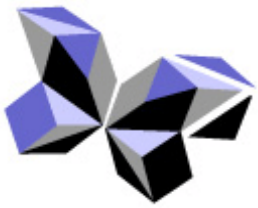

Digital Business Ecosystem

Contract n° 507953

WP14: DBE Knowledge Base

Deliverable D14.6: Final Release of the Recommender

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: D14.6

Due date: 10/2006

Delivery Date: 11/2006

Short Description:

The challenge in designing and developing the DBE KB is to enable the effective support of rich semantics and complex queries across multiple sources in an unstructured and self-organised P2P network such as the DBE network. The objective of the first P2P implementation of the DBE KB was to exploit the current P2P infrastructure of the ExE environment, and to offer high availability in knowledge access and an effective distributed knowledge discovery mechanism.

In the second release of the Recommender all the mechanisms to support knowledge access in DBE were developed. The knowledge access mechanisms provide the underlined functionality for querying metamodels, models and instances in the DBE environment. They also support recommendations (i.e. matching preferences of users expressed with the User Profile Metamodel), with the knowledge base Query Metamodel Language (QML), which is based on OCL2.0. Extensions to support fuzzy queries were also implemented. At that time, mechanisms to support a first approach for semantic query routing through indexing in the P2P distributed DBE implementation were also described. In the final release of the Recommender the whole DBE framework and infrastructure are exploited, including the metamodels and ontologies in order to enhance the querying mechanisms and support the heterogeneous environment through mechanism like ontology mapping, query expansion and semantic indexing and routing. This document accompanies the Recommender software delivered under the DBE Studio and Swallow sourceforge projects.

Partners owning: Technical University of Crete (TUC)

Partners contributed: Technical University of Crete (TUC)

Made available to: Public

Versioning

Version	Date	Author, Organization	Description
0.1	12/10/2006	GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS – TUC	Initial Creation
0.2	10/11/2006	FOTIS KAZASIS -TUC	Revisions
1.0	14/11/2006	GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS – TUC	Version 1.0 Submitted
2.0	8/12/2006	GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS – TUC	Revisions based on the feedback provided by the internal reviewers

Quality check

1st Internal Reviewer: Victor Bayon (UCE)

2nd Internal Reviewer: Paul Malone (WIT)

Table of Contents:

List of Figures:

List of Tables:

Executive Summary

The challenge in designing and developing the DBE Knowledge Base (DBE KB)[1] is to enable the effective support of rich semantics and complex queries across multiple sources in an unstructured and self-organised peer-to-peer (P2P) network such as the DBE network. The objectives of the P2P implementation of the DBE KB were to exploit the P2P infrastructure of the ExE environment and to offer high availability in knowledge access as well as an effective distributed knowledge discovery mechanism. The technical approach followed in order to fulfil the first objective was presented in Deliverables D14.3 – "1st P2P Distributed Implementation of the DBE KB and SR" [41] and D14.5 – "Final P2P implementation of the DBE Knowledge Base and SR" [49] whereas the one that was followed in order to fulfil the second objective was presented in Deliverable D14.4 "Second Release of the Recommender" [42] and in this document.

In the second release of the Recommender all the mechanisms to support knowledge access in DBE were developed. Knowledge comes from contextualization and personalization of information. In DBE contextualization of information is supported with the management of metamodels, which give context to information and models as well as with the management and use of domain specific ontologies that have community accepted semantics for their concepts and relationships. Personalization in DBE is supported with profiles of users (WP7 "User Profiling" [25]) and filtering mechanisms. The DBE knowledge is managed by the Knowledge Base (KB). The Knowledge Base is compatible with the OMG's MOF metadata framework therefore each segment of information stored in the KB has been produced and placed as instance of a model at a higher layer of the MOF meta-data architecture. The KB manages MOF metamodels, models, and instances providing the full functionality of a MOF repository, and uses XMI documents for metadata and data interchange. The knowledge access mechanisms provide the underlined functionality for querying metamodels, models and instances in the DBE environment. It also supports recommendation (i.e. matching preferences of users expressed with the User Profile Metamodel), with the knowledge base Query Metamodel Language (QML), which is based on OCL2.0, [2], adapted for MOF1.4 [1], and extended to allow similarity matching in order to accommodate user preferences. The Recommender is using the DBE knowledge base implementation, which is based on the MDR [21] repository and a native XML database to store and manage DBE models and metamodels. Similarity matching queries include a mechanism for retrieving knowledge that is expressed in different models.

In the final release of the Recommender the use of rich domain specific ontologies is allowed in order to enhance the description capabilities. This heterogeneous environment posed the need for the extension of the querying mechanism in order to support complex queries for the full use of the ontology framework. An ontology mapping mechanism has been implemented and is now available during the query analysis process. In addition, knowledge in DBE is distributed over a peer-to-peer network, making very difficult the efficient discovery of such knowledge. In order to face

[1] With the term Knowledge Base we mean the entire infrastructure needed to handle the DBE Knowledge. This infrastructure provides different services like Semantic Registry Service and Knowledge Base Service as it has been described in various documents.

this challenge, mechanisms that support query expansion and semantic query routing through indexing have been applied.

The document accompanies the software of the final distributed implementation of the DBE Recommender that has been integrated as part of the DBE Studio and Swallow projects. The Appendixes C and D briefly present the available interfaces for accessing the DBE Recommender. The source code is available within the swallow project in http://swallow.sourceforge.net inside the cvs branch recommenderFinal_210206.

1 Introduction

This document describes the mechanisms developed in order to support knowledge access in DBE. Since the previous release of the recommender the entire infrastructure has been stabilised and the whole framework was ready for use and testing. As the project is now mature the goal was to extend and enhance the implemented mechanisms in order to meet all the challenges for accessing knowledge in DBE. These challenges have to deal with the scalability of the system, the P2P nature of the system and the semantic exploitation of the existing framework. In the first part of the current introduction the final knowledge access framework is described. Based on this, the final characteristics of the system are introduced with emphasis on the querying mechanisms including all the enhancements referring to the query formulation, the use of ontologies and the query expansion implemented to facilitate the use of heterogeneous ontologies and finally the implementation of the query routing mechanism with the exploitation of the semantic overlay network.

DBE information is hosted in the Knowledge Base (KB). The DBE KB provides a common and consistent description of the DBE world and its dynamics, as well as the external factors of the biosphere affecting it. Its content includes:

- Representations of domain specific ontologies (common conceptualization in a particular domain);
- Semantic Descriptions of the SMEs themselves in terms of business models, business rules, policies, strategies, views etc.;
- Semantic Description of the SME value offerings (description on how the services may be called) and the achieved solutions (service chains/compositions) to particular SME needs.
- Models for gathering usage and accounting/metering data and statistics.
- User Profiles where SME's declare their preferences on the characteristics of demanded services and partners.

The DBE Knowledge Base (KB) follows the OMG's Model Driven Architecture (MDA) approach for specifying and implementing knowledge structuring and organization. The MDA "*…defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. The MDA approach and the standards that support it allow the same model specifying system functionality to be realised on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go*" [25]. Roughly speaking, this is done by separating the system design into Platform Independent Models (PIM) and Platform Specific models (PSM). Following this principle, the DBE Knowledge Base specifies the organization of the DBE knowledge in platform independent models that could be made persistent using many different platforms. To do that, one has to provide the corresponding Platform Specific Models and to provide the mapping from PIM to PSM knowledge structures. The DBE Knowledge base provides a PSM knowledge organization based on a Native XML Database Management System. Other implementations could be also possible. Note that

during the first release of the recommender the KB provide a PSM based on a Relational Database Management System [18] [24].

In addition the Knowledge Base follows the OMG's Meta Object Facility (MOF) approach for metadata and data[2] modelling and organization. The DBE Knowledge Base supports the four levels of the MOF architecture. The level M0 of the architecture consists of the data that we wish to describe, the level M1 comprises the metadata that describe the data and are informally aggregated into models, the M2 level consists of the descriptions that define the structure and semantics of the metadata and are informally aggregated into metamodels and the M3 level consists of the description of the structure and semantics of the meta-metadata. Thus, each segment of information that is stored in the KB is placed as an instance of a modelling element of a higher layer of the MOF meta-data architecture. That is, MOF based languages or mechanisms should be used in the upper levels of the architecture for defining each segment of information. Different kinds of metamodels[3] have been already developed and represented in the KB:

- the metamodel for Ontology Definition (ODM), which enables the representation and storage of existing OWL domain ontologies into the KB
- the metamodel for the business modelling and semantic description of services (BML, SSL), which enables the representation and storage for the business models and the semantics of the services offered by SMEs into DBE.
- other metamodels for the technical description of single and composite services (SDL, BPEL)
- the metamodel for expressing user profiles and user preferences (UPM)

Thus, the exploitable knowledge spectrum in DBE will range from ontologies, to business models, to semantic and technical service descriptions, to user profiles, to usage data, etc. Each one of these knowledge segments is represented using a different metamodel. In order to support efficient knowledge access over all these metamodels there was a need for a query mechanism that is quite generic so that it can specify, in a uniform way, knowledge access requests over all types of knowledge (both data and meta-data) that are kept in the DBE KB. Such kind of functionality is a prerequisite for implementing explicit querying of DBE Knowledge (information retrieval / pull-mode) as well as knowledge personalization (information filtering / push-mode) [27] functionality of the recommender component. Note that the framework and the techniques presented here are independent of metamodels. Thus, if DBE evolves and needs to use new metamodels, this framework will not need to change not only at design level but also at implementation level. This is one of the major advantages of using the MDA and MOF technologies.

Moreover, the DBE knowledge is distributed across SMEs in a decentralised peer-to-peer network. An SME may act as a knowledge base node that comprises knowledge, which

[2] Although MOF is typically used for describing metadata, it can be also used for specifying data by defining an instantiation metamodel. This is the approach followed by the DBE Knowledge Base.

[3] In the rest of the document the terms MOF Language, Metamodel and MOF Model will be interchangeably used for the same meaning

concerns both the SME's knowledge (metamodels, models, instances) and knowledge replicated from other nodes.

To this end, in this document we describe a query mechanism, which is based on a query metamodel that is quite generic so that the expressions (query models) that form the instances of this metamodel are capable of querying all types of knowledge (models and corresponding data), which are available in the Knowledge base in a uniform way. Moreover it describes mechanisms that provide functionality to effectively route queries to knowledge bases that have relevant information using semantic indexes.

As described, the DBE KB follows the MDA and MOF specifications. Given that MOF is strictly following the object-oriented paradigm, it is easily understood that, at the PIM level, all the DBE knowledge is also organised in a manner that follows the object-oriented paradigm (at the PSM level several implementations can be supported). In order to further support this decoupling between PIM and PSM knowledge manipulation there is a need for a knowledge access language that also follows the same paradigm in order to exploit all the semantic information.

Moreover, the DBE knowledge is distributed in decentralised manner over a set of nodes. Any node owning the DBE infrastructure can connect to the network at any time and automatically the rest of the nodes will become aware of its existence either directly from neighbouring nodes or through a random path connecting the new node with a second node, already part of the network. As a result the topology of the network is dynamic and not following any specific pattern, being a random graph network. This is what is called the physical network in the P2P environment; that is, all the nodes with the actual (physical) network connections between them allowing the transferring of data between them. As the framework for the construction of the physical network in the DBE has been set and is not posing any constraints for the knowledge management, the searching mechanisms that will be described have to be met on a higher level. This higher level is the so-called overlay network.

Many information and database systems provided powerful query mechanisms that are widely known, understood and used. For example, SQL-99 [12] has introduced object-oriented concepts into the SQL language. However, there are significant differences between the object models of MOF and of object relational databases. Since the MOF models and their instances can be mapped to XML documents (using XMI [8]), an XML query mechanism can be easily integrated with the MOF technology. XQuery [13], standardised by the W3C, are some of the many query languages for XML documents. However, the lack of object-orientation (such as inheritance or polymorphism) in XML would constrain the expressive power of an XML-based query approach[4]. The Object Query Language (OQL) is a query language based on SQL defined by the Object Database Management Group (ODMG) as part of the Object Data Standard [15]. However, the standard does not define a language's abstract syntax compliant with the MOF technology. SPARQL [50] is another query language that offers constructs to query ontologies and RDF graphs which. But SPARQL can not be used to the MOF environment

[4] we acknowledge here the weakness of XQuery to act as a generic MOF-based query language. However we elaborate appropriate mappings of the object-oriented queries to XQuery statements in order to utilise XQuery as the language for querying the native XML database management system

as the representation of data is quite different in the MOF and RDF environments making the constructs offered by SPARQL poor in our environment.

MOF QVT (MOF 2.0 Query, Views, Transformations) [19] is a specification aiming at restructuring and transforming models, generating/deriving other models. QVT is a technology of high importance to the MDA initiative and its specification has been recently adopted by OMG. Nevertheless, currently the more mature QVT-based approaches [21], [10] define query languages that are unable to express complex query mechanisms needed for IR (Information Retrieval) with relevance rank. MQL [16] is one of the languages developed to address the querying problem in MDA, but it doesn't conform to the OCL standard, as there is a great interference in OCL metamodel. On the other hand our approach has left intact the OCL metamodel and used its power to boost query expressiveness. mSQL [17] is another language proposed to query MOF based repositories. It is based on SQL and it does not exploit the power of OCL to navigate through metamodels and models and be able to express very complex queries. Another query framework is the one offered by EMF [51]. The EMF query framework cannot be used as it constructed on top of EMF and it does not offer any language which could be used in MOF environment. It is just an implementation making it monolithic and unusable.

Many systems are developed in the domain of schema based P2P networks; however most of them are either decentralised that do not face the problem of heterogeneous schemas on each node or they are not decentralised systems like the DBE network. These systems are using centralised indexes like Napster [32], or follow most costly approaches using flooding like Gnutella [33]. Other systems like Freenet [40] try to use more efficient indexing approaches based on some global hash function and caching of the location of recently requested documents. There are a number of P2P research systems (CAN [34], CHORD [35], and Tapestry [36]) that efficiently search for documents in a P2P network. However they mandate a specific network structure and conduct searching on document identifiers rather on the content of documents, which again is not the case in the DBE. A survey for centralised-search P2P systems can be found at [37], while [28] contains a survey for schema matching approaches.

Systems for searching with the use of ontologies also exist like KAON [38], which is based on the use of an Ontology Registry. In the same category, considering also the schema-matching problem, are also [29] [39]. Some systems deal with the query routing problem [30] [31] but they either don't have the flexibility to handle complex queries on the contents of heterogeneous models or they miss the special requirements posed by the use of ontologies. However, they are based on very interesting approaches that could be extended to cover the DBE framework and requirements. The above approaches are based on the semantic overlay network and there is much ongoing work in this domain. Other examples of systems are [44] and [43]. The first is dealing with semantic clustering techniques for the creation of a semantic "small world" while the second is a query routing approach that exploits semantic similarity of queries and topical experts for the build of a semantic index. Finally some metrics are introduced for the evaluation of these networks. Experiments along with their results are presented in both papers while some more metrics are introduced in [45].

The approach that we describe here is based on the definition of an object-oriented knowledge access language, named Query Metamodel Language (QML), using the MOF

meta-language. To achieve as much integration as possible with MOF, we opted to leverage the Object Constraint Language (OCL2.0), which has been used as the formal basis of our query metamodel. The choice of OCL was also motivated from the fact that OMG advocates in the core of its business architecture the use of MOF and on the top of it the use of Unified Modelling Language (UML), which contains OCL for specifying constraints in the models. Thus mechanisms that support OCL would be also useful for efficiently supporting UML in a Knowledge Base that would support also UML functionality. The implementation provides a coherent framework for QML processing that incorporates (Information Retrieval) IR functionality and the theoretical approach is based on the Extended Boolean Model [5]. For this reason, we have provided in QML methodologies for specifying ranking and fuzzy Boolean operations. The design has left OCL intact and has provided high-level query functionality for semantic query reformulation and fuzzy retrieval. The exploitation of both the query elements semantics and of powerful business ontologies result in semantic query expansion with relevant query terms. In order to make use of ontologies we had to discover similarities between their concepts using innovative methodologies.

As previously mentioned the knowledge access mechanism aims to satisfy two needs of DBE. The first refers to support discovery requests in the KB. These requests are instances of the QML. The additional requirement here is to also support Information Retrieval (IR)-style approximate matching and allow the ordering of results by their relevance score. The second refers to mechanisms responsible for matching preferences (user profiles) with business descriptions and service descriptions. The design and implementation of the mechanisms utilises the existing business and service ontologies that capture the semantics of business models and service descriptions.

At a technical level (implementation and theoretical approach) the knowledge access approach is uniform for both desired functionalities. However, whereas the discovery process is based on answering the formulated query expressions based on the available metamodel and model specific information laid in the KB, the recommendation process is based on matching user (SME) profiles (that include preferences on business and/or service semantics) and the underlying information. The Query Metamodel also allows the users to express preferences and as such it forms the basis of user profiles. For that the existence of a MOF User Profile Metamodel is considered as a prerequisite. Such a metamodel is provided by the work carried out in WP7 "User Profiling" (responsible partner Forschungszentrum Informatik - FZI) and it is named User Profile Metamodel (UPM) [25].

The knowledge access mechanism utilises the functionality for processing valid query expressions based on the QML (suitably formulated by the Query Formulator). Such functionality includes query parsing and analysis, query syntax tree construction, semantic query expansion and code generation using the KB infrastructure. From a technical point of view, the KB infrastructure is based on a combination of a MOF/JMI-compliant repository and a data management system. Two alternatives are currently available: a) the data is queried in MOF object representation; b) the object-oriented queries are mapped to XQuery statements and executed by the native XML database management system. The implementation of the code generator allows the generation of XQuery statements (enhanced when needed with fuzzy extensions) that correspond to the submitted queries and can be executed by the XML data base management

system. Moreover the code generator is capable of producing indexing related queries for the update and managing of the index.

Our work also aims at providing the functionality related to the evaluation of candidate services and candidate business partners in the process of creating composite services and establishing partnerships respectively. The major assumption behind the design and implementation of the Recommendation mechanisms is the existence of powerful business and service Ontologies that capture the semantics of business models and service descriptions. Although in the DBE environment this kind of Ontologies are difficult to be created (by regional catalysts for example), powerful OWL/RDF ontologies can be imported into Knowledge Base making it even simpler to reuse existing work from the semantic web. These Ontologies can be also used to define the corresponding preferences for businesses and services. The recommender exploits the XML database system to store preferences and all recommendation mechanisms use XQuery statements to implement the necessary matching functions between preferences and business/service descriptions.

