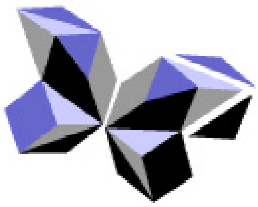
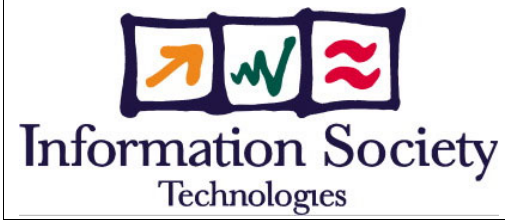


Digital Business Ecosystem

Contract n° 507953

WP14: DBE Knowledge Base

D14.4: Second 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°: D14.4

Due date: 31/12/2005

Delivery Date: 11/01/2006

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-organized 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.

This document describes the mechanisms developed in order to support knowledge access in DBE. 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. Extensions to support fuzzy queries are also described. Fuzzy queries include a mechanism for retrieving services that are expressed in different models. Finally, mechanisms that support semantic query routing through indexing in the P2P distributed DBE implementation are also described. This document accompanies the Recommender software delivered under the DBE Studio and Swallow sourceforge projects and will be demonstrated during the 2nd DBE Audit (January 2006).

Partners owning: TUC

Partners contributed: TUC

Made available to: All project partners and the EC

Versioning

Version	Date	Author, Organization	Description
0.1	10/11/2005	GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS – TUC	Initial Creation
0.2	03/01/2006	NIKOS PAPPAS - TUC FOTIS KAZASIS – TUC	Revisions
1.0	12/1/2006	GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS – TUC	Final Version Submitted

Quality check

1st Internal Reviewer: Miguel Vidal – SUN (Technical Coordinator)

2nd Internal Reviewer: All DBE Computing Partners

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-organized peer-to-peer (P2P) network such as the DBE network. The objectives of the first P2P implementation of the DBE KB were to exploit the current 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 fulfill the first objective is presented in Deliverable D14.3 – 1st P2P Distributed Implementation of the DBE KB and SR [41] whereas the one that was followed in order to fulfill the second objective is presented in this document.

This document describes the mechanisms developed in order to support knowledge access in DBE. Knowledge is coming 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 current implementation of the recommender is described. 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. Knowledge in DBE is distributed over a peer-to-peer network, making very difficult the discovery of such knowledge in reasonable delays. In order to overcome this drawback, mechanisms that support semantic query routing through indexing are applied.

The document accompanies the software of the 2nd distributed implementation of the DBE Recommender that has been integrated as part of the DBE Studio and Swallow projects and will be demonstrated during the 2nd DBE Audit (January 2006). This work is also associated with the prototype milestone M14.3 "2nd Release of the

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

Recommender" (month October 2005) that provides the basic distributed platform for supporting the recommendation services. The Appendix briefly presents the available interfaces for accessing the DBE Recommender. The source code is available within the swallow project in http://swallow.sourceforge.net.

1 Introduction

This document describes the mechanisms developed in order to support knowledge access in DBE. 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 realized 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 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] modeling 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

[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.

semantics of the meta-metadata. Thus, each segment of information that is stored in the KB is placed as an instance of a modeling 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 modeling 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 will be represented using a different metamodel. In order to support efficient knowledge access over all these metamodels there is a need for a query mechanism that will be 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.

Moreover, the DBE knowledge is distributed over SMEs in a decentralized peer-to-peer network. An SME may act as a knowledge base node that comprises knowledge, which 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 to query 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 organized 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 will also follow the same paradigm in order to exploit all the semantic information.

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

Moreover, the DBE knowledge is distributed in decentralized 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], standardized 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.

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 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.

Many systems are developed in the domain of schema based P2P networks; however most of them are either decentralized that do not facing the problem of heterogeneous schemas on each node or the case is not a decentralized system like the DBE network. These systems are using centralized indexes like Napster [32], or follow most costly

[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 utilize XQuery as the language for querying the native XML database management system

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 is 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 centralized-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]. Finally 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 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 Modeling 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.

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 utilizes 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 utilizes 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 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 current 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.

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. These Ontologies will 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. In the current implementation basic recommendation mechanisms are being provided based on the User Preferences part of the User Profiles Metamodel. The SME preferences (profiles) are the instances (models) of this metamodel and functionality for storing, updating, deleting, and listing of profiles is provided to the XML database system.

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, decentralized 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.

The rest of the document is organized as follows: section 2 presents the general architecture of the knowledge access module that supports the formulation and the evaluation of involved discovery requests to the peer-to-peer network of knowledge bases. Next, in section 3 the QML2.0 along with representative examples is presented. Sections 4 and 5 present the methodology used for the formulation of similarity queries and the code generation process. Section 6 presents the supported functionality of

Recommender Services to manage user profiles and suggest recommendations based on the profiles. Section 7 presents methodologies for the semantic exploitation of queries, in order to enhance the recall of the system. Next, in section 8 we present the query routing mechanism that uses semantic indexes and learning to retrieve knowledge from the peer-to-peer network of knowledge bases. The last section presents the conclusions and the issues that are under further consideration.

