Tiago André Rolo Teixeira

Licenciado em Ciências de Engenharia
Electrotécnica e de Computadores

Organising Interoperability Information on Highly Dynamic and Heterogeneous Environments

Dissertação para obtenção do Grau de Mestre em
Engenharia Electrotécnica e de Computadores

Orientador : Adolfo Steiger Garção, Professor Catedrático, FCT/UNL

Co-orientador : Pedro Miguel Maló, Professor Assistente, FCT/UNL

Júri:

Presidente: Doutor Tiago Oliveira Machado de Figueiredo Cardoso - FCT/UNL

Arguente: Doutor Manuel Martins Barata - ISEL/IPL

Vogais: Doutor Adolfo Steiger Garção - FCT/UNL
Mestre Pedro Miguel Maló - FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2012

Organising Interoperability Information on Highly Dynamic and Heterogeneous Environments

Ao meu pai, mãe
e irmão

Acknowledgements

Este espaço é dedicado a todos aqueles que me acompanharam durante a elaboração desta tese. Para todos eles, o meu mais sincero obrigado!

Primeiro, gostaria de agradecer a toda a minha família, e em especial aos meus pais e irmão pela compreensão, carinho e muita paciência durante todo este tempo. Ao Telepe, Né e ao meu afilhado Martim, por todas os momentos de descontração e diversão. À Adriana, por toda a amizade, amor, dedicação e por ter estado sempre presente em todos os momentos ao longo da minha formação.

Ao Pedro Maló e ao Bruno Almeida, por me terem possibilitado fazer este trabalho de investigação, por me terem orientado na direcção certa, pela objetividade e exigência. Ao meu companheiro de tese Márcio Mateus, por todas as horas perdidas a discutir ou divagar, e principalmente pela disponibilidade que sempre demonstrou para contribuir para a conclusão desta dissertação.

À malta do GRIS, Baeta, Casanova, Fábio, Edgar, Raquel, Hugo, e a todos os colegas da FCT, Vitor, Gonçalo, Manta, Catarina, Zé Manel, Marcelo, Edu, Chalaça, Vanessa e David.

Muito obrigado a todos.

Abstract

The "*Internet of Things*" is a dynamic global network infrastructure where physical and virtual "*things*" communicate and share information amongst themselves. Plug and Interoperate is an approach that allows heterogeneous "*things*" to plug (into data) and seamlessly exchange information within the environment. To allow that, Plug and Interoperate needs to have the comprehension about the existing interoperability information. For this, the interoperability information needs to be duly organised. However, and in the "*Internet of Things*", this presents major challenges. First, it is difficult to index all interoperability information due to the "*things*" heterogeneity (many and different languages and formats) and due to the dynamics of the system (disparate things entering/leaving the environment at all times). Also, that the environment can be used with much different purposes, which hinders the way on how the interoperability information should be organised. So, an architecture of an Interoperability Repository System is presented, in order to organise all interoperability information in this kind of environments. The solution handles heterogeneous interoperability information and allows users to add a User Space to the repository in order to customise it to specific needs. It also provides a notification mechanism in order to notify users of new or updated interoperability information.

Keywords: Internet of Things, Interoperability, Heterogeneity, Repository

Resumo

A "*Internet of Things*" é uma infraestrutura de rede dinâmica e global, onde "*things*" físicas e virtuais comunicam e trocam informação entre si. O "*Plug and Interoperate*" é uma abordagem que permite que "*things*" heterogéneas se liguem, e troquem informações dentro do ambiente sem problemas. Para o permitir, o "*Plug and Interoperate*" precisa do conhecimento sobre a informação relativa à interoperabilidade existente. Portanto, é necessário que essa informação relativa à interoperabilidade esteja devidamente organizada. No entanto, esta organização apresenta grandes desafios no ambiente da "*Internet of Things*". Primeiro, porque é difícil indexar toda a informação relativa à interoperabilidade, devido à heterogeneidade das "*things*" (muitas e diferentes linguagens e formatos) e devido ao dinamismo do sistema (diferentes "*things*" estão continuamente a entrar e sair do ambiente). Além disso, este tipo de ambiente pode ser usado com muitos e diferentes propósitos, o que dificulta a maneira como a informação relativa à interoperabilidade deve ser organizada. Assim, é apresentada uma arquitetura de um Repositório de Interoperabilidade, a fim de organizar toda a informação relativa à interoperabilidade neste tipo de ambiente. A solução apresentada manipula informação de interoperabilidade heterogénia e permite aos utilizadores adicionar um "*User Space*" ao repositório a fim de personalizá-lo às suas necessidades específicas. O repositório também disponibiliza um mecanismo de notificação, para notificar os utilizadores quando surgir nova ou atualizada informação relativa à interoperabilidade.

Palavras-chave: *Internet of Things*, Interoperabilidade, Heterogeneidade, Repositório

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
AM3	Atlas MegaModel Management
API	Application Programming Interface
ATL	Atlas Transformation Language
CDO	Connected Data Objects
CORBA	Common Object Request Broker Architecture
CRUD	Create, Read, Update, Delete
DB	DataBase
DB4O	DataBase for Objects
DFS	Deph-First Search
DSTC	Distributed Systems Technology Center
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
GME	Generic Modeling Environment
GMM	Global Management Model
GMT	Generative Modeling Technologies
GRIS	Group for Research in Interoperability of Systems
GUI	Graphical User Interface
HQL	Hibernate Query Language
HSQLDB	HyperSQL DataBase
ICT	Information and Communication Technologies
IoT	Internet of Things

iRM	integrated Repository Manager
IRS	Interoperability Repository System
IT	Information Technologies
JDBC	Java Database Connectivity
JMI	Java Metadata Interface
LAN	Local Area Network
MDI	Model Driven Interoperability
MDR	MetaData Repository
MIC	Model Integrated Computing
MIM	Model for Interoperability Management
MLS	Meta Language Space
MODL	Meta Object Description Language
MOF	Meta Object Facility
MS	Modeling Space
mSQL	model Structured Query Language
OMG	Object Management Group
OSI	Open System Interconnection
PnI	Plug and Interoperate
RMS	Repository Management System
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SQL	Structured Query Language
TTCN	Tree and Tabular Combined Notation
UML	Unified Modeling Language
WAN	Wide Area Network
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivating scenario: Plug and Interoperate

Internet of Things (IoT) refers to the general idea of "*things*", especially everyday objects, that are readable, recognizable, locatable, addressable, and/or controllable via the Internet whether via wireless sensor networks, wireless LAN - Local Area Network, WAN - Wide Area Network, or other means. Everyday objects includes not only the electronic devices we encounter everyday, and not only the products of higher technological development such as vehicles and equipment, but "*things*" that we do not ordinarily think of as electronic at all such as food, clothing, house materials, car parts, city landmarks and monuments; and all the miscellany of commerce and culture (IERC, 2011; NIC, 2008).

IoT is based on standards and interoperable communication protocols, such as IEEE 802.11 (Wireless LAN), IEEE 802.15 (Bluetooth), etc. where physical and virtual "*things*" have identities, physical attributes, virtual personalities, use intelligent interfaces and are seamlessly integrated into the information network (CERP-IoT, 2008). This means IoT is a dynamic, resource limited and heterogeneous environment, where different "*things*" need to communicate and share information among themselves, i.e. they need to interoperate.

Interoperability *"is the ability of two or more systems or components to exchange information and to use the information that has been exchanged"* (IEEE, 1990) and is a key challenge in the realms of the *Internet of Things* (CERP-IoT, 2010). As an example, shown in figure 1.1 lets consider an environment where the temperature in the truck and the refrigerator trailer is being monitored. To do that, within the truck and trailer there are several disparate temperature sensors,

from different manufacturers, reading in different scales, using different languages, etc. These temperature sensors are in this example considered as smart objects[1]. So, the Truck Unit will receive the temperature from the smart objects in order to monitor the temperature within the truck cargo. However, due to the heterogeneity of the smart objects, the Truck Unit is unable to read and use the temperature information correctly, i.e. they can not fully interoperate.

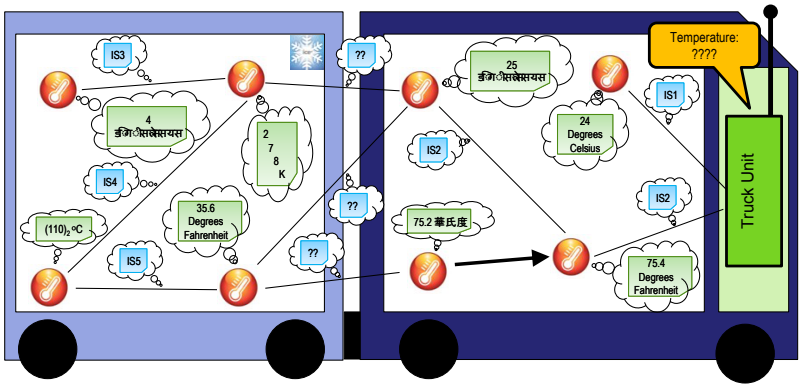

Figure 1.1: Example: Monitoring temperature in truck and trailer

One solution is that sensor manufacturers could add support for new data formats in the devices, but this would imply that the sensors need to be remanufactured. A preferable solution would be to have some kind of mechanism where sensors could simply plug (as they are, i.e. without suffering any changes) and seamlessly interoperate within the environment, i.e. the Plug and Interoperate. Plug and Interoperate (PnI) concept has been coined within the research group at UNINOVA-GRIS, allowing "*things*" to plug (into data) and seamlessly exchange information within the environment. One need to understand that the focus of PnI is in the data domain, and not how to physically connect the smart objects to the middlewares (a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems (Bakken, 2001)).

PnI is realised by the use of interoperability artefacts, which comprehends all the information that can aid different "*things*" achieving interoperability. Interoperability artefacts are a set of: Interoperability information models, Interoperability specifications, Languages and Tools. Using an example, such as the Eclipse Modeling Framework (Steinberg, Budinsky, Paternostro, & Merks, 2008), which have a model driven approach, they can be defined as:

1. Interoperability Information models: information model is an abstract, formal representation of entity types that includes their properties, relationships and the operations that can be performed on them (e.g. data formats). In EMF they are both source and target entity, and are expressed in EMF;

2. Interoperability specifications: Interoperability specification is information relating similarities / differences between two data formats, i.e. information on how to interoperate two

[1] a machine, system, sensor or device equipped with electronic control mechanisms and capable of automated and seemingly intelligent operation

data formats. ATL could be used as the interoperability specification because it does the mapping between the two entities within EMF;

3. Languages: Languages are systems of signs, symbols, gestures, or rules used in communicating, that, in this case, describe both information models and interoperability specification. The EMF framework includes a language (Ecore) for describing models;

4. Tools: tools are implements to work with interoperability artefacts, such as virtual machines. EMF framework uses a Virtual Machine as a tool to execute the data transfer using ATL.

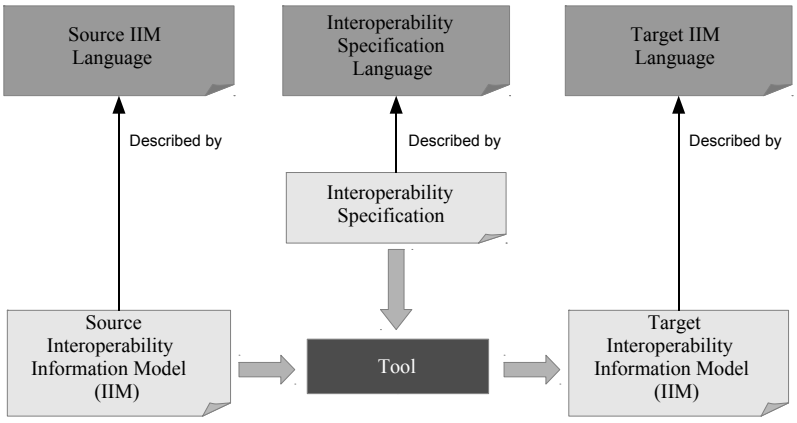

Figure 1.2: PnI Interoperability Artefacts

PnI also aids the integration of new smart objects in a system. It enables smart objects to be connected to the system without the need of being remanufactured, because manufacturers only need to provide interoperability specifications of their own hardware. This way, interoperability is assured at the middleware level with technology independence via methods that enforce interoperability to be implemented where it is needed (from embedded to high-end systems). The concept goes even to a next level of supporting interoperability between sources that are not explicitly defined in the system, either by generating and composing new interoperability specifications using existing ones, or by acquiring new interoperability specifications via sharing mechanisms.

1.2 Problem: Managing Interoperability Artefacts

An issue within the PnI scenario, is that within this heterogeneous environment there are many devices with disparate data formats, described by different languages. There may be also several different possible interoperations between data formats, which can also be described by different languages. Handling all the heterogeneity within this kind of environment means that systems need to know what data formats are supported and what interoperations are available within the environment. So, there is the need to comprehend the possible interoperations between systems and provide this as a service.

There is the need to work with all kinds of information models, interoperability specifications and the languages that describe them, known so far, and be able to deal with new interoperability artefacts whose data formats and languages are still unknown. In order to achieve it, there is the need to acknowledge what interoperability specifications exist within the environment, and which languages they refer to. So, all the available interoperability specifications between known data formats, need to be stored within the environment. This place would also need to be reusable and updatable so new specifications could be added.

Another characteristic of PnI is the need to generate and compose new interoperability specifications using existing ones. So it is needed some type of processing on top of the interoperability artefacts. Using a specific case, one could be able to find all the paths between two different information models, using available interoperations. These results, could then enhance the interoperability specifications available. They can also represent all the interoperability information using any graphical representation schemes deemed appropriate. This type of environment is in constant change, due to new smart objects keep entering the environment. This characteristic leads to the need that somehow the integrity of the environment needs to be kept, by updating the interoperability specifications being used by the smart objects.

This leads to the natural research question, which supports this master thesis work:

How to organise interoperability artefacts in PnI environments?

