still, having a database that can handle huge amounts of data efficiently, is a better choice.

5.3.6. Final Evaluation

Both eXist and Berkeley DBXML have a good performance querying small files, but as files grow larger and queries get more complex (especially queries that involve joins) their performance takes a big hit, while Sedna can scale very well. Querying over collections, the difference between Sedna and the other two, is still relevant, but not as much as querying a single file of the size of the collection. Storage performance is the clear advantage of Berkeley DBXML, especially over eXist that takes huge amount of time simply loading the documents to the database. Although loading a document to the MDR isn't simply storing in the database (and can take some time, as there are some operations to be performed) the underlying database must provide efficient storage in all cases, and eXist only provides this for small documents and a relatively small number of documents. Sedna also has another advantage over the other two systems, as it provides transactions and users/permissions features. Considering all the information gathered through the benchmark, the Sedna XML database was considered the best choice for the underlying database of the MDR as it can support operations against small and large files, being few or many documents.

5.4. Storage Model

In order to support the Metadata Repository' Information Model and features, a storage model must be created for its underlying database. This model defines where Concepts, Fragments and Instances are stored, including management information extracted from them (to support the repository operations) and additional resources (such as cached files, to improve performance). The logical structure of this model is described as a hierarchy of collections (resembling directories in file systems) since a Native XML Database is being used (although the chosen database, Sedna, does not natively support sub-collections, the Repository will "see" the storage model as a hierarchy of collections). All databases are initialized with this model, and the collection's resources are accessible through the database API, by either direct retrieval or querying. The storage model is intended to be internal to the Metadata Repository in such a way that external applications are unaware of it, particularly when performing metadata querying and transforming operations, since these operations are executed in the database environment. Next sections incrementally present this model.

5.4.1. Concept Storage

This section will present how Concepts are stored in the database. A Concept is defined as a XML file (which contains, as explained previously, the definition of the Instances structure (using XML Schema, which may include other XML Schemas), how Instances are identified, additional validations, XSL associations, relations, etc). Figure 5.6 depicts the Concepts storage model, which is presented in the following paragraphs.

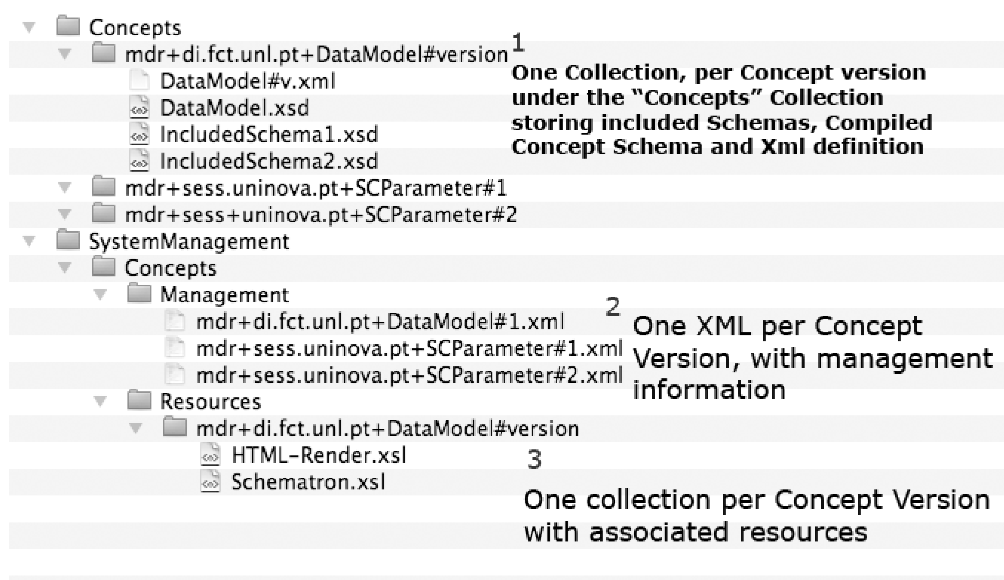

Figure 5.6 Concept's Storage Model

The Concept storage model features a high-level collection named “Concepts” where, for each version of each Concept, there will be a sub-collection named with the identifier and version of the Concept. In (1) we have an example of the “DataModel” Concept, that is associated with the “di.fct.unl.pt” namespace and is the first version of the Concept (although, in the figure, the word “version” is used to depict where the version number is to be placed) but in the following two examples (sess.uninova.pt/SCParameter#1 e #2) the version number is present. Inside each of these collections, the Concept’s included schemas are placed, as are the XML definition of the Concept and a “compiled” XML Schema (for caching purposes) created from the structure declared in the definition. A compiled XML Schema is the result of analyzing the Concept definition and producing a valid XML Schema from it (in order to validate Instances).

A second high-level collection named “SystemManagement”, whose purpose is to store management information, will have a sub-collection named “Concepts” where there will be two sub-collections. One named “Management” (2) and one named “Resources” (3). In the Management collection a XML file with management information (extracted initially from the definition of the Concept, and updated with subsequent repository operation) for each version of each Concept is kept. In the Resources collection, a sub-collection for each version of each Concept will hold resources associated with that Concept. If a Concept definition has embedded XSLT code, then, that XSLT code will be extracted from the definition, and placed in the Resources collection of that Concept. In the same way, if there are XSLTs “compiled” from the definition (i.e. that use XSLT templates from fragments) they will be placed in the same collection.

5.4.2. Fragment Storage Model

Fragments are stand-alone (or compositions of) XML Schema fragments and are used as a base to produce Concepts. The Fragments storage model is depicted in Figure 5.7.

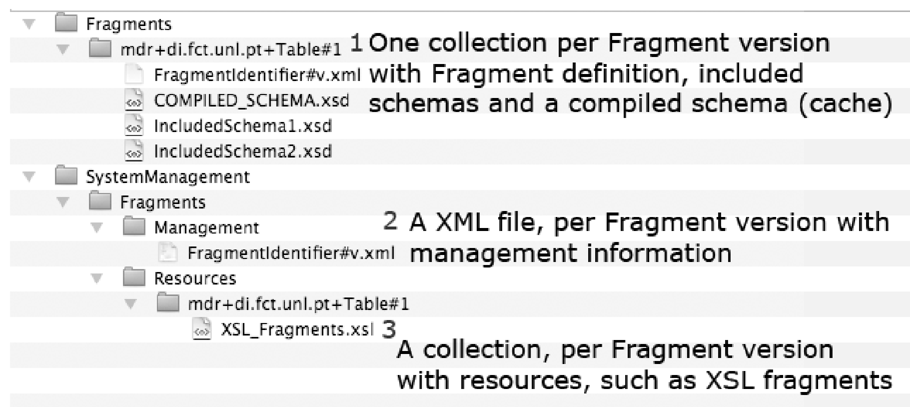

Figure 5.7 Fragments Storage Model

This storage model has, just like the Concepts storage model, a high-level collection named "Fragments" which has one sub-collection [named with the identifier of the Fragment (Figure 5.7, area marked by "1") and version] for each Fragment version, where the included schemas and fragment definition are stored, including a "compiled" schema, for caching purposes. In the SystemManagement collection, there's a sub-collection named "Fragments" (2), where a XML file for each Fragment version, exists. This file holds several management information, such as Concepts that use this Fragments, XSLT templates associated with this Fragment and metadata about the Fragment (descriptions, key-words, dates), etc. Some Fragments, may have XSLT templates associated (that will be used by Concepts to generate on-the-fly complete XSLTs, to process Instances) and, as such, for each Fragment version, a sub-collection in "Resources", named after the identifier of the Fragment is present, and holds this kind of resource.

5.4.3. Instance Storage Model

Instances are XML documents, compliant with the structure of a certain XML Schema, defined by a single Concept. Instances will be the primary target for queries and updates, since they represent the metadata in the MDR. The Instances storage model is depicted in Figure 5.8.

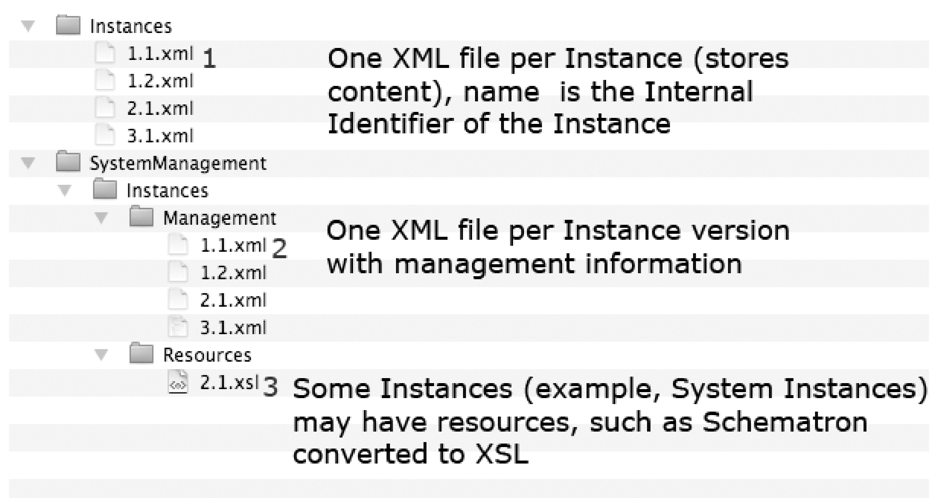

Figure 5.8 Instances Storage Model

At a high-level there is an "Instances" collection (Figure 5.8, area marked by "1"), where all Instances are stored. Instances are named using an Internal Identifier (IID). The internal identifier works as follows: When a Concept is added to the repository, it's given a unique number (The Concept's *Magic*[2] number). When an Instance of that Concept is added to the repository it's identified with the concatenation of that Concept's number, with a sequence number given to that Instance. So, for example, if a Concept's unique number is "1" and its first Instance is being put into the repository, its internal identifier will be "1.1" and it will be the latest version of that Instance.

[2] The name "Magic Number" is inspired in http://en.wikipedia.org/wiki/Magic_number_(programming)

When, after that, a second Instance (or a new version) of that Concept is added it will have identifier "1.2", and so on. The use of these identifiers enables that Instances can change their name over time (if they are identified using the XPath method, described earlier), but their identifiers are maintained (so as their relations).

As with Concepts, under the SystemManagement (2) collection, an "Instances" sub-collection is present and, for each version of each Instance (using the previous internal identifiers as names) a XML file with management information for each Instance is stored (this file keeps information regarding relations, resources and metadata about that instance). In some cases (3) Instances may have some resources associated. For example, Instances of the System Concept Schematron, may have a XSLT resulting of transforming the Schematron file, so that in can be applied with the XSL processor. System Concepts are Concepts associated with a system namespace and will be responsible for managing the Schematron library, the XSLT library and the XQuery library (which is composed by Instances in all cases).