2 Knowledge Access Module Architecture

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

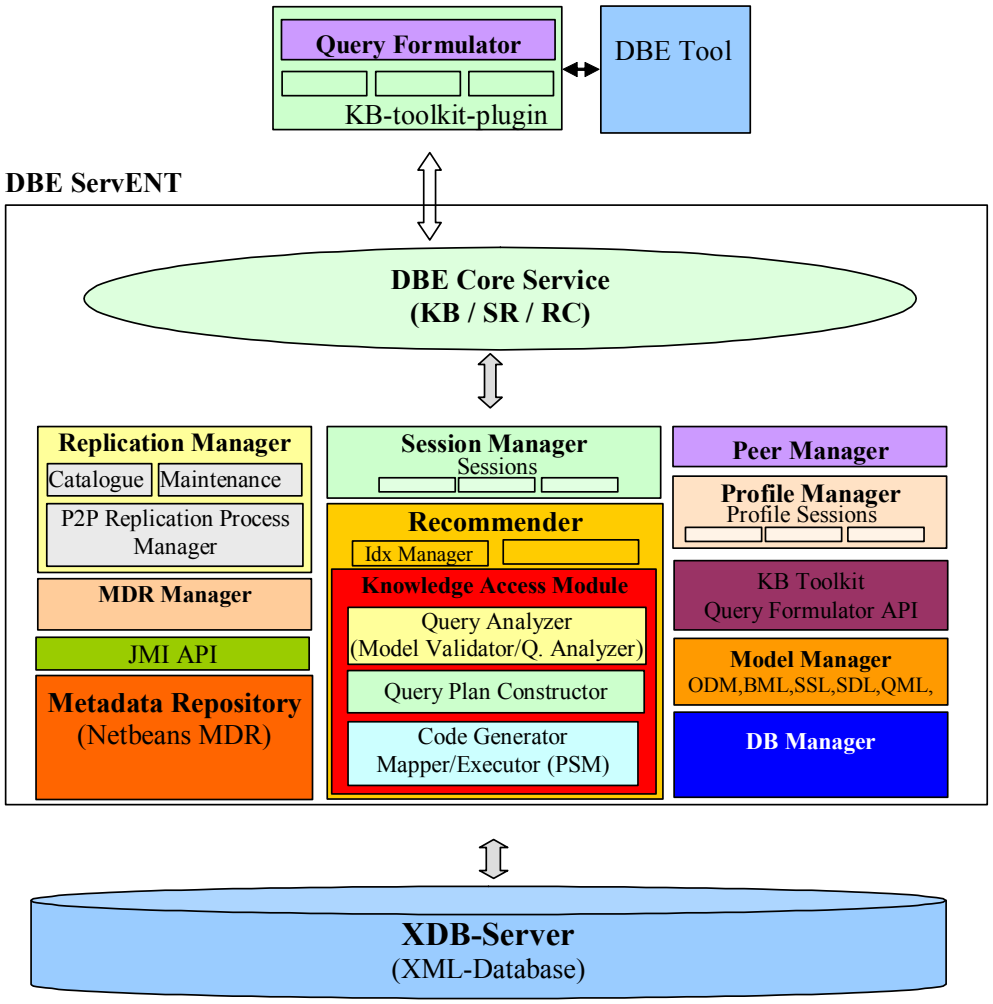


Figure 1: The architecture of the Knowledge Access Module as it was deployed in the second DBE integrated prototype.

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 deliverable of the recommender (D17.1, [24]). 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 Analyzer 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. In order to provide such facilities it uses the JMI reflective API of the MDR Manager to access the MDR repository.

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

The Code Generator module parses the execution plan and constructs queries that can be executed in the current data management system. The Knowledge Access Module is now supported by a powerfull 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.

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 decentralized semantic index based on a learning process and exploiting the rest of the DBE infrastructure for knowledge management.

3 The Knowledge Access Language

This section discusses in detail the terms of the second version of the Query Metamodel Language (QML2.0) and how simple keyword queries can be formulated into QML.

QML 1.0 was presented in detail in the first release of Recommender [24] and it leverages the Object Constraint Language (OCL2.0), which has been used as the formal basis of its metamodel. QML 1.0 supports powerful expressivity of queries on any metamodel and between them.

QML is the platform independent model (PIM) of the knowledge access module. We need a way to pose queries against semantics of the metamodels, models, and ontologies in a uniform way, in order to support semantic analysis of the queries instead of just syntactic analysis that SQL, XQuery, etc offer. This capability will be better shown in this deliverable where the distributed infrastructure of DBE rise a number of problems which can only be solved one layer higher than PSM (the XML database management system), i.e. at the PIM level.

QML 2.0 structure has slight changes from QML 1.0 in order to provide a more flexible way for defining queries. This includes ease support of multiple contexts, straightforward support for input parameters, and concrete definition of the result type, outside OCL.

In this section we will present only the changes from QML 1.0 to QML 2.0. For a detailed description of QML 1.0, the packages of QML 2.0 that have remained the same, and further examples please refer to deliverable D17.1 (1st release of Recommender) [24].

3.1 The QML metamodel

This section presents the QML metamodel in terms of a MOF metamodel.

The core QML metamodel consists of two packages; the Expressions package and the Types package, where the OCL expressions and types are defined respectively. QML also contains the Context Declarations Package, which makes use of the core QML package (OCL package) in order to express queries and constraints separate from the core of a MOF model. Furthermore, QML core uses the core MOF metamodel, so as to both refer directly to a MOF model's elements and express constraints incorporated in a MOF model. The core QML package (OCL package), since it is the same with the abstract syntax of OCL 2.0 properly aligned for MOF, is not presented here. Please refer to the OCL 2.0 specification [2] for details about these packages.