This problem presents a set of characteristics that need to be addressed:

- Heterogeneity: In data systems, heterogeneity is considered an unwelcome feature because it proves to be an important obstacle for the interoperation of systems (Gruber et al., 1995; Sciore, Siegel, & Rosenthal, 1994). The lack of standards is an obstacle to the exchange of data between heterogeneous systems (Visser, Jones, Bench-Capon, & Shave, 1997). One of the problem characteristics is the need to handle heterogeneous information models and the interoperations between them. The heterogeneity of the information models, interoperability specifications and languages that describe them, leads to the need to represent this information in a way that enables any user or application to consult it independently of its nature.
- Abstraction: There is the need to organise the interoperability artefacts with a high level of abstraction so it could be used with several different purposes. This may enable, for example a thorough exploration of all paths (set of interoperability specifications) between information models. If at some point there is the need to interoperate two different formats that don't have a direct interoperability specification between them, this type of processing would retrieve paths between the two formats, making the interoperability between those two formats possible. As an example, the use of analytic processing may be important because they do a mathematical analysis to determine behaviours, errors, representations and other things possible to calculate mathematically.

- Integrity: The need to keep the interoperability specifications updated within the environment is another characteristic of this problem. The environment needs to be offered the possibility to add / update interoperability artefacts whenever there is a new / better one, i.e. the environment needs to be updated when there is a updated / newer version of the interoperability specification being used. For example, one device enquires for an interoperability specification between two data formats, if the interoperability specification previously used is updated, then the device should be informed that a newer version is available.

1.3 Work Approach

The master thesis work approach is based on the Scientific Method, composed by the following steps (Schafersman, 1997):

1. Characterize the Problem;
2. Do a Background Research;
3. Formulate Hypothesis;
4. Setup an Experiment;
5. Test Hypothesis through an Experimentation;
6. Hypothesis Validation;
7. Publish Results.

The work approach steps are depicted in figure 1.3, and are defined and explained as following:

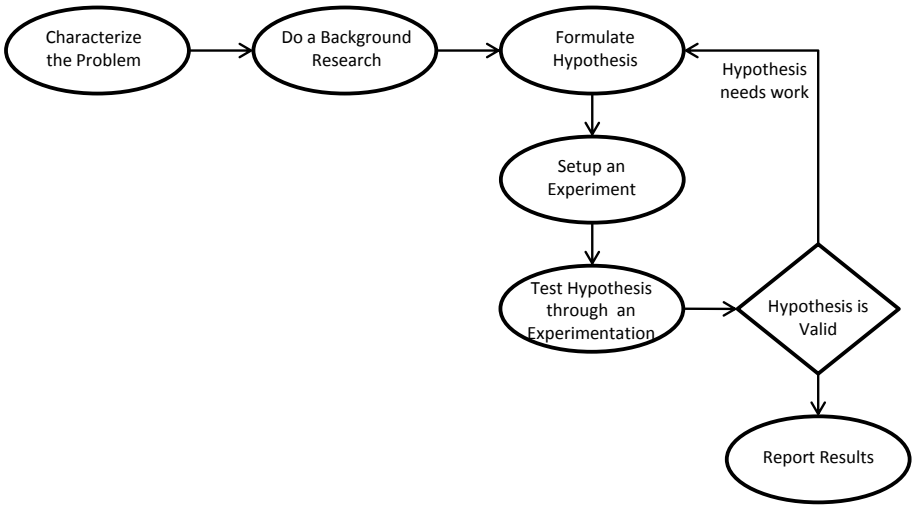

Figure 1.3: Overview of the Work approach

1. **Characterize the Problem:**

   It defines the "area of interest", the problem, its characteristics and the Research Question that drives this master thesis work. The problem in this dissertation is how to manage interoperability artefacts in IoT environments.

2. **Do a Background Research:**

   This step is based on the study of prior work on the subject, such as similar work that handles this type of information. So, it is necessary to gather information about the characterized problem. This means that there is the need to gather scientific information about the existing work that handle interoperability artefacts in IoT-like environments.

3. **Formulate Hypothesis:**

   Hypothesis states the "predicted" (as an educated guess) relationship amongst variables and is stated in a declarative form, brief and straight to the desired point. The hypothesis formulated in this master thesis work, serves to bring clarity, specificity and focus to the problem of managing interoperability artefacts in IoT environments. It is required that the hypothesis solves the presented problem.

4. **Setup an Experiment:**

   This phase includes all the detailed planning and execution of the experimental phase, which in this case is composed by the design of a technological architecture for managing interoperability artefacts in IoT environments. Since the hypothesis must be validated, it is necessary to setup an experiment, which can be replicated by others in a feasible way, and so, a proof of concept was implemented.

5. **Test Hypothesis through an Experimentation:**

   Firstly, a test battery should be defined taking into account the characteristics of the problem and the formulated hypothesis. In order to evaluate the hypothesis proposed, it is necessary to evaluate the outcomes of the system / architecture designed. The hypothesis needs to be tested using the designed experimentation. For each test, data should be collected for further analysis and hypothesis validation. After all tests applied and data outputs collected, it is time to interpret and analyse the results. If applicable, qualitative and quantitative data analysis should be applied to the results.

6. **Validate Hypothesis:**

   After the analysis of the experimentation results, it is necessary to verify the validity of the hypothesis purposed. This validation needs to take into account the problem characteristics. The results can lead to weakening of the confidence of the hypothesis, or even put in jeopardy all of the assumptions made in the very beginning of the research. This should not be interpreted as a failure, but as a way to improve the original approach and try another one with new expertise of the subject, re-iterating from step 3.

7. **Report Results:**

   This is the step where, when positive results are attained, is possible to consider the future and define the recommendations for further research. Discussion regarding literature, research objectives and questions should be taken into account, and draw conclusions out of it. The outcome of solids results should result in a contribution to the scientific community.

Accordingly to the type of research, scientific papers should be written to present intermediate results (e.g. in conferences), consolidated results (e.g. in journals), and finalised with a dissertation about the hypothesis, such as this one.

1.4 Dissertation Outline

This dissertation is divided in five chapters, starting with this introduction. Where it was presented a motivating scenario, the Plug and Interoperate, then the problem was presented and characterized, which drove to the research question: "How to organise interoperability artefacts in PnI environments". Afterwards, it was presented the approach used to do this work, which was based on the scientific method. So, following the scientific method based approach, the following chapters are:

2. **Related Work: Study of Model-driven Repositories** The second chapter presents the related work elements studied, which are systems that can handle interoperability artefacts. In the end of the chapter, there is an analysis which demonstrate the usefulness of each element in accordance to the characterized problem defined in the first chapter. It is also presented the contribution of each element to this work, i.e. which features were considered important and were then used in the creation of the system architecture.

3. **Interoperability Repository System:** The third chapter presents the architecture created in order to support the interoperability repository system. Firstly it presents the concept behind the architecture created, using some examples. Secondly, it is presented the three layered logical architecture with its logical modules, which are then defined and their methods explained. The chapter ends with the detailed architecture, where one can see the "full picture" (layers, logical modules and methods) of the created architecture.

4. **Testing and Validation:** This chapter presents the tests used to validate the formulated hypothesis. It begins by describing the adopted methodology used to test the hypothesis. It also describes the implemented proof of concept and both the tests definition and execution. Afterwards, the results of testing phase are presented. Finally, it is verified if the initial objectives set forth this dissertation were achieved, through an analysis of the test results.

5. **Conclusions and Future Work:** In this chapter, it is presented a summary of this dissertation, presenting the contributions of this work. It also presented a potential direction for future research, regarding the obtained results.

2

Related Work: Study of Model-driven Repositories

2.1 Review

Having defined all the problem characteristics, there is the need to understand the current state-of-the-art elements that can handle interoperability artefacts. This section presents several systems that can handle interoperability artefacts, which were studied and will be used as a base in order to achieve the objective (Organising Interoperability artefacts in IoT environments).

In order to present relevant related work elements, a research was made, in order to identify systems, technologies and/or approaches that handle interoperability artefacts. This research, along with the GRIS work group experience, revealed the interoperability state-of-research is of Model Driven Interoperability (MDI), which has the advantage of being easily deployable in heterogeneous systems, and the mapping between formats is technology independent (Bézivin, Soley, & Vallecillo, 2010). Model Driven can be characterized by the use of models to represent elements in a system, model transformations to represent relations between models and metamodels which are the models meta-concepts. In this specific case, the models are used for representing the existing information models, the model transformations to represent interoperability specifications between different information models and the metamodels to represent the languages that describe each one. This representation is technology independent, and the translators for applying this models to technological ones (e.g. XML) can be implemented without knowing the specific format to use.

2.1.1 Individual Review

The systems found and presented in this chapter are model based, i.e. they handle model driven concepts. The state-of-the-art elements are (listed in alphabetical order):

- Atlas MegaModel Management (AM3): is the only model-based repository found that enable the representation of Model Transformations, i.e. relationships between information models (ATLAS, 2011);
- Connected Data Objects (CDO): it was studied because it is a distributed shared Model framework, which can be used as a model repository (Eclipse, 2011);
- DTSC dMOF: dMOF was also chosen because of the engines used to query the data model (DSTC, 2000);
- Generic Modeling Environment (GME): GME was also studied due to its persistence methods. GME uses a database model as a storage mechanism (Ledeczi et al., 2001);
- Integrated Repository Manager (iRM): this repository was chosen due to mSQL query engine used to query the repository (Petrov, Jablonski, Holze, Nemes, & Schneider, 2004).
- Sun Netbeans Metadata Repository (MDR): it was chosen due to its persistence methods, which allows B-tree file based persistence storage (Matula, 2003);

Atlas MegaModel Management (AM3)

One of the systems found, was the AM3 (Atlas MegaModel Management) (ATLAS, 2011), which is part of the Generative Modeling Technologies (GMT), the official research incubator project of the top-level Eclipse Modeling Project (EMP) (Foundation, 2011). EMP focuses on the evolution and promotion of model based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations, and so it is widely used by the model driven community. AM3 objective is to deal with global resource management in a model-engineering environment. The AM3 environment is a tool based on the Global Model Management (GMM) (Allilaire, Bézivin, Brunelière, & Jouault, 2006) approach, which is based on several general concepts (including the OMG reference architecture). AM3 offers a set of services for handling and querying models, such as: megamodel view constructor; megamodel query language; model locator facility and a model identifier facility, which can be depicted in figure 2.1. The AM3 environment also provides storage mechanisms and uses some existing model manipulation tools.

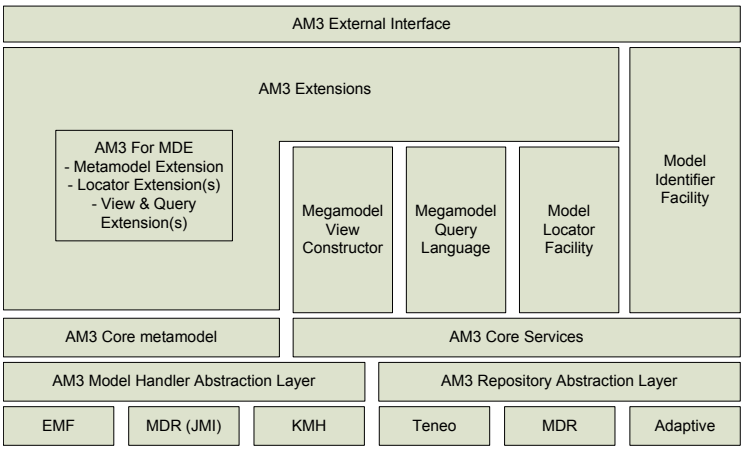

Figure 2.1: Atlas MegaModel Management logical architecture (MODELPLEX, 2007)

Analysis:

AM3 is a system based on the GMM representation approach. The GMM approach enables the representation of heterogeneous models, as long as they obey to the defined specifications. Although, it suites better on an environment such as Model Driven Software Development. Analysing the AM3 tool, some specific services were noticed: it provides a language to query the model, using a model handler abstraction layer; a view constructor, which enables the creation of a view of a specific part of the model; an identifier and locator facility, which is important to identify a specific model and to locate it. It also possesses the AM3 Extensions facility, which may enable the addition of processing tools, that may run algorithms on top of the information within AM3. Another important aspect of AM3, is that although the model storage method is strong related to the model locator's type, the authors of AM3 did not link AM3 to any specific storage method, enabling that it could be used in different environments. This may also be important due to the heterogeneity within the environment. Some storage mechanisms may be better suited for some type of information, and so the information may be stored in the storage mechanism deemed appropriate. They created a repository abstraction layer which enables AM3 to use different repository implementations.

Connected Data Objects (CDO)

The Connected Data Objects (CDO) Model Repository (Eclipse, 2011) is a distributed shared model framework for EMF models and metamodels. CDO is also a model runtime environment with a focus on orthogonal aspects like model scalability, transactionality, persistence, distribution, queries and more. The CDO server consists of a set of framework components, which are represented in figure 2.2. Each component manages a particular aspect and communicates with the storage back-end through a pluggable storage adapter. The storage back-end is pluggable and migrations between direct JDBC, Hibernate, Objectivity/DB, MongoDB or DB4O are seamless for CDO applications. The main functionalities of CDO are (Eclipse, 2011): Persistence, the persistence of models in all kinds of database back-ends like major relational databases or NoSQL

databases; Multi-User access, Multi-user access to models is supported through the notion of repository sessions; Transactional Access, Transactional access to models with ACID properties is provided by optimistic and/or pessimistic locking on a per object granule. Transactions support multiple save points that changes can be rolled back to; Scalability, the ability to store and access models of arbitrary size, is transparently achieved by loading single objects on demand and caching them softly in the end-user application; Data integrity, which can be ensured by enabling optional commit checks in the repository server such as referential integrity checks and containment cycle checks, as well as custom checks implemented by write access handlers.