5.4.4. Additional System Management Information

Some additional information is required in order to ease the management functions of the repository. Information such as the list of allowed namespaces used by Instances/Concepts/Fragments or the mapping between the MRI of an Instance and its internal identifier. The list of Fragment versions (and respective MRI) as well as Concept version (again with the respective MRI) must also be kept. To finalize the storage model, under the SystemManagement collection, there will be a sub-collection named "SystemControl", storing XML files holding the previously mentioned information, respectively in *Namespaces.xml*, *IdentifierList.xml*, *FragmentList.xml* and *ConceptList.xml* as seen in Figure 5.9.

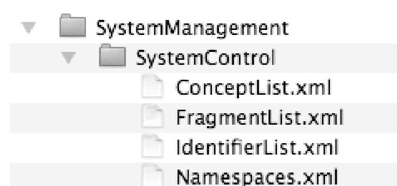

Figure 5.9 Additional System Management Information

The *Namespaces.xml* file stores a sequence of *Namespace* elements, whose content is the value of each namespace, under a root element *Namespaces*. The *IdentifierList.xml* file is responsible for the mapping between an MRI and an internal identifier, as well as storing the sequence counter for Concept's internal number (the previously described *Magic* number). The structure of this document is depicted in Figure 5.10.

The document’s root node is the *IdentifierList* element, having two children: The *UniqueID* element and the *Identifiers* element. The *UniqueID* element stores the current sequence number for the Magic Number of the next Concept to be inserted. When a Concept is inserted in the repository, the number stored in the *UniqueID* element is assigned to the Concept and the value is incremented by one. The *Identifiers* element holds a sequence of *Identifier* elements and each of those elements has a mapping between an Internal Identifier (attribute *IID*) and a MRI (set of attributes *namespace*, *concept*, *conceptVersion*, *instance*, *instanceVersion*). The reason for separating the elements of the MRI is that it’s easier to make XPath queries to find all Instances of a given Concept or in a given Namespace, etc.

```
<IdentifierList>
    <UniqueID>5</UniqueID>
    <Identifiers>
       <Identifier IID="5.1" namespace="di.fct.unl.pt"
        concept="DataModel" conceptVersion="1" instance="ExampleDatabase" instanceVersion="1" />
        <Identifier IID="6.1" namespace="di.fct.unl.pt"
            concept="Student" conceptVersion="1" instance="ExampleStudent" instanceVersion="1" />
    </Identifiers>
</IdentifierList>
```

Figure 5.10 Internal Structure of the IdentifierList.xml

The *FragmentList.xml* file stores the list of each Fragment in the repository, having a *FragmentList* root element, which holds s sequence of *Fragment* elements, each one of them with three attributes (*namespace*, *name* and *version*) to allow easier querying, as in the *IdentifierList.xml* file. A sample structure of the FragmentList.xml file is depicted in Figure 5.11.

```
<FragmentList>
    <Fragment namespace="fct.unl.pt" name="Clip" version="1" />
    <Fragment namespace="fct.unl.pt" name="Clip" version="2" />
    <Fragment namespace="fct.unl.pt" name="MscThesis" version="1" />
<FragmentList>
```

Figure 5.11 FragmentList.xml structure

The ConceptList.xml file stores the list of each Concept in the repository, using the structure depicted in Figure 5.12 (which is the same as the structure of the FragmentList.xml, but with different syntax).

```
<ConceptList>
    <Concept namespace="www.itds.pt" name="xeoModel" version="1" />
    <Concept namespace="di.fct.unl.pt" name="DataModel" version="1" />
</ConceptList>
```

Figure 5.12 ConceptList.xml structure

5.4.5. Complete Storage Model

Merging the previous storage models, under a common “MetadataRepository” collection, the full storage model for the repository is depicted in Figure 5.13.

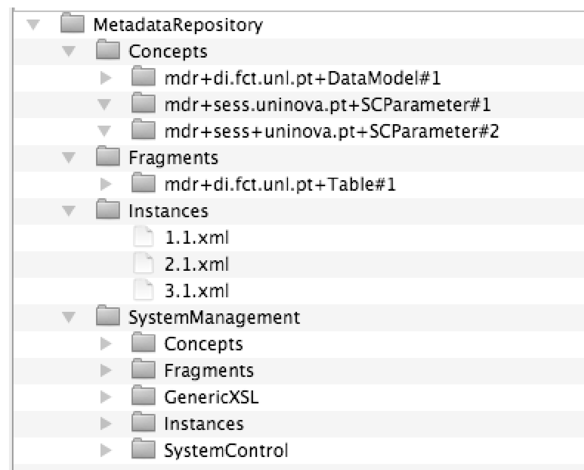

Figure 5.13 Metadata Repository Complete Storage Model

5.4.6. Access Permissions

Fragments, Concepts and Instances are to be updated/queried by external applications using the web services interface and by the metadata repository (the former, using the Web-Services API, and the later using it’s own internal methods), but the SystemManagement collection is to be queried/updated only by the repository’s management methods, so access permissions will have to be imposed. XQuery has a limited scope, based on the collections that are supplied by the native XML database, which means that external applications will be limited to Concept/Instance/Fragment querying, and will not be able to query internal resources. The web services API will enable external application to retrieve information in the SystemManagement collection such as Instance relations, but preventing direct access to the management information, as seen in Figure 5.14.

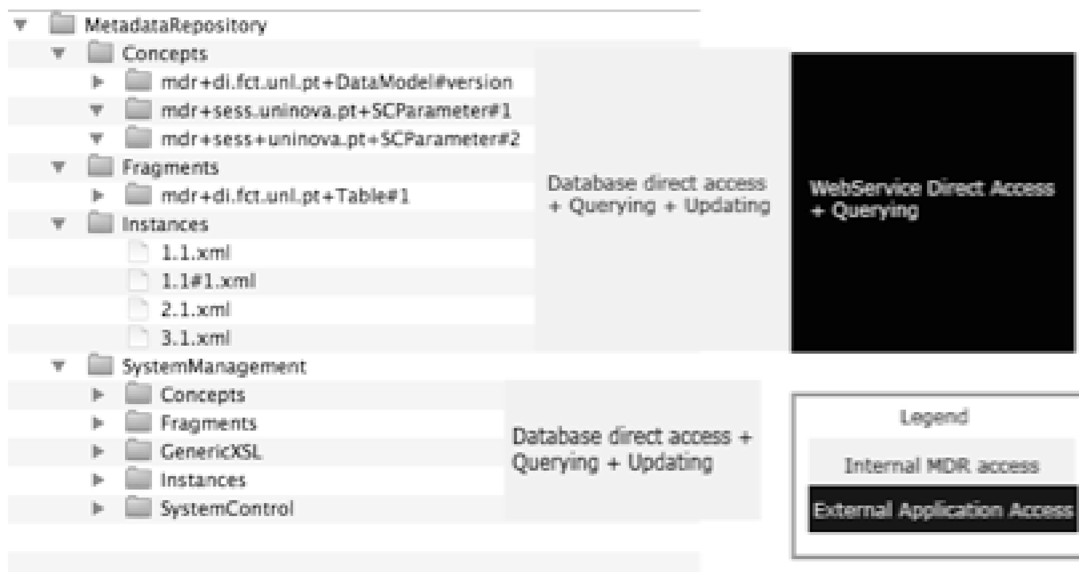

Figure 5.14 Storage Model's access permissions

5.5. Information Model

The Information Model presented in section 3.3 is fully implemented in the Metadata Repository; the next sections present implementation details of each of the layers of the Information Model, starting with the M2 layer and finalizing with the M1 layer. The M3 layer does not require the presentation of implementation details as the layer is composed of a XML Schema that verifies the validity of a Fragment (or Concept) XML definition and Java code that checks if the values present in the XML definition are, as previously described in section 4.3.

5.5.1. M2 Layer – Meta-model

This layer is where the basis for the Information Model is implemented, as presented earlier in section 3.3 and beyond. Each Fragment and Concept that is stored in the repository has an associated management file (as described in section 5.4). This management file stores its content as XML and includes the following information about each Fragment:

- Target Namespace and Prefix
- Included Files
- Imported Files
- List of Fragments it reuses
- List of Fragments that reuse this Fragment
- Metadata about the Fragment
- XSLT Templates
- MRI of the Fragment
- Version of the Fragment

Each Fragment defines a XML structure that can have a target namespace and to ease the building of another Fragment (or Concept) structure, the target namespace URI and prefix are stored in the management file. The Fragment structure can use embedded XML Schema without any restrictions and, thus, can use the *include* and *import* elements, these included and imported files content is stored alongside the Fragment definition and the list is stored in the management file. To ease integrity checks on Fragment removal, the list of Fragments that a Fragment reuses and the list of Fragments that reuse the Fragment are also stored in the management file. Metadata about the Fragment and the list of XSLT templates are also stored in the management file, so that the Fragment definition is stored and does not need to be used as a source of metadata. The Fragment's MRI and version number are also kept in the management file.