Finally, the incorporation of the above-mentioned querying mechanisms in the DBE services for each node and the deployment of the system in the P2P environment allowed the P2P functioning of the Recommender. However, the need for an efficient query processing and routing mechanism was thereafter, imperative. The DBE framework, in accordance with the DBE P2P infrastructure, set the basis for a semantically strong manipulation of the semantic overlay DBE network. The common DBE metamodels and the capability to use rich ontologies for the description and management of knowledge in the DBE, play a decisive role in the sharing and discovery of knowledge. Based on all this, a semantic, decentralised routing and indexing mechanism was developed. The evaluation, testing and fine-tuning of the above mechanism result in a concrete, consistent and efficient framework for the P2P management and utilization of knowledge in the DBE.

The rest of the document is organised as follows: chapter 2 presents the general architecture of the recommender and the knowledge access modules that support the formulation and the evaluation of involved discovery requests to the peer-to-peer network of knowledge bases. Next, in chapter 3 the QML2.0 along with representative examples is presented. Chapters 4 and 5 present the methodologies used for the formulation of similarity queries and the semantic exploitation of them. Chapter 6 demonstrates how XQuery code is generated while, in chapter 7, we present the query routing mechanism that uses semantic indexes and learning to retrieve knowledge from the peer-to-peer network of knowledge bases. Chapter 8 presents the supported functionality of Recommender Services to manage user profiles and suggest recommendations based on the profiles. The last section presents the conclusions and the issues that could be further considered.

2 Recommender Module Architecture

This section presents the general architecture of the recommender and knowledge access modules. Special consideration has been given on how the module will operate on the distributed p2p environment.

Figure 1 shows the architecture of the Knowledge Base (KB) infrastructure. The KB is compliant with the JMI 1.0 specification that is the result of a Java Community Process (JCP) effort to develop a standard Java API for metadata access and management. The advantages to comply with JMI 1.0 are:

- the provision of a standard metadata management API for the Java 2 platform,
- the definition of a formal mapping from any OMG standard metamodel to Java interfaces,
- the support of advanced metadata services (such as reflection and dynamic programming) and
- the interoperability between tools that are based on MOF metamodels and are deployed in the DBE environment.

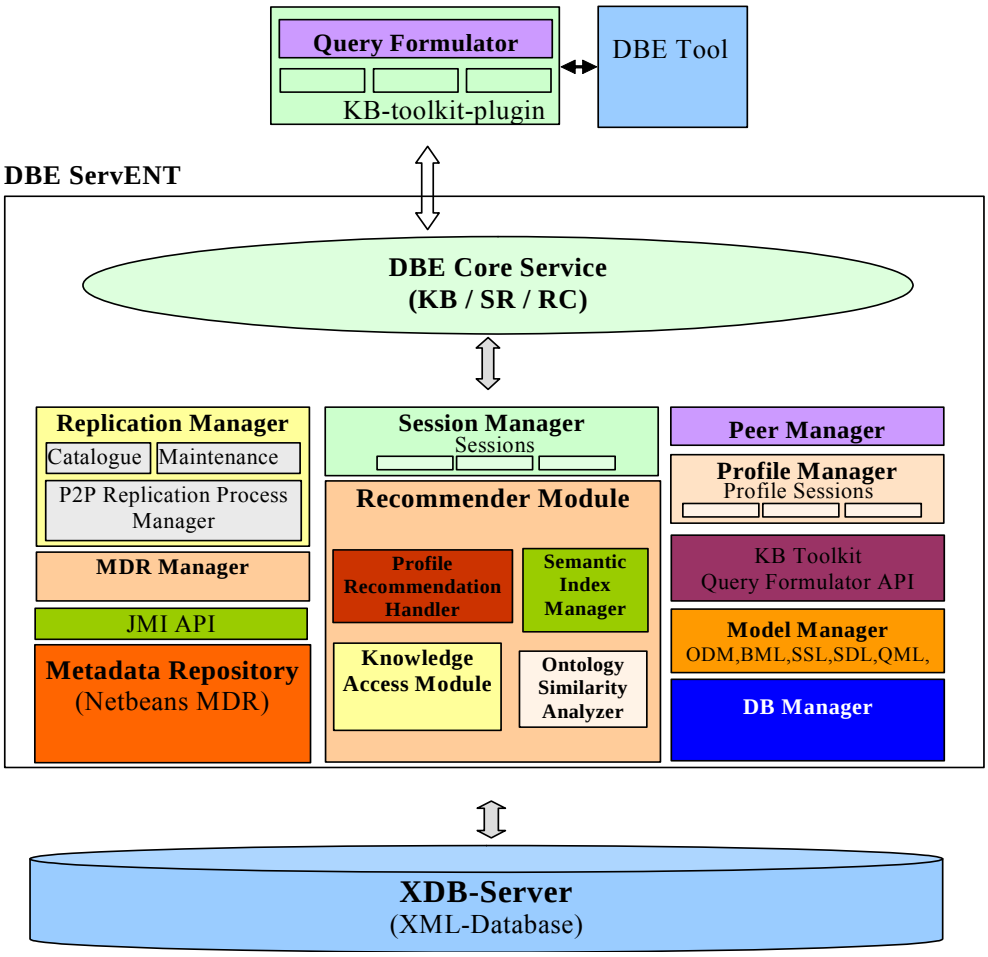

Figure 1: The architecture of the Knowledge Base Infrastructure.

The KB infrastructure provides a common persistence and knowledge management layer in the digital ecosystem. It offers a set of APIs and tools for accessing the DBE Knowledge (M1 Model information and M0 data). The components that comprise the basic KB infrastructure are described in deliverable D14.3 [41].

Figure 2 shows the architecture of the Recommender Module in detail. Recommender implements the components related to the evaluation of candidate services and candidate business partners in the process of discovering or composing services and establishing partnerships respectively. The Recommender Module is responsible for matching preferences with business descriptions and service descriptions. The major assumption behind the design and implementation of the Recommendation mechanisms is the existence of powerful business and service Ontologies that capture the semantics of business models and service descriptions. These Ontologies are used to define the corresponding preferences for businesses and services. The recommender exploits Profile Manager to store preferences and all recommendation mechanisms operate on top of the native XML database to implement the necessary matching functions between preferences and business/service descriptions.

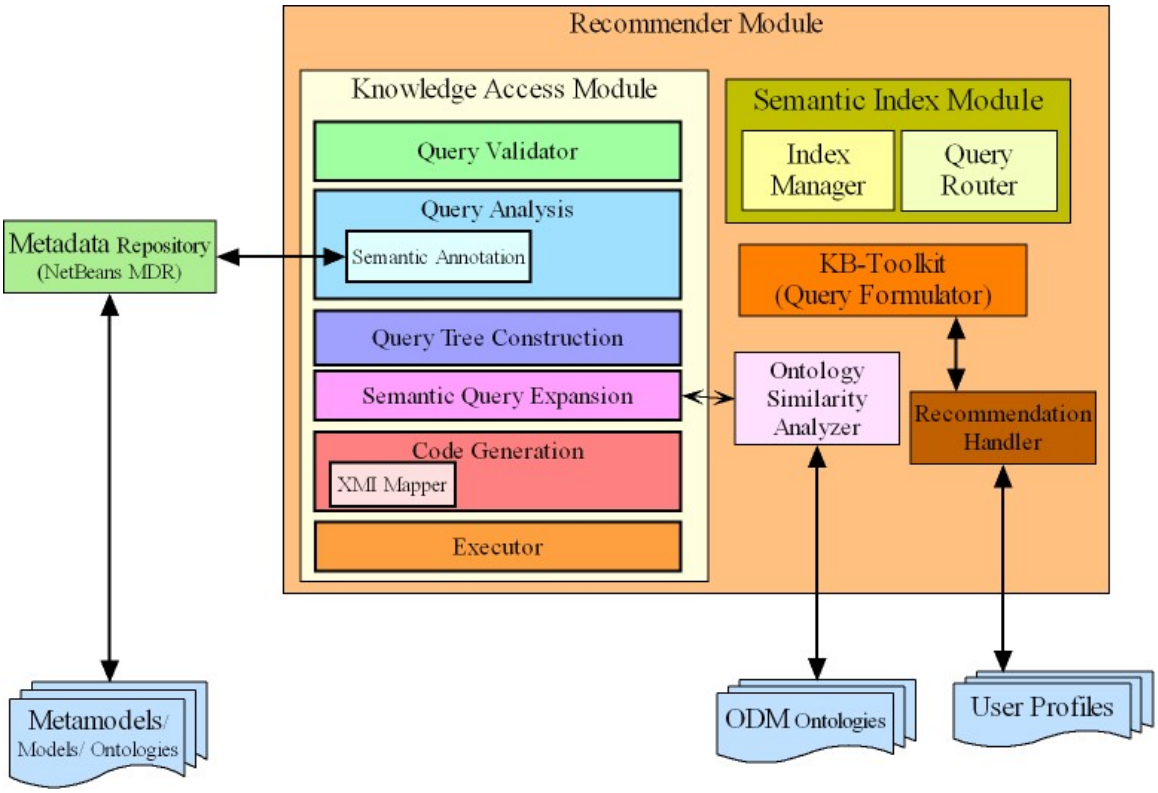

Figure 2: The architecture of the Recommender Module.

The knowledge access module uses a knowledge access language, the Query Metamodel Language (QML) that has been specially designed to make use of the semantic information from metamodels, models, and ontologies. QML has been already presented at the previous deliverables of the recommender (D17.1 [24] and D14.4 [42]). Slight changes occurred and now QML is even more powerful in expressing complex queries that use metamodel, model, and ontology terms. Queries can be formulated by using the Query Formulator module which automatically constructs fuzzy

queries in terms of QML that use senses from metamodels, models, and ontologies. The query formulator module uses the JMI Reflective API from the MDR Manager in order to understand the context of the query terms.

The Query Analysis module analyses the QML query in understandable, for the specific peer, terms. Thus, a query is re-analyzed in each peer and it is reformulated in order to be able to retrieve as much relevant information as possible. These terms are annotated semantically with metamodel, model and ontology information located in each peer. In order to provide such facilities it uses the JMI reflective API of the MDR Manager to access the MDR repository.

The Query Tree Constructor is responsible for creating an execution tree of the query that will provide to the executor module all required information for executing the query. This information is retrieved from the metamodels, models, and ontologies by using the JMI reflective API.

The Semantic Query Expander expands the query tree with new query branches which are semantically equivalent or semantically related to the original query. The information required for the expansion are provided from the semantic annotation process and the from ontology elements similarities provided by the Ontology Similarity Analyzer. For this purpose an ontology mapping mechanism has also been implemented.

The Code Generator module parses the execution tree and constructs queries that can be executed in the current data management system. The Knowledge Access Module is now supported by a powerful native XML database (Berkeley XML DB). This database supports XQuery language and, thus, the code generator produces XQuery code. In order to produce XQuery, code generator needs information on how metamodels, models, and data are mapped into XML. The Mapper module provides this information.

The Mapper module, as mentioned above, is responsible for keeping information of how metamodels, models, and data are mapped into XML documents. Although we use the XMI provided functionality for mapping models into XML, which is straightforward, there is a number of issues that the Code Generator needs to know when producing XQuery that retrieves XMI documents. These issues refer to knowledge of the models and to the way that the XMI contains this information. This information is provided partly by the query execution plan and partly by the Mapper.

The Executor is responsible for executing the query and retrieving the results. Although, the XQuery engine of the XML database does the actual execution of the XQuery statements, the Executor provides a common API for all queries and results for the other modules.

The Ontology Similarity Analyzer parses all ontologies in the system and based on a specific number of rules and criteria finds and creates similarities between ontology elements with a respective similarity weight. It also creates and keeps a string index of all ontology elements.

The Profile Manager manages the storage, retrieval and updates of the SME profiles in the database. The profiles are represented as XMI documents following the User Profile Metamodel (UPM).

The KB toolkit contains useful software components and export APIs for Query Formulation, Service Manifest processing and others.

Three services of DBE use the above functionality. Namely: the Knowledge Base Service (KB), the Service Registry Service (SR), and the Recommender Service (RC). The first two use these modules to answer queries posed against them and the latter (RC) uses the modules to provide recommendations depending on user preferences. All of them are services distributed on the DBE P2P network, i.e. each peer may have a knowledge base, a service registry, and a recommender.

As the DBE knowledge (models, services, etc) is distributed in a P2P environment, certain mechanisms are needed in order to make the query processing efficiently, based on the general architecture and infrastructure of the DBE. The knowledge access module provides a semantic indexing mechanism used for the query routing needs. This mechanism makes use of a completely decentralised semantic index based on a learning process and exploiting the rest of the DBE infrastructure for knowledge management. The indexing functionality is based on the exploitation of the semantic overlay network and the available infrastructure for searching and querying with the use of ontologies and the MDA architecture over the P2P physical network.

3 The Knowledge Access Language

This section discusses the final version of Query Metamodel Language (QML) and its integration with the adopted MOF architecture. Note that this version was already discussed in deliverable D14.4 (Second release of Recommender) [42] and is added here for convenience.

QML leverages the Object Constraint Language (OCL2.0), which has been used as the formal basis of its metamodel. QML supports powerful expressivity of queries on any metamodel and between them.

QML's main aim is to be flexible enough to express queries for all MOF metamodels, models, and data. Moreover, QML's metamodel should be such that queries can be formally analyzed in order efficient evaluation and semantic expansion to be possible. As last aim QML should enable in a simple manner fuzzy queries to be expressed. In order to achieve the first goal we need a language that is expressed in terms of MOF and defined as a MOF Metamodel that "moves" between layers and for the second goal we need a strongly typed language. Both these prerequisites are met by the Object Constraint Language (OCL2.0), which has been used as the formal basis of QML's metamodel.

OCL formal semantics are based on UML 1.4 and we had to align them to MOF 1.4 in order to be able to use it for querying and/or applying constraints. In our work UML metaclasses referred by the OCL meta-model have been suitably aligned to MOF metaclasses, using similar ideas with Loecher at al. at [12]. The elaborated formal semantics refer to these metaclasses. The differences and the alignment adopted can be seen in Table 1.

UML Metaclasses (referred from the OCL2.0 Abstract Syntax metamodel)	**MOF1.4 Metaclasses (referred from the QML metamodel)**
ModelElement	ModelElement
Classifier	Classifier
DataType	DataType
PrimitiveType	PrimitiveType
Attribute	Attribute
AssociationEnd	AssociationEnd
Operation	Operation
EnumerationLiteral	EnumerationType (with multivalued attribute labels)
AssociationClass	*MOF does not support it and therefore QML does not include it.*
Messages	*MOF does not support messages, therefore QML does not provide messages support.*

Table 1: UML metaclasses that have been deprecated (EnumerationLiteral, AssociationClass and Messages) as well as the aligned UML metaclasses to the MOF ones.

The QML is defined as a MOF metamodel. As it is a properly aligned version of OCL it essentially enriches the MOF language with a constraint language. The OCL constraints do not have a mechanism for defining result types. Moreover, an OCL constraint refers to one and only class or object (the so called *context*), which is a drawback for complex queries. For these reasons we introduced the Query Context Declaration meta-class. This meta-class's semantics are somewhat equivalent to the SQL *select* statement. For the rest of its structure QML utilises the concrete and abstract syntax of OCL. Due to space limitations this part is not formally presented in this paper but it can be found at [18, 2]. By extending the OCL core QML supports highly expressive queries as it inherits a set of valuable characteristics from OCL; it is an object-oriented, strongly typed language able to navigate through not only metamodels (M2 layer) but also on models (M1 layer) – as explained in detail at subsection D. Therefore, a query expression can be semantically analyzed and be able to query both models and data.

Most important functionalities of OCL, and therefore QML, are to declare (*Let* expression) and use variables (*Variable* expression), to navigate through model elements (*PropertyCall* expression), to loop through collections (*Bags, Sets* and *Sequences*) of model elements (*select*, *collect, exists* and *forAll* operations), to express *if-then-else* statements and literals. Operations are defined as a model-element navigation process. Next subsection discusses in more detail the QueryContextDeclaration meta-class.

3.1 The Query Context Declaration metaclass.

Figure 3 shows only a part of the context declarations package with the QueryContextDecl meta-class's syntax which is explained in detail here. It has to be noted that this extension approach stands outside the QML core metamodel and therefore it does not affect the compatibility of QML with OCL.

An instance of the QueryContextDecl class represents formally a query which is defined with a name (*simpleName*). The query returns a set of MOF objects of type *result* (may be any *Classifier*) that hold for the criteria defined at the OclExpression *bodyExpression*. Note that the result type typically is a Collection of *Tuples*. These types are defined at the Types package of the OCL specification. The bodyExpression is analogous to the "where" part of an SQL statement but more powerful. QueryContextDecl has a set of VariableDeclarations as contexts. Contexts are analogous to the "from" operand of an SQL statement. The context of a query is the object type where the constraint bodyExpression refers to. An instance of QueryContextDecl may have any number of contexts which allows queries to combine semantic information from more than one metamodels or models. The QueryContextDecl also has a set of VariableDeclarations as input arguments and another set as output arguments. The output arguments are analogous to the "select" part of an SQL statement. Note that the result type is automatically derived by the stated output arguments. The input parameters are used in order to make queries reusable and modular. In this manner, one can form and store query templates and reuse them at any time.

In the next and in the following subsections we present representative query expressions formulated with QML. The objective is to give an informal presentation of the semantics of the QML expressions, whereas the formal presentation can be found at

[18]. The first simple example explains the use and the semantics of the QueryContextDecl meta-class. The second example makes use of the let expression of OCL (please refer to [2]) to show how aggregation can be performed. When query expressions refer to M2 (i.e. available M2 metamodels) they obtain, as a result, qualified M1 models. These examples demonstrate also how to query for models and the last example shows how a query for data can be expressed.