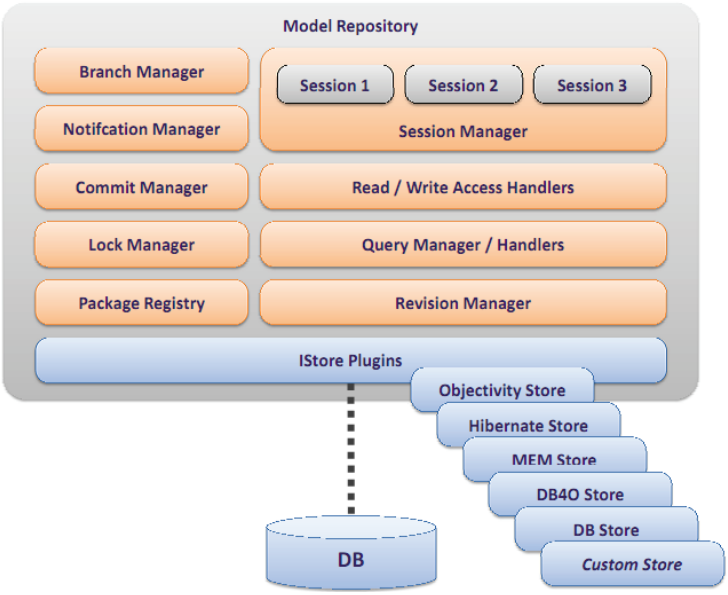

Figure 2.2: CDO server architecture (Eclipse, 2011)

Analysis:

The CDO model repository is another repository system within the Eclipse Modeling Framework, which allows the sharing of an EMF model. This presents an issue in order to deal with heterogeneous artefacts, which is the fact that it mainly supports model artefacts that conform to the Ecore metamodel. However, CDO presents several storage mechanisms due to the fact that each client accesses a server which shares the same model instance. Some of the storage mechanisms, referred as Stores in the CDO server architecture (figure 2.2) are (Eclipse, 2011): Mem store, which stores without real persistence, the server cannot be restarted; Db store, store that connects via JDBC to a relational database and manages revisions and models through a built-in object-relational mapper; Hibernate store, store that uses Teneo/Hibernate, which provides superior runtime Object Relational Mapping, HQL (Hibernate Query Language) and the TENEO's automatic mapping of an Ecore model to a relational database schema. CDO provides data integrity functionalities within and with the Notification Manager, it may notify users upon changes in specific models and with it, keep the user information also updated. This model repository does not possess the ability to process algorithms within its architecture.

DSTC dMOF

Another system is dMOF (DSTC, 2000), which is a repository developed by Distributed Systems Technology Center - DTSC. dMOF is a shared (the metadata in a dMOF repository can be shared among multiple repository applications), server-based (dMOF is CORBA based) MOF repository. Its architecture is depicted in figure 2.3. The metamodel is expressed in a MODL (Meta Object Description Language) file, which is a text based neutral format and similar to *C* language. Then it uses a tool called *modl2mof* to generate the MOF representation.

Initially dMOF has been designed as main-memory repository, but in version 1.1 a database persistence server has been added (Petrov & Buchmann, 2008). Its transaction model is JDBC based that give access to a SQL database introduced with the persistence service. The dMOF repository API comprises several sets of interfaces used to handle different model views. The *MultiRepository* (DSTC, 2000) interface allows an application to view (using the Repository-View manager) the models as conforming to different metamodels, the *SimpleRepository* (DSTC, 2000) interface provides a single metamodel view of the model metadata. The repository interfaces also handle naming and naming contexts.

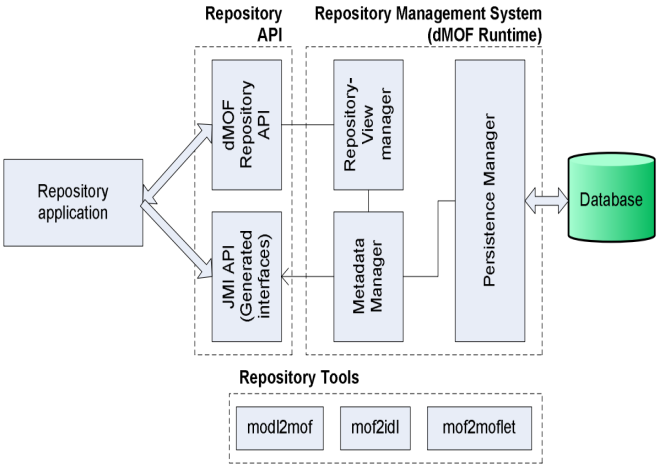

Figure 2.3: Architecture of DSTC dMOF (Petrov & Buchmann, 2008)

Analysis:

dMOF is yet another MOF based repository, which indicates that it may only represent models with the MOF representation approach, i.e. only models confirming to the MOF metamodel. However it provides tools that allow the use of different languages, such as MODL. dMOF situation is similar to the MDR situation, i.e. few information is available besides a user guide (DSTC, 2000) and some random articles that are also based on the same user guide. This repository has a very interesting module which is the Repository-View manager which is believed to enable the creation of views form the data model. The persistence service that uses JDBC allows dMOF to automatically save and restore the state of metadata from disc. dMOF uses a regular database to persist the information within.

Generic Modeling Environment (GME)

One other system, called Generic Modeling Environment (GME) (Ledeczi et al., 2001) was studied. This system is based on models, and provides a flexible framework, originated in the School of Engineering of the Vanderbilt University (USA), to address essential needs of embedded systems. It is part of the Model-Integrated Computing (MIC) (Karsai, Sztipanovits, Ledeczi, & Bapty, 2003), which focuses on the formal representation, composition, analysis, and manipulation of models during the design process. It places models in the center of the entire life-cycle of systems, including specification, design, development, verification, integration, and maintenance (ISIS, 2011).

The GME approach, and its architecture is presented in figure 2.4. It is used for creating domain-specific modeling and program synthesis environments. GME allows a modeller to visually manipulate underlying model data structures. It possesses a graphical user interface (GUI) at the top, a model Browser, Add-ons and a constraint manager. This system also has a persistence Storage Model database and a storage interface. The storage interface includes components for different storage formats (e.g. XML format and a fast proprietary binary file format).

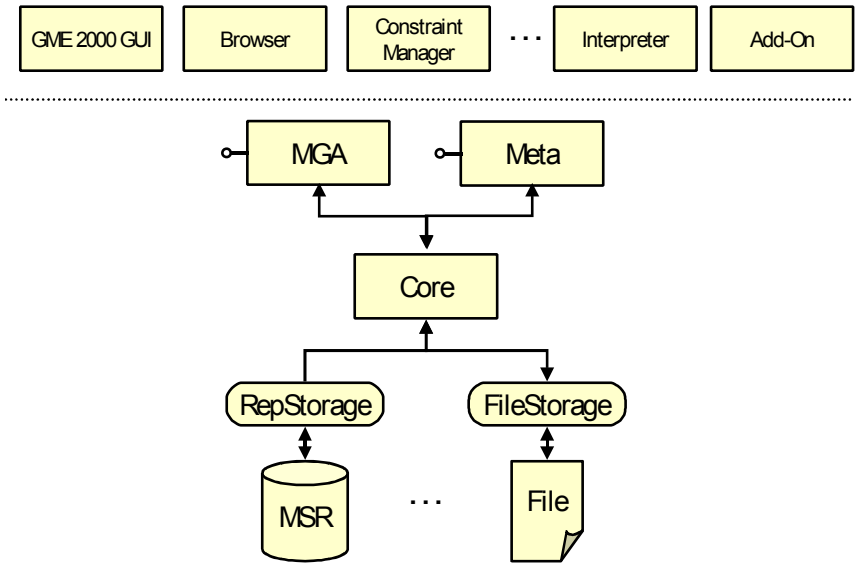

Figure 2.4: Generic Modeling Environment architecture (Ledeczi et al., 2001)

Analysis:

GME is a system created to work with the model elements (editing models and metamodels). However, it only supports partial model heterogeneity, i.e. it has a representation approach that obligates that the interoperability information has two levels (model and metamodel) of abstraction. GME also does not support model transformations. It presents some functionalities, such as a model browser, add-ons and a constraint manager which gives support for editing the information within GME. The add-ons may support processing on top of the information within. In terms of storage method available, GME uses a Storage Model database.

Integrated Repository Manager (iRM)

The Integrated Repository Manager (iRM) is another OMG MOF-compliant repository system (Petrov et al., 2004). The architecture of iRM, is presented on figure 2.5. Within the iRM Repository Management System (RMS), there are four logical modules: Metadata Manager; Lock Manager; Consistency Manager and the Data Store Manager. The Metadata Manager is the logical module of the IRM RMS that is directly associated with the MOF metadata architecture, i.e. it organizes the repository objects into the 4-layered OMG reference architecture (Petrov & Buchmann, 2008). The Consistency Manager is the module which checks and enforces the structural consistency of the repository data, i.e. it checks if every object has a type, or that every association has two association ends. The Lock Manager ensures that an object is not accessed by multiple repository applications in an incompatible manner, it is also used as a session manager, keeping track of the repository applications working with the repository. The Data Store Manager works as an interface to the storage approach, i.e. the Data Store layer in iRM is realized in terms of storage managers implementing a storage manager interface, in order to store data or metadata.

The iRM mSQL engine processes queries formulated in a declarative query language called mSQL, against the repository data. mSQL allows querying attribute values in classes on meta-layer instances of a specified meta-class, i.e. it allows model independent querying (Petrov et al., 2004). The mSQL module has two main sub-modules: the mSQL parser which includes partial semantic checker and a logical query plan generator (Petrov & Buchmann, 2008); and the mSQL wrapper containing the implementation of the different operator and mSQL funtions (Petrov & Buchmann, 2008).

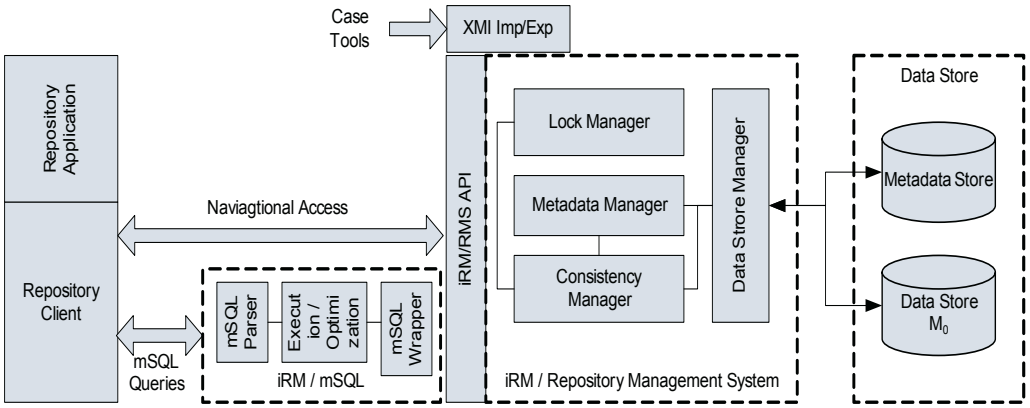

Figure 2.5: Architecture of iRM (Petrov et al., 2004)

Analysis:

The iRM is another repository based on 4-layered OMG reference architecture. As the iRM data model organizes each element in the 4-layered architecture, it is impossible to manage data that is not compatible with that structure. Both the consistency Manager and Lock Manager are useful modules to ensure the integrity of the information within the repository. The Data Store Manager enables different types of storing, i.e. iRM can persist both data and metadata, which means

that the elements represented by the data model are stored and the metadata associated with each element may also be stored using the Metadata Store. Another interesting iRM module is the mSQL query engine. This engine allows model independent querying which may ease the way users interact with the information within iRM.

Sun Netbeans Metadata Repository (MDR)

Sun Netbeans Metadata Repository (MDR) is another method encountered (Matula, 2003), and its architecture is presented in figure 2.6. This metadata repository is able to load any Meta Object Facility (MOF) metamodel and store instances of that metamodel. MDR can import/export metamodels and metadata using XML that conforms to the XMI standard. Metadata within the repository can be managed using the JMI API and the MDR API. MDR provides several functionalities, such as enabling that a user can model the language using UML (Matula, 2003). It provides a selection filtering support for querying (Petrov & Buchmann, 2008).

All repository metadata in MDR is persisted using the persistence Service Provider Interface (SPI). The primary storage structure assumed by the SPI is a key-value index. The key is the repository object identifier, while the value contains the repository object in a serialized form. It can be realized by different structures as B-Trees, Hash-Tables, database tables with indices, etc. The default realization is a file based B-Tree store (Matula, 2003). In order to enable users to respond to any changes in the repository, the implementation of the interfaces provided by MDR also handles notifying all the registered listeners of any changes in the metadata they are interested in (Petrov & Buchmann, 2008). Other MDR feature is the support of additional indexing, i.e. enables object indexing using a specified combination of values of several attributes/references. Indexing is used as an add-on to the default B-Tree storage.

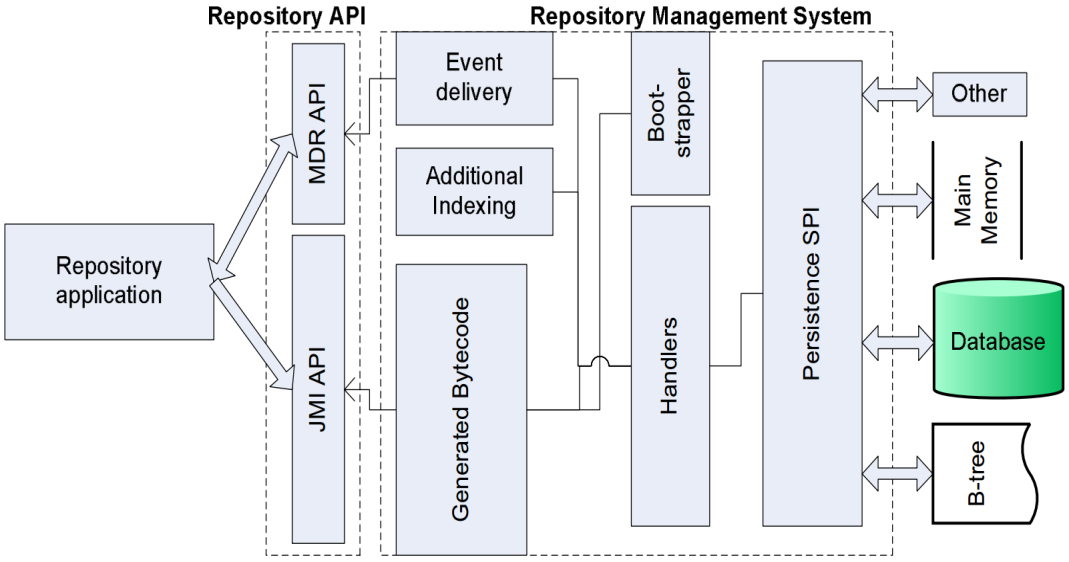

Figure 2.6: Architecture of MDR (Petrov & Buchmann, 2008)

Analysis:

The biggest issue about MDR is that it is no longer available, i.e. Netbeans stopped working on the MDR project and all the latest official documentation is now unavailable. However it is still used in some other projects such as AndroMDA (Bohlen et al., 2007), ArgoUML (ArgoUML, 2005) and EMF (Steinberg et al., 2008). All the information presented here (this analysis included), is based on the white paper (Matula, 2003) and in some articles that during its availability described it (Petrov & Buchmann, 2008). MDR is a repository that stores instances of MOF metamodel as metadata using different storage methods. MDR enables the use of UML to model the language because it has a tool to generate the MOF file based on the given UML model, because the UML language conform to the MOF metamodel. The support of MOF metamodel is important because many of the artefacts use the MOF metamodel, but MDR cannot represent models that conform to metamodels that are not MOF based. It possesses a persistence SPI which supports the use of several storage implementations, which is important because it allows the use of MDR in different environments. It does not allow processing the information within the repository. The query facility provided by MDR could not be analysed due to the fact that no specific information was found.