3.2 The Context Declarations Package

Context declarations are not needed in OCL, because OCL constraints are meant to be directly attached to the model elements they refer to. Nevertheless, a concrete syntax for them is given in the OCL2.0 specification [2] in order to facilitate the declaration of the OCL expressions in separate text files. Based on the concrete syntax we developed the Context Declarations package, which does not belong to the Core part of QML but is rather a set of helper meta-classes. These helper meta-classes are used to express information on where and how an OclExpression will apply. In particular, they express

information about the context of an OclExpression, its type (invariant constraint, operation or attribute definition) and any other specific information needed for each type. The adopted OCL2.0 specification explains in detail the concrete syntax of Context Declarations.

To express the idea of a query as a constraint on model elements resulting to a set of values with a specific type we have added the QueryContextDecl meta-class. Figure 2 shows only a part of the context declarations package with the QueryContextDecl meta-class, which is explained in detail at the following section.

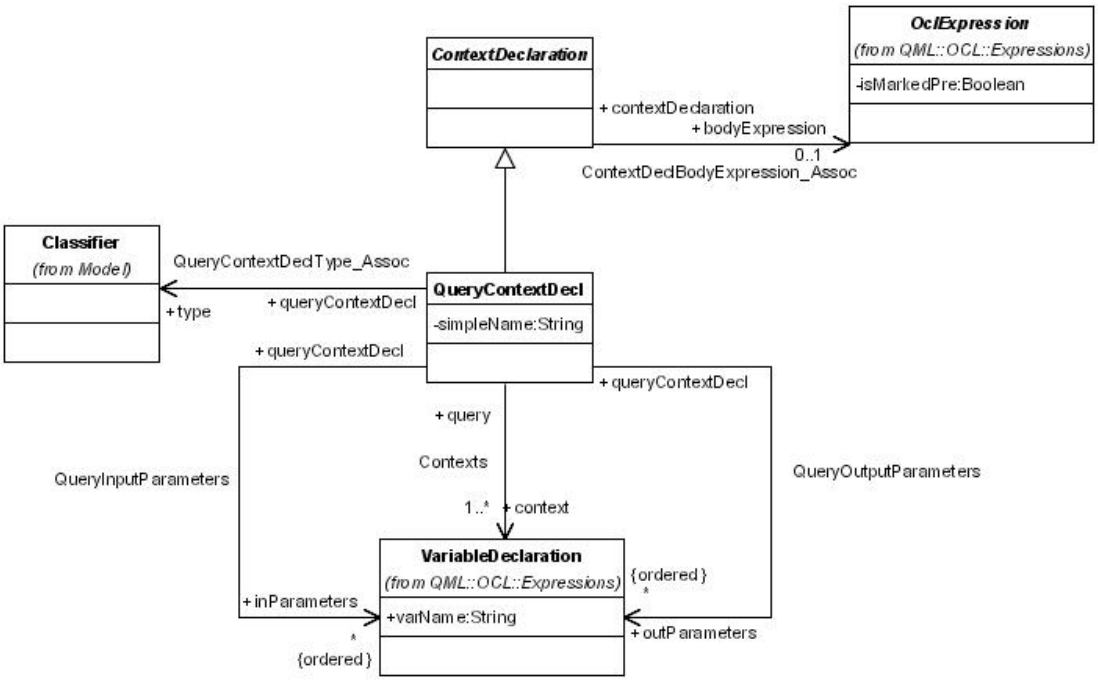


Figure 2: Part of the Context Declaration Package, showing the Query Context Declarations metaclass and its associations.

3.2.1 The QueryContextDecl Metaclass.

An instance of the *QueryContextDecl* class represents formally a query. It has a name for quick reference (*simpleName*). The query returns a set of 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 2.0 specification. The *bodyExpression* is analogous to the *"where"* part of an SQL statement but more powerful. *QueryContextDecl* has a set of *VariableDeclarations* as *contexts*. The *context* of a query is the object (model elements) where the *constraint bodyExpression* refers. An instance of *QueryContextDecl* may have any number of *contexts* that allows queries to combine semantic information from more than one metamodels or models. *Contexts* are analogous to the "*from*" operand of an SQL statement. The *QueryConetxtDecl* 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. This functionality is analogous to stored procedures of SQL. In this manner, one can form and store query templates and reuse them at any time.

3.3 Representative Examples

This section presents a number of simple examples that demonstrate QML and its abilities.

3.3.1 A Simple QML query example

In this and the following subsections, we present representative query expressions formulated with the QML. The objective is to give an informal presentation of the semantics of the QML expressions. The query expressions refer to M2 (i.e. available M2 metamodels) and obtain, as a result, qualified M1 models. We will first examine a simple example that demonstrates the usage of the QML metamodel. In section 5, 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 [18], 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, which are also shown in **Error! Reference source not found.**, are used for the formulation of the example queries:

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
```

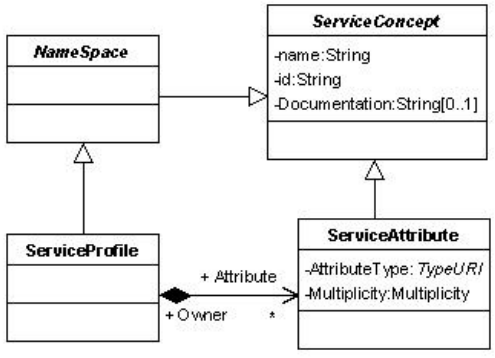


Figure 3: A part of the Service Semantics Language (SSL) metamodel

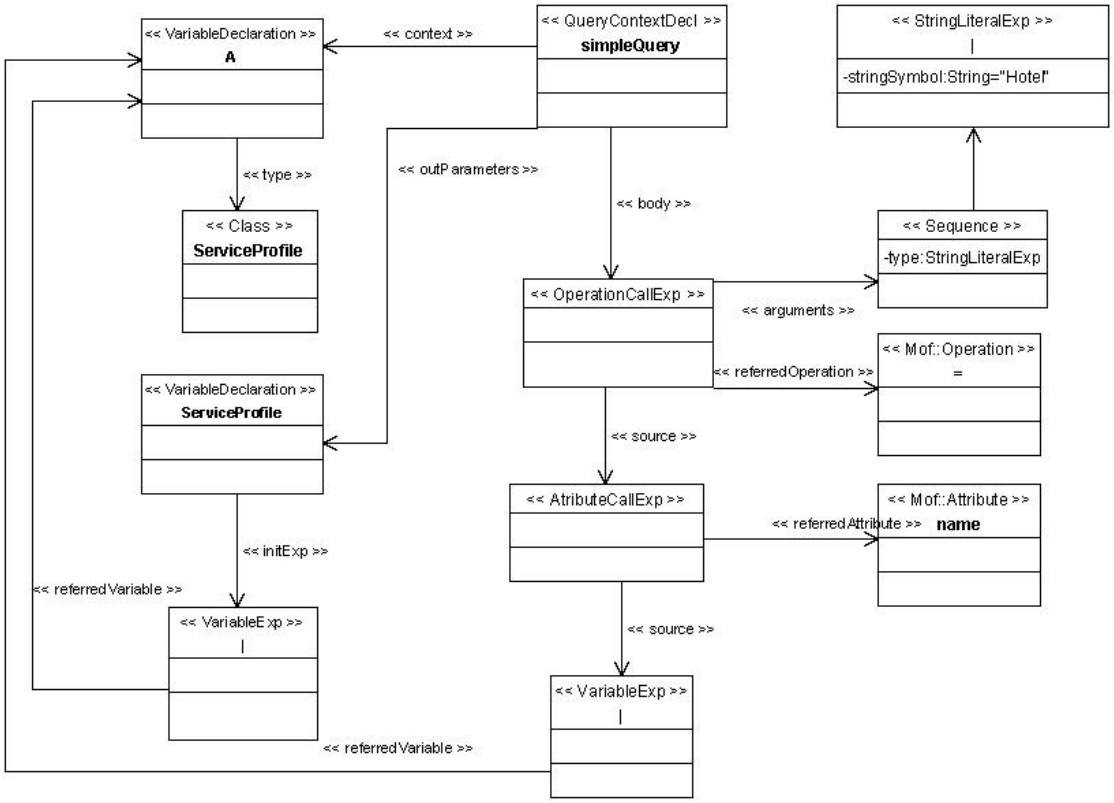


Figure 4: QML representation of the query: Context A: SSL::ServiceProfile simpleQuery A.name = "Hotel" out ServiceProfile: = A.

Figure 4 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 the OperationCallExp referring to the operation "=" of the primitive type String. This OperationCallExp has only one argument: the StringLiteralExp "Hotel". Moreover, the OperationCallExp has a navigation source where it must apply. The source is the AttributeCallExp referring to the Attribute "name" of the ServiceProfile class. The source of the AttributeCallExp is a VariableExp referring to the variable A, which is the context of the query.

3.3.2 Aggregating objects

In the previous section, we showed a simple query that selected the ModelElements of type "ServiceProfile" that matched a set of criteria. 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, 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 5 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.

3.3.3 Querying instances

All the previous examples showed how QML is used to find models (i.e. models that have a "ServiceProfile" with name "Hotel" etc.). 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.

Consider the following statement that searches for the "Hotels" that have at "HotelName" ServiceAttribute the value "Hilton" and are located at "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 frequently. We will discuss this problem and propose a solution at section 7 (Semantic Exploitation).

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 formulated into QML ones.

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 1. 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 1: Examples of criteria for the query formulator API

When the user has formulated all the criteria into QML expressions, he/she then creates the general expression by joining the simple expressions with disjunction and conjunctions. The general expression is then formulated, along with the result arguments, into the final QML query.

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 release or recommender [24].

As already mentioned, query results and recommendations in an environment where information is modeled with different metadata structures even in same domains 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. It has to be noted that this extension approach 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 recommendation mechanisms. It has to be noted that at a technical level the information filtering/retrieval 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 repository, the recommendation process is based on matching user profiles (that include preferences on models) and the underlying information. All recommendations and discovery requests computed by the Query Service could be considered as similarity based retrieval requests.

Referring to table 2 all different kinds of recommendations encapsulated in the three candidate use cases (see the "Universe" column) can be modeled using the same general mathematical framework based on information retrieval theory. This framework is summarized here and the implementation of this framework on top of a MDA repository is given.

A generalized 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 2.

The objective now is to define a generalized 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 should extend the generalized framework to work on such a system and develop mechanisms that use pure QML in order to support all kind of recommendation functionalities.

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

Table 2: Different kinds of Recommendation functionality expressible by a general information retrieval system

4.1.1 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 fNOT, fAND, and fOR. There are numerous possible definitions of these functions. Table 3 presents the definitions that correspond to the p-Norm Extended Boolean Model, which is the most general one. Note that the functions fAND, and fOR are n-ary instead of binary. This is because these evaluation functions are not commutative as their Boolean counterparts.

$f_{AND}((a_1,w_1),\ldots,(a_n,w_n))$	$f_{OR}((a_1,w_1),\ldots,(a_n,w_n))$	$f_{NOT}(a)$
$1-\left(\frac{\sum_{i=1}^{n}(1-a_i)^p \cdot w_i^p}{\sum_{i=1}^{n} w_i^p}\right)^{1/p} \quad 1 \le p \le \infty$	$\left(\frac{\sum_{i=1}^{n} a_i^p \cdot w_i^p}{\sum_{i=1}^{n} w_i^p}\right)^{1/p} \quad 1 \le p \le \infty$	$1-a$

Table 3: Weighted n-ary basic evaluation functions for the p-Norm Extended Boolean Model.

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 actually QML path expressions and a query weight 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 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; the *a* element of table 3 equations.

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 table 3.

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 recognized by the system are simple conjunctive queries.

4.1.2 Handling Arbitrary Complex Queries

What happens if the queries to be evaluated are arbitrary complex queries instead of simple conjunctive or disjunctive queries? This is a critical question in order to make the techniques applicable in our context. One can follow a solution that is based on two steps. The first one is the decomposition of the complex queries that should be evaluated into simple (either conjunctive or disjunctive) hierarchical components that take place during the construction of the query's execution plan. The second step evaluates these components following a bottom up approach that is taking place during the code generation phase. We are currently investigating these issues to provide a more generic framework.

4.1.3 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 the table 3 equations.

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 set of criteria which are shown in table 4.

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 original value
Numeric	<	<=	>	Larger as closer to original value
Numeric	>	>=	<	Larger as closer to original value

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 7 where the semantic exploitation of the query will be discussed a different value will be assigned to *a*.

4.2 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 realizes 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.

4.3 Formulating Keyword Expressions

This section discusses how keyword expressions can be formulated into valid queries. Keyword expressions can be either unstructured or semi-structured as was shown at table 1.

Unstructured query terms (i.e. keyword expressions) 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 feature of a model or ontology, the latter 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 it is assumed that all query terms have the same weight, i.e. are equally important.

5 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 current 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 as a parallel to the SQL query produced for the previous release of the recommender.

5.1 The Mapper

The use of the XMI format for the management of the models in combination with the use of an XML database has resulted in a more straightforward manipulation of the models and data. The need for a mapping between an XML schema or an XML like schema and a relational schema no longer exists. The queries can directly be generated in the XQuery language and posed against he XMI files holding the models or the data of the system. The current role of the mapper comes exactly at this point to allow XMI files to be treated as XML files without any limitation.

In the XMI files used, there are two cases in which the direct generation of an XQuery based on paths in the file is impossible. The first and most critical is resulting from the fact that XMI allows referencing of elements based on their IDs anywhere in an XMI file. This can result in elements containing other elements that are situated not in a sub-path of the current element, which is not the case of an XML file that would allow the construction of an XQuery. The role of the Mapper in this case is to hold a list of the elements that are positioned outside the main structure of the XMI document and thus referenced with the XMI referencing mechanism. When such elements are met during the generation of a query path, they are recognized based on the look-up structure, described above and the path to them is rewritten using the XMI referencing mechanism. In other words the sub-path is substituted with a selection containing the specific element ID referencing to the actual position of the element in the file. The second case that has to be handled by the mapper is when there is a reference to an element outside the file in the case for example of a reference to an element in an ontology which is held in an other file, or to be speaking with XML database terms, in another container. These elements are also kept in a second look-up structure to allow them not only to be recognized but also to recognize their base container. When they are met during the construction of the query their container has to be included in the query and selection referencing them in their original document has to be added in the query path. Below an example produced query containing a reference to an ontology is given:

```
FOR $p IN collection('sm.dbxml')//XMI.Content,
      $o1 IN collection('odm.dbxml')//Class
WHERE
$p//SSL.Core.ServiceParameter/SSL.Types.OntologyClassURI
      [@lexicalForm = $o1/@id and $o1/@name = 'CreditCard']
RETURN $p
```

In this example the original query contained a ServiceParameter with a type reference to an ontology concept 'CreditCard'. For the production of the final query a second collection with the ontology is added and a selection with a reference to the concept in the original ontology is added to the path expression.

5.2 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 syntax 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[5]:

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

- recognize all the contexts involved in the query
- construct the full path for each expression involved in an operation
- recognize 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-

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

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 list of any strictly Boolean conditions, the hard constraints, is still found in the list contained in the where clause. The additional part is the list of the LET expressions. For each fuzzy condition `i` in the original query, three things are calculated: the original path expression with its operator and constraint, this is the `fei` part, the original expression using a fuzzy matching pattern, this is the `fci` part, and finally a function returning a weighted value for the specific term used in each specific conditions. The results of all these terms are summed and normalized 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 above pattern example works for values that correspond to string literals. In the case of numerical values and operators like (<, >), some other fuzzy partial matching has to take place instead of the 'contains' function. This has been implemented with a new function checking the range of values around the numerical value of the query either starting from a range query or a complete match one.

In the example below an implementation for the `fuzzyor` function is given along with a complete example of a query containing both soft and hard constraints.

```
declare function local:fuzzyor($n1 as node()*, $n2 as node()*,
$wp as xs:float) as xs:float {
    if (exists($n1))
    then $wp
    else if (exists($n2))
    then 0.5*$wp
    else 0 cast as xs:float
};
```

As one can notice the parameters for the `fuzzyor` function are the two calculated expressions for each term, `fei` and `fci`. If the first of them has a returning node a

complete much for the value has been found and the function returns the initial weight of the term. If the second only expression has a returning node, only a partial match for the value has been found and the function returns a percent of the initial weight.