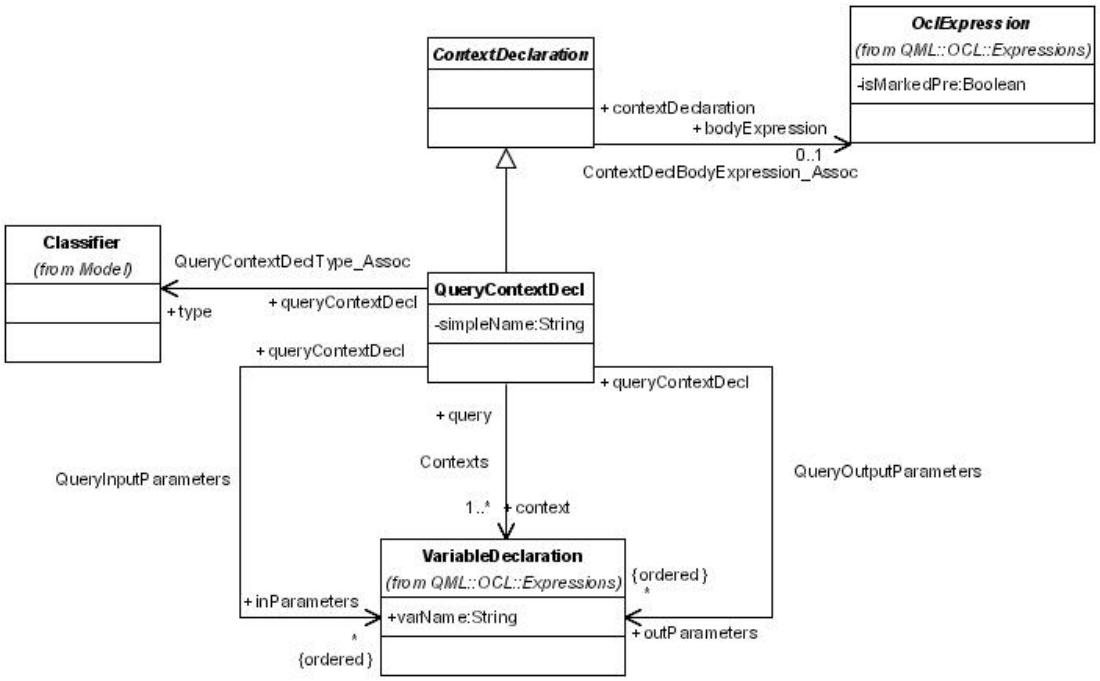

Figure 3: Part of the Context Declaration Package, showing the Query Context Declarations metaclass and its associations.

3.2 A Simple QML query

We will first examine a simple example that demonstrates the usage of the QML metamodel. In section IV, we will explore, through more complex examples, the expressiveness of QML and its support for similarity ranking. The example queries are driven by the Semantic Service Language (SSL) metamodel [8], which is one of the metamodels imported and supported in the DBE knowledge base that allows the semantic description of services.

The following SSL primitives are used for the formulation of the example queries and are depicted in figure 4:

ServiceProfile: A service profile is a model according to which a service will be semantically described.

ServiceAttribute: An attribute (of a service profile) defines a slot of semantic information for a particular profile.

Consider the following statement which retrieves all the service profiles that appear to have a name equal to "Hotel":

Context A: SSL::ServiceProfile simpleQuery

A.name = "Hotel"
out RServiceProfile := A

Figure 5 shows the QML representation of this statement. In terms of the QML metamodel the semantics of the above query can be intuitively explained as follows: There is a query defined in a QueryContextDecl object with the name "simpleQuery". It has a context named "A" with type the MOF Class "ServiceProfile" of the "SSL" metamodel. It has an output parameter named "RServiceProfile" that is assigned a value from the variable "A". Note that, we can imply from the output parameters the result type and in the case we had multiple result arguments the type of the query would be implied as TupleType. The body of the expression is an operation on a property of the context. The OclExpression is responsible for the navigation (A.name) in a formal manner and the operation ("=").

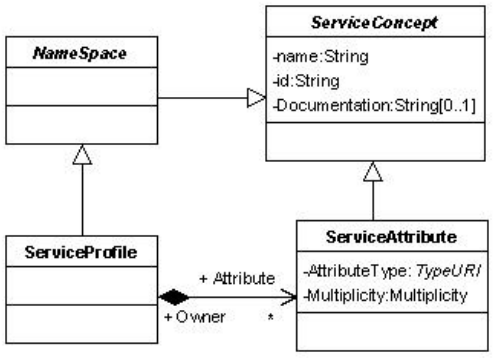

Figure 4: A part of the Service Semantics Language (SSL) metamodel

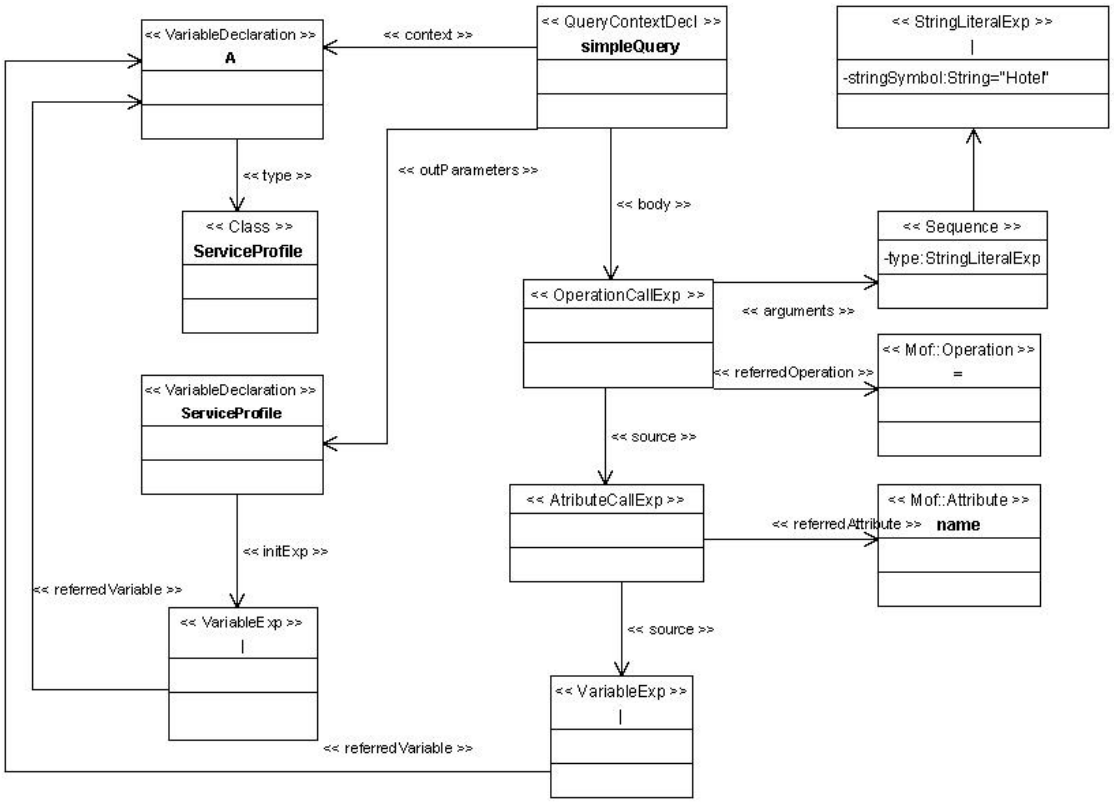

Figure 5: QML representation of the query: Context A: SSL::ServiceProfile simpleQuery A.name = "Hotel" out ServiceProfile: = A.

3.3 Aggregating objects

Many queries involve forming data into groups and applying some aggregation function such as count or sum to each group. The following example shows how such a query might be expressed in QML, using the part of the SSL metamodel shown in the previous section.

The following QML query finds all "ServiceProfiles" that are named "Hotel" and have at list one ServiceAttribute named "HotelName" and have totally more than two "ServiceAttributes".

Context A: SSL::ServiceProfile simpleQuery
let B := A.Attribute in
A.name = "Hotel" and
B->exists(name = "HotelName") and
B->count() > 2
out RServiceProfile := A,
NumberOfAttributes := B->count()

Note that "A" bound by the context clause, represents an individual "ServiceProfile", while "B" bound by a let clause, and represents a set of "ServiceAttribute" items. "Attribute" is the MOF AssociationEnd connecting the class "ServiceProfile" to the class "ServiceAttribute". Note that, Each MOF Association connects two classes. The two ends which have a name and multiplicities are the MOF Association Ends. The "Attribute"

association end has a multiplicity of zero to many, and as such the statement "A.Attribute" will result to the set of "ServiceAttribute" that belong to "A" rather that a single "ServiceAttribute". We generally treat the MOF Associations as join conditions of classes.

For grouping objects we follow the same approach with XQuery. There are a number of known limitations on this approach. For example we do not allow aggregating results and this is mainly due to high complexity in matching elements (classes, objects, etc) between themselves, as they have properties or depending classes. As it is discussed on section V these limitations are not a drawback; we can express very complex queries in just these terms. Nevertheless, we focus on addressing these limitations in later stages of our research when answering more complex problems, as, for example, how we aggregate results that come from different repositories, etc.

3.4 Querying instances

The previous examples showed how QML is used to find models and in this section we will show how we can query instances of these models. Note that instances of the models are the M0 level data, the actual data. The query has to be expressed in terms of the models. From a technical point of view this is possible because MOF is defined in terms of itself and, thus, it resides not only on level M3 but on all levels. That is the reason OCL can be used to express constraints of the MOF metamodel. The only requirement is the M2 metamodels not only to be instances of MOF but also extend it, as for example UML (see also [1, 2]).

Consider the following statement that searches for the "Hotels" that have at the "HotelName" ServiceAttribute the value "Hilton" and are located in "Athens".

Context A: HotelModel#Hotel instanceQuery
A.HotelName = "Hilton" and
A.HotelAddress.City = "Athens"
out Hotels := A

The difficulty in this case is that this query is expressed in terms of a specific model (the "Hotel" model) and may not be able to retrieve the Athens Hilton hotel if it is expressed in terms of another model, for example "HotelModel2" that has a different structure. This is not a critical problem when searching for models because metamodels are not considered to change. This is where the semantic expansion of the query is vital.

4 The Query Formulator Module

This section describes how QML queries are formulated using the Query Formulator, note that QML queries can be directly created and run, but we offer this module to make things simpler as QML is quite complex. This section also presents how fuzzy queries are formulated and how keyword queries are analyzed. The advanced query formulator module, a module used to offer simpler and more advanced capabilities to users, is also presented here.

The Query Formulator module is responsible for producing fuzzy QML queries based on a set of weighted criteria. The criteria may be structured, semi-structured, or unstructured (as in keyword-based search). A criterion is described by five parameters, which are shown in Table 2. Context is the model element (in either M1 or M2 level) that must exist and conform to the criterion. Path is the path from the context to an attribute (i.e. name, price, etc). We do not want for example any price to be less than 70 but only the room price of a Hotel element. The operation is the operation of the criterion. The supported operations vary depending on the type of attribute (numeric types support "<", ">", etc and string type support "=", "like", etc); for a complete reference please refer to OCL2.0 specification [1]. Value refers to the value that the attribute must have exact, greater than, etc. Finally, weight refers to how important this criterion is for the general query. The results of a query come to a ranked order of relevance, user may define which criteria are more important and which not. The weight value ranges from 0 to 1.

Context	Path	Operation	Value	Weight	Description
ServiceProfile	[attribute, name]	=	"Address"	0.5	a structured criterion searching for models
Hotel	[rooms, price]	<	70	1.0	A structure criterion searching for data
	[rooms, price]	<	70	1.0	A semi-structured criterion searching for data
			"Finland"	1.0	A unstructured criterion

Table 2: Examples of criteria for the query formulator API

When users formulate all the criteria into QML expressions, they create the general expression by joining the simple expressions with disjunctions and conjunctions. The general expression is then formulated, along with the result arguments, into the final QML query.

When value is of string type it can either be a single word or a phrase. The value "Los Angeles" will formulate a query searching for the phrase "Los Angeles" and not for the two words "Los" and "Angeles". If users require the second option they can create

queries with two criteria. The case of values is preserved during the formulation process and it is up to the execution engine on how to handle it.

The idea of fuzzy queries construction is explained in the next sub-section.

4.1 Formulating QML Fuzzy Expressions

In this section we present a small introduction to the information retrieval techniques that we used to support fuzzy queries and then we demonstrate with an example how queries are formulated into QML. For a detailed description of the IR techniques please refer to the 1st and 2nd release or recommender [24, 42]. Note that this framework is independent of metamodels. Thus, if DBE evolves and uses new metamodels, this framework is not needed to change not only at design level but also at implementation level.

Query results and recommendations in an environment where information is modelled with different metadata structures even in the same domain have poor precision and recall. As a result we needed to apply techniques and concepts from Information Retrieval (IR) with relevance rank into QML. Many of the known techniques are platform specific and therefore not applicable in our context. QML was enhanced with the Extended Boolean Model in order to address the problem in a platform independent way.

We have developed a framework for QML processing that incorporates Information Retrieval functionality and that is based on the Extended Boolean Model. This extension stands outside the QML metamodel and therefore it does not affect the compatibility of QML with OCL.

The knowledge access context that we consider is twofold. It is related to pure search functionality as well as to recommendation mechanisms. At a technical level the information filtering/retrieval approach is uniform for both desired functionalities. All recommendations and discovery requests computed by the Query Service could be considered as similarity based retrieval requests and can be modelled using the same general mathematical framework based on information retrieval theory. This framework is summarised here and the implementation of this framework on top of a MDA repository is given.

A generalised request for retrieving items, which belong to an items universe I that is described in terms of a feature space F, corresponds to a query q that consists of a structured set of features F', which is a subset of F. This general scheme can be used to describe the kinds of functionality (see the "Interpretation" column) shown in Table 3.

The objective here is to define a generalised information retrieval framework that could be used in all of the above scenarios. Moreover, taking into account that the correspondence between information items and features, as well as queries and features, could be implemented in a MDA-based repository, we extended the generalised framework to work on such a system and developed mechanisms that use pure QML in order to support all kinds of recommendation functionalities.

Universe I	Feature space F	Queries Q	Interpretation
Models and Data	Metamodel and Model features	Preferences of the model in terms of possible partners	Retrieve models and data that are similar to some preferences
Models	Metamodel features	Desired metamodel features	Retrieve models that are similar to a given query
Data	Model features	Desired model features	Retrieve data that are similar to a given query

Table 3: Different kinds of Recommendation functionality expressible by a general information retrieval system

4.2 Implementation of the p-norm extended Boolean model using QML

The Extended Boolean Model is a generalization of the Boolean logic based on the fuzzy set theory. It provides formulae for the evaluation of complex Boolean expressions so that the qualifying information items can be given a rank in the range of [0, 1] instead of just a Boolean true/false result. Various studies [5] prove its superior performance in comparison with the traditional information retrieval models.

In order to actually evaluate the queries, one should give the evaluation functions f_{NOT}, f_{AND}, and f_{OR}. There are numerous possible definitions of these functions. Study [5] presents the definitions that correspond to the p-Norm Extended Boolean Model, which is the most general one. Note that the functions f_{AND}, and f_{OR} are n-ary instead of binary. This is due to the fact that these evaluation functions are not commutative as their Boolean counterparts.

There are numerous strategies for the implementation of Extended Boolean Model on top of a RDBMS [7],[9]. But we need an implementation that is platform independent, and as such we need to implement it on top of QML. However, there is a straightforward implementation of the p-Norm by using QML in case that the queries that are accepted by the system have a simple form (either conjunctive or disjunctive queries).

To demonstrate the technique, let us assume that the queries accepted by the system are simple disjunctive queries. Let us further assume that Items are the Model elements or Objects depending on the level the query refers to (i.e. M0 or M1). Features are the elements connected with the Items. Note that a Model Element may be a Feature for another Model Element or an Item. Depending on the query context the Items are resolved. Every connection between an Item and a Feature is a path from the Item to Feature and the Feature's value. This path can be defined by a QML path expression and has a weight denoting how relevant the Feature is for the Item. Every query consists of query terms (t_{qi}) which are QML path expressions and a query weight (w_i) denoting how important the term is for the overall query. Thus, every term must match a feature item path. In other words, if an Item (i.e. Model Element) is connected to a Feature with the query term (i.e. the QML path expression) a weight (a_i) is returned on how "strong" this connection is. For example if an Item has the same path to the Feature as the query term the weight 1 is returned; if the path is "alike" the query term an intermediate weight is returned denoting how relevant the two paths are; and if the Item-Feature

path has nothing to do with the Query path zero is returned. This weight is the Item-Feature weight. Equation 4-1 is the f_{OR} evaluation function of the p-norm model.

$$\left(\sum_{i=1}^{n} a_i^p \cdot w_i^p \Big/ \sum_{i=1}^{n} w_i^p \right)^{1/p} \quad 1 \le p \le \infty \qquad (4\text{-}1)$$

To join all the query terms with the fuzzy OR operation, we have to calculate the fuzzy OR evaluation function following the p-Norm model from equation (4-1).

An example of a QML query expression that implements the abovementioned ideas is shown below:

Context A: SSL::ServiceProfile
Let fe1:= A.Attribute->exists(name = "HotelName"),
fc1:= A.Attribute->exists(name.contains("HotelName")),
fe2:= A.Attribute->exists(name = "Address"),
fc2:= A.Attribute->exists(name.contains("Address")),
a1 := if (fe1) then 1 else if (fc1) then 0.5 else 0,
a2 := if (fe2) then 1 else if (fc2) then 0.5 else 0,
*r := pow (pow(a1*w1, p)+ pow(a2*w2, p), 1/p)*
in(fe1 or fc1 or fe2 or fc2) and r>0
out Rank := r, ServiceProfile := A

ServiceProfile is the Model Element that plays the role of context and Item. The path *A.Attribute->exists(name = "HotelName")* plays the role of Item-Feature relation. The variables fe1, fc1 and fe2, fc2 as pairs are needed for calculating the Item-Feature weight (i.e. *a*) for each query term. The evaluation function for this calculation is that if the value "HotelName" (for the first case) exists in this path return 1, otherwise if is a substring of the Feature return 0.5, and otherwise zero. This function is calculated in variable *a1*. The final rank comes form the evaluation of the fuzzy OR function at variable *r*.