2.1.2 Synthesis

Having already presented and described each of the state-of-the-art element, now the synthesis of each element will be presented and related with the problem characteristics presented before. Table 2.1 presents the author's point of view in how each state-of-the-art element relates to the previously defined characteristics. Each line of the Table 2.1 represents one system and how that system handles each characteristic (columns).

One can notice that the majority of systems presented can only represent models conforming to one Metamodel. MDR, dMOF and iRM are MOF compliant, and CDO is ECORE compliant. AM3 and GME are not MOF nor Ecore compliant, but use a strict representation approach, meaning that any model not in accordance with it, can not be represented. So, it can be concluded that none of this elements can fully represent heterogeneous interoperability artefacts.

Regarding the need to keep the integrity of the environment, iRM uses a module to verify the consistency of the persisted information. This same system and dMOF also have a module responsible for all the management of the metadata persisted. MDR use a different module, which is called Handlers. As the name suggests it handles all the information within the system, and verifies its consistency. This MDR module presents an alternative to the metadata manager and consistency manager modules.

The studied systems also possess some add-ons, which can enable the use of execution of algorithms within the repository, such as the possibility of creating views from the data model, provided by dMOF. This feature, which is also used by AM3, may facilitate the use of algorithms on top of the data model, i.e. the algorithms may work on top on a view instead of working directly

on top of the data model. Some systems also provide specific query languages to allow model independent querying, which is the case of AM3 (provides a megamodel query language), iRM (provides a mSQL Engine) and CDO (Query Manager / Handlers).

In terms of interfaces, all six systems presented, provide methods of interaction with the information within each by having well-defined interfaces. The majority of them have a repository specific API, but MDR and dMOF also have automatically generated interfaces, i.e. Java Metadata Interface (JMI) (Dirckze, 2002) which is the standard for metadata management. The JMI specification enables the implementation of a dynamic, platform-independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata. The presented systems also have persistence managers which enables the connection to the persistence method available. Systems like dMOF, iRM and GME only provide one (available) persistence method, but AM3, MDR and CDO allows different methods of persistence and different implementations, making it indispensable to have a persistence manager, which is independent of the storage method used.

Table 2.1: Overview of the Related Work elements

	Heterogeneity	Execution Space	Environment Integrity
AM3	AM3 only supports partial model heterogeneity due to its strict representation approach. However it uses some disparate storage mechanisms	It possesses a megamodel view constructor, so it may construct views on some parts of the megamodel. It also enables Extensions	Does not provide
CDO	CDO only supports models conforming to the Ecore Metamodel	Does not provide	Is provides Notifications through the Notification Manager module
dMOF	It only support models conforming to the MOF metamodel	dMOF has a Repository-View manager that allows the creation of views from any specific part of the data model	Does not provide
GME	GME only supports partial model heterogeneity due to its strict representation approach	Does not provide, however it can be added as an Add-on	Does not provide
iRM	iRM uses the OMG reference architecture, so it only supports artefacts that can be represented using that structure	iRM has an iRM/mSQL Engine which allows model independent querying	iRM possesses a Consistency Manager and Lock Manager modules to ensure the integrity of the information, but only within the repository
MDR	MDR only supports instances of the MOF meta model	Does not provide	MDR uses Handlers, which are used as a constraint manager to guarantee the consistency of the information, but only within MDR

2.2 Advancement

After studying all the systems presented, it was concluded that none of those can fully handle heterogeneous interoperability artefacts. The heterogeneous characteristic of this work is of vital importance due to the need of various interoperability artefacts to achieve interoperability among themselves. AM3, GME and iRM are not metamodel specific, due to using representation approaches not linked with any specific metamodel. From those three, the better suited representation approach is the one used by AM3, the GMM representation approach. This approach is the only one that represents relationships between information models, and so this work will have a similar approach so it can represent heterogeneous information models, interoperability specifications and the languages that describe each one.

In terms of persistence, the studied systems have one or more storage mechanisms, which may aid the support of heterogeneous information. It is believed that this work must not be linked to any specific storage method. Being tied with some specific implementations, as some of the systems presented, would lead to a non scalable or non evolutionary system. So these work needs to have a persistence manager that will allow the use of different persistence methods and implementations. This is a fundamental characteristic due to the need of supporting heterogeneous interoperability artefacts. With it, the system may be able to choose which storage mechanism fits better to store each type of interoperability artefact.

In terms of interfaces, there is the need to provide methods to interact with the information. Every system studied has a repository specific API which provides this type of methods.Due to these API being repository specific they provide methods in accordance with the information within in order to provide their functionalities to the end users. So, this work will also have a repository specific API in order to enable the user to interact with the information.

Another important aspect is how each state-of-the-art element provides execution spaces. Table 2.1 shows how each one of them handle that characteristic (second column). Both AM3 and dMOF provide a view constructor module so that any application may create views of sections of the information within. MDR and GME only allow model editing and browsing respectively. iRM provides an mSQL Engine. None of them allows the possibility of executing algorithms within the system. So, this work will provide execution spaces, as a major advancement from the state-of-the-art elements. These User Spaces are spaces where a user or application can upload an algorithm and run it from within the system using the interoperability artefacts available.

Another characteristic of this work is the need to keep the integrity of the information in the environment, and for that only CDO possesses a module that accomplishes it. The CDO Notification Manager does exactly what the characteristic needs, it notifies the users or applications upon changes in the artefacts within the repository. MDR and iRM have a Consistency Manager module that keeps the integrity of the information, but only within the repository. This work will use a solution as the one present in CDO, which is a Notification Manager module, allowing the system to keep the environment updated with the last interoperability specifications available.

It was also noticed that the majority of the presented systems are considered repositories. So, the obvious choice to overcome the problem and its characteristics is the use of a repository. Repositories facilitate more efficient storage and management of data formats and interoperability specifications, they also enable sharing and discovery of new information.

3

Interoperability Repository System

3.1 Concept

Repositories facilitate more efficient storage and management of resources, they enable users to share and discover resources shared by others. In this section it is presented some examples of a repository end-user who interacts with it in order to use the functionalities provided. Those users may interact with the repository system in different ways and with different objectives.

One type of end user activities that can be accomplished with the use of the repository is using it exactly as a repository, i.e. a place where the user can store something. In this particular case the user can store an interoperability artefact in the repository and consult it afterwards. In order to accomplish this, the user only has to give the artefact that needs to be stored in the repository, and the repository automatically stores it. This adding, consulting and deleting is just some of the functionalities provided by the repository. This activities represents a simple interaction that a end-user can have with the repository.

It is also needed that the repository provides spaces, in which allows the user to add its own algorithms and execute them from within the repository. The ability to update the interoperability related information in the environment is another functionality needed. Another example that may use this repository is an heterogeneous environment where two different formats want to communicate. Imagining two data formats that are unable to understand the messages exchanged between them, so one of them enquires the Repository System for specifications on how to understand that message. The repository than replies with the interoperability related information enabling the communication between those two formats.

This repository is to store and manage all this interoperability artefacts and would allow any smart objects to query it for specifications on how to communicate with different data formats whenever needed.

The repository handles information about the interoperability artefact as it handles the actual file, however the repository considers the file as information about the artefact. If any user or application is in need of an interoperability specification, it can enquire it, as shown in figure 3.1. It is important to understand that the centralized aspect of the figure is due to all the interoperability artefacts be stored in the same place, and not that the repository will be the centred in the environment.

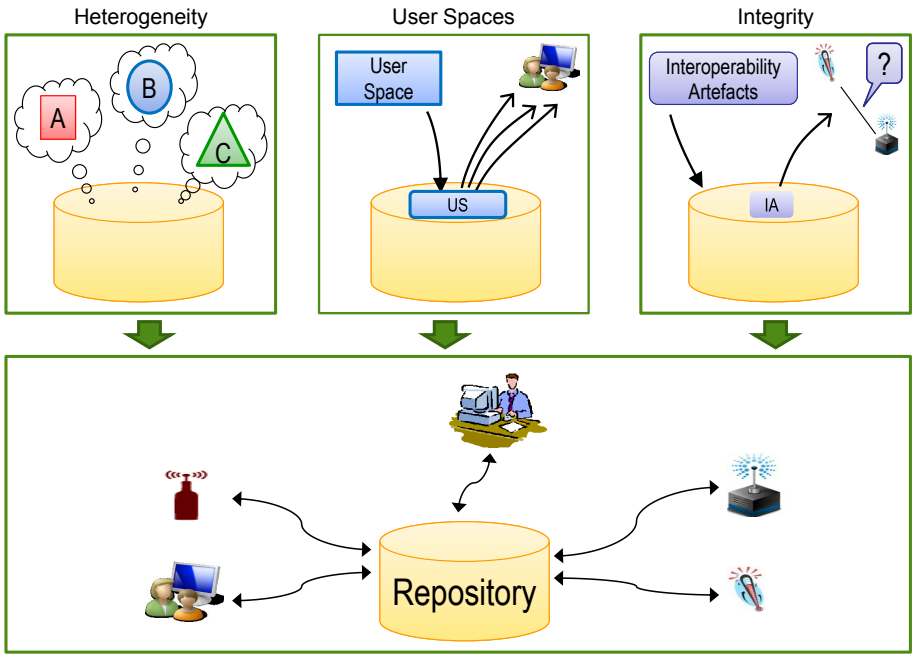

Figure 3.1: Concept

3.2 Architecture

In order to create the repository architecture, it was needed to identify which characteristics this system needed. Those characteristics are:

- Handle Heterogeneity: The repository needs to handle heterogeneous interoperability artefacts. It needs to represent information modes, interoperability specifications and the languages that describe each one.

- Provide User Spaces: The repository must provide spaces so the end user of the repository can upload its algorithms and then execute them fro within the repository, allowing more complex processing using the interoperability artefacts stored.

- Update the Environment: The repository must keep the environment up-to-date in respect to interoperability. It needs to guarantee that the interoperability specifications being used in the environment, are the newer available.

The architecture used in this work is based on the reference architecture for repository systems (Petrov & Buchmann, 2008). In the next few paragraphs it is described the architecture created which obeys to the three layered structure defined by the reference one, with the logical modules within each layer.

- **Repository Interface layer:** defines a set of methods to expose the repository functionalities to the Repository Applications. This layer exposes functionalities for storage and retrieval of information, life cycle management, execution of queries against the repository data, etc. The logical modules within this layer are:
  - Information Manipulation Interface: enables the manipulation of the information about interoperability artefacts within the repository and allows a user access to the user space interface;
  - Configuration Interface: enables the repository configuration, i.e. adding / deleting storage mechanisms, enables subscription of artefacts and notifies the application which subscribed to a new / updated artefact, allows one to manipulate the user spaces provided by the repository, etc;
  - User Space Interface: provides access to the Execution Engine, by means of the User Space Specific interface. This allows a user to interact with a specific User Space.
- **Repository Management System (RMS) layer:** it is in this layer that all the processing occurs, i.e. this layer may have a set of modules responsible for managing the information, allowing several operations to be executed; This layer possesses the following logical modules:
  - Metadata Manager: It provides the comprehension of all the artefacts stored in the repository;
  - Notification Manager: module responsible to manage subscriptions of artefacts;
  - Execution Engine: module that executes algorithms to enhance the functionalities of the repository;
- **Persistence layer:** This layer represents the set of persistence providers working with the repository system and the module that handles them. This persistence providers can be a relational database, flat files, index files, etc.:
  - Persistence Management: Module that handles the persistence of the actual files and manages the storage mechanisms available;
  - Storage Mechanism: it represents any storage mechanism available.

The architecture created is presented in figure 3.2, where these three layers are explicit, and all the logical modules are presented.

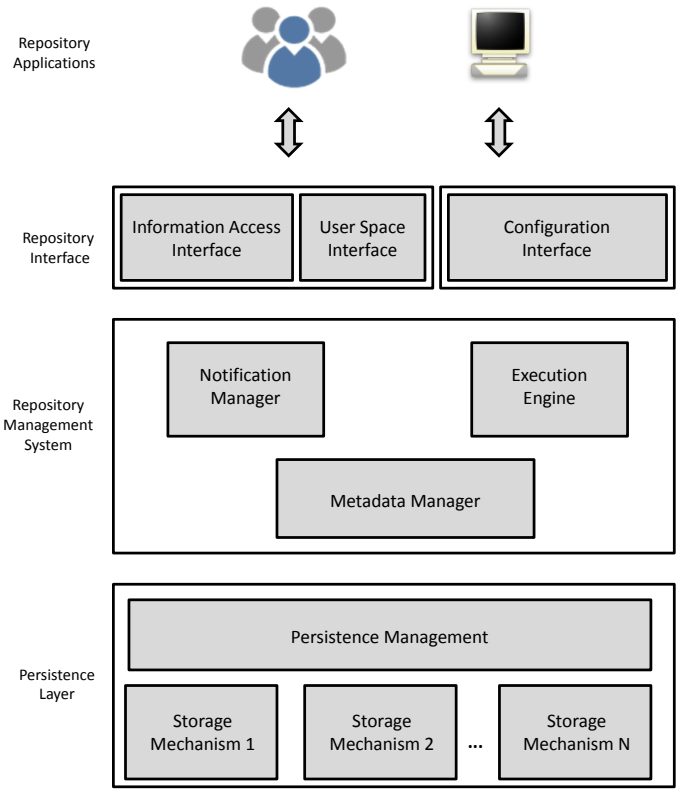

Figure 3.2: Logical Architecture

3.3 Logical Architecture Specification

This section will provide a specification of the logical modules present in each layer. Each logical module has a description of its objectives and the methods each one provides. Figure 3.3 presents an example of a logical module. It has sets of methods on top (API), which are methods available to other modules to use. The ones in the bottom (Caller Interface) are methods that the module uses to communicate with the other modules. Each of the logical module described next has a figure such as this one in order to explain its own methods.