```
FOR $p IN Collection('dbe-ssl.db')//XMI.content

LET $fe1:=
$p//SSL.Core.ServiceProfile/@name[.='HotelReservation'],

    $fc1:=
$p//SSL.Core.ServiceProfile/@name[fn:contains(.,
                                          'HotelReservation')],

    $fe2:=
$p//SSL.Core.ServiceProfile.Attribute/@name[.='ServiceProvider'],

    $fc2:=
$p//SSL.Core.ServiceProfile.Attribute/@name[fn:contains(.,
                                          'ServiceProvider')],

    $r:= (local:fuzzyor($fe1, $fc1, 0.44) + local:fuzzyor($fe2,
$fc2, 1.0)) div 1.44

WHERE $p//SSL.Core.ServiceProfile/@name like 'as'

ORDER BY $r descending

RETURN <group><docid>{$p}</docid> <rank>{$r}</rank></group>
```

The above query constitutes of three constraints. The two soft constraints on the values "HotelReservation" and "ServiceProvider" and a hard constraint on the value "as".

The construction of more complex paths for the support of selection nodes or and the construction of paths for referenced nodes either in another part of the same document or an external ontology with the use of the mapper has been also implemented.

6 Recommender Service

As described in the introduction of the current report, the recommender component in the DBE is used to personalize 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.

6.1 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 5. 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'*).

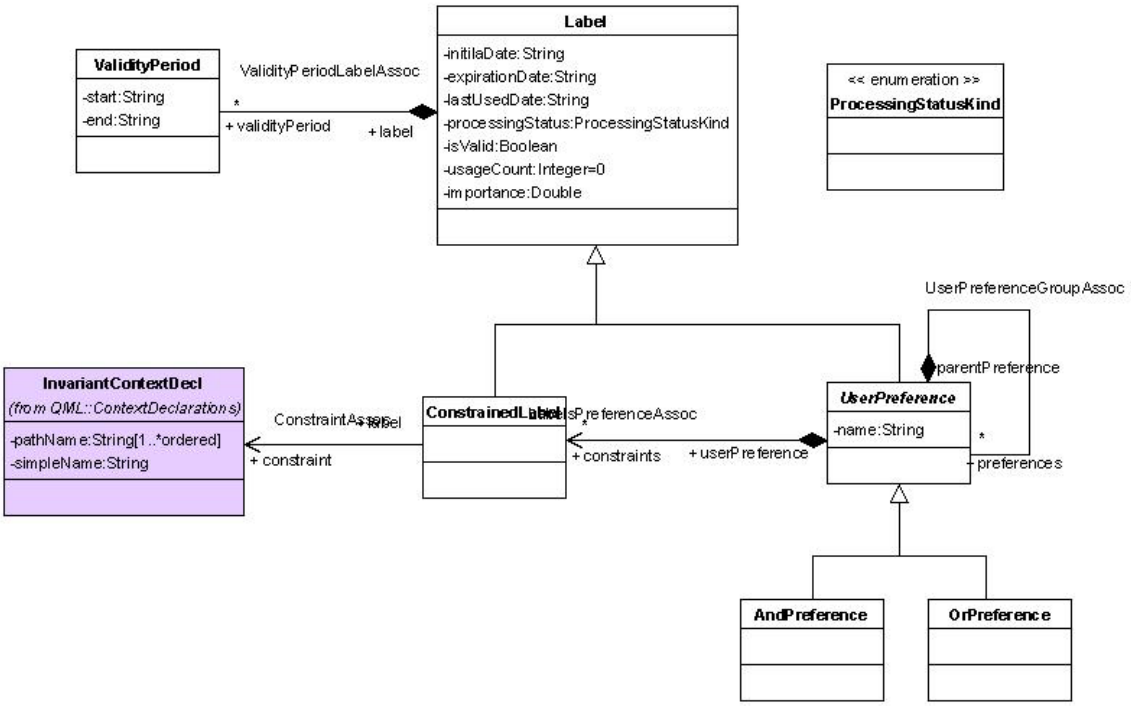


Figure 5: The User Preferences package of the User Profile Metamodel.

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

6.2 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.

6.3 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 in 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 are subject of one of the following sections. 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.

7 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 original one, but refer to the same kind of service.

Recall the problem stated at section 3.3.3 where a data object could not be retrieved because the query was expressed in terms of a different metamodel. 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 the reason we selected a query language that is expressed in semantic terms instead of directly using a PSM language as XQuery or SQL.

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.

7.1 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 can assume that each domain is described by only one ontology, as the idea of ontologies is that are a set of commonly agreed concepts that can describe a domain fully.

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 or refer to similar services. However, in DBE's environment, where knowledge is highly distributed, how can the term domain be 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 table 3 equations 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 section 4.1 is the v_i and thus, the issue is to resolve the value of c_i.

7.2 The three approaches

There are many different ways to assign a value to c_i. In this section we will explore three of them. All these approaches share in common the use of ontologies to find equivalent concepts and the metamodels to find relevant concepts. Example of the first is the "Hotel" and "Hostel" equivalent classes and of the second the inheritance of the concepts from the same metamodel concept (inheritance from Business Event of BML or Contact Information of SSL for example).

The first is to explore the model and ontology concepts of the query terms and find equivalent or sub-concepts of them for each of which a different query is formulated with a corresponding c_i and forwarded to the executor. The results are finally joined and forwarded back to the originator of the query. This approach has as drawback the high complexity of finding relevant concepts for each model and ontology concept and the creation of a large number of queries to be executed.

The second approach is to execute only two queries, the original one "as is" and a reformulated one that uses only the sub-concepts of query terms those refer to ontologies, i.e. a "model query" and an "ontology query". The "ontology query" has as c_i a value smaller that 1 and the "model query" the value 1. This approach has as advantage that only two queries are executed and it is quite simple to implement. The drawback is that sub variations of models are not captured and are treated in a global manner.