Every Concept also has a XML management file that includes several information to help manage the Concept and to deal with the integrity requirements. The list of information includes:

- Target Namespace and Prefix
- Instance Identification
- Create and Update dates
- Concept's Magic Number
- Concept's Sequence Counter
- List of Instances
- Included Files
- Imported Files
- Fragment References
- Schematrons compiled
- Embedded XSLT
- Definition of Relations
- Inverse Relations
- Metadata about the Concept
- Rules to create metadata about Instance

A Concept structure can reuse a Fragment or use embedded XML Schema and, as such, they can define a target namespace, which is stored in the management file. The Instance identification method (via XPath or Sequence Numbers) is also stored as well as the creation and last update dates. The Concept's *Magic* Number, a unique sequence number that identifies the Concept from all others (and is used to create the internal identifiers for Instances) that's generated in the Concept insertion process, is stored in this file, as well as the Sequence counter to generate the other component of the internal identifier (as described in 5.4). A list of all the Concept's Instances is also stored in the management file as well as the list of possible included/imported files (because the Concept, like the Fragment, can use embedded XML schema code and that code can make use of the include/import element). Since embedded Schematron (and XSLT) code can be declared in the Concept definition, if present, they will be extracted from the definition and stored as resources associated to the Concept, to speed the process of validation/transform, the name of the "compiled" (in case of the

Schematron, that uses the XSLT implementation) files is kept in the management file. The definition of relations is also ported to the management file, to be separated from the Concept definition; the Concepts that declared this Concept as a target of a relation are also listed to ease the integrity checks. The metadata about the Concept and the rules to created metadata about Instances that are declared in the Concept definition are also stored in the management file. As with Fragments, the management file also stores the MRI of the Concept, as well as the Concept version. In Figure 5.15 a sample Concept Management file is depicted; the figure does not contain the full information about the Concept, but one can see the internal identifier generation, as the Concept's Magic Number has value "4" (element *MagicNumber*) and the sequence counter (element *SequenceNumber*) has value "3", meaning there are already two Instances of this Concept, which can be verified by checking the *ListOfInstances* element that has two Instances with identifier 4.1 and 4.2, respectively.

```
<ConceptManagement mri="mdr://lol.fct.unl.pt/Concept" version="1">

    <Info>
        <targetNamespace>http://di.fct.unl.pt</targetNamespace>
        <targetNamespacePrefix>ex</targetNamespacePrefix>
        <InstanceIdentification>
            <Xpath>/root/child</Xpath>
        </InstanceIdentification>
        <CreationDate>2003-08-22T03:23</CreationDate>
        <UpdateDate>2003-10-05T22:14</UpdateDate>
        <MagicNumber>4</MagicNumber>
    <SequenceNumber>3</SequenceNumber>

    </Info>

    <IncludedFiles> [2 lines]
    <ImportedFiles> [2 lines]

    <FragmentReferences> [2 lines]

    <ListOfInstances>
        <Instance>4.1</Instance>
        <Instance>4.2</Instance>
    </ListOfInstances>

    <SchematronsCompiled> [6 lines]
    <XSLListEmbedded> [7 lines]
```

Figure 5.15 Sample Concept Management File

5.5.2. M1 Layer - Model

Instances are XML documents that obey the vocabulary defined by a Concept. To ease the management of Instances, each Instance has a management file that stores information about the Instance, such as:

- Instance Identification Information
- Relations with other Instances
- Relations that other Instances have with this one
- Namespaces

Instance Identification Information is a set of values about identifiers. This includes the Instance's identifier name (that string return by the XPath that the Concept uses to identify Instances, or the sequence number), the MRI of the parent Concept, the version number and the internal identifier (in the format described in section 5.4).

Relations between Instances are managed with the help of the management files. Each management file stores information about each relation that an Instance has with other Instances as well the inverse situation. Each relation has the following set of properties:

Table 5.5 Properties of a relation in an Instance management file

Property	Description
Identifier	An auto-generated identifier to this arc
Target	The Internal Identifier of the target Instance
Type	The type of the relation, can be a manual relation, a relation created automatically via MRI found in content or via content matching in Instances (as described in 3.5)
Concept	The MRI of the parent Concept of the target Instance
Behavior	The behavior of the relation in case a target is updated/removed. See section 4.2.
Relate to Last Version	Attribute that locks the relation with a given version of the target or if the relation should be with the latest version of the target

The management file also includes the list of internal identifiers of every Instance that relates to this Instance, this eases the integrity management when trying to remove an Instance. A list of all namespaces declared in the Instance (and their respective prefix) is stored to ease querying, since XQuery is used and any namespace used in the query must be declared at the beginning of the XQuery expression. An example Instance management file is depicted in Figure 5.16.

The previous figure depicts the management file for an Instance with internal identifier “7.1” and it has a relation with Instance with internal identifier “7.2”. Instance 7.2 also has a relation with “7.1” as can be seen in the *InverseRelations* element, which means there’s a cyclic reference between these two Instances.

```
<InstanceManagementFile>
    <IdentifierName>Ebo_Filter</IdentifierName>
    <ParentConcept>mdr://www.itds.pt/xeoModel&amp;1</ParentConcept>
    <Version>1</Version>
    <InternalIdentifier>7.1</InternalIdentifier>
    <Relations hasLockingRelations="0">
        <Relation>
            <target>7.2</target>
            <type>AutoRel</type>
            <behavior>IgnoreContent</behavior>
            <concept>mdr://www.itds.pt/xeoModel&amp;1</concept>
            <relateLastVersion>true</relateLastVersion>
        </Relation>
    </Relations>
    <InverseRelations>
        <Instance>7.2</Instance>
    </InverseRelations>
    <Namespaces/>
</InstanceManagementFile>
```

Figure 5.16 Sample Instance Management File

5.6. Querying and Transforming

Metadata querying and transforming is implement as described in 4.5 and this chapter will highlight the capabilities of these mechanisms in the metadata repository. As an example, suppose a “Club” concept is in the repository and it has a XML Schema structure like the one depicted in Figure 5.17. A Club has a name (that’s used as an identifier), the name of the country where it’s located and the name of its stadium.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.fct.unl.pt"
    elementFormDefault="qualified">
<xs:element name="Club">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name"></xs:element>
            <xs:element name="Country"></xs:element>
            <xs:element name="Stadium"></xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 5.17 Club Concept XML Schema structure

Stored in the repository are two Instances of this Concept, depicted in Figure 5.18 and Figure 5.19 (Instance *Manchester United* and *Arsenal*, respectively).

```
<ex:Club xmlns:ex="http://www.fct.unl.pt"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ex:Name>Manchester United</ex:Name>
    <ex:Country>England</ex:Country>
    <ex:Stadium>Old Trafford</ex:Stadium>
</ex:Club>
```

Figure 5.18 Instance Manchester United of Concept Club

```
<ex:Club xmlns:ex="http://www.fct.unl.pt"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ex:Name>Arsenal</ex:Name>
    <ex:Country>England</ex:Country>
    <ex:Stadium>Highbury Park</ex:Stadium>
</ex:Club>
```

Figure 5.19 Instance Arsenal of Concept Club

For this example, let's consider the *Manchester United* Instance has a manually created relation with the *Arsenal* Instance and the *Arsenal* Instance is not related with any other Instance (and these are the only relations these Instances have). Figure 5.20 presents a XQuery expression that retrieves every Club Instance, iterates through them, outputs the name, stadium and country of the Club as well as the name of the Club to which it's related to (if it's related).

```
declare namespace ex="http://www.fct.unl.pt";

(: Get the MRI of all Instances of the Club Concepts :)
let $docs:= mdr:getInstanceMRIofConcep("mdr://example.pt/Clubs&amp;1");

<Clubs>
{
    (: Iterate through the MRIS :)
    for $p in $docs

    (: Get the Instance for this Iteration :)
    let $instance := mdr:getInstance($p)
    (: Get the relation of this CLubs with its rival, the MRI of the rival :)
    let $rival := mdr:getRelationsInstance($p)
    (: Get the Instance representing the rival :)
    let $targetInstance := mdr:getInstance($rival/Relations/target/text())

        <Club>
            (:Output the club name :)
            <Name>{$instance/ex:Club/ex:Name/text()}</Name>
            (: Output the club country :)
            <Country>{$instance/ex:Club/ex:Country/text()}</Country>
            (: Output the club stadium :)
            <Stadium>{$instance/ex:Club/ex:Stadium/text()}</Stadium>
            (: Output the club rival :)
            <Rival>{$targetInstance/ex:Club/ex:Name/text()}</Rival>
        </Club>
}
</Clubs>
```

Figure 5.20 XQuery example

XQuery expressions are always executed in the context of the Instances collection (see section 5.4 for details on the storage model) and to separate them from the repository's storage model, a set of XQuery functions is provided in the form of a module that's automatically included by every XQuery Instance. The module is defined in namespace http://mdr.di.fct.unl.pt and mapped to the prefix "mdr" and provides a set of functions to deal with relations, identifiers and document retrieval, the content of the module will be described in 5.6.1. In the example XQuery, the *mdr:getInstanceMRIOfConcept(MRI)* function is used to get the MRIs of all Instances of the Club Concept, while the mdr:getInstance(MRI) function retrieves a given Instance. The

```
<System.Query>
    <Name>HTML.Clubs</Name>
    <Description>Selects every Club and output a HTML visualization, including relations</Description>
    <QueryDefinition>
        declare namespace ex="http://www.fct.unl.pt";

        (: Get the MRI of all Instances of the Club Concept :)
        let $docs := mdr:getInstanceMRIOfConcept("mdr://example.pt/Club&amp;1")
        return

        <Clubs> [24 lines]
    </QueryDefinition>
    <Transforms>
        <Transform>mdr://transform.system.di.fct.unl.pt/Transform/Clubs&amp;1</Transform>
    </Transforms>
</System.Query>
```

Figure 5.21 Instance of the Query System Concept

mdr:getRelations(MRI) function is retrieves the MRIs of Instance to which this Instance relates to. The XQuery expression depicted in Figure 5.20, could be stored in the repository as an Instance of the Query System Concept where it would be related with the Instance of the Transform System Concept identified with the MRI mdr://transform.system.di.fct.unl.pt/Transform/Clubs&1 (which is a XSLT to output HTML), the definition of the Query Instance is depicted in Figure 5.21.

The result of invoking the previous query, using the web service interface and invoking the *executeQuery(QueryName)* method is depicted in Figure 5.22.

```
<Clubs>
    <Club>
        <Name>Manchester United</Name>
        <Country>England</Country>
        <Stadium>Old Trafford</Stadium>
        <Rival>Arsenal</Rival>
    </Club>
    <Club>
        <Name>Arsenal</Name>
        <Country>England</Country>
        <Stadium>Highbury Park</Stadium>
        <Rival></Rival>
    </Club>
</Clubs>
```

Figure 5.22 Result of XQuery execution

Since the result of the query is XML, another method could be invoked in the web service interface to execute the query and pass its results to an associated XSLT (like the one referenced in Figure 5.21) and the result is depicted in Figure 5.23.

Clubs

Name : Manchester United
Country England
Stadium Old Trafford
Rival Arsenal

Name : Arsenal
Country England
Stadium Highbury Park
Rival

Figure 5.23 XSLT applied to the result of a query

5.6.1. Repository Built-in XQuery Functions

Querying is a very important functionality of the Metadata Repository and the use of XQuery was a design choice; to abstract from the storage model of the repository and from the implementation choices regarding internal identifiers, a set of XQuery functions is provided by the repository. These functions are grouped in a XQuery module that's automatically included in each XQuery expression that's executed within the Metadata Repository (with the prefix "mdr"). The list of functions is presented and described in the following table.