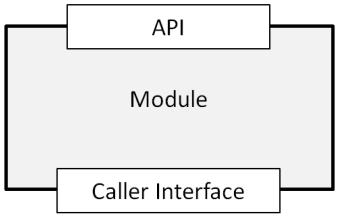

Figure 3.3: Example of a module with the API and Caller Interface

3.3.1 Persistence Layer

This layer, as the Repository Applications layer, presents all the storage mechanisms that may be used by the repository to store the artefacts that it has to persist. This storage mechanism can be of many types, i.e. databases, files or main memory. It also presents the Persistence Management module. It is important to understand that is in this layer that the actual files of the interoperability artefacts are handled and stored. The Persistence layer and its logical modules are present in figure 3.4.

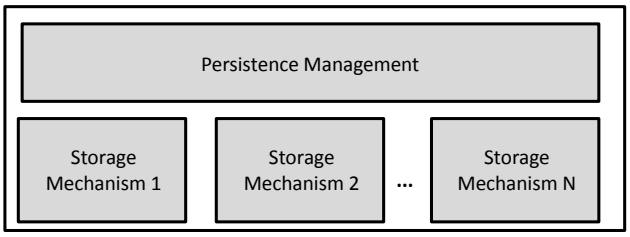

Figure 3.4: Persistence layer and its logical modules

Storage Mechanism

All the storage mechanisms within this Persistence layer need to be registered in the Persistence Management module, so the repository knows its existence and which artefacts are being persisted for each mechanisms. Present in figure 3.5 is an example of a Persistence Layer module. This example presents a general storage mechanism which can be a database, File system or main memory. It presents CRUD methods in the API, so the repository can store or delete the actual files.

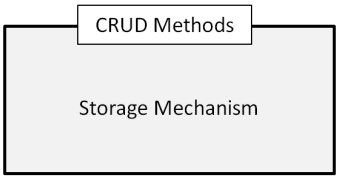

Figure 3.5: Example of a Persistence Layer module

Persistence Management

This module is responsible for the management of the persistence mechanisms being used by the Repository. It also keeps track of which files are stored within each storage mechanism. Whenever this module receives a file it stores it in one of the storage mechanisms available. The Persistence Management module and its methods are present in figure 3.6.

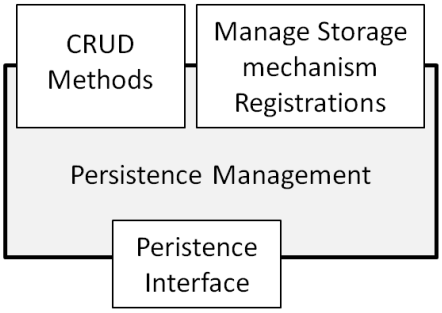

Figure 3.6: Persistence Management module

This Persistence Management module indexes the files with the storage mechanism where it is stored, it is also where the storage mechanisms are registered so it can always know which storage mechanisms are available. Another important characteristic of this module is that this module is the one responsible for choosing which storage mechanism to use whenever a new file needs to be stored. That choice is based on the type of the interoperability artefact, considering which storage mechanisms available.

This module has a set of CRUD methods, which are the four basic functions of persistent storage (Create, Retrieve, Update and Delete) (Martin, 1983). This methods allow the other modules to add, update, delete or retrieve information from this module. The Manage Storage mechanism registration is a set of methods to subscribe, unsubscribe or manage the storage mechanisms available. The module also has a persistence interface (as a Caller Interface) which allows the communication with the storage mechanisms in order to request the addition or removal of a specific file.

3.3.2 Repository Management System Layer

The RMS layer possesses all the logical modules responsible for processing the information within the repository. This layer is where the interoperability artefacts (excluding the files, which are handled in the persistence layer) is handled. Within this layer there are three different logical modules, which can be observed in figure 3.7:

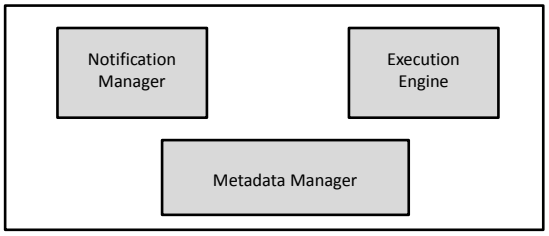

Figure 3.7: Repository Management System layer and its logical modules

Metadata Manager

Metadata Manager is the logical module responsible for the representation of all interoperability artefacts. This module needs a representation approach similar to GMM (the representation approach used by AM3), that can represent heterogeneous information. Figure 3.8 presents the Metadata Manager module and its methods.

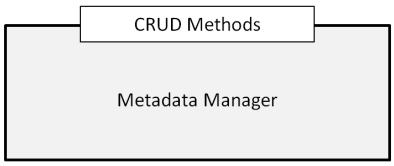

Figure 3.8: Metadata Manager module

The interaction with this logical module mainly consists in basic adding, retrieving, updating or deleting (CRUD methods) information about artefacts. To have the meta-information of all artefacts means that it needs use unique identifiers. The metadata manager is responsible to generate these unique identifiers so each interoperability artefact has its own identifier. This Metadata Manager module possesses CRUD Methods on its API, in order to enable the other modules to manipulate the information within.

Notification Manager

This module is responsible for notifying repository application whenever there is a change in the artefacts within the repository. This notification services are useful for several modelling applications that are performing complementary operations with the same artefact. This is also important to maintain updated the interoperability specification being used within the environment. Figure 3.9 shows this Notification module with its own methods.

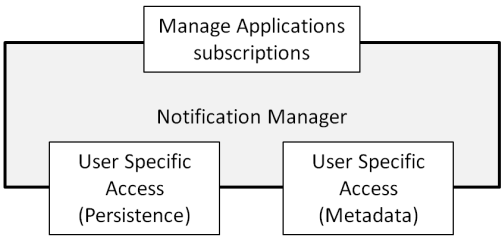

Figure 3.9: Notification Manager module

This module possesses an API method called: Manage Application subscriptions, which exists in order to provide the methods to manage subscriptions. It allows an application to subscribe to one or several artefacts and when there is a change in the subscribed artefacts it notifies the application. In the Caller Interface, it possesses two methods that serve as a connection between this module and both the Metadata Manager and the Persistence Management module, so the Notification Manager module can notice whenever an artefact (file and/or meta-information) is changed in the model module.

Execution Engine

This is the module responsible for all the user defined operations executed, using the Generic Data Interface to communicate with the other modules. This execution engine can perform more complex tasks, such as traverse algorithms and statistic determinations. Figure 3.10 shows the Execution Engine module and both its API and Caller Interfaces.

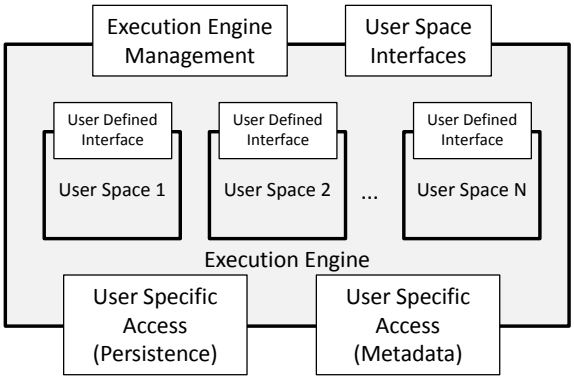

Figure 3.10: Execution Engine module

This Execution Engine module provides User Spaces in order to allow one user to add its own algorithm to the repository. The repository provides two sets of methods in its own API: the Execution Engine Management, which are the methods provided to manage the user spaces, such as the addition of removal of a algorithm from one User Space; and the User Space Interfaces, which are the methods provided by each algorithm, that are present in figure 3.10 as the API of each User Space. Each algorithm may use the User Specific Access methods (The Execution Engine Caller Interface) to use the functionalities provided by the other modules.

3.3.3 Repository Interface Layer

In order to allow user interaction, the repository needs to support a well-defined interface with different methods. Each method may have different abstraction levels, and have distinct objectives. From a more functional point of view, this repository needs to be associated to a set of services. These services can be used to query the repository for interoperability artefacts, allowing the user to enhance the repository's interoperability specification, add/remove entities or relationships to/from the repository, as well as, to attach new information about an existing artefact.

One important aspect, is that all the control flow is on this layer, which means that it is the Interface modules that control the execution flow of each action that the user wants executed. For example, in the case that a user need to add a file to the repository, he calls the Add method (included in the CRUD methods) from the Information Manipulation Interface module and inserts the file there. Afterwards, this interface module interpret the file to retrieve its meta-information and asks the Model module for a Unique Identifier to use with this new artefact. After gathering all that

information it asks both the Model and the Persistence Management module to store both meta-information and actual file respectively, using the unique identifier generated for that artefact.

The Repository Interface layer is where all the services provided by the repository are exposed. This layer, which can be seen in figure 3.11, has two logical modules which provide methods to access the information within the repository.

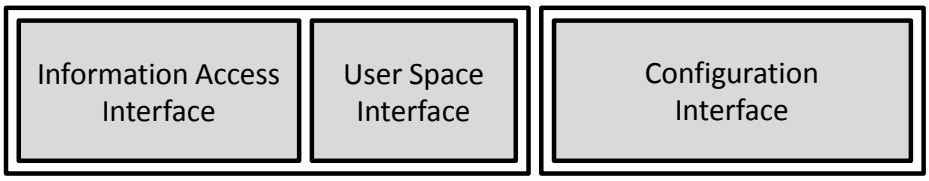

Figure 3.11: Repository Interface layer and its logical modules

Information Access Interface

This Information Access Interface module offers methods (figure 3.12) to allow applications to add, retrieve, update or delete artefacts. These CRUD methods will both work with the actual files and the meta-information associated with them, which means that this module is the one that will differentiate the files from the metadata, in order to standardize the process of adding and retrieving artefacts. This interface allows a end-user to use any CRUD method to interact to the artefact specific information. These are relevant methods because some applications may need to consult the information regarding specific artefacts.

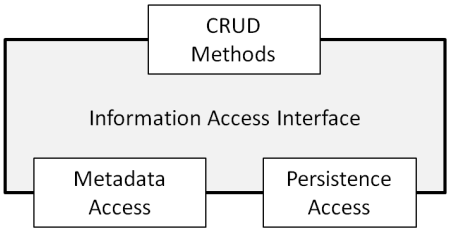

Figure 3.12: Information Access Interface module

This Interface module is responsible for the interoperability artefacts manipulation within the repository. It contains all the methods available to manipulate both actual files (stored via the Persistence Management module) and the artefacts metadata (stored in the Metadata Manager module). It provides CRUD methods, present in the API of the module in figure 3.12, to the end users and then based on what the user want, may communicate with both the Metadata Manager and Persistence Management module through the Caller interface, Meta-Information Access and Persistence Access respectively. One have to notice that in any of the services provided by this module, it is this module that controls the flow of information. Another functionality of this module is the capability to interpret the files added to the repository, in order to extract the meta-information from it.

User Space Interface

This User Space Interface module provides access to the user spaces, by means of the User Space Specific interface, enabling a user to interact with a specific user space by using the user space interface provided by the Execution Engine. The user may then request the execution of the algorithm present in the chosen user space.

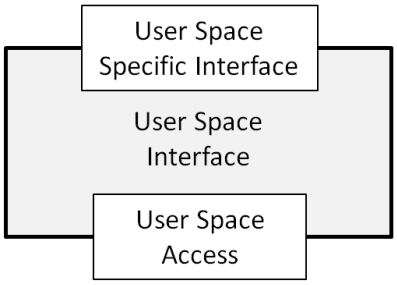

Figure 3.13: User Space Interface module

Configuration Interface

This Configuration Interface module allows the configuration of the repository system. The objective of this module is to offer methods to manage the repository functionalities, such as the configuration of the user spaces, the subscription of notifications and the registration of new storage mechanisms. Figure 3.14 presents this module and its own methods.

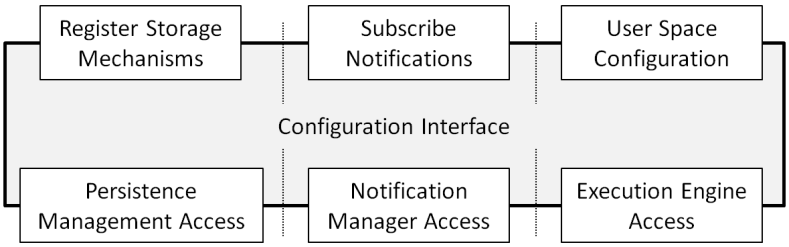

Figure 3.14: Configuration Interface module

This interface module provides methods to configure the repository. One can register and unregister storage mechanisms to the Persistence layer, in order to provide more persistence options to the repository. It also allows a user or application to subscribe/unsubscribe any artefact present within the repository, meaning that when that specific artefact changes, the notification manager notifies the user about the change. This module also provides communication support to the Execution Engine module. It gives users the possibility to configure the User Spaces provided by that module, that the user may be able to add, retrieve, delete or update algorithms from the User Spaces. In order to accomplish all of this, the Configuration Interface module possesses a set of Caller interfaces which provide access to the other repository modules.

3.4 Detailed Architecture

In this section, a detailed architecture is depicted. Figure 3.15, shows a detailed IRS architecture with its modules and the specific interfaces.