The third one is bottom up and is a hybrid approach of the previous two. It executes only one query that retrieves the concepts that their data conform to the most general query. For example for the query term *Hotel.Address.City="Chania"* the system searches for *Country="Finland"* and retrieves the concepts for service. For example, it retrieves "Hotel.Address.Country", "Hostel.Address.Country", "Hotel.Country", and "Restaurant. Address.Country" with their corresponding v_i values. The next step is to assign a value to c_i for each one of these concepts using information from ontologies and metamodel. Some demonstrative values could be: 1, 0.5, 0.8, 0.5, and 0.1 respectively.

8 Query Routing using Semantic Indexes and Learning

As already described in section 3 of the current report, the DBE environment is based on a P2P network. The P2P paradigm has emerged and radically become widely used providing a new more straightforward and decentralized way for accessing data. The advantages of the nature of P2P networks are well known, however the behaviour of such networks is based on the specific characteristics of each deployed system. In the DBE the case is a fully decentralized 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.

8.1 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.

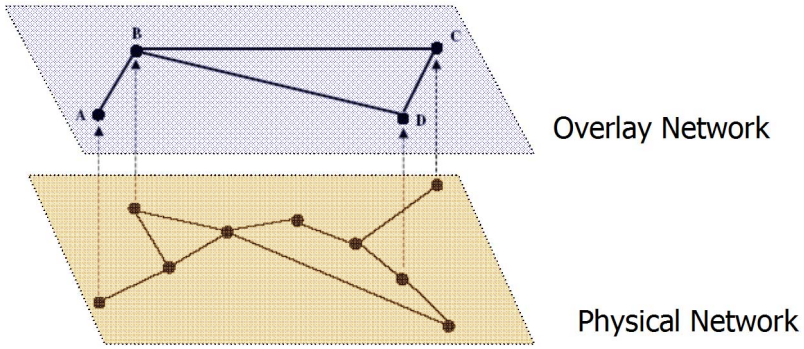


Figure 6: A P2P overlay network example

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 decentralized 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. The organization of this information and knowledge in general in the DBE is described in the next paragraph. Next, the searching mechanisms will be analyzed. A first approach for the construction and use of a fully decentralized index will also be presented. Finally the evolution of the whole framework towards a fuzzy semantic overlay network will be introduced.

8.2 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 existent in the DBE, the process of constructing a model description personalized 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. So the first characteristic of the network is a different model for the description of each single node constructed locally but based on some domain ontology. Below in figure 7, an example of two models created by two different nodes is shown.

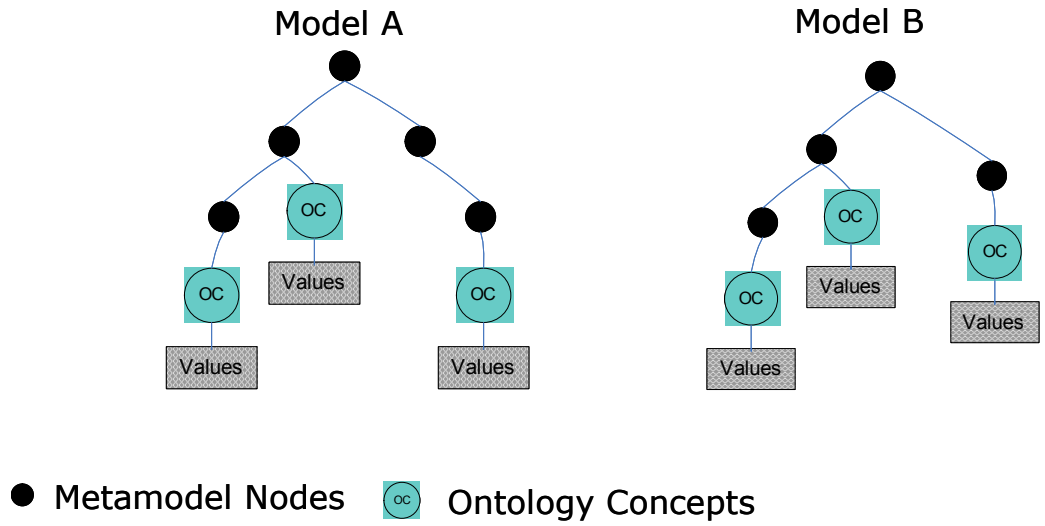


Figure 7: Creation of models based on the common metamodels with the use of domain ontology concepts

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

Before going further to describe the searching mechanisms we must shortly refer to the formulation of a query that will then be processed using the searching mechanisms in order to come up with some results. Regardless of who is initializing the query formulation the important part is that it will be based on one of the above-mentioned models, residing on one of the nodes. As the queries are based on QML they can directly contain the model or part of the model used for the formulation, completed with the requested values. That is, the original model will be edited and supplemented with the correct values and a QML query will be constructed and send for processing.

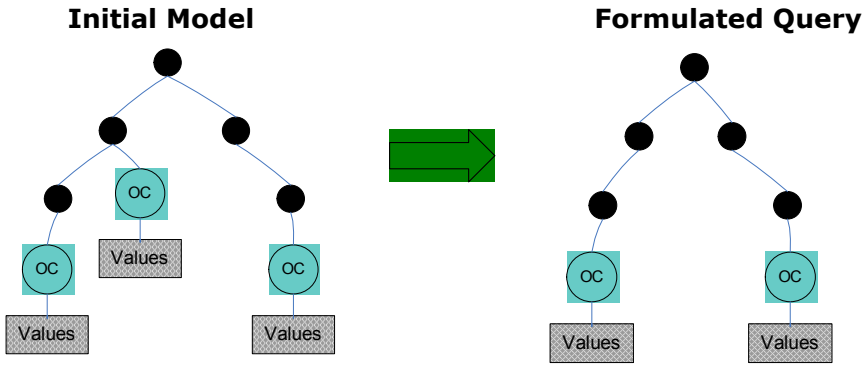


Figure 8: Query formulation, based on an existent model

In the above figure the original model is truncated and for the remaining nodes the requested values are provided in order to formulate a complete query. After the formulation of the query, processing will have to take place both locally and distinctly, in the rest of the nodes. It is obvious that the local node can straightforwardly process the query as its description is based on the same model used for the formulation of the query. For the rest of the nodes two are the major issues that have to be answered. The

first is the routing mechanism that will be used in order to avoid flooding of the query to the entire network and the second is the processing mechanism that will be used before forwarding any query and after receiving a query from another node.