Function	Description
getRelationsInstance(MRI)	Given the MRI of an Instance, returns a list of MRIs of Instances that the Instance relates to.
getInverseRelationsInstance(MRI)	Given the MRI of an Instance, returns a list of MRIs of Instances that have a relation with this Instance.
getAllInstances()	Returns all Instances of the repository, similar to the use of the XQuery collection("CollectionName") function.
getInstance(MRI)	Returns a single Instance given it's MRI
getInstancesOfConcept(MRI)	Returns all Instances of a Concept, given the Concept's MRI
getLatestVersionInstances(MRI)	Returns the latest version of each Instance of a Concept, given the Concept's MRI
getInstanceMRIOfConcept(MRI)	Returns the list of MRIs of every Instance of a Concepts, given the Concept's MRI
getInstanceMRILatestVersion(MRI)	Returns the list of MRIs of the latest version of each Instance, given the MRI of the parent Concept

Table 5.6 List of XQuery functions provided by the repository

5.7. Implementation Status

This section lists the implementation status of the features of the Metadata Repository and presents the reasons for the status of each partially or non-implemented feature.

Table 5.7 Implementation status of the features of the Repository

Functionality	Implementation		
	Complete	Partial	Notes
Metadata Storage		X	
Supporting database	X		
Multiple Databases			a)
Storage Model	X		
Batch Storage	X		
System Concepts and Instances		X	b)
Metadata Validation and Integrity		X	c)
Metadata Updates	X		
Metadata Querying and Export	X		
Metadata and Search			d)
Users and Authentication		X	e)

Notes:

a) Multiple database support is a feature that, for the purpose of validating the repository's importing of metadata features and promotion of existing definition, adds no value and due to time constraints it was not implemented.

b) Schematron System Concept and Instances, were not implemented due to time constraints and their value in extra validations were a small gain to the repository.

c) Metadata Validation and Integrity are fully implemented, except for relations using XPath to select part of the content of a target Instance. The automatic relation's behavior of updating the content of the origin Instance in case the target instance is changed is also not implemented, but the necessity of such a feature is arguable as most users will probably want to maintain control over the content of Instances (and who updates it) and not leave it to a metadata repository.

d) Metadata about the elements in the Information Model (Fragments, Concepts, Instances and Relations) was not implemented as well as search functions, due to time constraints and the fact that without a user-friendly graphical interface, searching is a not very important feature as well as it was not the mains focus of the thesis.
e) Users are managed by the database system and authentication is done against the database by the repository, but the repository only stores information about a system user in its configuration files as all other user-related information are stored in the database.

Chapter 6
Validation

This chapter presents validation tests that were performed to assess if the repository complied with the requirements defined as objectives

6.1 Space Environment Support System - SESS…… 120
6.2 ITDS - Xeo…… 129

To provide evidence that the Metadata Repository complies with the requirements listed in section 3.1, two validation tests were conducted: The SESS tests and the ITDS test. The tests involved the creation of Fragment and Concept definitions (including definition of automatic relations based on content) and loading those Fragments and Concepts in the repository as well as loading a set of Instances and afterwards checking if the relations were successfully captured and the external metadata integrated.

6.1. Space Environment Support System - SESS

The Space Environment Support System (SESS) project, previously presented in section 1.2, was developed to monitor space weather and spacecraft phenomenon. In the project, a metadata repository was developed to store and manage metadata. The repository's information model was also based on MOF and featured the notions of Concepts and Instances represented as XML Schemas and XML documents, respectively. To model all of the domain metadata, a set of Concepts was created and Instances were produced as a result of normal system operation. To test if this Metadata Repository complied with the requirements presented in section 3.1, two tests were made using the content of the SESS project. The first test consisted in the integration of the XML Schemas of SESS's Concepts and loading of the XML documents; the second required modeling the set of SESS's Concepts into Fragments and Concepts of this repository and then loading Instances. The list of Concepts used in this validation includes the full list of Concepts that modeled the domain of the project (and are described in

Table 6.1) as well as a small set of Concepts that represented technical metadata (depicted in Figure 6.1).

Table 6.1 List of Concepts from SESS project

Concept	Description	Instances
Ground Base	Stations located on the earth that perform S/W measurements using dedicated instruments	28
Ground Station	Stations located on the earth that are used for transmitting information to or receiving information from a S/C	4

S/C Event	Types of temporal occurrences with the S/C during its operating phase, described start time, end time and value.	0
S/C Parameter	Types of numeric or textual S/C telemetry measures in time, as functions in time – f(t)	113
S/C Position	Types of components of a S/C position in time – f(t)	9
S/W Event	Types of temporal S/W occurrences, described by start time, end time and value.	188
S/W Parameter	Types of single numeric S/W measures in time – f(t), or multiple component S/W measures in time – f1(t), f2(t), f3(t)	226
S/W Parameter Component	Types of component S/W measures in time of a S/W measure – f(t)	174
Space Agency	Space Agencies that operate S/C missions	2
Spacecraft	Spacecraft that performs S/W measures or belongs to S/C missions	8

Domain concepts are related with each other and those relations are expressed through instance relation elements. In Figure 6.1, the relationships between domain Concepts (as well as the relations with the technical Concepts) are depicted.

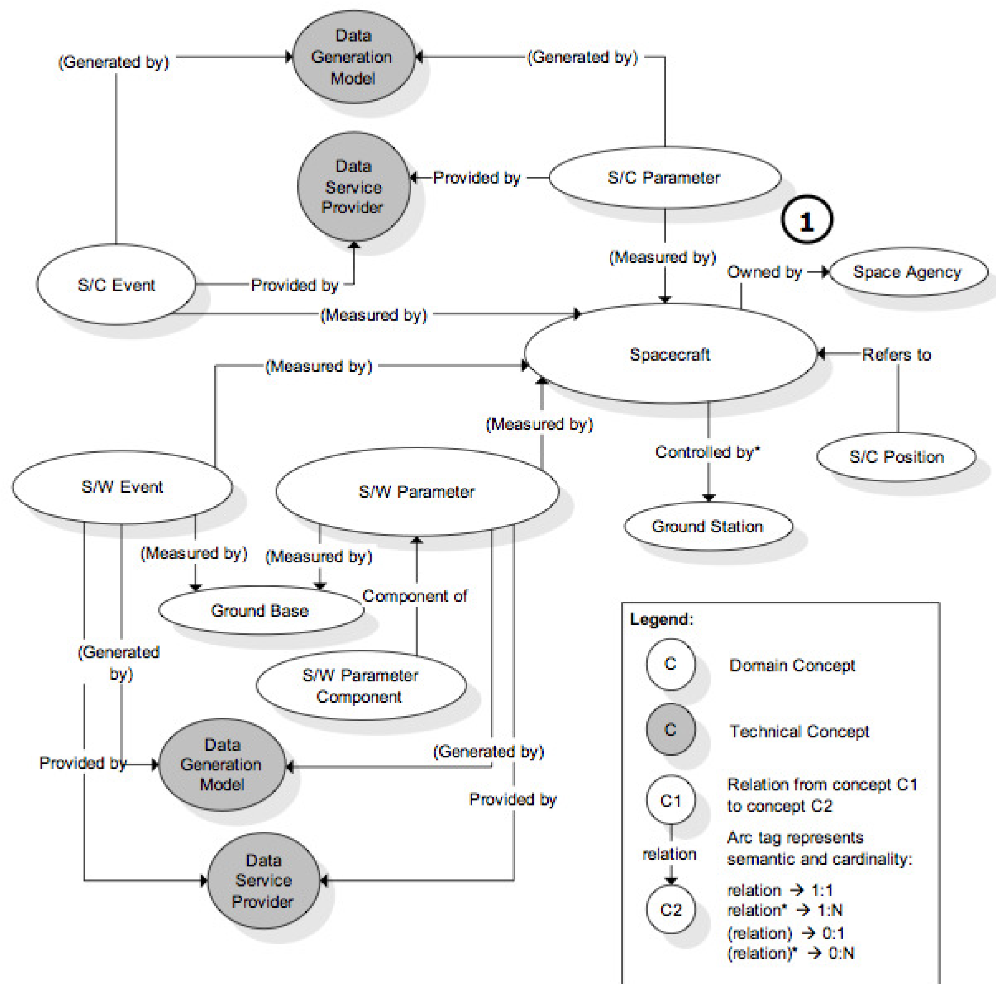

Figure 6.1 SESS domain concepts relationships, taken from [1]

All the concepts and instances of the SESS project were imported in the repository, but for demonstration purposes and to limit the extent of the example, in this chapter the results will be limited to Concepts in the gray area in Figure 6.2.

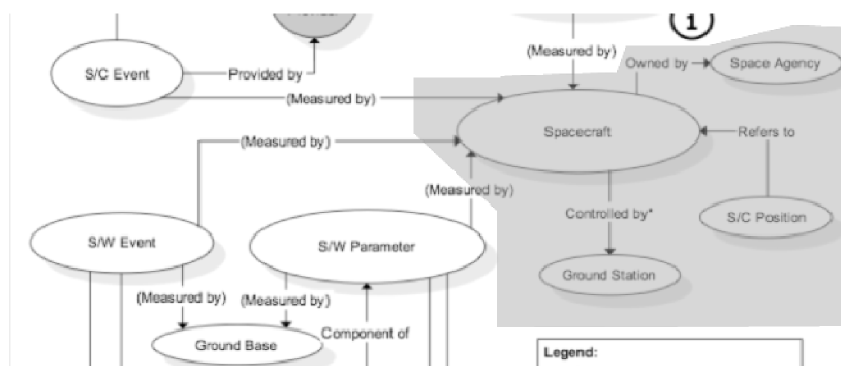

Figure 6.2 SESS Concepts used as an example in import

The example will feature the Spacecraft Concept (related to the Space Agency and Ground Station Concept), the Space Agency Concept, the Ground Station Concept and the S/C Position Concept (related to the Spacecraft Concept), with a total of four concepts and twenty three (23) instances. To test if the repository met the various requirements present in section 3.1, two tests were made, which are described in the following sections.