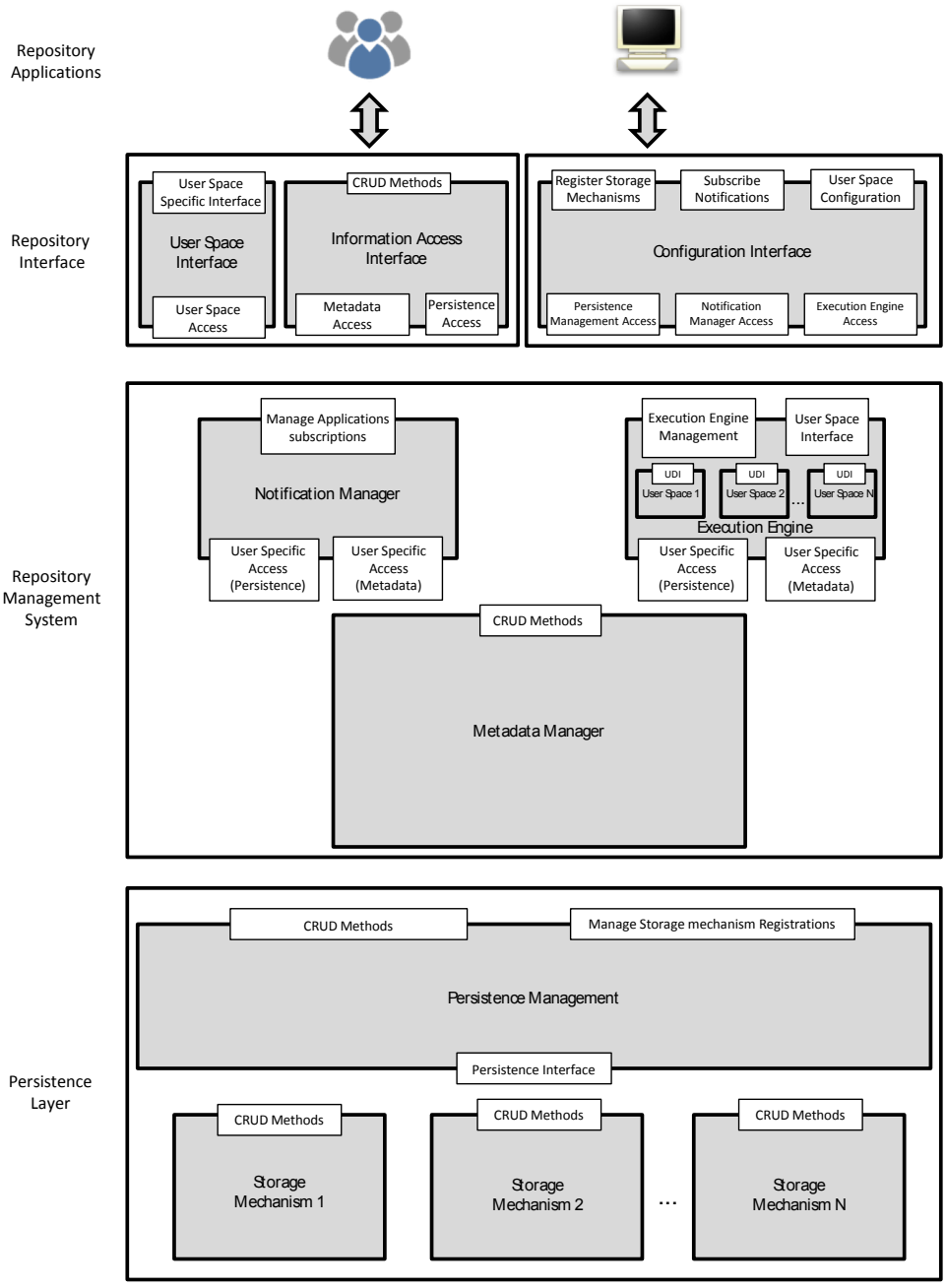

Figure 3.15: Detailed Architecture

4

Testing and Validation

4.1 Testing Methodology

Testing is the process of trying to find errors in a system implementation by means of experimentation, which is usually carried out in a special environment where normal and exceptional use is simulated. One have to notice that testing cannot ensure complete correctness of an implementation. It can only show the presence of errors, not their absence (Tretmans, 2001).

There are several methods to test the suitability of solutions to meet their requirements, each with its specific field of application (Onofre, 2007). Although not all geared for the same purpose, some have similarities, being the most evident the use of international standard for conformance testing of Open Systems, i.e. the ISO-9646: *"OSI Conformance Testing Methodology and Framework"* (Technology, 1991), as a starting point. As such, and since it is necessary an abstract testing methodology, the concepts defined by this standard will be used.

The general purpose of this standard is "to define the methodology, to provide a framework for specifying conformance test suites, and to define the procedures to be followed during testing", which leads to "comparability and wide acceptance of test results produced by different test laboratories, and thereby minimising the need for repeated conformance testing of the same system" (Technology, 1991).

This standard was originally developed to provide a platform and define a terminology for the application of tests on OSI ("Open System Interconnection") systems. But due to its low usage, the methodology has been little used for compliance testing of these type of systems. Nonetheless, methodology has been applied to other types of protocols and systems, being used as a basis for other methods of compliance tests, as used in standard ISO 10303 (*"ISO 10303 part 30 - Conformance testing methodology and framework"*).

The testing process described by this methodology is divided into three stages, as depicted in figure 4.1. The first phase is the specification of an abstract test suite for the system in question, and is referred as test definition. This test suite is abstract in the sense that it is developed independently of any implementation. The second phase consists of defining the tests so that they can be executed, and is called test implementation. It takes into account the implementation that will be tested, adapting the previous defined tests to the system implementation. The last phase, test execution, consists in its execution and observation of results. Which leads to a verdict on the compliance of the system under test with the initial requirements defined (Tretmans, 2001). The system implementation is a proof of concept, which is classified under the third level (Proof of Concept Demonstrated, Analytically and/or Experimentally) according to Technology Readiness Levels (Defence Research & Engineering, 2009).

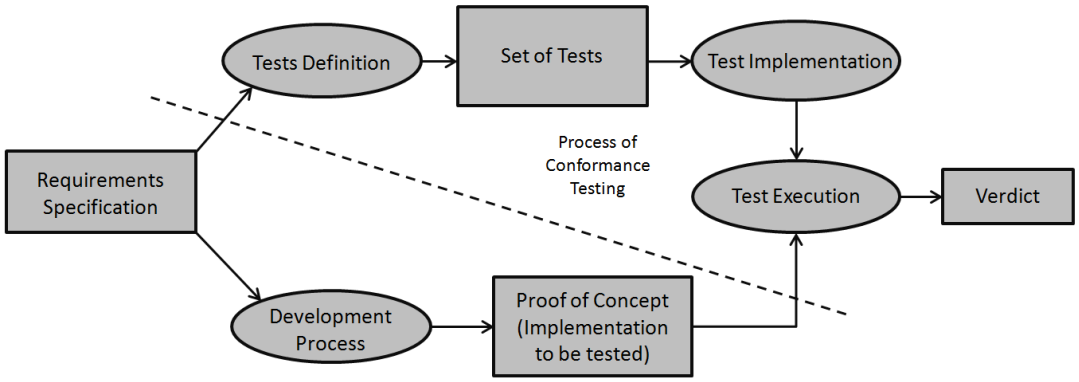

Figure 4.1: Global View of the Conformance Testing Process, based on (Technology, 1991)

Because abstract tests suites are standardized, they must be specified with a well defined classification, independent of any implementation and be globally accepted. The standard ISO / IEC 9646(Technology, 1991) recommends the use of the semi-formal language: TTCN-2 - *Tree and Tabular Combined Notation*. In TTCN-2, the behaviour of tests is defined by the sequence of events that occur during the test (Tretmans, 1992).

This behaviour is defined in a tabular form, and a chain of successive events is indicated by increasing the indentation of several events. The alternative events are defined using the same indentation. A sequence ends with the specification of the verdict that is assigned when the execution of the sequence terminates. After a completion of a TTCN-2 test table a verdict must be deliberate: "Success", "Fail" or "Inconclusive". Each test table possesses an header, where there is defined the test name, its purpose, and the inputs needed during the test execution.

Table 4.1: Example of a TTCN-based table test

Test Case		
Test name:	Test the establishment of a Basic Connection	
Purpose:	Check if a phone call can be established	
Inputs:	[I1]: Phone number	
Line number	**Behaviour**	**Verdict**
1	! Dial number [I1]	
2	? Connected line	
3	! Connection Established	SUCCESS
4	! Busy Tone	INCONCLUSIVE
5	? No connection	FAIL

An example is depicted in Table 4.1. This created example exemplifies a phone call establishment evaluation. After a series of actions and evaluations (question events) a different verdict is attained. Firstly, the user dials the phone number, then it verifies if the line is connected. If there is no connection, then the verdict is "FAIL". If there is a connected line, the connection could be either established or be busy, making the verdict "SUCCESS" or "INCONCLUSIVE" respectively.

In order to present the results of the tests execution, it will be used test cases. The test cases should contain the parameters of the test to be conducted, including the inputs needed and the expected results. A matrix will be used to present it, illustrated in table 4.2, in which each row represents a specific test case and the columns represent the test ID, Inputs, and expected and actual results.

Table 4.2: Test Case example

Test	Input	Result (Line number)	
	I1: Phone Number	Expected	Actual
1	(+351) 913 456 789	Success (3)	Success (3)
2	(+351) 913 456	Fail (5)	Fail (5)

4.2 Proof of Concept Implementation

The language used to create this proof of concept implementation was the Java programming language, due to be platform independent.

Due to the existence of several modules, and the need to control them, a module interface was created, which defined methods, such as the *getAPI*. This method is implemented by all the repository modules, so that it knows which modules exist. This specific method is needed so that modules know the methods provided by all the other modules. This is specially important, so that users when creating their own User Spaces, know which modules are "registered" within the repository and the services they provide. To work correctly, an API class needed to be defined, so that all APIs use the same structure.

Each module will have information that needs to be persisted, and a database is the logical choice to persist that information. But due to the object oriented nature of the solution, there was the need of a Object Relational mapping technique, so the objects can be persisted in the relational

databases. To do that, it was used EclipseLink, which is a project that provides a complete persistence solution that runs in any Java environment, reading and writing objects to virtually any type of data source, including relational databases. These databases are then managed by a SQL relational database engine, HSQLDB (HyperSQL DataBase).

Another important aspect of the proof of concept implementation is the manage of the storage mechanisms. In this proof of concept, it was implemented two different storage mechanisms, file system and Eclipse Teneo (a database persistence solution for Ecore models), as shown in figure 4.2. In order to implement these two storage mechanisms, it was used a Java Interface, which is an abstract type that is used to specify an interface that classes must implement.

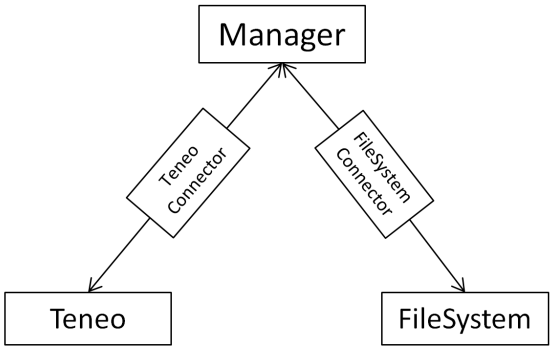

Figure 4.2: Storage mechanisms implementation using Java Interface

In this Java interface, present in figure 4.3, all the methods that the storage mechanisms need are here defined, and implemented in the storage mechanism connector, the FileSystem connector and Teneo Connector. Figure 4.3 presents the definition of the Persistence Java interface. This interface defines the CRUD methods that each storage mechanism need to provide and one other method, called *getSupportedTypes*, that retrieves the file type that the storage mechanism can persist. The Manager uses the supported file types in order to choose where to persist each file. Considering an ECORE model, the manager looks for the storage mechanism that explicitly persists Ecore files, and if there is one, it uses it to store file. In the case that there isn't one, it looks for storage mechanisms that doesn't have an explicit type (supports every file type), such as the File System.

```
package persistence.manager;
import java.io.File;
public interface Persistence {
            public void addFile(File ficheiro);
            public File getFile(String nome);
            public String getSupportedType();
```

Figure 4.3: Persistence Interface

The notification manager relates the user subscriptions to the interoperability artefacts stored, and so, it needed an interface implementation. This defines methods to notify users whenever actions like adding, updating and deleting are executed to the subscribed artefact. It also needs to define methods to enable user to subscribe to specific artefacts and events. The concept behind its creation is just like a "*wiretap*", shown on the left of figure 4.4. The notification manager "listens" to

the communication between two modules, and checks if there is any subscription to the listen event. In practice, it was implemented as seen in the right part of figure 4.4, a encapsulation module was created with an API, which redirects the requests made by the source module to the target module API. After redirecting the request, this encapsulation module API invokes the Event method provided by the Notification module, with information such as: method, interoperability artefact and which is the target module of the request.

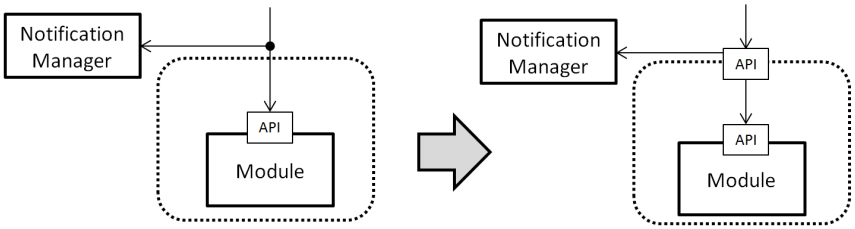

Figure 4.4: Notification mechanism implementation

The user interfaces were created to allow the users to interact with IRS and access all the functionalities provided. The technology chosen was SOAP (Simple Object Access Protocol), due to its generalized and extensive use, and because SOAP is a protocol specification for exchanging structured information, which was considered necessary.

The User Space module allows users to add their own user spaces to the repository. In this proof of concept implementation, two user spaces were created:

- US1 - "AllTrans": The first User Space was intended to determine the transformations between the interoperability artefacts. In order to accomplish that, it was used a path finding algorithm using DFS - Deph-first search, which is an algorithm for traversing a tree or graph. One starts at the root and explores as far as possible along each branch before backtracking. This "AllTrans" US possesses a method to find all paths between two interoperability artefacts, and has two arguments: the source and target interoperability artefacts;
- US2 - "ConformsTo": The second user space is intended to do a simple statistic analysis to the repository, so it possesses one method, that will analyse the interoperability artefacts within the repository and return the number of artefacts "described by" each of the meta-languages (e.g. MOF, ECORE, XSD, etc.) present in the repository. This method has one argument, the meta language.