It is quite easy to develop a similar QML query in case that the queries recognised by the system are simple conjunctive queries.

4.3 Improving relevance ranking

As it was demonstrated in the previous section, the formulated query of the example does not retrieve only *ServiceProfiles* that have exactly the name "Hotel" but it also retrieves those that have a name *like* "Hotel" (for example "HotelReservation"). The latter do not have the same weight as those of the exact mach. This happens to improve the recall of the system.

This section presents the policy we follow in all cases in order to produce the argument *a* of equation (4-1).

Generally, argument *a* represents how relevant is feature *i* to the query term. In many applications, the value of this argument is 0 if the feature does not exist and 1 if exists. In our application, this value can vary from 0 to 1 depending on a set of criteria which are shown in table 4. The weight of qualifying operations is always 1.0 and of non-qualifying operations is always 0.

The actual formulae which results the weight of the numeric semi-qualifying operation is:

$$a_{numeric} = \begin{cases} \frac{1 - |best - avgSemiValues|}{best}, if\ 1 - |best - avgSemiValues| > 0 \\ 0, \text{otherwise} \end{cases}, \quad (4\text{-}2)$$

where *best* is the value of the query (either upper or lower bound) and *avgSemiValues* is the average of the values resulted from the semi-qualifying operation. Note that if

Type	Operator	Qualifying Operator	Semi-qualifying Operator	Value of a for semi-qualifying operator
String	=	=	like	0.5
String	like	=	like	0.5
Numeric	=	=	!=	Larger as closer to best value (eq. 4-2)
Numeric	<	<=	>	Larger as closer to best value (eq. 4-2)
Numeric	>	>=	<	Larger as closer to best value (eq. 4-2)

Table 4: The value of the matching factor *a* depending on the operation and the type of original query

Moreover, the relevance of a feature can be aligned not only to its relevance to the value of a query term, but also to how exact the mach of the feature path to the query is term's one. For example if we look for the path: *Hotel.City* and the feature has the path *Hotel.Address.City* then the feature, with what we have describe so far, would have a relevance value 0. Keep in mind that in Chapter 5, where the semantic exploitation of the query will be discussed, a different value will be assigned to parameter *a*.

4.4 Advanced Query Formulator

The advanced query formulator was designed to provide a more sophisticated way to formulate queries. It introduces reusable components called templates. In each template there are declared the contexts and the attributes of the query, result types and join conditions. It is common that most queries use the same main attributes of the many offered by a model or a metamodel and the template realises this idea.

Advanced query formulator offers the ability to create templates and store them. Later they can be used to build queries by just adding criteria on the attributes. These criteria may form conjunctions and/or disjunctions.

Templates and advanced query formulator are used by sophisticated user interfaces for building queries or user preferences. They can be used by other services for common queries. For example a service from a Travel Agent that needs to search for Hotels could create a standard template with all the needed attributes and reuse it at all queries made by the system. Code Snippet 1 shows a usage example for the travel agent.

```
Template template = new Template();
//Adds a template elememt named "location" searching
//on the BML path "Hotel/Locality" of type String
template.addTemplateElement(new TemplateElement("location",
                           "Hotel::Locality", "String"));
template.addTemplateElement(new TemplateElement("country",
                           "Hotel::Country", "String"));
template.addTemplateElement(new TemplateElement("starCategory",
                           "Hotel::StarCategory", "Integer"));
template.addTemplateElement(new TemplateElement("roomPrice",
                           "Hotel::Rooms::Price", "Integer"));
…
//Create a Query formulator
AdvancedQueryFormulator form = new AdvancedQueryFormulator(
       (QmlPackage)qmlTool.getModelPackage(),
       AdvancedQueryFormulator.INSTANCE_QUERY);


//Initialise the formulator with the static query Template created earlier.
form.setTemplate(template);
//Create a query expression for each criteria (if exists) and added to a
Vector exprs = new Vector();
//Note the last number is a weight of how important query expression is
QueryExpr expr = new QueryExpr("=", "location", "Tampere", 1.0);
exprs.add(expr);


QueryExpr expr = new QueryExpr("<", "starCategory",  "5", 1.0);
exprs.add(expr);


//make an array of query expressions
QueryExpr[] queryExprs = new QueryExpr[exprs.size()];
…
//put the expressions inside the formulator
form.getQuery(queryExprs);
```

Code Snippet 1: A usage example of advanced query formulator API for a travel agent searching for Hotels.

4.5 Formulating Keyword Expressions

This section discusses how keyword expressions can be formulated into valid queries. Keyword expressions consist of query terms which can either be unstructured or semi-structured as was shown at table 2.

Unstructured query terms refer to those that include only a single word. Examples of those query terms include "Hotel" and "Finland" and while the first one refers to a feature of a model or ontology, the second refers to instance data. The mechanism of keyword formulation make sure that the keyword expression is queried against on both layers i.e. models and instances.

Semi-structured query terms refer to those that provide a path (not full path), an operation, and a value. Examples of this case include: Country="Finland" and Hotel/Room/Price<100. It is assumed that the path refers to a model path and the value to an instance of that path. Thus, the keyword query mechanism searches for services that the model path expressed exists and has as instance the value of the expression. Both the path and the value may not mach exactly but with a similarity rank.

The general keyword query may include any number of both unstructured and semi-structured keyword query terms. The mechanisms expressed in this chapter that refer to the similarity rank also hold for this case while each query term may have a weight denoted as float after the query term (i.e. Hotel^0.5 or Country=Finland^0.9). If no weight is assigned then value 1.0 is assumed.

The parsing of the text into a query expression is done by java program produced by JavaCC application based on the grammar shown in appendix B.

This query is also expanded using ontology information explained in the following section.

5 Semantic Exploitation

This section discusses how the semantic information of the query can be exploited in order the system to be able to recommend services (in other terms data) that have different structures (models) from the model the query was expressed, but refer to the same kind of service.

Recall the problem stated at section 3.4 where a data object could not be retrieved because the query was expressed in terms of a different model. This problem is addressed if we reformulate the starting query in terms of the second model adapting properly (downscale) the weights of each term. The reformulation process is not an easy task in the relational world of different schemas for each database [3] [14]. That is another reason we selected a query language that is expressed in semantic terms instead of directly using a PSM language as XQuery, SQL or SPARQL.

This semantic information can be used to reformulate the query by understanding, which classes between models are semantically equivalent. This can be done by considering information from the metamodel (e.g. two model elements that are instances of the same metaclass and have similar properties) or if the metamodel is using information from ontologies (please refer to section 3) we can use equivalent relationships from ontologies and reformulate the query by the means of the ontologies.

5.1 Defining the Problem

Before addressing the problem, we have to formally define it. The case is that any SME can describe its service and its business following a model of its own. Thus, there is a large diversity of models even for the same kind of businesses (e.g. for Hotels). The main case is that each domain has a small number of models, which are reused from the large variety of SMEs with few or none changes. In other words, each domain is described by a small number of model groups, where each model group consists of a main model and a number of sub variations of this model. Moreover, all the models use concepts from domain ontologies, which in DBE are models of the ODM metamodel [4, 18]. We expect that each domain is described not by one but by many ontologies.

With this in mind, we can state that when we search for services, which match a set of criteria, we want to discover all the services of a specific domain that these criteria apply even when the services follow different models/ontologies or refer to similar services. However, in DBE's environment, where knowledge is highly distributed, how can domains be strictly defined? The answer is: it cannot. Each peer/knowledge base might give a different classification of services into domains depending on its knowledge.

If models were just different structures for data, one could simplify the problem into reformulating the query for each structure. However, in our case, each model is not just a structure but it has also senses. For example, Hotel and Hostel are related as they offer accommodation, but they share nothing in common with Restaurants. This kind of information comes from ontologies where it is stated that the concept Hotel is equivalent with the concept Hostel, but no equivalence or any other kind of relation exists for Restaurants.

Another issue to keep in mind is the performance issue. Imagine a peer having 1000 services following 800 different models, for each of which a different query will be reformulated, according to structure differences and ontology senses, and executed. That is; create 800 queries, execute them, and finally, join the results. Thus, even though you can explore all information to enhance the recall and the precision of the system it is prohibited by performance costs.

To include all the above statements in a general case, we specify the problem as follows: The a_i term of equation (4-1) is the matching factor of the feature for query term i. The matching factor is decomposed in two factors c_i and v_i as $a_i=c_i*v_i$, where c_i is the matching factor of the feature's concept to the query term's concept and v_i is the matching factor of the feature's value to the query term's value. The former a_i as described in Table 4 of section 4.3 is the v_i and thus, the issue is to resolve the value of c_i.

The approach we followed to resolve related paths and their corresponding c_i's and the model for creating an expanded query which makes use of this information to improve the system's recall is explained in this chapter. First we discover the related paths for each query term of the query by making use of the semantic information of the query and the ontology concept similarities found at an earlier stage (see section 5.2). Along with the related paths we compute their corresponding weight, which denotes their relevance to the original query term. Finally we reformulate the query by adding new query terms. The final query can then be processed to produce a validating XQuery expression, which will be executed by the execution engine.

5.2 Ontology Similarity Analyzer

In this section the creation rules of ontology mappings are discussed. Although this section applies for ODM ontologies keep in mind that it is also valid for OWL/RDF ontologies as there is a one to one mapping between them [18]. In order to formally define these rules we need first to define the ontology space.

5.2.1 Definition of Ontology Space

The set of all ontologies is the Ontology Space O. O consists of the ontology elements e_i which are divided into four categories: Classes (o_i), Object Properties (op_i), Data Types (d_i) and Data Type Properties (op_i), i.e $\{o_i\}\cap\{op_i\} \cap\{d_i\} \cap\{dp_i\} = \{e_i\}$. Moreover Data Type Properties and Object Properties consist Properties (p_i), i.e. $\{op_i\} \cap\{dp_i\} = \{p_i\}$.

Inside O there exist a number of functions as follows:

- Equivalence (~). $\mathbf{e_1 \sim e_2}$
- Subclass (or subproperty) (IsA). $\mathbf{e_1}$ **IsA** $\mathbf{e_2}$
- Domain: $\mathbf{p_1}$ **domain** $\mathbf{\{o_i\}}$
- Range: $\mathbf{op_1}$ **range** $\mathbf{\{o_i\}}$, $\mathbf{dp_1}$ **range** $\mathbf{\{d_i\}}$

In the ontology space paths between a class and a datatype exist:

$$\mathbf{o_1 \text{ -> } p_1 \text{ -> } \{o_i \text{ -> } p_i\}^n \text{ -> } d_{n+2}}$$

where the arrow (->) is defined as follows:

- $\mathbf{o_1 \text{ -> } p_1 \text{ => } p_1}$ **domain** $\mathbf{o_1}$

- **p_1 -> o_1 => p_1 range o_1**
- **p_1 -> d_1 => p_1 range d_1**

5.2.2 Definition of Ontology Similarity Rules

We define a new non-symmetric function in the ontology space *O* denoted as similarity (s) between two ontology elements with a similarity rate *s* € [T_{sim}, 1], where T_{sim} is a threshold below which no mapping can exist and can take values in the interval [0, 1).

Definiton 1. A similarity *s: O x O* € [T_{sim}, 1] is a function from a pair of entities to a real number expressing the similarity between two objects such that

s(a, b) =1 *iff a = b* (definiteness)

The following rules define how similarities are created:
Rule 1. if **e_1 ~ e_2** then **s(e_1, e_2) = re** and **s(e_2, e_1) = re** (equivalence rule)
Rule 2. if **e_1 IsA e_2** then **s(e_1, e_2) = rb** and **s(e_2, e_1) = rp** (IsA rule)
Rule 3. if **op_1 range {o_i}** and **op_2 range {o_i}** then **s(op_1, op_2)=rt** and **s(op_2, op_1)=rt** (type rule)
Rule 4. if **op_1 range {o_i}** and **op_2 range {o_i'}** and **s({o_i}, {o_i'})=r** then **s(op_1, op_2)=rt*r** (type rule 2)

Rules 3 and 4 are indirect as we guess that properties with the same range (i.e. type) have some equivalence. This is needed because most of the times an ontology creator might define that two classes are equivalent but drop out that two properties are equivalent. Four similarity parameters are defined: *re* for equivalence, *rb* for subclasses, *rp* for superclasses, and *rt* for types. Some indicative values are: *re* = 0,9, *rp* = 0,8, *rb* = 0,9 and *rt*=0,8.

The similarity function is transitional with the following rule:
Rule 5. if **s(e_1, e_2)=r1** and **s(e_2, e_3)=r2** and **e_1 ≠ e_3** and **r_1*r_2 > T_{sim}** then **s(e_1, e_3)=r1*r2** (transitional rule)

In order to create the similarities you need one step for the rules one to four and *n* for the transitions where *n* is proven at appendix A that equals to:

$$n = \left\lceil \log(T) \Big/ \log(\max(re, rb, rp, rt)) \right\rceil$$ if max(*re, rb, rp, rt*) ≠ 1.

For a threshold T_{sim} = 0,5 and the abovementioned weights the number of transitions required is *n* = 7.

Although all ontologies belong on the same space and equivalences and sub classing may exist between different ontologies, the common practice in our environment does not incorporate these properties as a general rule. Thus, all the above rules ((1) to (5)) will not perform well when searching for similarities between different ontologies. In order to overcome this problem we introduced the following rule to allow automatic extraction of similarities when information about equivalences and sub classing is not present. The rule is based on similarity between strings and is often described as the edit distance (also called the Levenshtein edit distance [48]), that is, the minimum number of changes necessary to turn one string into another.

Rule 6. if ***levenshtein*****(e_1, e_2)=r** and **r>T_{leven}** and **$e_1 \neq e_2$** and **¬s(e_1, e_2)** then **s(e_1, e_2) = f_{leven}*r**

where T_{leven} is a threshold for the levenshtein distance and f_{leven} is a factor in the interval (0, 1]. For the string based Levenshtein search we used the Apache Lucene[5], which is a high-performance, full-featured text search engine library written entirely in Java. It is distributed under the Apache Licence version 2.

Ontology Links

Apart from the similarity function in order to be able at later steps to find similar paths we need another to keep the links between the ontology elements indexed for fast reference. Moreover these links may be used in future stages for more powerful expansions.

Definiton 2. An ontology link *ol: O * O* -> [T_{link}, 1] is a function from a pair of ontology elements to a real number expressing how strong the connection between these elements is. It is created with the following rules:

Rule 1. if **p_1 domain o_i** then **ol(o_i, p_i) =1**
Rule 2. if **p_1 range o_i** then **ol(p_i, o_i) =1**
Rule 3. if **p_1 range d_i** then **ol(p_i, d_i) =1**

Note that these ontology links have rate one (1). For the time being no transitions exist, but in the future we could provide transitions with respective reduce of the cost. The meaning of transitions here is that there is a path between two elements. The farer two elements are the lowest their link mapping rate. Thus a transition rule would be:

Rule 4. if **ol(e_1, e_2) =r1** and **ol(e_1, e_2) =r2** and **$e_1 \neq e_3$** and **f_{link}*r_1*r_2 > T_{link}** then **ol(e_1, e_3) =f*r1*r2**

where f_{link} is the reduce factor and its value belongs in the interval *[0, 1)*. The number of steps that this transition rule must run is:

$$n = \left\lceil \log(T) \Big/ \log(f) \right\rceil .$$

Both similarities and ontology links are stored as XML files into the XDB Server repository for sound storing and fast retrieval and processing.

5.3 Retrieving Related Paths

In previous sections we defined and created similarities in order to be able to find related paths to a given ontology path. In order to continue with the actual process the following definitions must apply.

Definiton 3. We define a path in the ontology space as *<>: O^n-> O^n* denoting a sequence of ontology elements starting with a class and ending to a datatype:

<e> or **{o_i -> p_i}^{n} -> d_i.**

[5] http://lucene.apache.org/

Definiton 4. We define the function length of a path as $l: O^n \rightarrow N$ denoting the number of elements consisting a path.

Definiton 5. We define the distance between two elements as $\delta: O \times O \rightarrow [0, 1]$. The complement of the distance (δc) is their similarity rate:

$$\delta c(e, e') = 1 - \delta(e, e') = s(e, e') \quad (5\text{-}1)$$

If the no similarity exists then the distance is 1 and the distance complement 0.

Definiton 6. We define the distance between two paths as $d: O^n \times O^m \rightarrow [0, 1]$ denoting the distance between the two paths. We are interested in the complement of the distance which is:

$$dc = 1 - d. \quad (5\text{-}2)$$

Definiton 7. The distance complement between two paths with only one element is their similarity rate (if exists):

$$\text{if } l_{<e>} = l_{<e'>} = 1 \text{ then } dc(<e>, <e'>) = s(e, e') \quad (5\text{-}3)$$

Definiton 8. We define two paths as **related** if their complement distance is greater than zero (0):

$$dc(<e>, <e'>) > 0 \text{ then } <e> \text{ and } <e'> \text{ are related} \quad (5\text{-}4)$$

Before continuing in calculating the distance complement the algorithm for retrieving the related paths will be presented. It is based on the similarities and the ontology links starting from the datatype and looping till the root class is reached. At first it is going to be presented formally and then an example will be given.

5.3.1 Formal Definition of Related Paths Discoverer Algorithm

Problem: Find the paths and their relatedness as a weight is the interval [0, 1] that are related to a given ontology path called the original path.

The algorithm loops, in a bottom up fashion, the elements of the original path. Each loop results in a frontier F of the mapping paths found so far, which is used for the next steps.

Given the original path: $\{o_i \rightarrow p_i\}^n \rightarrow d$ we perform the following steps.

Step 1: Create the frontier F with the similarities of the datatype **d**.

$$F(d) \leftarrow getSimilarities(d)$$

Step 2a: Current element is the next element of the original path, e_i.

Step 2b: Find the elements that are linked to any mapping path of the frontier and are similar to the current element.

Step 2c: Create the corresponding path for the elements found in step 2b and add them the frontier.

$$F(e_i) \leftarrow F \cap getSimilarities(F, e_i)$$

Step 3: Repeat step 2 for all elements of the original path.

Step 4: All paths of the last frontier are mapping paths of the original query and are called candidate paths.

Step 5: Calculate a weight for each candidate path and select those whose weight is greater than a threshold T_{path}.