6.1.1. Standalone Test

To test the capacity of the repository of loading external metadata "as-is", concepts of the SESS project were converted into Concepts of this repository by declaring the structure of each Concept as an embedded schema with the full XML schema definition of the SESS one. An example is in Figure 6.3, where the definition of the GroundStation Concept is depicted. In area "*1" (*of Figure 6.3) the definition of the XPath to identify Instances, because every SESS Instance had a unique *Name* element, which is a very good choice for using XPath as the method to identify Instances. The structure of the Concept is depicted in "*2"* and is the entire schema of the Groundstation XML Schema (not visible in the picture due to size restrictions). The Grounstation Concept defines a relation with the SpaceAgency Concept and in "*3"* the valid target is declared. In "*4"* is the declaration of the automatic rules (XPath) to create of relations in an automatic way.

```
<Concept xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="GroundStation" namespace="sess.uninova.pt">
    <InstanceIdentification>
      1 <Xpath>/ex:GroundStation/mdr:Name/text()</Xpath>
        <NamespaceBinding prefix="ex" uri="http://www.esa.int"/>
    </InstanceIdentification>
    <Structure><GlobalEmbeddedSchema>
            <xs:schema xmlns:mdr="http://di.fct.unl.pt/MetadataRepository" [67 lines]
    </GlobalEmbeddedSchema></Structure> 2
    <Relations>
        <Relation>
            <Targets>
                <Enumeration>
                    <target>mdr://sess.uninova.pt/SpaceAgency&amp;1</target> 3
                </Enumeration>
            </Targets>
            <Cardinality max="1"/>
            <AutoRelContent>
              4 <TargetMatch> [4 lines]
                <LocalInstanceXPath>
                    <Xpath>/ex:GroundStation/ex:SpaceAgencyRelation/mdr:Relation/data(@globalId)</Xpath>
                    <Namespace prefix="ex" uri="http://www.esa.int"/>
                </LocalInstanceXPath>
                <RemoteInstanceXPath>
                    <Xpath>/ex:SpaceAgency/data(@globalId)</Xpath>
                    <Namespace prefix="ex" uri="http://www.esa.int"/>
                </RemoteInstanceXPath>
                <Behavior>
                    <GenerateArc>maintain</GenerateArc>
                </Behavior>
            </AutoRelContent>
        </Relation>
    </Relations>
</Concept>
```

Figure 6.3 Definition of Concept Groundstation from SESS

Concepts were all declared in the same way and loaded in the repository, then Instances were loaded and the relations automatically captured by the XPath rules. The captured relations are depicted in Figure 6.4, where the Instances of each of the four Concepts used for the example are named using the value of the *ShortName* element in their content (each Instance has a unique *ShortName*). The analysis of the figure shows that every SCPosition Instance was related to the same Spacecraft Instance and that the *ESA* Instance of the GroundStation Concept had eight Instances related to it, as opposed to the *Nasa* Instance that only had four relations to it.

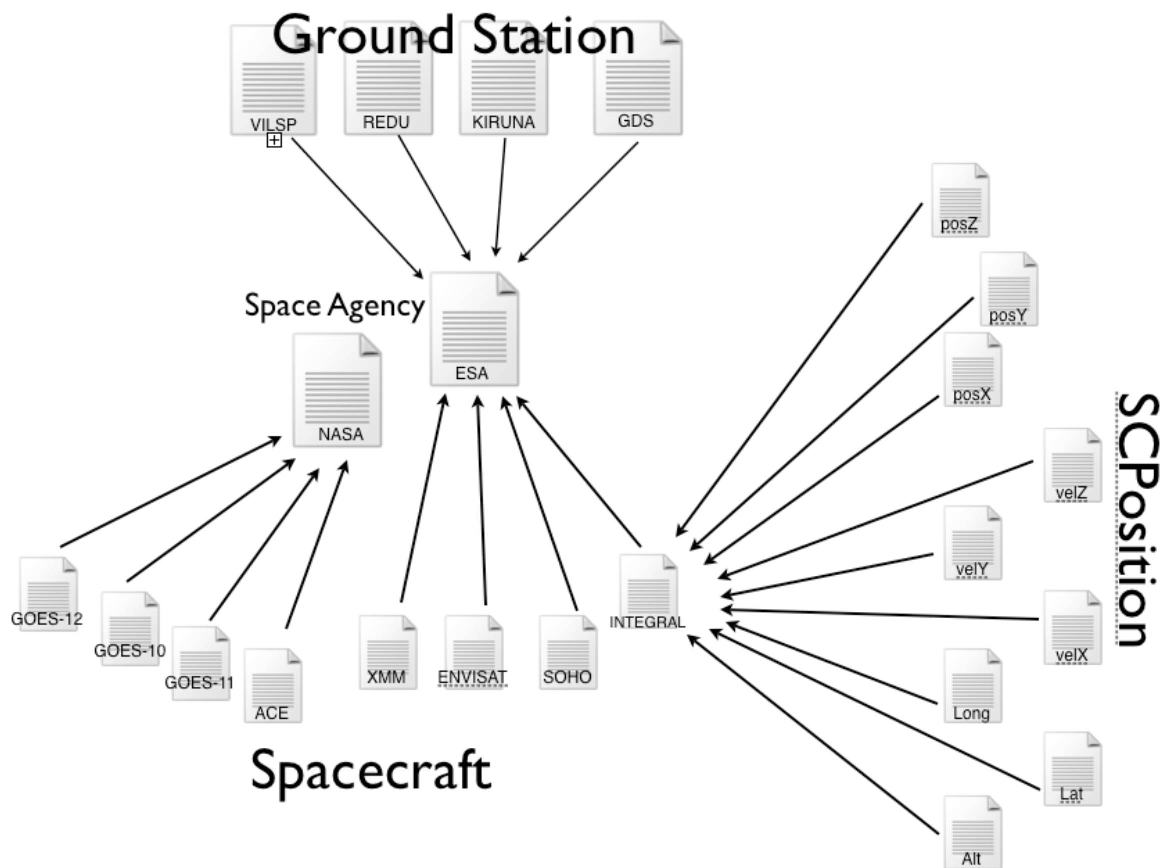

Figure 6.4 Graph of captured relations from Instances of SESS

6.1.2. Reusability Test

To validate the reusability requirement, a test that consisted in converting the same schemas from SESS into Fragments and creating Concepts by reusing those Fragments wherever possible was made. The use of Fragments is better suited in an Information System being built from scratch, where information can be defined as separate parts and be constantly reused as Fragments, but in order to show the use of Fragments the Concepts of SESS were "re-engineered" in order to make use of Fragments. In this chapter, the Concepts used are the same ones used in the previous test (which were chosen because they make use of other schemas and, thus, those schemas are a good choice to be converted in Fragments). To illustrate the dependencies between the existing Concepts, in Figure 6.5 is depicted which schemas import or include other schemas and if those schemas have a target name space or not.

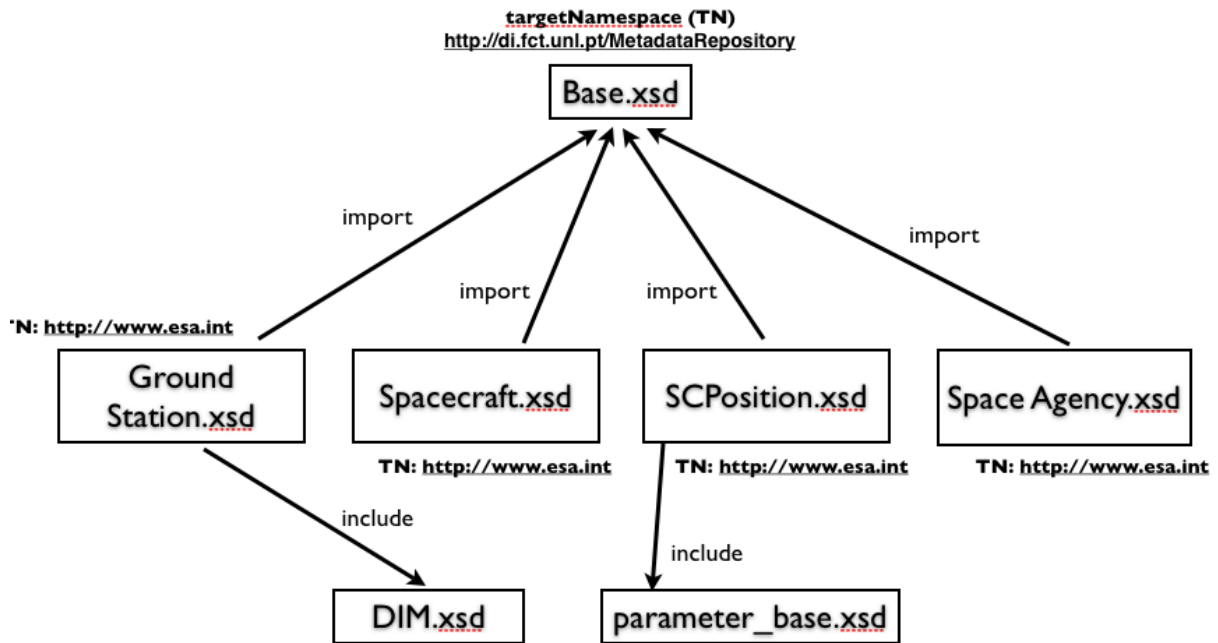

Figure 6.5 Relations between Concepts and included Schemas

Figure 6.5, at the center, features the four Concepts (Ground Station, Spacecraft, SCPosition and Space Agency) used in this example. Each of them imports the base XML schema, as required by the SESS repository's rules, and the Ground Station concept includes the *DIM* schema, while the SC Position concept, includes the *parameter_base* schema. After analyzing the relations between the schemas (and their content) the choice was to transform *Base.xsd*, *DIM.xsd* and *parameter_base.xsd* into Fragments and using the composition method to recreate the four Concepts, including locally embedded schema to use the elements that are part of each Concept and weren't part of any of the included/imported schemas.

Fragments

The build process of Fragments was simple, as none of the schemas (DIM, base and parameter_base) included/imported other schemas. Each Fragment was given a name equal to their schema name and all of them associated to the repository namespace "sess.uninova.pt" which means the following MRI's were associated to the Fragments:

- *mdr://sess.uninova.pt/DIM (Dim.xsd)*
- *mdr://sess.uninova.pt/baseMdr (base.xsd)*
- *mdr://sess.uninova.pt/parameterBase (parameter_base.xsd)*

Each Fragment's structure was defined by using the embedded schema option, as depicted in Figure 6.6 (example for the DIM Fragment, the same was done with the other two).

```
Fragment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="DIM" namespace="sess.uninova.pt">
   <Structure>
       <GlobalEmbeddedSchema>
           <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
           <xs:element name="Location">
               <xs:annotation> [2 lines]
               <xs:complexType>
                   <xs:sequence>
                       <xs:element name="GeodeticCoordinates" minOccurs="0"> [26 lines]
                       <xs:element name="Orientation" minOccurs="0"> [30 lines]
                       <xs:element name="CityName" type="xs:string"> [4 lines]
                       <xs:element name="CountryName" type="xs:string"> [4 lines]
                   </xs:sequence>
               </xs:complexType>
           </xs:element>
           </xs:schema>
       </GlobalEmbeddedSchema>
   </Structure>

/Fragment>
```

Figure 6.6 Fragment definition of the DIM Fragment

Concepts

Each Concept in the example was built in the same way. The structure created was a composition of already existing Fragments with the inclusion of local embedded schema where needed. In Figure 6.7, the definition of the GroundStation Concept is depicted and will be described in the following paragraphs.