The Metadata Manager is the module responsible for representing the interoperability artefacts stored within IRS. In order to do that, a representation approach was needed. This representation approach needed to represent information models, interoperability specifications and the languages that describe each one, and on top of that it needs to handle the heterogeneity of the interoperability artefacts. An approach was developed within the research group at GRIS-UNINOVA by Bruno Almeida and Pedro Maló. This theory and method approach is called Model for Interoperability Management (MIM), depicted in figure 4.5, and its objective is to represent and manage interoperability artefacts.

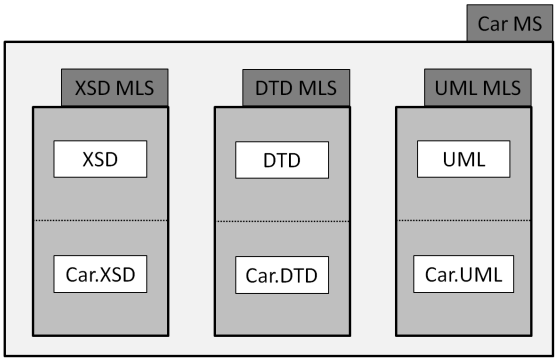

Figure 4.5: Model for Interoperability Management approach

This approach is based on the notion of a Modelling Space (MS), which is a modelling architecture representing a specific model that can be represented in various different Meta Language Spaces (MLS). The MLS represents the model and the specific language that describes it. Figure 4.5 presents an example where three different MLS (XSD, DTD and UML) compose the Car MS.

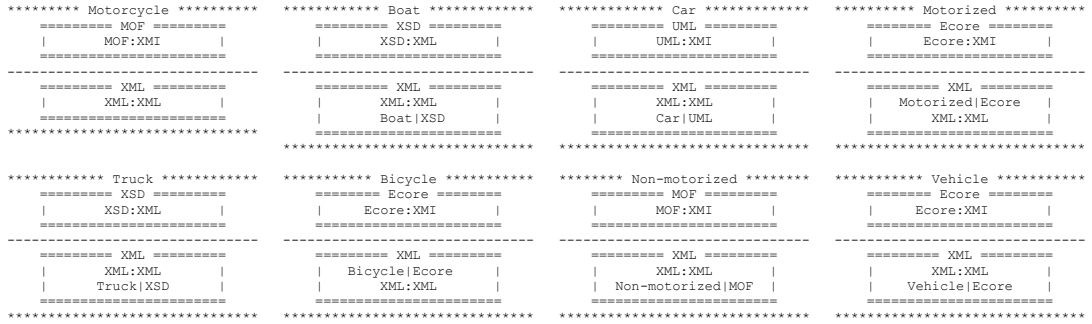

Figure 4.6: Interoperability Information Models representation using MIM

This MIM representation approach used in the Metadata Manager, enabled managing heterogeneous interoperability artefacts, as seen in figure 4.6. The figure presents Interoperability information models. Each information model (e.g. Motorcycle, Boat, Car, etc.) represents a MS and is divided in 2, the top half represents the information model MLS. The bottom half represents how the information model data is described. Considering the information model representing the Truck present in figure 4.6, one can notice that it conforms to the XSD MS, and its data is described using XML (represented as XSD:XML in the figure, which means MS:DATA). The XSD MS is present in figure 4.7. The Truck information model data is described in XML, so it conforms to the XML MS and its data is described using XML.

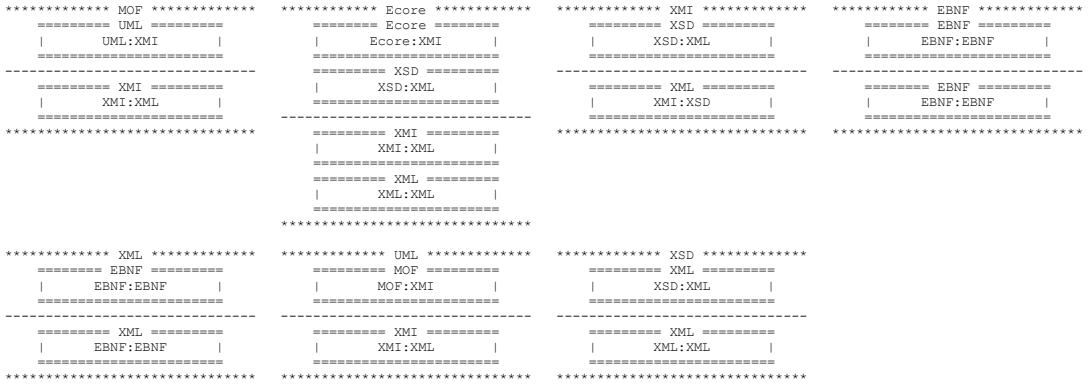

Figure 4.7: Languages representation using MIM

4.3 Test Definition and Execution

The test cases will result in the verdict on the compliance and suitability of the proposed solution. The tests will be divided into three groups, each one corresponding to a requirement, and so, the following tests were defined:

1. Handling Heterogeneity: the first test is to verify if the repository can handle heterogeneous artefacts. For that, it will be used the most basic operation of the interoperability repository system, i.e. adding and retrieving interoperability artefacts. Adding and retrieving different types of interoperability artefacts will test the capability of the repository to handle heterogeneous information.

2. Environment Integrity: the second test is intend to verify the repository's ability to notify users that subscribe to a specific interoperability artefact, keeping this way the integrity of the environment;

3. User Spaces: this will test users spaces. This test is intended to verify the abstraction of the repository, and to do so, verify if one user can successfully add its own user space to the repository. This will verify if the repository can be used with different purposes.

The tests will be done manually, due to the fact that the prof of concept developed requires human interaction in its operation. And also, the fact that the presentation of the results is carried out in a variety of ways, that hinder an automatic approach. The definition of each test is presented in the next sections.

4.3.1 Adding and Retrieving an Interoperability Artefact

Test Definition

This first test scenario describes the steps needed to test some repository CRUD methods. In this case, a user adds an Interoperability Artefact to the repository, and afterwards another user tries to retrieve the same Interoperability Artefact. This interoperability artefact is considered as both the file and its metadata. Table 4.3 exemplifies this test.

Table 4.3: Add and Retrieve an Interoperability Artefact test definition

Add and Retrieve Interoperability Artefact		
Test name:	Adding and Retrieving an Interoperability Artefact	
Purpose:	Test the addition and retrieval of an interoperability artefact to and from the repository	
Inputs:	[I1]: Interoperability Artefact file; [I2]: Interoperability Artefact metadata	
Line Number	**Behaviour**	**Verdict**
1	! Add Interoperability Artefact ([I1] + [I2])	
2	? Interoperability Artefact added successfully	
3	! Retrieve added Interoperability Artefact	
4	? Interoperability Artefact Retrieved	
5	! Compare to original Interoperability Artefact	
6	? Same Interoperability Artefact	SUCCESS
7	? Different Interoperability Artefact	FAIL
8	? Interoperability Artefact not found	FAIL
9	? Failed to add Interoperability Artefact	FAIL

Test Execution

As explained, the test cases matrix can present more than one test execution and its outcome. So, this "Adding and Retrieving an Interoperability artefact" will be tested with different inputs in order to verify if the actual results are consistent with the expected results.

Table 4.4: Add and Retrieve an Interoperability Artefact test execution

Test	Input		Result (Line number)	
	I1: IA file	I2: IA metadata	Expected	Actual
1	Vehicle.ECORE file	Vehicle.ECORE metadata	Success (6)	Success (6)
2	Car.UML file	Car.UML metadata	Success (6)	Success (6)
3	Truck.XSD file	Truck.XSD metadata	Success (6)	Success (6)
4	Motorized.ECORE file	Bicycle.ECORE Metadata	Success (6)	Success (6)
5	Motorcycle.MOF file	Vehicle.ECORE Metadata	Fail (9)	Fail (8)

The tests were executed with different inputs, such as ECORE, MOF and XSD. As expected, the actual results are in line with the expected ones, i.e. the tests were successful. The tests 4 and 5 were made using metadata that didn't conform to the actual file. The result in test 4 was as expected, however the repository should not allow the addition of a file with different metadata. This is due to a implementation flaw, because the repository is not extracting metadata form the

file, and so it cannot compare it with the one added by the user. In test 5, it successfully failed to add the file, due to the fact that there is a storage mechanism (Eclipse Teneo) specific for Ecore models. However it should have failed when the file is added and not when trying to retrieve the file. This maybe due to the fact that the repository is not alerting when it fails to add a new interoperability artefact.

4.3.2 Environment Integrity

Test Definition

The notification mechanism test was created in order to test if the repository can keep the integrity of the environment. In order to verify that the repository successfully notifies users that subscribed to an Interoperability artefact. The definition of this test is present in table 4.5.

Table 4.5: Environment Integrity test definition

Environment Integrity		
Test name:	Notifications upon events on Interoperability Artefacts	
Purpose:	Test the ability keep the integrity of the environment by means of alerting a user that the subscribed artefact as been updated	
Inputs:	[I1]: Interoperability Artefact; [I2]: Clause ; [I3]: Event	
Line number	**Behaviour**	**Verdict**
1	! Subscribe to [I1] with Clause [I2]	
2	? Subscription added successfully	
3	! Apply [I3] on [I1]	
4	! Wait for Notification	
5	? Notification Received	SUCCESS
6	? Notification not Received	FAIL
7	? Failed to subscribe to [I1]	FAIL

This Environment Integrity test is meant to notify the repository users of specific event on the repository. For this, this test has three inputs: I1 - the interoperability artefact which the user wants to subscribe; I2 - a clause that means which events the user want to be notified about; and I3 - the event input, which means, for testing purposes, which event will be forced on the interoperability artefact in order to test if the user is notified.

Test Execution

In order to execute this test, it was needed that the repository already possessed Interoperability artefacts. Figure 4.8 shows this test initial conditions, i.e. the information present within IRS when this test was executed. The notation used to describe the information models is "*Model_Name.Meta_language*" (e.g. Vehicle.ECORE) which means that this information model represents a vehicle and it is described by the ECORE metamodel.

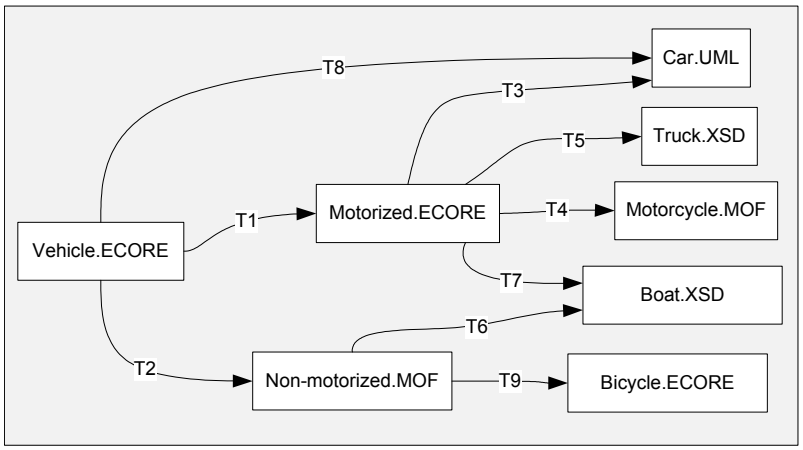

Figure 4.8: Initial Conditions

To ensure that the repository can keep the integrity of the environment, several tests were executed with different input combinations. The results of those tests are present in the Environment Integrity test execution table (table 4.6).

Table 4.6: Environment Integrity test execution

Test	Input			Result (Line number)	
	I1: IA	I2: Clause	I3: Event	Expected	Actual
1	Vehicle.ECORE	Update	Update	Success (5)	Success (5)
2	Motorcycle.MOF	Delete	Delete	Success (5)	Success (5)
3	Car.UML	Update & Delete	Update	Success (5)	Success (5)
4	Truck.XSD	Delete	Update	Fail (6)	Fail (6)

Initial Conditions: present in figure 4.8

As one can see in table 4.6, the actual results correspond 100% to the expected ones, which means that the repository can indeed keep the integrity of the environment. The executed tests used random interoperability artefacts in order to ensure that it would work with every artefact and not only with specific ones. The clauses used in the testing process were the Update and Delete clause, becuse they are the most common clauses used in this type of subscriptions.

4.3.3 User Spaces

Test Definition

The User Space execution test is the last test defined. Its objective is to test the management and execution of the the User Spaces present within the Execution Engine module. It defines the steps needed to add a User Space to the repository, and then execute it. Table 4.7 provides the definition of this User Space Execution test.

Table 4.7: Definition of the User Spaces test

User Spaces		
Test name:	Test the Management and execution of User Spaces	
Purpose:	Test the ability to add a User Space to the repository, to access it and execute it	
Inputs:	[I1]: User Space; [I2]: Arguments; [I3]: Return Expected	
Line number	**Behaviour**	**Verdict**
1	! Add [I1] to the repository	
2	? [I1] added successfully	
3	! User access [I1] with [I2]	
4	? Return Result	
5	!Compare Returned Result with [I3]	
6	? Same Result	SUCCESS
7	? Different Result	FAIL
8	? Return no Result	FAIL
9	? Failed to add [I1]	FAIL

Test Execution

The first step defined in this User Space execution test is the addition of a User Space to the repository. The User Spaces used in this test, were the ones implemented in the proof of concept, i.e. the "AllTrans" and the "ConformsTo" User Spaces. Each of the User Spaces possesses methods to interact with the information within the repository.

To execute this test, both User Spaces required that the repository was already populated with interoperability artefacts. So, when this test was realized, the repository had the same interoperability artefacts that the previous test had (figure 4.8). The results of the executed tests are present in table 4.8.