The computational and space complexity are exponential to the length (n) of the original path and namely:

Computational Complexity: $\mathbf{O\{(2^{n-1}-1)*b\}}$ and Space Complexity: $\mathbf{O\{(2^{n}-1)*b\}}$,

Where b is the branching factor of each step's mapping classes.

5.3.2 Example

Let's examine the algorithm for the following path: Hotel->AddressPr->Address->CityPr ->City and the similarities shown in Figure 6. The black lines are the ontology links and the dashed red lines are the element similarities.

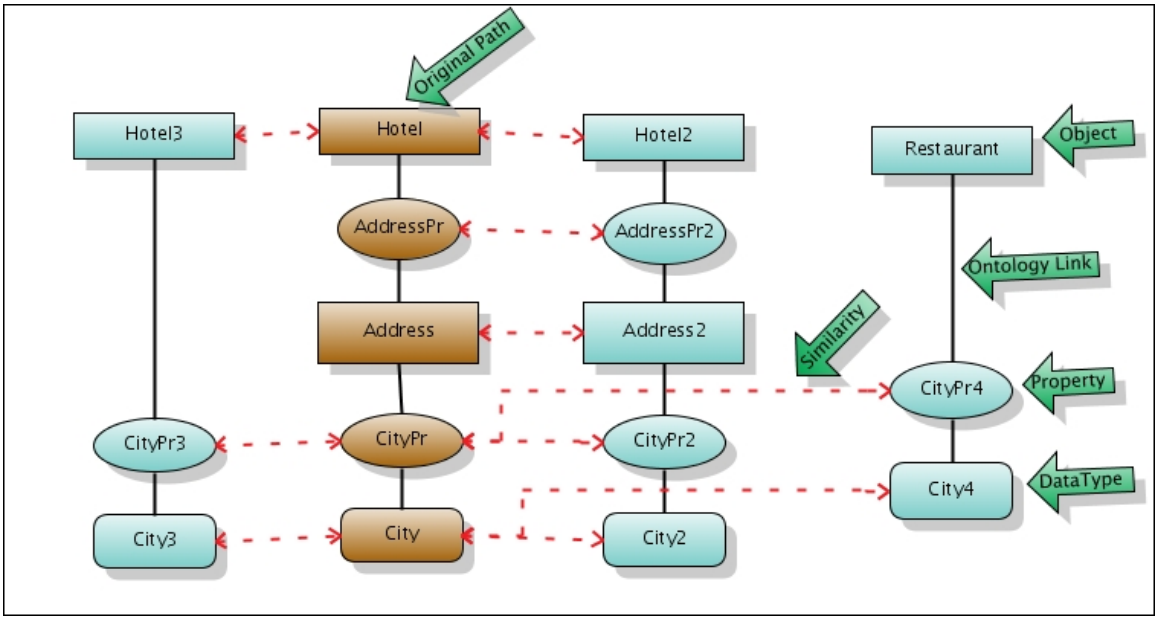

Figure 6: The ontology mappings stored for the main path.

For simplicity reasons we will not examine the properties but just the objects and the datatypes, i.e. the original path now is Hotel->Address->City. The same rules apply for the whole path. In each loop the frontier will look like:

Start: $F\left(\left\langle City\right\rangle\right)\rightarrow\begin{Bmatrix}\left\langle City2\right\rangle\\\left\langle City3\right\rangle\\\left\langle City4\right\rangle\end{Bmatrix}$

Loop 1: $F\left(\left\langle City,Address\right\rangle\right)\rightarrow\left\{\left\langle City2,Address2\right\rangle\right\}\bigcap\begin{Bmatrix}City2\\City3\\City4\end{Bmatrix}$

Loop 2: $F\left(<City,Address,Hotel>\right)\rightarrow$

$$\begin{Bmatrix}<City2,Address2,Hotel2>\\<City3,Hotel3>\end{Bmatrix}\bigcap\left\{<City2,Address2>\right\}\bigcap\begin{Bmatrix}City2\\City3\\City4\end{Bmatrix}$$

The frontier of loop 2 has the related paths. If we didn't add the frontier in each step (which is responsible for exponential complexity of the algorithm) we couldn't retrieve the path <City3 , Hotel3> path.

5.3.3 Calculating Path Distance

Two functions for calculating the path distance exist in the bibliography:

$f_{Do}=d\left(<e_i>_{i=1}^{n},<e_j>_{j=1}^{m}\right)=\lambda.\delta(e_n,e_m)+(1-\lambda).d\left(<e_i>_{i=1}^{n-1},<e_j>_{j=1}^{m-1}\right)$ (Do at al., [46]) which can be written also as:

$$f_{Do}=d\left(<e_i>_{i=1}^{n},<e_j>_{j=1}^{m}\right)=\sum_{i=1,j=1}^{n,m}\lambda.(1-\lambda)^{n-i}.\delta(e_i,e_i)\ \text{ with }\ \lambda\in[0,1] \qquad (5\text{-}5)$$

$f_{Val}=d\left(<e>,<e>'\right)=\dfrac{\sum\delta(e,e')+|l-l'|}{l}$ where *l* and *l'* are the lengths of the <e> and <e>' respectively (Valtchev, [47]) (5-6)

The |l-l'|/l factor of f_{Val} is a compensating factor which does not occur on f_{Do} but f_{Do} as will be shown needs it in order to perform correctly in our case.

Table 5 shows the results of these two functions for the previous example and for all related paths. None of these two functions be used each one for different reasons.

l = 3, δ = 0,1 (for all mapping elements) and λ=0,5.				
#	***Related Path***	f_{Do}	f_{Do}***(with compensation)***	f_{Val}
1	**<City2>**	0.05	0.716	0.64
2	**<City3>**	<<	<<	<<
3	**<City4>**	<<	<<	<<
4	**<City2, Address2>**	0.075	0.40	0.4
5	**<City2, Address2, Hotel2>**	0.0875	0.0875	0.1
6	**<City3, Hotel3>**	0.0625	0.395	0.4

Table 5: The related paths of <Hotel, Address, City> and their distances using functions f_{Do}, f_{Do} with compensation and f_{Val}.

The function f_{Do} cannot be used because is very dependent on the similarity between the last element of each paths, which is not correct in our case because we need a more balanced function.

The second function, $f_{Va;}$, does not suffer from the previous problem, but as it can be seen from the table the cases 4 and 6 do not differentiate while being completely different. Case 6 is a complete path whereas case 4 is not.

Thus, we need function that produces balanced distances while it differentiates all cases. Function $f_{Va;}$ has a compensating factor which corrects the sum of element distances, namely: *l –l'.* This factor tells us that the distance of two paths is analogously to the difference of their lengths. In order to use a different factor we define a new measure; the bondage of two related paths.

Definiton 9. The tenacious bond $b: O^n \times O^m \rightarrow [0, 1]$ between a related path <e>' to an original one <e> is a real number in the interval [0, 1] denoting how close the last element of <e>' is to the last element of <e>. We calculate *b* as follows:

$$b(< e_i >_{i=1}^{n}, < e_j >_{j=1}^{m}) = \frac{k}{n} \ , \qquad (5\text{-}7)$$

where *n* is the length of the original path and *k* is such that e_k is similar to e_m.

Examples.

b(<Hotel, Address, City>, <City3>) = 1/3 = 0,33 because *City* (k=1) is similar to *City3*.

b(<Hotel, Address, City>, <Address2,City2>) = 2/3 = 0,66.

b(<Hotel, Address, City>, <Hotel3,City3>) = 3/3 = 1.

Definiton 10. We define the evaluation function of path distance as follows:

$$f = d\left(< e_i >_{i=1}^{n}, < e_j >_{j=1}^{m}\right) = \frac{\sum_{i=1,j=1}^{n,m} \delta(e_i, e_i)}{n} + (1-b) = \frac{\sum_{i=1,j=1}^{n,m} \delta(e_i, e_i)}{n} + \frac{n-k}{n} \Leftrightarrow$$

$$f = \frac{\sum_{i=1,j=1}^{n,m} \delta(e_i, e_i) + n - k}{n}, \quad (5\text{-}8)$$

where *m, n* are the lengths of the original path and the related one respectively. This evaluation function is somewhat the same with f_{Val}. At first glance function *f* does not return values in the interval [0, 1], but it actually does. Appendix A hosts the proof of it. The results of all the functions are compared and shown in Table 6.

n = 3, δ = 0,1 (for all mapping elements) and λ=0,5.						
#	***Related Path***	***f_{Do}(with compensation)***	***f_{Val}***	***f***	***k***	***m***
1	**<City2>**	0.716	0.64	**0.70**	1	1
2	**<City3>**	<<	<<	**<<**	<<	<<
3	**<City4>**	<<	<<	**<<**	<<	<<
4	**<City2, Address2>**	0.40	0.40	**0.40**	2	2
5	**<City2, Address2, Hotel2>**	0.0875	0.10	**0.10**	3	3
6	**<City3, Hotel3>**	0.395	0.40	**0.066**	3	2

Table 6: The related paths of <Hotel, Address, City> and their distances using functions f_{Do} with compensation, f_{Val}, and f.

Some of the related paths having very large distances can be omitted by using a threshold. We used a threshold of 0.3 and thus we select the cases 5 and 6. These will be used later on to semantically expand the query.

5.4 Semantic Query Expansion

In the previous section we showed how relevant paths can be discovered with their corresponding distances. In this section the modelling framework for expanding queries will be described.

A query consists of a number of query terms (qt_i) which should be matched over a dataset *D*. Each query term consists of a path expression (p_i), an operation (op_i), a literal value (v_i) and a weight (w_i) denoting the user importance on this term. The following formula describes this:

$qt_i = (p_i, op_i, v_i, w_i)$ (5-9)

The path expression of the query term consists of two parts the model part (or model path - *modelPath*) and the ontology part (or ontology path - *ontoPath*) any of which may be empty. Thus the path is:

$p = \{modelPath, ontoPath\}$ (5-10)

At this point we define a new element; the query branch (*qb*). The query branch will hold the information of an expanded query term and will consist of a number of query terms and a weight denoting how important this branch to the overall query is.

$qb_i = (w, \{qt_{i1},..., qt_{in}\})$ (5-11)

In order to semantically expand the query we apply the following algorithm:

```
1. EXPAND_QUERY_TERM(QT)
2. %input: the query term QT
3. %output: a query branch
4. QB := NEW-QUERY-BRANCH()
5. QB->w := QT->w
6. QT_1 = NEW-QUERY-TERM (QT->path, QT->op, QT->v, 1)
7. QB->addQueryTerm(QT_1)
8. QT_2 = NEW-QUERY-TERM (QT->path->ontoPath, QT->op, QT->v, 1)
9. QB->addQueryTerm(QT_2)
10. pathSet := FIND-RELATED-PATHS(QT->p->ontoPath)
11. for each path in pathSet
12.       newQT = NEW-QUERY-TERM (path, qt->op, qt->v, 1- path->distance)
13.       QB->addQueryTerm(newQT)
14. end for
15. return QB
```

Algorithm 1: The algorithm for expanding query terms with related paths.

Figure 7 shows diagrammatically the reformulation of the query term:

qt(<Hotel, Address, City>, "=", "Athens", 0.8)

with the related paths found on the previous section's example. I remind the reader that two paths were found: <Hotel2, Address2, City2> with distance and <Hotel3, City3> with distance 0.07.

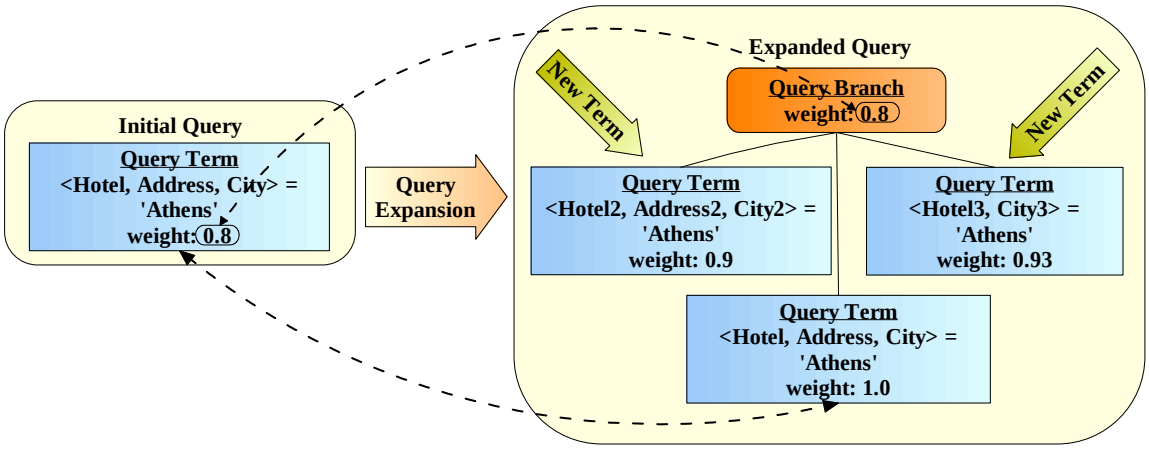

Figure 7: Query expansion example of path <Hotel, Address, City> to <Hotel2, Address2, City2> and <Hotel3, City3>.

In the process of reformulating a query term into a query branch we kept in mind how the result will be processed. In section 4.2 we described the fuzzy framework used to

process queries and we reproduce the evaluating formulae of the query terms here for convenience:

$$\left(\frac{\sum_{i=1}^{n} (a_i . w_i)^p}{\sum_{i=1}^{n} {w_i}^p} \right)^{1/p} \qquad (5\text{-}12)$$

where w_i is the user defined weight for the query branch i (before the expansion was the query term i) and a_i is a weight denoting how relevant is the data source to the query branch. Before the query expansion a_i was calculated using the weights of Table 4 of section 4.3 depending only on the data value. After the query expansion the next formulae calculates a_i.

$$a_i = \max_{j=1}^{k} \left(ad_{ij} * ap_{ij} \right) \qquad (5\text{-}13)$$

where ad_{ij} is factor of query term j of branch i depending on the data value given by Table 4 while ap_{ij} is the weight of the query term j which equals to the relevance of the path to the original one. Note that by selecting the maximum of all these values we actually use a boolean OR function between the query terms of each branch. Moreover, the use of product reflects as an AND operation between the path matching and the data matching.

6 Code Generation

The QML2.0 Query Metamodel Language, as described in the previous section, is used in order to pose a query against a specific metamodel. The role of the code generator as it was described in the previous version of the recommender [24], is to take as input the syntax tree, produced by the query execution plan constructor and generate the code to be executed in the appropriate query language. In the final implementation of the DBE the storage system used is an XML database. The models in order to be transferred, stored and queried are kept in the XMI format and are also kept in the database as XMI files. In order to be able to answer a question in this environment, the formulated query must be translated into an XQuery. Moreover the use of ontologies for the query expansion, results in an expanded query tree which must also be recognised during the code generation and a more complex final XQuery must be produced in order to include all the expanded leafs of each query term.

6.1 The Mapper

The role of the mapper, as described in the previous deliverable [42], is to allow XMI files to be treated as XML files without any limitation. This means that during the final code generation, some paths referencing elements in a different position of the XMI documents, or elements outside the document contained in an external ontology must be rewritten in order to produce the correct path expressions for the XQuery.

6.2 The XQuery Generation

As described above the generated code from the Code Generator must be an XQuery following the XQuery language syntax. To produce the XQuery query, the Code Generator parses the query tree created by the query execution plan constructor. During the parsing, parts of the final query are produced for each operation node in the syntax tree. The final query follows the syntax pattern of a simple XQuery query of this form[6]:

```
FOR list of contexts IN list of collections
WHERE list of conditions
RETURN return pattern
```

The generator must dynamically complete the query with the involved contexts and related collections. Here it must be noted that multicontext queries are also supported. The list of conditions must also be extracted and added as a list of path expressions with

[6] Other XQuery query patterns can be also used since the code generator is quite flexible to take them into account in case they exist.

their constraints or operators. The return pattern is usually an XML like list of recognizable return values. For this purpose the generator has to:

- recognise all the contexts involved in the query
- construct the full path for each expression involved in an operation
- recognise and handle the operation itself. This can be either a `simple` operation or a `Boolean` operation.

The `simple` operations are all the operations supported for constraining the value of an attribute (=, <, >=, like etc). In this case the query is constructed as above with the completion of the XQuery pattern. The `Boolean` operations refer to the Boolean AND, OR operators. As demonstrated in the previous version of the recommender extensions in QML had been considered in order to incorporate IR functionality. The p-norm Extended Boolean Model described in 17.1 Appendix A is supported and from the end-user's point of view 'hard' and 'soft' constraints had been introduced to allow the construction of fuzzy expressions for the ranking of the query results. In the current implementation all the Boolean and fuzzy expressions are incorporated in one concrete query following a syntax pattern of this form:

```
FOR list of contexts IN list of collections
LET
    $fei := path expression,
    $fci := path expression [fn:contains(. , value)],
    .... ,
    $r   := (local:fuzzyor($fei, $fci, w) + . . ) DIV ws
WHERE list of conditions
RETURN return pattern
```

The results of all the query terms are summed and normalised and the resulting value `$r` is the total rank of the query. All the terms are given a smaller weight if there is not a complete match of the value thus if the `fci` expression is the case.

The code generator can handle any query tree based on the QML and is used either to generate the code for a complex query consisting of many sub-queries or the simple queries used for the index update and query routing processes. Finally the code generator must be able to recognise an expanded query and use the query tree in order to produce a query branch for each expanded query term, grouping all the expanded leafs for the specific term. The pattern for an expanded query is of the following form:

```
FOR list of contexts IN list of collections
LET
     $fei := path expression,
         $fci := path expression [fn:contains(. , value)],
         .... ,
     $bri:=local:fuzzyor($fei,$fci, w), .. ,
```

```
        $r:=(($bri*bw)+ .. div sw)
    WHERE list of conditions
    RETURN return pattern
```

In the above pattern all the queries for each query branch are grouped using the existing fuzzy function from the previous release of the recommender. Each expanded term has its own weight *w* while each query branch has each own weight *bw*. Using the above pattern the p-norm Fuzzy Boolean model can be applied and the results for the total query are ranked as in the previous release of the recommender. Of course the advantage is that now full evaluation of an expanded query can take place maintaining the ability to support complex fuzzy queries over an heterogeneous environment with the use of ontologies.