The Instance identification method is the same as used in the previous chapter, using XPath and taking advantage of the fact that each Instance had a unique name (as seen in the area marked by "1" in Figure 6.7). The structure of the Concept is a composition whose wrapper element is the *GroundStation* element, this is required as all Instances of the SESS Ground Station Concept had this element as its root element and, as explained in the Concept Definition Language (4.2.2), this wrapper element will be converted in a XML Schema element with the same name. The same target namespace of the SESS Concept is used in the Concept definition (which can be seen in the area marked by "2"). The definition of the Structure is highlighted in areas 3,4 and 5 in the figure and is built as a sequence of elements. The sequence starts by reusing two group elements (*identificationElementGroup* and *documentElementsGroup*, area 3) from the baseMdr Fragment and then use locally embedded XML schema to include an element that was part of the GroundStation

Concept (marked in area 4 and the same is true for the following *SpaceAgencyRelation* element). Area 5 depicts the reuse of the DIM Fragment, referencing the *Location* element. Finally, in area 6, Fragment baseMdr is reused again, but this time the Concept definition is reusing an attribute group definition.

```
<Concept xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="GroundStation" namespace="sess.uninova.pt">
    <InstanceIdentification>
        <Xpath>/ex:GroundStation/mdr:Name/text()</Xpath>                1
        <NamespaceBinding prefix="ex" uri="http://www.esa.int"/>
    </InstanceIdentification>
    <Structure>
        <GlobalComposition rootElement="GroundStation" targetNamespace="http://www.esa.int" targetNamespacePrefix="pt"> 2
            <Sequence>
                <Schema rootElement="identificationElementsGroup" type="group">
                    <Reference>mdr://sess.uninova.pt/baseMdr</Reference>
                </Schema>
                <Schema rootElement="documentationElementsGroup" type="group">   3
                    <Reference>mdr://sess.uninova.pt/baseMdr</Reference>
                </Schema>
                <Schema rootElement="GroundStationNumber" type="element">
                    <LocalEmbeddedSchema>
                        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
                        <xs:element name="GroundStationNumber" type="xs:integer"> [5 lines]
                        </xs:schema>
                    </LocalEmbeddedSchema>                                        4
                </Schema>
                <Schema rootElement="SpaceAgencyRelation" type="element">
                    <LocalEmbeddedSchema> [22 lines]
                </Schema>
                <Schema rootElement="Location" type="element">
                    <Reference>mdr://sess.uninova.pt/DIM</Reference>           5
                </Schema>
            </Sequence>
            <Attributes>
                <attributeFragGroup refFrag="mdr://sess.uninova.pt/baseMdr" name="identifierAttributesGroup"/>  6
            </Attributes>
        </GlobalComposition>
    </Structure>
</Concept>
```

Figure 6.7 Groundstation Concept Definition

In Figure 6.7 there are no declaration of relations (as were presented in the Concept definition in Figure 6.3) but only because of space restrictions, they were present when that definition was loaded into the repository.

The result of loading the definition in the repository is a XML Schema, that's depicted in Figure 6.8

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:mdr="http://di.fct.unl.pt/MetadataRepository"
    xmlns:pt="http://www.esa.int"
    targetNamespace="http://www.esa.int"
    elementFormDefault="qualified">
    <xs:import  schemaLocation="sess.uninova.pt.baseMdr.1.xsd" namespace="http://di.fct.unl.pt/MetadataRepository"/>
    <xs:include schemaLocation="sess.uninova.pt.DIM.1.xsd"/>                                                          1

    <xs:element name="GroundStationNumber" type="xs:integer"> [5 lines]
    <xs:element name="SpaceAgencyRelation" type="mdr:relationType"> [18 lines]      2

    <xs:element name="GroundStation">
        <xs:complexType>
            <xs:sequence>
                <xs:group ref="mdr:identificationElementsGroup"/>                  3
                <xs:group ref="mdr:documentationElementsGroup"/>
                <xs:element ref="pt:GroundStationNumber"/>
                <xs:element ref="pt:SpaceAgencyRelation"/>                         4
                <xs:element ref="pt:Location"/>
            </xs:sequence>
            <xs:attributeGroup ref="mdr:identifierAttributesGroup"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 6.8 Result of converting a Concept definition in XML Schema

In the area marked as "1" the import and includes of the Fragments is done, while in area 2 is the set of elements that were included with the use of local embedded Schema. Area 3 shows the reference to elements from the baseMdr Fragment and area 4 the reference to elements locally embedded (*GroundStationNumber* and *SpaceAgencyRelation*) and referenced of the DIM Fragment (*Location*). After the conversion of each Concept and Fragment they were loaded in the repository and each Instance was also loaded in the repository, with every relation being captured in the same automatic way as it was previously described. The graph of relations between instances is depicted in Figure 6.4.

Evaluation

The purpose of the test was to evaluate importing of external metadata "as-is" and the use of Fragments to promote information reuse. Although the content from SESS was not particularly suited for this, it was possible to concert the content in Fragments and Concepts and still loading Instances with no trouble. Relations between Instances were also automatically captured and stored in the repository and, thus, the repository is able to deal with the requirements such as metadata storage, import, validity, integrity, relationships and reuse.

6.2. ITDS - Xeo

ITDS (Internet, Tecnologias e Desenvolvimento de Software) [119] is a Portuguese software company that has created and developed the XEO platform. XEO stands for eXtensible Enterprise Objects and is a platform for business modeling and applications development that allows professional teams of business analysts and users to develop and maintain complex business

applications with much less effort and risk, simply by modeling their real world's business requirements in the form of business objects.

ITDS uses XML to store XEO's business objects, which are basically metadata for their framework. The framework uses objects as input and transforms them in an application according to their content. ITDS was a company that was contacted during this thesis for feedback about the repository's features and capabilities and they kindly agreed in supplying a small set of their XML objects to validate the metadata repository as a solution for metadata integration.

All of XEO's objects are instances of one XML Schema, which was used to create a Concept in the repository. Objects are related to each other and those relations are reflected in the XML content of those objects, making them a candidate for the automatic capturing of relations, based on content. ITDS supplied a set of one hundred and forty three (143) objects that are part of the base of their framework and these objects we imported as Instances of the Concept previously created (which includes the definition of automatic relations).

To load XEO's objects in the repository a *xeoModel* Concept was created under the namespace *www.itds.pt* and the structure of the Concept was the embedded schema supplied by ITDS as seen in Figure 6.9 (content of the schema omitted).

```
<Concept name="xeoModel" namespace="www.itds.pt">
    <InstanceIdentification>
        <Xpath>/xeoModel/general/@name</Xpath>
    </InstanceIdentification>
    <Structure>
        <GlobalEmbeddedSchema>
        <xsd:schema xmlns:xm="xeoModel" xmlns:xsd="http://www.w3.
        </GlobalEmbeddedSchema>
    </Structure>
```

Figure 6.9 xeoModel Concept definition

The content of a XEO object has an attribute that is unique among all objects and, as such, is the natural choice for identifying objects with XPath as seen in the previous figure (*Xpath* element). As an example, the full MRI of object *Ebo_Flag* is *mdr://www.itds.pt/xeoModel/Ebo_Flag*. Since XEO objects relate only to other XEO objects, the definition of a relation is done with a single target Concept (itself) and several automatic relation rules, since it can relate to other objects in several ways. A subset of the relation definition is depicted in Figure 6.10.

```
<Relations>
    <Relation>
        <Targets>
            <Enumeration>
                <target>mdr://www.itds.pt/xeoModel</target>
            </Enumeration>
        </Targets>
        <AutoRelContent>
            <TargetMatch> [4 lines]
            <LocalInstanceXPath><Xpath>/xeoModel/general/implements/interface/text()</Xpath></LocalInstanceXPath>
            <RemoteInstanceXPath><Xpath>/xeoModel/general/data(@name)</Xpath></RemoteInstanceXPath>
            <Behavior> [2 lines]
        </AutoRelContent>
        <AutoRelContent>
            <TargetMatch> [4 lines]
            <LocalInstanceXPath><Xpath>/xeoModel/general/objects/object/text()</Xpath></LocalInstanceXPath>
            <RemoteInstanceXPath><Xpath>/xeoModel/general/data(@name)</Xpath></RemoteInstanceXPath>
            <Behavior> [2 lines]
        </AutoRelContent>
        <AutoRelContent>
```

Figure 6.10 xeoModel relation definition sample

The relation definition in the previous figure is only part of the definition and features two automatic relation rules, of the total of five declared in the full Concept definition. After the insertion in the repository, the number of relations was counted and ascended to one hundred and sixty eight (168) relations between the set of objects.

6.2.1. Evaluation

The test case with XEO objects required that the repository imported external metadata (of a type that is not available anywhere) stored the metadata after its validation and captured automatic relationships between objects. This situation allows to validate the repository in some of the requirements presented in section 3.1, such as 1,2,3,4 and 6. Making the metadata repository a good solution for the integration of external, real-world, data.

Chapter 7

Conclusions and Future Work

This chapter draws final conclusions on the design and implementation of this thesis and presents future work activities

7.1 Conclusions………………………………………………………………………………………………. 134
7.2 Future Work……………………………………………………………………………………………. 136

This chapter draws final conclusions on the design and implementation of this thesis and presents future work.

7.1. Conclusions

The metadata repository, whose design and implementation is presented in this thesis, was built to comply with several requirements in metadata management (as listed in chapter 3.1) with strong emphasis on integration of external metadata and reusability. By complying with the requirements, the metadata repository becomes a solution for metadata management in any small or medium enterprise and it was designed to be the support for other systems. It has a lightweight architecture with a single web service interface so that other systems and management applications can connect to, using the stored metadata as source of input with quality guarantees.

The use of an adequate information model combined with XML technologies to represent, validate and process metadata provides the repository with the extensibility and flexibility to ensure compliance with several of the requirements. The absence of restrictions imposed to documents eases the integration process and the use of a XML language to define the properties of elements in the M2 layer of the Information Model enables a separation between the XML vocabulary defined by the elements and the properties items in that layer can have inside the repository. This means documents can be imported "as-is", without any effort of conversion. The use of Fragments (and XSLT templates associated to them) provides the mechanism to reuse available knowledge and avoid building definitions from scratch when they could be reused, thus, promoting reusability and being an advantage in the adoption of the repository. Still part of the information model, the repository features a relationships mechanism that allows representing dependencies between objects in the real world, by enabling the creation of connections between the metadata that represents these real world objects and maintaining the integrity of these relations through time (while assuring the metadata is valid at all times).