Table 4.8: User Space test execution

Test	Input			Result (Line number)	
	I1: User Space	I2: Arguments	I3: Return Expected	Expected	Actual
1	AllTans	Vehicle.ECORE; Boat.XSD	[T1, T7]; [T2, T6]	Success (6)	Success (6)
2	AllTans	Vehicle.ECORE; Car.UML	[T8]; [T1, T3]	Success (6)	Success (6)
3	AllTans	Car.UML;	[ ]	Fail (8)	Fail (8)
4	Unknown US	Vehicle.ECORE; Car.UML	[ ]	Fail (9)	Fail (9)
5	ConformsTo	ECORE	[3]	Success (4)	Success (4)
6	ConformsTo	Boat.XSD	[ ]	Fail (8)	Fail (8)

Initial Conditions: present in figure 4.8

The results were consistent with the expected results. There were executed three tests using the first User Space, which were a success. The third one was using invalid arguments, which resulted in a expected "Fail". Test 4 was made to test the addition of a invalid User Space, which the repository, as expected, failed to add. The last two tests were executed using the second User

Space, the first using valid arguments, and the second one with invalid ones. Again, the results were as expected.

4.4 Verdict

The first conclusion to be drawn from the executed tests, is that the proof of concept implemented successfully passed all the tests. It successfully added and retrieved heterogeneous artefacts to/from the repository; it notified a user when the subscribed event is triggered; and it allowed a user to add and use a User Space within the repository.

One can notice that these three tests were devised taking into consideration the characteristics of this work, to handle heterogeneity: the repository needed to handle several types of interoperability artefact described with different meta languages; Provide User Spaces in order to enable users to add their own User Spaces (containing their own algorithms), and execute them form within the repository; and Update the Environment, so that the repository may keep the information in the environment updated.

The first test, the add and retrieve an Interoperability artefact test, proved that the repository can successfully add and retrieve heterogeneous artefacts. This functionality was testes with different type of artefacts and the actual test results were consistent with the expected ones. However, it demonstrated that the repository was not notifying the user when failed to add the interoperability artefact. Although it is an implementation flaw, it does not compromise the functioning of the repository.

The second test, was intended to verify if the repository would keep the information updated within the environment, and as shown, the repository successfully handled the notification mechanism. It notified when the subscribed event happened.

The third and last test, was developed in order to verify if the IRS could indeed handle the User Spaces added by the users. Once again the actual results were consistent with the expected ones. It successfully managed the User Spaces and was able to retrieve all the transformations to and from one interoperability artefact, as well as all the transformations between two different interoperability artefacts. It was also able to identify the number of information models “conforming to” the given meta-language.

After the analysis of the executed tests, it can be concluded that the hypothesis formulated in this work is valid. The created architecture is capable of handling all the problem characteristics.

5

Conclusions and Future Work

The "*Internet of Things*" (IoT) is a dynamic global network infrastructure where physical and virtual "*things*" communicate and share information amongst themselves. Plug and Interoperate is a scenario within IoT, which allows things to plug (into data) and seamlessly exchange information within an heterogeneous environment. PnI appears due to the need to address interoperability issues within IoT, i.e. due to the existence of many systems that use disparate data formats and need to interoperate.

The study of this scenario led to the definition of its main characteristics: Heterogeneity, which presents the need to handle heterogeneous interoperability artefacts (information models, interoperability specifications, languages that define them and tools); Abstraction, which is the need to enable the use of the information with different purposes; and Integrity, which presents the need to keep the environment updated with the newest interoperability specifications available. The characterization of the problem led to the research question: "How to organise interoperability artefacts in IoT environments".

In order to gather information about the characterized problem, a background research was made, to identify systems, technologies and approaches that handle interoperability artefacts. Each of the presented elements has been studied and analysed, in order to identify the different approaches that could be taken into account in order to solve the characterized problem. The interoperability state-of-research is using model driven concepts, so the elements studied and presented are model based: the AM3, which is a model based environment used in the Eclipse Modeling Project; CDO, which is a distributed and shared model repository used in EMF; GME, a system based on information models, which focuses on the formal representation, composition, analysis, and

manipulation of models during the design process; MDR, which is a metadata repository used to store instance of the MOF metamodel; dMOF, which a shared, server-based MOF repository; and iRM, which is another MOF compliant repository system with querying capabilities.

Some of the elements provided interesting approaches to solve specific problem characteristics, such as the CDO notification manager, which has the potential to solve the environment integrity problem characteristic. An advancement from the state-of-art was presented, where all the characteristics/approaches considered relevant were discussed, in order to understand which were going to be used in this work. One of the relevant approaches noticed, was that the majority of systems are considered repositories, which facilitate more efficient storage and management of information, and so it was decided that the system that this work needed to create was also a repository, the Interoperability Repository System (IRS).

Repositories facilitate more efficient storage and management of resources, enabling users to share their resources and find resources shared by other repository users. The concept, is that the repository stores all the interoperability artefacts, and when enquired, the repository would answer with the interoperability artefact required. In order to create this repository, some characteristics where identified so that it could fulfil its objectives: Handle Heterogeneity, which represents the need of representing heterogeneous interoperability artefacts; Provide User Spaces, which enables the users to attach their User Spaces to the repository and allow their execution from within; Update the Environment, which provides notifications to the users who subscribed information within IRS, allowing that the repository can keep the integrity of the environment.

The IRS architecture obeys to the three layers defined in the reference architecture for repository systems (Petrov & Buchmann, 2008), and each of them possesses its own modules. Each module possess an API, which have methods provided by the module to other modules or users to use. Some of them also have a Caller Interface, which represent the methods that the module use in order to communicate with other modules. The first layer is the Repository Interface layer, which defines a set of methods in order to expose the repository functionalities to the end users. There are two modules within this layer:

- Information Manipulation Interface: which enables the manipulation of the information within the Interoperability Repository System;
- Configuration Interface: which enables the configuration of the repository;
- User Space Interface: provides access to the Execution Engine, by means of the User Space Specific interface. This allows a user to interact with a specific User Space.

The second layer is the Repository Management System layer (RMS), which possesses modules responsible for all the processing and management of the information within Interoperability Repository System. Within this RMS layer, there are three modules:

- Metadata Manager: module responsible for the representation of interoperability artefacts stored within the repository;
- Notification Manager: the notification manager handles the user / application subscription to specific artefacts, as well as notifies them of any changes occurred;
- Execution Engine: this module provides User Spaces, so that users can upload their own algorithms and execute them from within the repository.

The third layer is the Persistence layer, which is responsible for the management of the storage mechanisms available to the repository. And in order to accomplish that, it possesses two modules:

- Persistence Management: which is the module responsible for managing the storage mechanisms available and to keep track of the stored files;
- Storage Mechanism: represents the storage mechanisms available and usable to the repository.

The followed work approach, defined a Testing and Validation step which purpose is to perform functional testing in order to assess the system compliance with the previously defined characteristics. Firstly, it was defined the methodology used in the testing phase, which was based in the ISO 10303 ("*ISO 10303 part 30 - Conformance testing methodology and framework*"). This methodology defines a creation of a proof-of-concept and both the definition of a set of tests and its execution. Due to the standardization of the test suits, it was used a semi-formal language to present the behaviour of tests, which was the "*TTCN - Tree and Tabular Combined Notation*".

So, three tests were defined and executed. The first test: Adding and Retrieving an Interoperability Artefact, was intended to test the most basic activity of a repository system. The execution of this test showed that the repository successfully allowed the addition and removal of different interoperability artefacts, which means that the repository can handle heterogeneous artefacts, but it also showed that the repository is not alerting when it fails to add a file to the repository. Although, it did not hinder the result of this test execution.

The second test, Environment Integrity was created in order to check if the repository can successfully keep the integrity of the environment. Meaning that it notifies users that specific events happened in the subscribed interoperability artefacts. In order to execute this test, some initial conditions were defined, such as the interoperability artefacts that were present within the repository at the time of the test execution. The execution of this test revealed that the repository is handling the notification mechanism successfully, as the actual results were consisted with the expected ones.

The final test, intended to test the User Spaces mechanism was created in order to verify if the repository could manage and execute the User Spaces from within. The execution of this test used

the same initial conditions used in the second test, plus two created User Spaces. This test was also a success.

The results of the three executed tests were a success, so, as a verdict it can be said that the hypothesis purposed in this master thesis work is valid, which means that the Interoperability Repository System can indeed handle heterogeneous interoperability artefacts, can be personalised by the user for different purposes, and it successfully notifies users of specific events occurred within the repository, keeping this way the integrity of the environment.

5.1 Future Work

An interesting advancement that can be accomplished, is to make the repository distributed and decentralised, i.e. to enable that both the control and the contents of the repository are dispersed throughout the environment. Any IoT environment is a distributed environment, where there are several heterogeneous sensors and devices, that work together in order to achieve some goal. Actually, the repository requires the interoperability artefacts to be stored in one single place, and so, a distributed repository would allow that all interoperability artefacts within IRS would be spread throughout the environment, which will enable that the information within the repository be closer to where it is needed. With a distributed repository, the information would be stored on more than one place, preventing information loss.

In the application domain, one can notice that the scenario used as a motivation for this master thesis work was the IoT, however it is also believed that this repository is generic enough to be applied to other scenarios. This repository could be used to solve several issues in the Enterprise Interoperability domain. Enterprise Interoperability describes a field of activity with the aim to improve the manner in which supply chains, extended enterprises, or any form of virtual organizations, by means of Information and Communications Technologies (ICT), interoperate with other enterprises, organisations, or with other business units of the same enterprise, in order to conduct their business (Cluster, 2006). This work could be useful to provide interoperability specifications between different IT systems, applications and business processes, enabling this way Enterprise Integration.

5.2 Publications

From this work resulted two scientific articles, one with the title “Towards an Interoperability Management System” which was published in the 6th Iberian Conference on Information Systems and Technologies (CISTI) 2011, which presented an early development stage of this work. Another, with the title “Towards measuring information interoperability based on model transformations”, which was also published in the same conference, presented a measuring method in its early stage, that will allow the evaluation of model transformations.

6

Bibliography

Allilaire, F., Bézivin, J., Brunelière, H., & Jouault, F. (2006). Global model management in Eclipse GMT/AM3. In *Eclipse technology exchange workshop in conjunction with ecoop* (Vol. 6).

ArgoUML. (2005). *Argouml tool.* Available from `http://argouml.tigris.org/`

ATLAS. (2011). *The Eclipse-GMT AM3 (Atlas MegaModel Management) project.* Available from `http://www.eclipse.org/gmt/am3/`

Bakken, D. (2001). Middleware. *Encyclopedia of Distributed Computing*.

Bézivin, J., Soley, R. M., & Vallecillo, A. (Eds.). (2010). *Mdi '10: Proceedings of the first international workshop on model-driven interoperability.* New York, NY, USA: ACM.

Bohlen, M., et al. (2007). *Andromda.* Available from `http://www.andromda.org`

CERP-IoT. (2008). Internet of things: Strategic research roadmap. *Cluster of European Research Projects on the Internet of Things (CERP-IoT)*.

CERP-IoT. (2010). Vision and Challenges for Realising the Internet of Things. *Cluster of European Research Projects on the Internet of Things (CERP-IoT)*.

Cluster, E. I. (2006). Enterprise interoperability research roadmap. *An Enterprise Interoperability community document. Published by European Commission in*.

Defence Research, O. of the Director of, & Engineering. (2009). Technology Readiness Assessment (TRA) Desk book.

Dirckze, R. (2002). Java metadata interface (jmi) specification. Available from `http://java.sun.com/products/jmi/`

DSTC. (2000). dMOF Version 1.1. User Guide [Computer software manual].

Eclipse. (2011). *The CDO Model Repository project.* Available from `http://www.eclipse.org/`

cdo/

Foundation, E. (2011). *Eclipse modeling project.* Available from http://www.eclipse.org/modeling/

Gruber, T., et al. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies*, *43*(5), 907–928.

IEEE. (1990). *IEEE standard computer dictionary. a compilation of IEEE standard computer glossaries.* Available from http://ieeexplore.ieee.org/servlet/opac?punumber=2267

IERC. (2011). Internet of things - global technological and societal trends. *IoT European Research Cluster (IERC), Second Ed.*.

ISIS. (2011). *Institute for software integrated systems - model integrated computing.* Available from http://www.isis.vanderbilt.edu/research/MIC (Last checked: 28 March 2011)

Karsai, G., Sztipanovits, J., Ledeczi, A., & Bapty, T. (2003). Model-integrated development of embedded software. *Proceedings of the IEEE*, *91*(1), 145–164.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., et al. (2001). The generic modeling environment. In *Workshop on intelligent signal processing, budapest, hungary* (Vol. 17).

Martin, J. (1983). *Managing the data-base environment*. Prentice-Hall. Available from http://books.google.com/books?id=ymy4AAAAIAAJ

Matula, M. (2003). Netbeans metadata repository. *NetBeans Community*.

MODELPLEX. (2007). *Deliverable d2.1.a: Global model management principles* (Tech. Rep.).

NIC. (2008). National Intelligence Council - Disruptive Civil Technologies, Six Technologies with Potencial Impacts on US Interests out to 2025.

Onofre, S. M. (2007). *Plataforma para testes de conformidade de sistemas baseados em módulos conceptuais step*. Unpublished master's thesis, Departamento de Engenharia Electrotécnica ; Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia. Available from http://hdl.handle.net/10362/1870

Petrov, I., & Buchmann, A. (2008). Architecture of omg mof-based repository systems. In *Proceedings of the 10th international conference on information integration and web-based applications & services* (pp. 193–200).

Petrov, I., Jablonski, S., Holze, M., Nemes, G., & Schneider, M. (2004). irm: An omg mof based repository system with querying capabilities. *Conceptual Modeling–ER 2004*, 850–851.

Schafersman, S. (1997). An introduction to science. *Scientific Thinking and the Scientific Method*.

Sciore, E., Siegel, M., & Rosenthal, A. (1994). Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems (TODS)*, *19*(2), 254–290.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). Emf: Eclipse modeling framework.

Technology, I. (1991). *Open systems interconnection, conformance testing methodology and framework*.

Tretmans, J. (1992). *A formal approach to conformance testing*. Twente University Press.

Tretmans, J. (2001). An overview of osi conformance testing. *Samson, editor, Conformance Testen, in Handboek Telematica*, *2*, 4400.

Visser, P., Jones, D., Bench-Capon, T., & Shave, M. (1997). An analysis of ontology mismatches; heterogeneity versus interoperability. In *Aaai 1997 spring symposium on ontological engineering, stanford ca., usa* (pp. 164–72).