7 Query Routing using Semantic Indexes and Learning

7.1 The Routing Mechanism

With DBE, we consider a fully decentralised service based network with different schemas on each node. The challenges for such networks have also been studied and probably the most common is the efficient query processing and routing. The methodology for searching in the DBE will be described in the rest of this section.

First of all it must be clear that there is no assumption constraining the technical characteristics of each node or the topology of the network. Any node owning the DBE infrastructure can connect to the network at any time and automatically the rest of the nodes will become aware of its existence either directly from neighbouring nodes or through a random path connecting the new node with a second node, already part of the network. As a result the topology of the network is dynamic and not following any specific pattern, being a random graph network. This is what is called the physical network in the P2P environment. That is, all the nodes with the actual network connections between them allowing the transferring of data between them. As the framework for the construction of the physical network in the DBE has been set and is not posing any constraints for the knowledge management, the searching mechanisms that will be described have to be met on a higher level. This higher level is the so-called overlay network. This means that all the semantic information must be taken into account making use of the available metamodels and of course the ontologies used for the descriptions of knowledge and the SMEs. The querying mechanism has also been enhanced to allow query expansion and the creation and use of an advanced routing mechanism.

7.2 The Overlay Network in the DBE

An overlay network is a logical abstraction of the underlying physical network. As shown in the next Figure, we can represent an overlay network as a graph $G = < V,E >$, where the vertex set V is the collection of all overlay nodes, and the edge set E is the collection of paths between overlay node pairs that are determined by the application. The construction of an overlay network is equivalent to the selection of a subset E of Ec where Ec is the set of all the paths in the physical network.

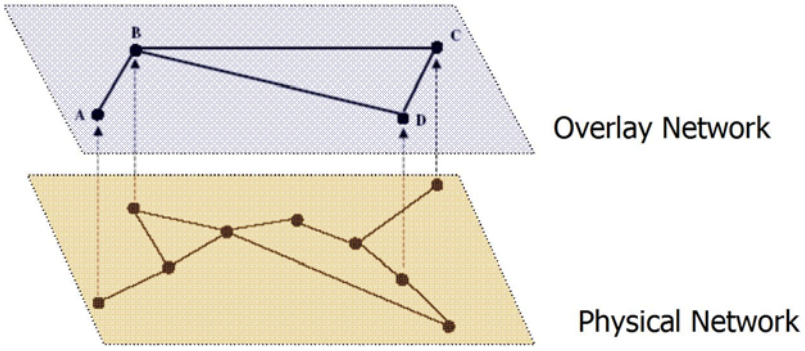

Figure 8: A P2P overlay network example

The set of nodes and paths in the overlay network are formed based on the knowledge and connection mechanisms its node is equipped with, comprising the logic for the construction of the network. This logical organization allows the nodes to have partial knowledge of the network and a definite scope for knowledge manipulation, which means searching, sending, retrieving etc. These are clearly represented in the above figure. In the overlay network each node is actually connected only with those nodes from the physical layer that add something to its own knowledge or capabilities. Of course the topology of the overlay network is also dynamic and adaptive as knowledge is added, updated or deleted during the lifetime of each node. Finally the knowledge of each node follows an evolutionary process based on the interaction between the nodes.

In the DBE the nodes hosting KB core services are treated equally supporting the fully decentralised approach followed up to here. The knowledge, the searching and the indexing mechanisms for the construction and function of the KB overlay network in the DBE are described in the rest of the paragraph. The DBE KB overlay network is actually a semantic overlay network as the protocol for its construction is based on the semantic information stored in each node.

7.3 Knowledge Representation Framework and Query Mechanism

Knowledge representation in the DBE is following the MDA approach and is based on a common set of metamodels [18]. These metamodels have been constructed to cover all the different aspects of knowledge representation in the DBE (service description, business description etc) and are available to all the nodes, members of the network. Based on these metamodels each node is capable of building its own models for describing its services, information and data. However the common paradigm should be providing some models to the nodes and giving them the capability to edit these models and adapting them to their own needs for description. These needs are directly dependent on the domain classifying each node. This can be a key factor for providing nodes of the same domain with a common "vocabulary" available for their descriptions in the model level. Following the framework set by the DBE, this can have the form of domain ontologies available to everyone in the network. As the metamodels are general enough in order to cover all the different domains existing in the DBE, the process of constructing a model description personalised for each node should gain from the

existence of domain ontologies in order to semantically communicate knowledge with other nodes based on the common concepts contained in the domain ontologies.

It must be clear that each model consists of three parts: the common DBE metamodels, ontology concepts derived from the domain ontologies and finally the values for the instance level description of the data.

In the previous section of this report, the use of ontologies for the description of the nodes has been described and the mechanisms for the query formulation and processing have been explained. The query formulation takes place locally and makes use of the local model. However, the domain ontologies available are globally known to the rest of the nodes and the formulator may freely choose which to use. After the formulation of the query the query expansion phase is needed before flooding the query to the network. The important thing is that the final query, as described above, is expanded in order to include all the relevant ontology concepts based on the path and ontology matching mechanisms. The indexing and query routing mechanisms described in the next paragraph try to classify the nodes and the queries making use of these ontology concepts.

7.4 Semantic Indexing

7.4.1 Indexing Overview

Up to here we have described the framework for the complete query formulation procedure and analyzed the infrastructure that will allow the efficient query routing in the DBE environment. Based on the above, the DBE nodes have access to the physical layer of the network where they have knowledge of their physical neighbours. This of course, allows only random flooding of the network and can be used only for bootstrapping reasons. The framework described up to this point is used in order to build a semantic overlay layer where query routing will be more efficient. There are two factors that must be taken into account. The first is the forwarding mechanism available within the DBE P2P infrastructure and the second is the characteristics of knowledge representation in the DBE referring to the use of a common set of metamodels and ontologies as described in the previous chapter.

As far as the forwarding mechanism is concerned, the important characteristic is that every node has connection only to its physical neighbours. If a node is considered the originator of a query, this must be forwarded to all or a subset of the originator's neighbours. After receiving the query its node is responsible for locally executing it and forwarding it to its own neighbourhood. Of course the reverse procedure of communicating back the results must also be considered. The important thing about this procedure is that every node is sending the results directly back to the originator of the query rather than using the inverse path followed during the initial forwarding of the query. This has the advantage of avoiding the flooding of the network with the results of each node but must be taken into consideration when trying to build an indexing mechanism for the system.

Building a semantic overlay network, the indexing mechanism must take advantage of the knowledge representation infrastructure. As each node has a complete and formal

semantic description of its contents the aim here is to construct an efficient index of the nodes in the network and not an index of the available documents. It's dealt as the problem of finding the "right" person to ask, avoiding flooding or forwarding the queries to nodes that are not likely to have relevant content. To achieve this we try to classify the nodes and the arriving queries based on a common procedure having to do with the use of specific ontologies and ontology concepts for their description. The use of heterogeneous ontologies is covered with the ontology mapping mechanism described in the previous section of the current report. This kind of processing completed with the expansion of the query takes place in the originator node. This node is then responsible for routing the query making use of its local index and collecting information for updating the index.

The index maintained by each node can be divided into two semantic layers: the one created after collecting results about a query asked by the node itself and one created about forwarded queries received through the network. The first layer contains nodes which had content related with the query and we can call them content providers. The second layer contains nodes which asked a specific query and are likely to have collected indexing information about that query and can be called recommenders.

7.4.2 Index Creation

In the general case, a query contains a number of sub-queries. For the forwarding of the query the expanded query tree is used. For the creation and update of the index the query tree is split into as many sub-trees as the count of the sub-queries. Of course, this has the cost of the number of messages that will have to be forwarded through the network, not forgetting the feedback messages with the results. As the network is not flooded and the results are send directly back to the originator of the query this may not influence the response time for the initial query. However, it is probably more efficient to forward the index update queries during another period, when for example the network traffic is low or some time after the forwarding of the initial query.

At this point we must make clear that its node can trigger the index update procedure only for queries the node itself requested. This leads to the updating of the indexing information on content providers, the top layer. For the next layer a node can collect information about recommender nodes only after the reception of a forwarded query. These two cases are analyzed in the rest of this section.

After the expansion of the query and before forwarding the final query, it is divided into as many sub-query trees as the count of the original query's sub-queries. Next, code is generated to produce the final XQuery both for the total query and the individual sub-queries. The difference is that the total query will be forwarded immediately, or after using the index in order to route the query, while the sub-queries are created for the index update procedure and can be forwarded sometime later trying to avoid increasing the network load. When this time comes the node collects the results for each sub-query and updates its index for the concepts contained in them. The structure of the index is described in the next paragraph. The nodes sending back results are added in the top layer with the content providers along with the concepts they contained and a relative rank for each concept. The above procedure results in creation of the top level index.

For the creation of the second level index every node after the reception of a query just keeps track of the originator of the query and the concepts contained in the query. The originator of the query is the only one that collects indexing information about content providers and so he can be used as a node in the second level index where just recommender nodes form a semantic layer themselves.

For the use of the index after the concepts contained in it are recognised in the originator node, they are matched against the top level index. A total rank for each indexed node is returned and the query is routed to the nodes with a rank above some threshold. If not enough nodes exist in the index, or if the number of the results is not considered sufficient, or even from the beginning, the node can also create a list of nodes from the second level index where the query can be forwarded as they are nodes with the most likelihood to contain some indexing information about the query and also route it to some content providers.

7.4.3 Index Structure

The semantic indexing applied in the DBE takes place in the semantic overlay network, as described previously. To create the overlay network, each node must have knowledge of a group of other nodes in the network based on some criteria. This knowledge derives from the semantic description of the nodes. Each node, owns a unique node ID and has all its data and metadata described, using the common metamodels and domain ontologies of the DBE. Each document uses a specific ontology and is connected with the ontology concepts of the specific ontology. The final query, resulting after the query expansion, contains all the related concepts from the rest of the known ontologies.

Regardless of whether the query consists of more than one sub-query the indexing mechanism keeps track of information about each individual ontology concept avoiding to group concepts belonging in the same query. However this is considered an open issue and can further be investigated in the future. Each sub-query contains one or more ontology concepts which will be used in order to classify the answering nodes. At this stage we decided to use the sub-queries independently and not as a part of the original query aiming to get more precise information for a each smaller set of ontology concepts, contained in each sub-query and not the whole set of them. For the indexing update procedure, the individual sub-queries are forwarded to the network and the result is of the following type

Path | Ontology Concept | Rank | Node ID

In the above structure each node managing to match an ontology concept with its contents returns a result to the originator which is then responsible for updating the index with the concept and an appropriate rank. It must also be noted that the concepts used in the index associated with the answering nodes, are the ones from the initial query and not the expanded siblings of it. Having described the index structure and the querying mechanism it is obvious that the creation of the index is based on a continuous learning procedure. The learning procedure is initiated upon the collecting of the results as a part of the querying mechanism. The result of this learning procedure is the

acquisition of knowledge about the contents of the different nodes. After a short or a longer learning period the contents of a node's index look like the contents of the following table:

Path	Ont. Concept	Rank	Node ID
ServiceProfile/ServiceName/..	"Hotel"	0.75	KBfd#2345..
ServiceFunctionality/Name/..	"Cancellation"	1.0	KBfd#2345..
ServiceAttribute/..	"Address""	0.4	KBfd#2349..
. . . . .			

Table 7: The contents of an index residing on a DBE node

The use of the index is described in the next paragraph. Some more interesting issues originating from the P2P nature of the network are also the joining of new nodes and the so called "cold start" problem for the index.

7.5 Using the Index and "Cold Start" Routing

Up to here the framework for the construction of the semantic overlay network in the DBE has been set. In addition the searching mechanisms that exploit this network have been described. The P2P nature of the network allows the joining and leaving of nodes at anytime. On the physical network this issue is handled by the DBE P2P infrastructure. The result is that the "physical" existence of any node is automatically "felt" by the rest of network upon its join or disconnection. The critical point is the discovery of a new node in the semantic overlay network and the training of its index.

When a new node joins the network it is not a part of the index of any other node so the probability of participating in any forwarding list of nodes is rather low. A first approach for actually advertising the node's existence is the formulation of a set of queries that will be controlled (using a horizon or a maximum number of nodes) flooded to the network. The random discovery of other nodes to make the above flooding is easy due to the existence of neighbouring nodes in the physical network. The important thing is that the set of queries will be created based on the locally stored models of the new node. Any nodes that will receive the flooded queries will make an insertion to their indexes about the new node and will also send back some results that can be used by the new node for starting the training of its own index. The learning process for an empty index can also be quickened with a request of a new node for the indexing information of its neighbouring nodes. All the above procedure is a first approach to face the challenges emerging from the semantic indexing and searching mechanisms needed for the efficient searching of knowledge in the DBE P2P environment. In the last paragraph of this section the introduction of a fuzzy semantic overlay network will set the basis for a more concrete and flexible approach to the above challenges.

7.6 Towards a Fuzzy Semantic Overlay Network

The framework for the construction of the DBE semantic overlay network has been described in the current section. In this description the network has been treated semantically as a single set of all the nodes belonging to the overlay network along with their paths. In this paragraph a further exploitation of the semantic information and the semantic indexes will try to improve the behaviour and the efficiency of the overlay network. The core part of the idea is to allow the classification of all the nodes based on their semantic information in order to split the set of nodes, in the overlay network, into Semantic Overlay Clusters (SOCs). The searching mechanisms can take advantage of this classification by using the same mechanism for the classification of the queries. The whole mechanism must also be fully decentralised and dynamic leading to a fuzzy semantic overlay network.

As already stated a classification operator is needed for the creation of semantically separable clusters, the SOCs. This operator will be applied locally by each node and will allow the classification of the node itself and also the classification of all the received queries into a number of SOCs. In the current approach the basis for the classification is the set of ontology concepts contained in the model description of the node or the query. More specifically the classifier will be capable of identifying a SOC for each concept in the description of the query. In a more advanced approach, more general concepts or concepts semantically closely related can be used for the identification of nodes or queries in the same SOC. The result is that each node may participate in one or more clusters (SOCs) and each query may classify in one or more clusters. So the first step is to extend the indexing mechanism and use the semantic information acquired for the neighbouring nodes in order to classify them into SOCs. Of course the result of this classification needs to be stored for future use and is also following the learning mechanism of the index. This procedure will gradually build a table of SOCs in each node and each SOC will be associated with a list of nodes belonging to it. Every time a new query arrives at the node, the same classifier is applied to it and a list of SOCs is also associated with it. These SOCs are used for forwarding the query to those nodes that classified in the same SOCs according to the table of SOCs kept by the node that received the query. Each node maintains locally its own table of SOCs.

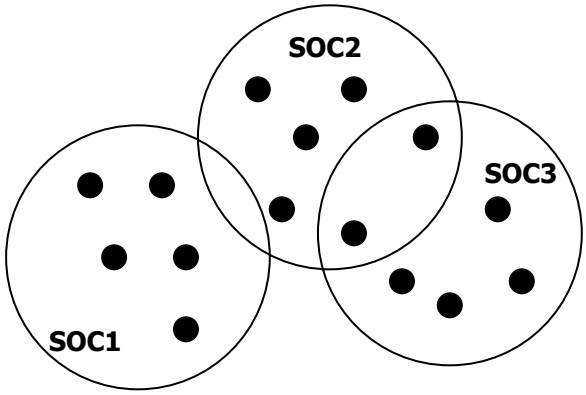

Figure 9: Formulation of semantic clusters of the nodes in the overlay network.

In Figure 9, an example of the creation of three semantic overlay clusters is presented. It must be clear that each node, applies locally the classification operator on the

contents of its own index. The result is a list of nodes and their corresponding SOCs, that is the clusters to which they belong. As this is a local procedure, each node has its own view of the network and its own table of SOCs. Moreover the table is built dynamically and gradually after the processing of each new query and the analysis of the results. Finally as the classification mechanism is based on the ontology concepts used for the descriptions of the queries and the nodes, it must be implied that the classification operator will have the ability to classify a node or a query with some probability to some SOCs, in case a complete match with some ontology concepts or some existing SOCs is not possible.

In conclusion, the nodes belonging to the semantic overlay network can be classified into clusters allowing the enhancement of the semantic indexing mechanisms. The organization of the overlay network into semantic clusters can result in a more efficient way for handling the query routing problem. The preservation of the decentralised character of the network in conjunction with the classification operator will lead to a fuzzy semantic overlay network consisting of locally recognizable and manageable lists of semantic clusters. The classification of the concepts into semantic overlay clusters provides a common way to efficiently face the challenge of query processing and routing in the semantic and decentralised P2P environment of the DBE.

8 Recommender Service

As described in the introduction of the current report, the recommender component in the DBE is used to personalise the DBE knowledge for each user (end user or SME). It must be capable of fulfilling both the classical information retrieval approach of providing access to knowledge, based on an explicit user request with the use of some criteria and in addition the role of an autonomous service, filtering the DBE knowledge and providing valuable recommendations to the user. Responsible for the delivery of the above functionality in the current implementation of the DBE is the Recommender Service. The user preferences and requirements necessary for the filtering and recommending procedure are provided in the user profile as described in the FZI's User Profile deliverable D7.2 [25]. This information is stored in the User Profile Model (UPM), which is shortly described, in the next paragraph. The main part of the profile is the one keeping the user preferences in the form of constraints related with a specific user. All the constraints are formulated in the form of OCL Expressions. The management of the profile and the collection of recommendations based on the profile is the subject of the rest of this section.

The Recommender Service acts as an autonomous process that manages SME preferences (either business preferences or service preferences) and matches these preferences with available business descriptions and service descriptions. The Recommender Service exploits the matching mechanisms of the Recommender module of the KB infrastructure. The Profile Manager searches in the database for stored SME profiles that are active and creates a new session for each profile. Each session takes over the process of searching in the P2P network of Semantic Registries for published services that match the preferences of the specific profile. The results are aggregated and processed per session and are available each time a recommendation based on a profile is requested or the SR service can automatically trigger a notification to the interested user whenever there is a change in the recommendations due to new service publishing or due to updates of existing services.

In Figure 10 the Recommender Service is presented as part of the DBE Servent. The framework remains the same while the modules have been extended in order to meet the new query analysis needs and support the semantic query routing mechanisms. The Recommender Service makes full use of the improved Knowledge Access Module benefiting from the Query Analyzer and Code Generator for the support of ontology based queries and the Semantic Index Module for the efficient forwarding and processing of queries.