To store metadata, the repository acts as a database management system and since XML technologies are extensively used, a native XML database was chosen as the support for persistent storage. Once the metadata is stored the repository provides management features such as a versioning system for each element of the information model (except the M3 layer) or querying and exporting mechanisms (via XQuery and XSLT) to enable sharing and exporting information with other systems or applications.

The promotion of reusability is extended to the XSLT field, as Fragments can have associated XSLT templates that can be reused by other XSLTS (via a special processing instruction) or by Instances of the System Concept "Generic Transform" which creates on-the-fly XSLT to process Instances based on the structure of a Concept and on available XSLT associated to Fragments that are used by the Concept. Queries are defined in XQuery (stored in the repository as Instances of a System Concept)

and can be linked to XSLTs or XSLT Pipelines so that applications connecting to the repository can request the execution of a query and subsequent XSLT execution over the results of the query, by just invoking a method in the API, specifying the query name and the transformation name associated to that query.

The metadata repository provides features that ease the integration of external metadata and captures relations from an already existing context by content matching in documents. This feature, however, has a severe penalty in performance, as the number of documents to be queried can be very high, if the number of target Concepts is considerable or if the total number of Instances of target Concepts is considerable. Automatic relations are a powerful feature, but using it in systems with heavy load or with a high number of documents can lead to performance degradation. It is up to the users of the repository to assess if the performance penalty is acceptable or not.

Fragments are at the core of the reusability features of the repository as they allow reusing already existing elements to create new definitions. However, the reusability mechanisms in the repository are “element-oriented” i.e. it’s possible to explicitly reuse elements, but XML Schema also has a property of defining “types” of elements that other elements can reuse, this mechanism is used when creating a new element and its structure is already defined in a “type”. Fragments are only built to reuse elements and not to create elements reusing structure (also because, in this way XSLT templates would be more difficult to associated to a Fragment, as XSLT are mainly designed to process elements and not “type” definitions).

Metadata about Instances, Concepts, Fragments and relations was not implemented due to time constraints and the lack of a graphical tool that supported search by end-users, despite the presence of search mechanisms in any repository application being essential as stated in [17]: “*It can therefore be concluded that the ability to find information is not just a “nice to have” but it drastically affects the bottom line*”. The lack of such a tool makes any search mechanism to users rather useless and to applications it makes no difference, as common searches are keyword-based and no meaning or context is associated to them, as such, they would have great difficulty to process the results of such a search. The Sedna database does not have a built-in XQuery full-text extension, but the database engine implements the XQuery function “contains” that can be used for simple searches and supports the integration of the *dtsearch* [120] module, which provides a full-text search extension to XQuery. The *dtsearch* module features several kinds of searches, such as word searching, boolean searching, wildcard, phonic searching, fuzzy searching and synonym search. As a result, the repository is prepared to support competent search mechanisms and in the event a graphical tool to manage the repository content is produced it can make the interface to these features.

Despite the problems described earlier, the metadata repository designed and implemented in this thesis was able to integrate metadata from external sources (SESS and ITDS) and capturing relations in an automatic way from the content of those sources. A “Fragment-version” of the SESS project in

order to achieve a higher degree of reusability was also designed and loaded successfully in the repository. The result of these tests is a reason to think of the repository as a viable solution for the support of information systems that base their integration strategy in a metadata solution.

7.2. Future Work

This section presents future activities for the work described in this thesis, for the design and implementation of the metadata repository. Regarding these activities the following points are suggested.

Performance – Implementation of caching mechanisms to enhance performance of the repository in several operations, such as the generation of XML Schemas, generation of XSLTs that reuse Fragments XSLTs or results of XQuery queries.

Management Console – For the use of the repository in an Information System, a standalone graphical management console (preferably a web-based one) is required so that users can manage the content of the repository.

Multiple-Database – Support for multiple databases, which can be used to support different systems with only one instance of the metadata repository.

Metadata and Search – Implement the storage of metadata about the elements in the Information Model (Instances, Concepts, Fragments and Relations) and provide a search mechanism, preferably associated to the management console.

Fault tolerance mechanisms – The metadata repository is designed to be a support for an Information System and high availability is required. This would require implementing fault-tolerance mechanisms and testing them in real world situations, with repositories in distinct geographic locations.

Concept and Fragment Editor – Fragments and Concepts are defined using a XML language. To abstract users of this design choice by providing a graphical tool to “build” Fragments and Concepts, using the repository as a source of information. The ability to easily reuse Fragments to build Fragments and Concepts would be another step in the promotion of reusability.

In the context of this thesis, the software-house ITDS showed interest in the metadata repository as a solution to manage their business objects library (XML objects) in order to help documents and reuse those objects, because of the lack of centralized source of these objects leads to the creation of new objects from scratch. Their interest went beyond and a meeting with the purpose of discussing the use (and customization) of the metadata repository as a tool to support their framework is in agenda.

References

1. Martins, R.F., *Extensible Metadata Repository for Information Systems*, in *Department of Computer Science*. 2007, Universidade Nova de Lisboa: Monte da Caparica. p. 200.
2. Schmidt, A., et al., *XMark: a benchmark for XML data management*, in *Proceedings of the 28th international conference on Very Large Data Bases*. 2002, VLDB Endowment: Hong Kong, China.
3. David Marco, M.J., *Universal Metadata Models*. 2004: Wiley Computer Publishing.
4. Tannembaum, A., *Metadata Solutions - Using Metamodels, Repositories, XML and Enterprise Portals to Generate Information on Demand*. 2002: Adisson-Wesley.
5. Vaduva, A. and K.R. Dittrich, *Metadata Management for Data Warehousing: Between Vision and Reality*, in *Proceedings of the International Database Engineering & Applications Symposium*. 2001, IEEE Computer Society.
6. SOA. *Service Oriented Architecture* 2008; Available from: http://www.opengroup.org/projects/soa/.
7. BPEL4WS. *Business Process Execution Language for Web Services*. 2008; Available from: http://www.ibm.com/developerworks/library/specification/ws-bpel/.
8. BPML. *Business Process Modeling Language*. 2008; Available from: http://www.ebpml.org/bpml.htm
9. ESA. *Space Environment Support System for Telecom/Navigation Missions (SESS)*. 2005; Available from: http://telecom.esa.int/telecom/www/object/index.cfm?fobjectid=20470.
10. Ferreira, R., et al., *XML Based Metadata Repository for Information Systems*, in *EPIA 2005 - 12th Portuguese Conference on Artificial Intelligence*. 2005: Covilhã, Portugal.
11. Marco, D., *Building and managing the Meta Data Repository: A Full Life-Cycle Guide*. 2000: John Wiley & Sons, Inc. 416.
12. XML. *eXtensible Markup Language*. 2008; Available from: http://www.w3.org/XML/
13. Sun-Microsystems. *Java EE at a Glance*. 2008; Available from: http://java.sun.com/javaee/. .
14. OMG. *Object Management Group - MetaObject Facility (MOF)*. 2008; Available from: http://www.omg.org/mof/.
15. OMG. *Object Management Group*. 2008; Available from: http://www.omg.org.
16. Schematron. *A language for making assertions about the presence or absense of patterns in XML documents*. 2008; Available from: http://www.schematron.com/.
17. Inmon, W., B. O'Neil, and L. Fryman, *Business Metadata: Capturing Enterprise Knowledge*. 2007: Morgan Kaufmann Publishers.
18. Murphy, L.D., *Digital Document Metadata in Organizations: Roles, Analytical Approaches, and Future Research Directions*, in *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 2 - Volume 2*. 1998, IEEE Computer Society.
19. Wootton, C., *Developing Quality Metadata: Building Innovative Tools and Workflow Solutions*. 2007: Focal Press.
20. Harold, E.R., *XML 1.1 Bible*. 2004: John Wiley & Sons.
21. SGML. *W3C. Standard Generalized Markup Language Overview*. 1995; Available from: http://www.w3.org/MarkUp/SGML/.
22. W3C. *World Wide Web Consortium*. 2006; Available from: http://www.w3.org.
23. HTML. *HyperText Markup Language - W3C*. 2006; Available from: http://www.w3.org/MarkUp/.
24. Namespaces. *Namespace in XML 1.0 (W3C)*. 2006; Available from: http://www.w3.org/TR/REC-xml-names/.
25. Berners-Lee, T., R. Fielding, and L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax*. 1998: RFC Editor.
26. DC, *Dublin Core Metadata Initiative.* 2008.
27. McDonough, P., *METS: standardized encoding for digital library objects.* Int. J. Digit. Libr., 2006. **6**(2): p. 148-158.
28. *XML Schema - Part 0: Primer Second Edition*. 2004; Available from: http://www.w3.org/TR/xmlschema-0.
29. Vlist, E.v.d., *XML Schema*. 2002: O'Reilly Media, Inc.

30. XMLSpy, A. *XML editor for modeling, editing, transforming, & debugging XML technologies*. Available from: http://www.altova.com/products/xmlspy/xml_editor.html.

31. Oxygen. *XML Editor and XSLT Debugger*. Available from: http://www.oxygenxml.com/.

32. LibXML. *The XML C parser and toolkit of Gnome*. Available from: http://xmlsoft.org/.

33. Xerces. *Java Parser*. Available from: http://xerces.apache.org/xerces-j/.

34. Tools, X.S. *List of XML Schema Tools (W3C)*. 2008; Available from: http://www.w3.org/XML/Schema#Tools.

35. *RELAX NG, a schema language for XML*. 2008; Available from: http://relaxng.org/.

36. *DTD – Document Type Definition*. Available from: http://www.w3.org/TR/REC-xml/#dt-doctype.

37. OASIS. *Relax NG Validators*. 2009; Available from: http://relaxng.org/#validators.

38. *Skeleton - An Implementation of Schematron 1.5 in XSLT*.

39. *XML Path Language (XPath) 2.0*. 2006; Available from: http://www.w3.org/TR/xpath20/.

40. Tidwell, D., *XSLT: Mastering XML Transformations*. 2007: O'Reilly Media, Inc.

41. *XSL Transformations (XSLT) Version 2.0*. 2006; Available from: http://www.w3.org/TR/xslt20/.

42. Holzner, S., *Inside XSLT*. 2001: New Riders Publishing. 616.

43. *XQuery 1.0: An XML Query Language*. 2006; Available from: http://www.w3.org/TR/xquery/.

44. Walmsley, P., *XQuery*. 2007: O'Reilly Media, Inc.

45. Evjen, B., et al., *Professional XML*. 2007: Wrox Press Ltd.

46. *XML:DB. XML:DB Initiative: XUpdate - XML Update Language*. 2000; Available from: http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html.