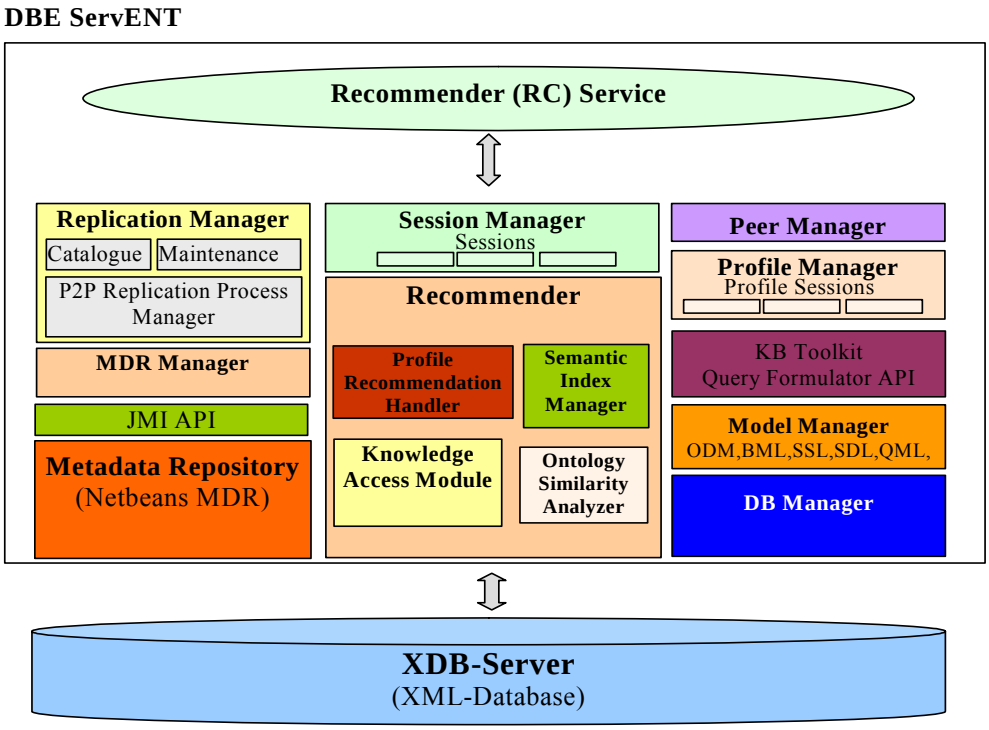

Figure 10: The Recommender Service (RC-Service)

8.1 Introduction to User Profile Metamodel

In this section a small introduction to the User Profile Metamodel (UPM) and its connection to the Query Metamodel Language will be given. UPM is able to express user preferences in a sophisticated way that the recommender can use to provide advanced recommendation facilities to end-users. For a detailed description please refer to the FZI's User Profile deliverable D7.2 [25].

User Profile Metamodel (UPM) expresses user preferences in terms of constraints on metamodels and models. UPM user preferences package is shown in figure 11. What is important to state is that user preferences can be grouped with others in conjunctions or disjunctions (AndPreference, OrPreference respectively). Each user preference (or group) has an importance weight, validity period and a set of other arguments used for performance reasons. Each simple preference (LabeledConstraint) has an actual constraint expressed in terms of QML (for example: *Hotel.Country= 'Finland'*).

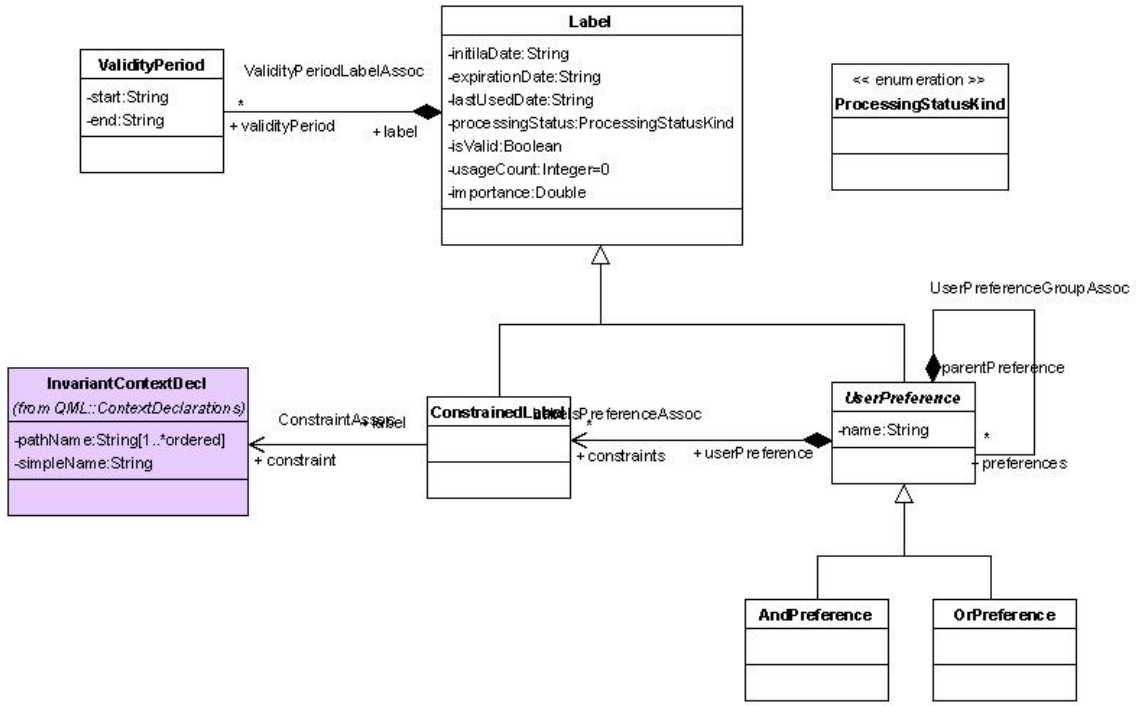

Figure 11: The User Preferences package of the User Profile Metamodel.

The core package of the UPM (not shown here) contains user information.

8.2 User Profile Management

The recommender service is responsible for providing a complete API for the management of the user profile. The Recommender Service API is described in detail in Appendix A of the current report. Any information regarding the User Profile description or preferences is following the UPM and is stored in documents locally for each user – node in the DBE network. It is essential to be able to manage one or more user profiles for each user. Once a profile exists it is stored and processed only locally. The propagation of a profile for the purposes of the recommender is done in the form of queries formulated by the recommender upon either a user or a service request. The functionality available for the management of the User Profile by the Recommender Service is the following:

- Storing of a UPM document. A document can contain more than one profiles associated with a specific user identity. Of course, following the user profile model, its profile can contain multiple constraints, that is, multiple preferences.
- Deletion of a UPM document for the case a user profile has to be repealed.
- Listing of all the available UPM documents for a specific user.

All the above functions can be accomplished for the local needs of any node in the DBE network. A newly created profile can be stored in a node at any time leaving the utilization of it, thereafter, to the Recommender Service. Here, it must be made clear that the Recommender Service is independent and makes use only of the common XML database available for all the DBE services in order to store the user profiles. As already stated, the Recommender Service is responsible for automatically finding and using all

the available profiles for providing recommendations to the referred user. The process for collecting recommendations is described in the rest of the section.

8.3 Collecting Recommendations in the DBE

The main role of the recommender service is that of an autonomous agent responsible for collecting and providing recommendations to the users. The whole procedure is based on the existence of the users' preferences as described in the previous paragraphs. Each node interested in keeping some preferences that will allow the collection of valuable recommendations, can have its own recommender service deployed. Once a recommender service is deployed it is assumed that it will start running and continue working in the background as long as the node hosting it is 'alive'. The procedure of collecting recommendations is set to be conducted periodically. Of course, this period is subject to change depending on a rational assumption on both the need for new recommendations and also the possibility for their existence. In the rest of the paragraph, the whole procedure for collecting recommendations in each of these runs will be described.

The procedure can be broken down into the following steps:

- The discovery of all the locally stored profiles
- The extraction of the constraints contained in each profile
- The formulation of QML queries based on the constraints
- The propagation of the queries to the network
- The collection of the resulting recommendations

It must be made clear that the propagation of the queries in the DBE network is based on the DBE searching mechanisms, which were mainly described in paragraph 7. The recommender service makes use of the Semantic Registry service in order to discover Service Manifests, matching the required preferences. In the current implementation the list of the matching Service Manifests is actually the list of the recommendations returned to the user.

At the beginning of each recommendation process, the Recommender Service asks for a list of all the locally stored profiles. This list is necessary as it is the only way to know if there have been any changes since the last recommendation run. As all the users are able to add a new profile or remove an already existing one at anytime the recommender must always be aware of the current profiles listing. For the rest of the process it is essential to remember that each profile is associated with a user identifier, which is supposed to be unique. The important thing is that the recommender has to create the group of queries referring to the preferences of a specific user identifier and then collect the results, recommendations, for its group of queries.

The second step in the process is the extraction of all the preferences, constraints, found in each profile and the formulation of QML queries containing these constraints. As it has already been described, each profile can contain many constraints. These constraints are in the form of OCL expressions. In this step of the process all the constraints have to be aggregated in a group of disjunctive queries. Actually the produced list of queries for each profile will be treated as a list of fuzzy queries. This will allow all the queries to apply in the collection of the recommendations and the returned recommendations to be ranked regarding the profile as a whole. For the recommender

to be able to make use of the rest of the DBE infrastructure for the query propagation and result collection, the queries are formulated in the QML which of course is the query language used by the rest of the DBE components.

The QML queries created in the previous step are propagated using the Semantic Registry's Service functionality. All the queries are first processed locally and then propagated to the network allowing a P2P collection of recommendations for each profile. The P2P searching mechanisms of the DBE for the query routing are analyzed in the following sections. The responsibility of the recommender is to invoke the querying functionality, supply the queries for the profiles and then collect the returned results.

In the second step of the recommendation process, the queries resulting from each profile were aggregated and grouped for each user identifier. The recommender is responsible for grouping the resulting recommendations in the same manner. Finally a ranked collection of recommendations is kept for each user. This list of recommendations is available until the next recommendation process begins. Also, the recommender service API allows the user to obtain his list of recommendations at anytime. With the whole process the recommender has to take into account two very important aspects of the system:

- a user can create and store a new profile at anytime. An existing profile may also be deleted
- a new node or a new service matching the user's preferences may also come up at anytime.

The discovery of any change in the profile(s) stored for a user has been covered in the first step of the recommendation process. The organizing of the resulting recommendations and the fact that they are ranked allows the discovery of any new results not only to be clear but also to be identified or even advertised based on their rank. The rest of the work, that is, the discovery of a new node or a new service is left to the P2P functionality of the DBE mechanisms.

9 Conclusions

In the final release of the recommender, the most important issue was the extension of its mechanisms and algorithms over the fully decentralised P2P network of the DBE. This includes the enhancement of the DBE framework with those mechanisms and characteristics that would allow the exploitation of all the advantages of the P2P network while facing the common problems and challenges of such networks.

First of all, the querying mechanisms where improved in order to become more flexible and capable of fully supporting the query formulation needs on the one hand and the query answering mechanisms on the other hand. To these needs, the exploitation of rich domain specific ontologies was added. The query analysis had to be supported by an ontology mapping mechanism and now, query expansion takes place to facilitate the information retrieval needs.

The MDA approach for the description of knowledge in the DBE in conjunction with the use of the XMI format for the exchange of data between the modules and between the users had already imposed the utilization of an XML database. The code generation module with the use of XQuery was improved in order to utilise the new expanded queries continuing to support fuzzy queries following the p-norm Extended Boolean Model. Moreover the code generator was extended in order to generate indexing queries for the creation, managing and use of the semantic index.

Finally, the incorporation of the above mentioned querying mechanisms in the DBE services for each node and the deployment of the system in the P2P environment enhanced the P2P function of the Recommender. However, the need for an efficient query processing and routing mechanism was thereafter, imperative. The DBE framework, in accordance with the DBE P2P infrastructure, set the basis for a semantically strong manipulation of the semantic overlay DBE network. The common DBE metamodels and the capability to use rich ontologies for the description and management of knowledge in the DBE, plays a decisive role in the sharing and discovery of knowledge. Based on all this, a semantic, decentralised routing and indexing mechanism was developed. The evaluation, testing and fine-tuning of the above mechanism will result in a concrete, consistent and efficient framework for the P2P management and utilization of knowledge in the DBE.

10 Glossary

Term	Description
API	Application Programming Interface: Is a technology that facilitates exchanging messages or data between two or more different software applications
BML	Business Modelling Language
DBMS	Database Management System: A software system that allows efficient manipulation (storage, organisation, indexing, and querying) of large amounts of data.
ExE	Execution Environment: It is where services live, where they are registered, deployed, searched, retrieved and consumed. This word is sometimes referred to as the "runtime of the DBE".
JCP	Java Community Process: The "home" of the international developer community whose charter it is to develop and evolve Java technology specifications, reference implementations, and technology compatibility kits
JDBC	Java Data Base Connectivity: A technology that provides cross-DBMS connectivity to a wide range of relational databases, as well as access to other tabular data sources such as spreadsheets and flat files
JMI	Java Metadata Interface: A Java Community Process (see JCP description) specification of a standard Java API (see description of API) for metadata access and management based on the MOF specification.
JavaCC	Java Compiler Compiler is a parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar.
KAM	See Knowledge Access Module
KB	Knowledge Base: the part of the DBE system where DBE knowledge is stored and managed. Such knowledge refers to ontologies, business and service descriptions, etc.
KB Service	Knowledge Base Service: A Service on top of the DBE Knowledge Base that provides functionality for storing and retrieving models.
Knowledge Access Module	A component used to provide uniform access to the DBE Knowledge.

MDA	Model Driven Architecture: An approach (proposed by OMG) to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology.
MDR	Meta-Data Repository: MDR implements the OMG's MOF standard based metadata repository based on the JMI specification (see JMI description)
MOF	Meta Object Facility: A generalised facility for specifying abstract information about very concrete object systems.
MOF Repository	A repository for storing, managing and retrieving meta-data (models) and meta-meta-data (metamodels) that have been described with MOF.
OCL	Object Constraint Language: OMG's standard for expressing constraints and well-formedness rules on object models. The latest release is also considered suitable for querying object models.
ODM	Ontology Definition Metamodel: A MOF model (metamodel) developed in the DBE for ontology representation according to the corresponding OMG RFP
OMG	Object Management Group: International standardisation body
P2P	Peer-To-Peer
QML	Query Metamodel Language: It is a Knowledge Access Language developed in DBE in order to provide uniform access to various kinds of DBE knowledge.
Query Analyzer	A component of the Knowledge Access Module that is used to analyse queries against the metamodel (used for knowledge representation) specific semantics.
Query Code Executor	A component used (by the Knowledge Access Module) to execute the generated query code
Query Code Generator	A component of the Knowledge Access Module that takes as input the query syntax tree and generates the code to be executed in the appropriate query language
Query Execution Plan Constructor	A component of the Knowledge Access Module that evaluates the QML expressions already analysed into a syntax tree representation.
Query Formulator Tool	A front-end tool, developed in DBE, allowing the user to formulate queries against the DBE knowledge using a tree-view representation of the Knowledge Structure.
Recommender	A DBE (autonomous) Core Service that will provide users

	(SMEs) with personalised knowledge by exploiting their profiles
SDL	Service Description Language: A MOF model (metamodel) that provides technical description of the programmatic interface of a service
Semantic Registry	The component of the DBE Knowledge Base that hosts the published services (in the form of Service Manifest Documents).
SFE	Service Factory Environment: This is devoted to service definition and development. Users of the DBE will utilise this environment to describe themselves, their services and to generate software artefacts for successive implementation, integration and use
SQL	Structured Query Language: A language for querying relational data
SR	Semantic Registry: It is the component of the Knowledge Base that hosts the service descriptions published in the DBE environment and available for discovery and consumption.
SSL	Semantic Service Language
XMI	XML Metadata Interchange: An SMIF (see SMIF description) standard specification based on XML.
XML Database	A database that is specialised for storing XML data and stores all components of the XML model intact. It has an XML document as its fundamental unit of (logical) storage and it is not required to have any particular underlying physical storage model (e.g. it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files)
XQuery	A Query language from the W3C that is designed to query collections of XML data.

11 References

1. Object Management Group (OMG), "Meta Object Facility(MOF) Specification," (2002), version 1.4, http://www.omg.org/
2. Boldsoft, International Business Machines Corporation, IONA and Adaptive Ltd. "OCL 2.0 OMG Final Adopted Specification", (OMG Document ptc/03-10-14), October 2003
3. Calado, Silva, Vieira, Laendar, Ribeiro-Neto "Searching web databases by structuring keyword-based queries" Proceedings of the eleventh international conference on Information and knowledge management, 2002.
4. N. Gioldasis, N. Pappas, F.G. Kazasis, G. Anestis, S. Christodoulakis:"A P2P and SOA Infrastructure for Distributed Ontology-Based Knowledge Management", Sixth Thematic Workshop of the EU Network of Excellence DELOS on Digital Library Architectures
5. Lee J. H., "Properties of Extended Boolean Models in Information Retrieval", In Proceedings of the 17th ACM SIGIR International Conference on Research and Development in Information Retrieval, 1994, 182-190.
6. F. Kazasis, N. Gioldasis, N. Pappas, G. Anestis, S. Christodoulakis: "MOF-based Knowledge Management for a Digital Business Ecosystem", 2nd IST Workshop on Metadata Management in Grid and P2P Systems (MMGPS): Models, Services and Architectures, December 2004.
7. Raghavan S., Garcia-Molina H.: "Integrating Diverse Information Management Systems: A Brief Survey", IEEE Data Engineering Bulletin,Vol.24,No.4,pp.44-52,December 2001.
8. OMG XML Metadata Interchange (XMI) Specification v1.2 http://www.omg.org/cgi-bin/apps/doc?formal/02-01-01.pdf, 2002
9. Rantzau R., Shapiro L.D., Mitschang B., Wang Q.: "Algorithms and applications for universal quantification in relational databases", Information Systems, Vol. 28, No. 1-2, pp. 3-32, 2003
10. Compuware Corporation and SUN Microsystems (2003) "XMOF Queries, Views and Transformations on Models using MOF, OCL and Patterns" OMG Doc. ad/03-08-07
11. MDA Guide Version 1.0.1: http://www.omg.org/docs/omg/03-06-01.pdf, 2003
12. "SQL", ISO/IEC 9075:1999.
13. "XQuery 1.0: An XML Query Language", http://www.w3.org/TR/xquery, November 2002.
14. S. Melnik, H. G. Molina, E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching" Proc. 18th ICDE Conf., 2002.