47. *XML:DB. XML:DB Initiative for XML Databases*. 2003; Available from: http://xmldb-org.sourceforge.net/.

48. Chamberlin, D.D., et al., *XQuery Update Facility 1.0*, W.W.W. Consortium, Editor. 2008.

49. XQilla. *XQuery and XPath 2.0 Library*. 2008; Available from: http://xqilla.sourceforge.net/XQueryUpdate.

50. MonetDB. *Database system with XQuery front-end*. 2008; Available from: http://monetdb.cwi.nl/XQuery/.

51. *Semantic Web*. 2006; Available from: http://www.w3.org/2001/sw/.

52. Daconta, M.C., K.T. Smith, and L.J. Obrst, *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*. 2003: John Wiley & Sons, Inc. 281.

53. Miles, A., et al., *SKOS core: simple knowledge organisation for the web*, in *Proceedings of the 2005 international conference on Dublin Core and metadata applications: vocabularies in practice*. 2005, Dublin Core Metadata Initiative: Madrid, Spain.

54. Antoniou, G. and F. vanHarmelen, *A Semantic Web Primer*. 2004: MIT Press.

55. *OWL Web Ontology Language Use Cases and Requirements*. 2004; Available from: http://www.w3.org/TR/webont-req/

56. Silberschatz, A., H.F. Korth, and S. Sudarshan, *Database Systems Concepts*, ed. B.T. Allen. 1997: McGraw-Hill, Inc. 821.

57. *Resource Description Framework (RDF)*. 2004; Available from: http://www.w3.org/RDF/.

58. Beckett, D. *New Syntaxes for RDF*. 2003; Available from: http://www.dajobe.org/2003/11/new-syntaxes-rdf/paper.html.

59. *An XML Syntax for RDF: RDF/XML*. Available from: http://www.w3.org/TR/REC-rdf-syntax/#rdfxml.

60. *RDF Schemas and Namespaces*. Available from: http://www.w3.org/TR/PR-rdf-syntax/#schemas.

61. *RDF Validation Service*. 2007; Available from: http://www.w3.org/RDF/Validator/.

62. *RDF Vocabulary Description Language 1.0: RDF Schema*. 2004; Available from: http://www.w3.org/TR/rdf-schema/.

63. *SPARQL Query Language for RDF*. 2006; Available from: http://www.w3.org/TR/rdf-sparql-query/.

64. *OWL Web Ontology Language*. 2004; Available from: http://www.w3.org/TR/owl-features/.

65. SKOS. *Simple Knowledge Organization System*. 2004; Available from: http://www.w3.org/2004/02/skos/.

66. *Common XML vocabularies*. 2008; Available from: http://www.service-architecture.com/xml/articles/common_xml_vocabularies.html.

67. Oracle. *Semantic Technologies Center*. Available from: http://www.oracle.com/technology/tech/semantic_technologies/index.html.

68. *MySQL Xml Functions*. 2008; Available from: http://dev.mysql.com/doc/refman/5.1/en/xml-functions.html.

69. *PostgreSQL 8.2.9 Documentation – XML Document Support*. 2008; Available from: http://www.postgresql.org/docs/8.2/static/datatype-xml.html.

70. *XML Support in Microsoft SQL Server 2005*. 2005; Available from: http://msdn.microsoft.com/en-us/library/ms345117.aspx.

71. OWL. *Web Ontology Language Guide*. 2004; Available from: http://www.w3.org/TR/owl-guide/.

72. Motik, B. and S. Grimm. *Closed World Reasoning in the Semantic Web through Epistemic Operators*. in *OWL: Experiences and Directions*. 2005. Galway, Ireland.

73. Andrew, A.M., *Rough-Neural Computing: Techniques For Computing With Words, ed. by Sankar Kumar Pal, Lech Polkowski and Andrzej Skowron, Springer, Berlin, 2004, xxv+734 pp., ISBN 3-540-43059-8, Cognitive Technologies Series, ISSN 1611-2482 and Modelling With Words: Learning, Fusion, and Reasoning Within a Formal Linguistic Representation Framework, ed. by Jonathan Lawry, Jimi Shanahan and Anca Ralescu, Springer, Berlin, 2003, xi+228 pp., ISBN 3-540-20487-3, LNAI Series no. 2873, ISSN 0302-9743.* Robotica, 2004. **22**(6): p. 698-699.

74. Horrocks, I. *OWL Rules, OK?* in *Rule Languages for Interoperability*. 2005. Washington, DC, USA.

75. Horrocks, I., et al. *Semantic Web Architecture: Stack or Two Towers?* in *Principles and Practice of Semantic Web Reasoning*. 2005: Springer.

76. Noy, N.F., *Semantic integration: a survey of ontology-based approaches.* SIGMOD Rec., 2004. **33**(4): p. 65-70.

77. Noy, N.F. *What do we need for ontology integration on the semantic web, position statement*. in *Workshop on Semantic Integration, jointed held with the 2nd International Semantic Web Conference*. 2003. Sanibal Island, Florida, USA.

78. Klein, M. *Combining and Relating Ontologies: An Analysis of Problems and Solutions*. in *Workshop on Ontologies and Information Sharing, IJCAI*. 2001. Seattle, WA.

79. Uschold, M. and M. Gruninger, *Ontologies and semantics for seamless connectivity.* SIGMOD Rec., 2004. **33**(4): p. 58-64.

80. *Repository in a Box*. 2006; Available from: http://icl.cs.utk.edu/rib/.

81. *Reuse Library Interoperability Group - The Basic Interoperability Data Model*. 1995; Available from: https://kspace.cdvp.dcu.ie/repository/doc/bidm.html.

82. MIT. *DSpace Federation*. 2006; Available from: http://www.dspace.org/.

83. *Mckoi SQL Database*. 2004; Available from: http://mckoi.com/database/.

84. *Dspace on Windows*. Available from: http://wiki.dspace.org/index.php/DSpaceOnWindows.

85. MIT. *DSpace System Documentation: Functional Overview*. 2006; Available from: http://dspace.org/technology/system-docs/functional.html.

86. *Dspace Repository Users*. 2008; Available from: http://www.dspace.org/index.php?option=com_content&task=view&id=596&Itemid=180.

87. *The Protégé Ontology Editor and Knowledge Acquisition System*. 2006; Available from: http://protege.stanford.edu/.

88. *What is Protégé? A Protégé Overview*. 2008; Available from: http://protege.stanford.edu/.

89. *Open Knowledge Base Connectivity*. 1995; Available from: http://www.ai.sri.com/~okbc/.

90. *What is protégé-frames? A Protégé Overview*. 2008; Available from: http://protege.stanford.edu/overview/protege-frames.html.

91. *What is protégé-owl? A Protégé Overview*. 2008; Available from: http://protege.stanford.edu/overview/protege-owl.html.

92. Jena. *A Semantic Web Framework for Java*. 2006; Available from: http://jena.sourceforge.net/.

93. *Fedora Digital Repository System*. 2008; Available from: http://www.fedora.info/.

94. *Introduction: Basic Concepts in Fedora*. 2008; Available from: http://www.fedora.info/download/2.2.1/userdocs/tutorials/tutorial1.pdf.

95. *Fedora Information Page*. 2008; Available from: http://www.fedora.info/documents/brochure/Fedora%20Page%20Final.htm.

96. *Fedora Development Team, Fedora White Paper*. 2005; Available from: http://www.fedora.info/documents/WhitePaper/FedoraWhitePaper.pdf.

97. CA. *Computer Associates AllFusion Repository for Distributed Systems* 2007; Available from: http://www.ca.com/us/products/default.aspx?id=1439.

98. SAS. *SAS - Metadata Server*. 2007; Available from: http://www.sas.com/technologies/bi/appdev/base/metadatasrv.html.

99. Fielding, R.T., *Architectural styles and the design of network-based software architectures*. 2000, University of California, Irvine. p. 162.

100. Bourret, R. *XML Database Products*. 2007; Available from: http://rpbourret.com/xml/XMLDatabaseProds.htm.

101. *Database 11g | Oracle Database 11g | Oracle*. 2008; Available from: http://www.oracle.com/database/index.html.

102. *SQL Server 2008 Overview, data platform, store data | Microsoft*. 2008; Available from: http://www.microsoft.com/sqlserver/2008/en/us/default.aspx.

103. *MySQL :: The world's most popular open source database*. Available from: http://www.mysql.com/.

104. *PostgreSQL: The world's most advanced open source database*. Available from: http://www.postgresql.org/.

105. *eXist Open Source Native XML Database.*

106. Meier, W., *eXist: An Open Source Native XML Database*, in *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*. 2003, Springer-Verlag.

107. Meier, W. *Index-driven XQuery processing in the eXist XML database*. in *XML Prague*. 2006. Prague, Czech Republic.

108. *JBoss*. Available from: https://www.jboss.org/.

109. *Apache Tomcat - An Open Source JSP and Servlet Container from the Apache Foundation*. 2009; Available from: http://tomcat.apache.org/.

110. *OASIS eXtensible Access Control Markup Language*. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

111. *Sedna XML Database*. Available from: http://modis.ispras.ru/sedna/.

112. Fomichev, A., M. Grinev, and S.D. Kuznetsov, *Sedna: A Native XML DBMS*, in *SOFSEM 2006: Theory and Practice of Computer Science*. 2006, Springer. p. 272-281.

113. Fomichev, A., M. Grinev, and S.D. Kuznetsov. *Descriptive schema driven XML storage*. in *Advances in Databases and Information Systems (ADBIS)*. 2004. Budapest, Hungary.

114. *XQuery Test Suite Result Summary*. Available from: http://www.w3.org/XML/Query/test-suite/XQTSReportSimple.html.

115. Lehti, P., *Design and Implementation of a Data Manipulation Processor for an XML Query Language*. 2001, Technische Universitat Darmstadt. p. 82.

116. *Oracle Berkeley DB XML*. Available from: http://www.oracle.com/database/berkeley-db/xml/index.html.

117. Srivastava, A.V., *Comparison and Benchmarking of Native XML Databases*, in *Department of Computer Science and Engineering*. 2004, Indian Institute of Technology: Kanpur. p. 6.

118. Mabanza, N., J. Chadwick, and G.S.V.R.K. Rao. *Performance evaluation of Open Source Native XML databases - A Case Study*. in *International Conference on Advanced Communication Technology*. 2006.

119. *ITDS - Internet, Tecnologias e Desenvolvimento de Software*. 2008; Available from: http://www.itds.pt/.

120. *dtSearch - Text Retrieval / Full Text Search Engine*. 2008; Available from: http://www.dtsearch.com/.