15. T. Attwood et al. "The Object database standard /ODMG-93", Morgan-Kaufmann, San Mateo, 1994.
16. David Heardean, Kerry Reymond, Jim Steel. "MQL: a Powerful Extension to OCL for MOF Queries", EDOC '03, p.264
17. Ilia Petrov, Stefan Jablonski. "Towards a Language for Querying OMG MOF-based Repository Systems". Wisme Workshop. UML 2004. 11-15 October 2004, Lisbon, Portugal
18. TUC, DBE Deliverable, D14.1 – DBE Knowledge Representation Models: https://dbe.digital-ecosystem.net/servlets/ProjectDocumentList ?folderID=16&expandFolder=16&folder ID=0, May 2005
19. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, October 2002, OMG Document: ad/2002-04-10.
20. Sten Loecher and Stefan Ocke "A Metamodel-Based OCL-Compiler for UML and MOF" in 6th International Conference on the UML and its Applications, UML 2003, volume 154 of ENTCS. Elsevier, October 2003.
21. T. Gardner, C. Griffin, J. Koehler, R. Hauser: "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard". MetaModelling for MDA Workshop, York, England, 2003
22. Metadata Repository (MDR) Project. http://mdr.netbeans.org/
23. Berkeley DB XML. http://www.sleepycat.com/products/bdbxml.html
24. TUC, DBE Deliverable, D17.1 – Recommender: https://dbe.digital-ecosystem.net/ servlets/ProjectDocumentList?folderID =16& expandFolder=16&folderID=0, March 2005
25. FZI, DBE Deliverable, D7.2 - Initial Description of Profiling mechanism design and rationale with respect to one or two use cases: https://dbe.digital-ecosystem.net/servlets/ProjectDocumentList?folderID=361&expandFolder=361&folderID=328, October 2005
26. MDA Guide Version 1.0.1: http://www.omg.org/docs/omg/03-06-01.pdf, 2003
27. Belkin N. J., Croft W. B. "Information Filtering and Information Retrieval: two sides of the same coin?" Communications of the ACM, 35(12), 29--38, 1992
28. Erhard Rahm, Philip A. Bernstein, "A survey of approaches to automatic schema matching". The VLDB Journal 10: 334–350 (2001).
29. S. Castano, A. Ferrara, and S. Montanelli, "H-match: an Algorithm for Dynamically Matching Ontologies in Peer-based Systems". In Proc. of the 1st SWDB VLDB Workshop, Berlin, Germany, 2003.
30. Arturo Crespo, Hector Garcia-Molina, "Routing Indices for Peer-to-Peer Systems".
31. Arturo Crespo and Hector Garcia-Molina, "Semantic Overlay Networks for P2P Systems".
32. Napster. http://www.napster.com
33. Gnutella. http://gnutella.wego.com

34. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. "A scalable content-addressable network." In ACM SIGCOMM, August 2001.
35. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In Proc. ACM SIGCOMM, 2001.
36. B. Zhao, J. Kubiatowicz, A. Joseph. "Tapestry: An infrastructure for fault-tolerant wider-area location and routing". Technical Report UCB/CSD-0101141, Computer Science Division, U.C. Berkeley, April 2001.
37. B. Yang and H. Garcia-Molina. "Comparing hybrid peer-to-peer systems". In VLDB, 2001.
38. A. Maedche, B. Motik, L. Stoljanovic, R. Studer, R. Volz. "An infrastructure for searching, reusing and evolving distributed ontologies". In Proc. WWW2003, Budapest.
39. S. Castano, A. Ferrara, and S. Montanelli, E. Pagani, G.P. Rossi. "Ontology-Addressable Contents in P2P networks", in Proc. of WWW'03 1st SemPGRIDWorkshop, (2003).
40. http://freenetproject.org/
41. TUC, DBE Deliverable, D14.3 1st P2P Distributed Implementation of the DBE KB and SR, December 2005
42. TUC, DBE Deliverable, D14.4 2nd Release of Recommender, December 2005
43. A. Loser, S. Staab, C. Tempich, "Semantic Methods for P2P Query Routing", MATES 2005.
44. M. Li, W. Lee, A. Sivasubramaniam, D.L. Lee, "A Small World Overlay Network for Semantic Based Search in P2P Systems", Proceedings of Workshop on Semantics in Peerto-Peer and Grid Computing (SemPGrid) 2004.
45. C. Schmitz, A. Loser, "How to model Semantic Peer-to-Peer Overlays?", Proceedings P2PIR Workshop, Informatik 2006.
46. Hong-Hai Do, Sergey Melnik, and Erhard Rahm. "Comparison of schema matching evaluations". In Proc. GI-Workshop "Web and Databases", Erfurt (DE), 2002.
47. Petko Valtchev. "Construction automatique de taxonomies pour l aide a la representation de connaissances par objects" These d' Informatique, Universite Grenoble 1, 1999.
48. I. V. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". Cybernetics and Control Theory, 1966.
49. TUC, DBE Deliverable, D14.5 – "Final P2P implementation of the DBE Knowledge Base and SR", November 2006
50. http://www.w3.org/TR/rdf-sparql-query/
51. http://www.eclipse.org/emft/

12 Appendix A – Mathematical Proof

In this appendix two proofs are shown.

Proof 1. We will show that the function:

$$f = \frac{\sum_{i=1,j=1}^{n,m} \delta(e_i, e_i) + n - k}{n} \qquad (12\text{-}1),$$

Belongs in the interval [0, 1]. We have just to show that $f \leq 1$ and $0 \leq f$.

First the sum: $\sum_{i=1,j=1}^{n,m} \delta(e_i, e_i)$ is greater than zero (0) and smaller than *min(m, n) = m* because δ belongs in the interval [0,1] and thus the maximum value is the number of term participating in the sum, i.e. *m*. Thus (12-1) can be written as:

$$f = \frac{m + n - k}{n}, \qquad (12\text{-}2)$$

Moreover, remember the definition of *k* in section 5.3 from which we have that the maximum value of *k* is *n* and the minimum is *m*, i.e.:

$$m \leq k \leq n, \qquad (12\text{-}3)$$

From (12-3) we have:

$$m \leq k \Leftrightarrow m - k \leq 0 \Leftrightarrow m - k + n \leq n \Leftrightarrow \frac{m - k + n}{n} \leq 1 \Leftrightarrow f \leq 1$$

The first part is proven.

$$k \leq n \Leftrightarrow n - k \geq 0, \qquad (12\text{-}4)$$

but also $m \geq 0$ and by adding parts we get:

$$m + n - k \geq 0 \Leftrightarrow \frac{m + n - k}{n} \geq 0 \Leftrightarrow f \geq 0$$

The second part is also proven thus *f* belongs to [0,1].

Proof 2. We will show the upper bound of the number of transitions required for rule 5 of section 5.2.2 is given by the formulae:

$$n = \left\lceil \log(T) \Big/ \log(\max(re, rb, rp, rt)) \right\rceil \text{ if } \max(re, rb, rp, rt) \neq 1. \qquad (12\text{-}5)$$

It is clear that the final similarity of any number of transitions cannot be smaller than the threshold *T.* If all the factors use to produce similarities is less than 1 (i.e. max(*re, rb, rp, rt*) ≠ 1) in each transition a smaller similarity is produced because $r = r_1 * r_2$.

Now, let *n* be the number of transitions such that the minimum threshold is reached. Then the following equation holds:

$r_{final}=r_1*r_2*...r_n=T$ (12-6)

The maximum product of the *n* factors will be produced if each one is the maximum allowed:

r_{max} = max(*re, rb, rp, rt*) (12-7)

Thus, equation (12-6) can be rewritten as:

$$r_{final} = r_{\max}^{n} = T \quad (12\text{-}8)$$

Finally if we use logarithms we get:

$$n = \left\lceil \log(T) \Big/ \log(r_{\max}) \right\rceil \quad (12\text{-}9)$$

Proved as said.

13 Appendix B – The Keyword Expressions Parser Grammar

In this appendix we present the grammar we used to produce a parser for keyword expressions. The grammar is in the javaCC syntax, application used to create the parser.

The grammar of the keyword expressions parser is as follows:

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  < ID: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )* >
|
  < NUM: ( ["0"-"9"] )+ >
|
  < FLOAT: ["0"-"9"] "." (["0"-"9"])+ >
|
  < OPERATOR: ["=", "<", ">"] >
|
  < SEPARATOR: ["/", "\\"] (["/", "\\"])?  >
|
  < STR: ["\"", "'", "(", ")"] >
}

java.util.Vector Expression() :
{
      java.util.Vector termimage = new java.util.Vector();
      QueryTerm queryTerm;
}
{
  ( queryTerm=Term()
```

```
      {
        termimage.addElement(queryTerm);
      }
  )*
      {
          return termimage;
      }
}

QueryTerm Term() :
{
      java.util.Vector path = null;
      Token op = null;
      Object value = null;
      Token w = null;
}
{
      LOOKAHEAD(2)

      path=path() op=<OPERATOR> value=Factor() ( "^" w=<FLOAT>)?
      {
        String oper = (op == null) ? null : op.image;
        String imp  = (w == null)  ? null : w.image;
        QueryTerm result = new QueryTerm(path, oper, value, imp);
        return result;
      }
|
       value=Factor() ( "^" w=<FLOAT>)?

      {
        String imp  = (w == null)  ? null : w.image;
        QueryTerm result = new QueryTerm(null, null, value, imp);
        return result;
      }
}

java.util.Vector path() :
```

```
      {
      java.util.Vector factorimage = new java.util.Vector();
      Token t;
      }
{
  t=<ID>
      {
        factorimage.addElement(t.image);
      }
  ( <SEPARATOR> t=<ID>
      {
        factorimage.addElement(t.image);
      }
  )*
      {
        return factorimage;
      }
}

Object Factor() :
{
      java.util.Vector factorimage = new java.util.Vector();
      String s;
}
{
   s=Simple()
      {
        return s;
      }
|
<STR> s=Simple()
      {
        factorimage.add(s);
      }
( s=Simple()
      {
        factorimage.add(s);
```

```
      }
)* <STR>
      {
            return factorimage;
      }
}

String Simple() :
{
      Token t;
}
{
  t=<ID>
      {
        return t.image;
      }
|
  t=<NUM>
      {
        return t.image;
      }
|
  t=<FLOAT>
      {
        return t.image;
      }
}
```

14 Appendix C - The Recommender Service (RC) API

The documentation for the Recommender Service

org.dbe.kb.proxy

Interface RCI

Public interface **RCI**

Using this interface one can store, update, list, delete User Profiles, and collect recommendations.

Method Summary	
void	**collectRecommendations**() Used to initiate the process for collecting recommendations for all the locally stored profiles. It is supposed to be called periodically in order to discover any changes in either the stored profiles or the available services and provide new recommendations. The returned recommendations can are instantly available through the getProfileResults() method.
void	**deleteUPMModel**(java.lang.String id) Used to delete a UPM model based on the document ID containing it.
java.util.Collection	**getProfileResults**(java.lang.String uid) Used to get all the available recommendations for a specific user identifier.
java.util.Collection	**listUPMs**() Returns a collection with all the locally stored profiles (UPM models).
java.lang.String	**retrieveUPMModel**(java.lang.String id) Used to retrieve a specific UPM model based on the document ID containing it.
void	**storeUPMModel**(java.lang.String id,java.lang.String data) Used for storing a UPM model-document given its document ID.

15 Appendix D – The Query Formulator

Interface IQueryFormulator

Title: QML Formulation API

Description: An API for formulating QML queries and Expressions (**org.dbe.kb.qi**)

Field Summary

static int	**AND**
static int	**OR**

Method Summary

void	**clearQuery**() All the objects created so far by the formulator are clear from the MDR Repository.
org.dbe.kb.metamodel.qml.contextdeclarations.InvariantContextDecl	**formulateConstaint**(java.util.Collection path, java.lang.String operation, java.lang.String value) This method formulates a constrained OclExpression from a Vector of Mof Classes, a string representation of an opearation, eg "=", and a String value.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpression**(java.util.Collection path, java.lang.String operation, java.lang.String value) This method formulates a constrained OclExpression from a Vector of Mof Classes, a string representation of an opearation, eg "=", and a String value.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpressions**(java.util.Collection expresions, int type) Formulates a conjuctive or disjuctive OCL expression
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateFuzzyExpression**(java.util.Collection exprs, double[] weights, int type) Formulates a fuzzy conjuctive or disjuctive OCL expression
org.dbe.kb.metamodel.qml.contextdeclarations.QueryContextDecl	**getQuery**(java.lang.String name, org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression body, java.lang.String result) Constructs QML expressions for fuzzy query

Class QueryFormulator

Title: QML Formulation API

Description: An API for formulating QML queries and Expressions (**org.dbe.kb.qi**)

Method Summary	
void	**clearQuery**() Keeps track of all Objects created by the formulator and clears every time needed.
void	**copyPackage**(java.lang.String fromExtend, java.lang.String toExtend, javax.jmi.reflect.RefPackage fromPackage) Copies one RefPackage from the from MDR extend to the toExtend.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpressions**(java.util.Collection expresions, int type) Formulates a conjuctive or disjuctive OCL expression
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateFuzzyExpression**(java.util.Collection expresions, double[] weights, int type) Formulates a fuzzy conjuctive or disjuctive OCL expression
org.dbe.kb.metamodel.qml.contextdeclarations.QueryContextDecl	**getQuery**(java.lang.String name, org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression exp, java.lang.String result) Constructs QML expressions for fuzzy query

Class ModelQueryFormulator

Title: QML Formulation API

Description: An API for formulating QML queries and Expressions (**org.dbe.kb.qi**)
This class creates model queries.

Constructor Summary

ModelQueryFormulator(org.dbe.kb.metamodel.qml.QmlPackage qml) Creates a new Query instance to be used later on.

Functionality

org.dbe.kb.metamodel.qml.contextdeclarations.InvariantContextDecl	**formulateConstaint**(java.util.Collection path, java.lang.String operation, java.lang.String value) Formulates a QML constraint
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpression**(java.util.Collection path, java.lang.String operation, java.lang.String value) Formulates a QML constraint

Class InstanceQueryFormulator

Title: QML Formulation API

Description: An API for formulating QML queries and Expressions (**org.dbe.kb.qi**)
This class creates instance queries.

Constructor Summary

InstanceQueryFormulator(org.dbe.kb.metamodel.qml.QmlPackage qml) Creates a new Query instance to be used later on.

Method Summary

org.dbe.kb.metamodel.qml.contextdeclarations.InvariantContextDecl	**formulateConstaint**(java.util.Collection path, java.lang.String operation, java.lang.String value) Formulates a QML constraint
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpression**(java.util.Collection path, java.lang.String operation, java.lang.String value) This method formulates a constrained OclExpression from a Vector of Mof Classes, a string representation of an opearation, eg "=", and a String value.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpression**(java.lang.String[] path, java.lang.String operation, java.lang.String value) This method formulates a constrained OclExpression from a Vector of Mof Classes, a string representation of an opearation, eg "=", and a String value.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**refineInstanceQuery**(org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression hard, org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression soft)

Class AdvancedQueryFormulator

Title: Advanced QML Formulation API

Description: An Advanced API for formulating QML queries and Expressions (**org.dbe.kb.qi.adv**)

Field Summary

static int	**INSTANCE_QUERY**
static int	**MODEL_QUERY**

Constructor Summary

AdvancedQueryFormulator(org.dbe.kb.metamodel.qml.QmlPackage qmlPackage, int type) Creates a new Advanced Query Formulator

Functionality

org.dbe.kb.metamodel.qml.contextdeclarations.QueryContextDecl	**getQuery**(QueryExpr[] expressions) Creates and returns a QueryCoonextDecl class.
Template	**getTemplate**() Gtes the template of the formulator.
void	**setTemplate**(Template template) Sets a template to the formulator

Class QueryExpr

Title: Advanced QML Formulation API

Description: An Advanced API for formulating QML queries and Expressions (**org.dbe.kb.qi.adv)**
The objects of this class are actual query expressions

Constructor Summary

QueryExpr()
QueryExpr(java.lang.String operation, java.lang.String id, java.lang.String value, double weight) Creates a new Query Expression for a specific operation, template element id, value and weight

Functionality

java.lang.String	**getOperation**()
java.lang.String	**getTemplateElementId**()
java.lang.String	**getValue**()
double	**getWeight**()
void	**setOperation**(java.lang.String operation)
void	**setTemplateElementId**(java.lang.String templateElementId)
void	**setValue**(java.lang.String value)

void	**setWeight**(double weight)

Class Template

Title: Advanced QML Formulation API

Description: An Advanced API for formulating QML queries and Expressions (**org.dbe.kb.qi.adv**)
This class denotes a reusable query component.

Constructor Summary

Template()

Method Summary

void	**addTemplateElement**(TemplateElement te) Adds a template element to the template
java.lang.String	**getDescription**() Gets the template's description
TemplateElement	**getTemplateElement**(int index) Gets the template Element at the specified index
java.util.Vector	**getTemplateElements**() Gets a collection of the template elements
void	**setDescription**(java.lang.String description) Sets the template's description

Class TemplateElement

org.dbe.kb.qi.adv

Constructor Summary

TemplateElement()
TemplateElement(java.lang.String id, java.lang.String path, java.lang.String type)

Functionality

javax.jmi.model.MofClass	**getContext**()
java.lang.String	**getDelimeter**()
java.lang.String	**getId**()
java.lang.String	**getPath**()
java.lang.String	**getType**()

void	**setContext**(javax.jmi.model.MofClass context)
void	**setDelimeter**(java.lang.String delimeter)
void	**setId**(java.lang.String id)
void	**setPath**(java.lang.String path)
void	**setType**(java.lang.String type)