In this paragraph the local processing of a query after its reception will be analyzed. In the previous section the semantic analysis and the query reformulation process have been described. Each query comprises of one or more sub-queries and each sub-query is made of a path to a specific concept in a model and the requested value for this concept. In order to answer a query, all the parts of the query that is all the sub-queries must be processed in the queried node, and a total rank for the node and for this query must be produced. The important thing at this stage is that in order to produce a matching for a sub-query, three parts of the query must be matched: the path in the metamodel that leads to the matching concept (A), the concept (B) itself and finally the value(C) that was filled in the specific query.

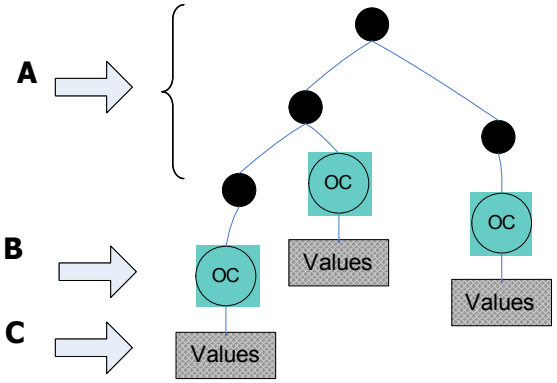


Figure 9: The three parts of a query that have to be matched: A) the path in the metamodel, B) the ontology concepts and C) the data values

The matching of the path and the value is a procedure already implemented and used by the DBE. The tricky thing here is that every node has potentially its own model. This can lead to a different concept used by the specific node. The key here is that these concepts are taken from the common domain ontologies available to all the nodes. Based on this assumption two are the possibilities: either the initial node used for the query formulation has used the same domain ontology with the queried node, or another ontology in the same domain has been used. In both cases the node receiving a query has to analyze the query and identify all the concepts contained in it. If the concepts originate from the same domain ontology the matching can be conducted with the already known mechanisms. In any other case the semantic analysis and query reformulation process presented in the previous section will allow the node to identify the concepts in the query and provide a matching for the query with the locally recognizable concepts. It is obvious that this may not be possible for all the concepts in all the sub-queries comprising the query. In that case, the part of the query will be left without a matching and will not contribute to the final results affecting however the total rank of the query.

The formulation of a query in the DBE environment has been analyzed and the important points in the answering procedure have been depicted. These mechanisms, alone, are enough for creating a query in the DBE P2P environment and with the use of the Recommender Service or the Semantic Registry Service for flooding the query in the network and collecting the available results. Of course this procedure has already been

implemented and tested but it must be considered as the naive approach for a searching mechanism in the DBE environment. The current implementation of the Recommender must take into account the Semantic Overlay Network in the DBE and base its searching mechanisms on an advanced semantic indexing mechanism that will allow a more efficient forwarding of the queries in the network. In the next paragraph the construction of an index with its structure and its learning mechanism will be described.

8.3 Semantic Indexing and Learning Mechanisms

All the queries in the DBE are formulated in the QML. On the other side all the data and metadata descriptions follow the MDA approach and reside on the different nodes waiting for the matching either directly with a query or with the reformulated query produced after the semantic analysis of the initial query. After having analyzed the querying mechanism there are some strong indications about the nature and structure of the index that would allow the efficient routing of the queries. It is quite clear that the important parts in the analysis of a query were the path of each sub-query, the ontology concepts contained and finally the requested data values. This information is exactly what is needed as a part of the indexing mechanism. To be more precise, in order for the index to be useful, every time a single query is processed the above-mentioned information has to be stored in the index. Going through the whole procedure, every time a new query arrives at a node, it will be broken down to the contained sub-queries and analyzed. Then it will be processed locally and forwarded to the network. Leaving aside for the moment the forwarding mechanism, the node will start collecting results from the network. Based on the results, the node has to keep track of the ability of each of the other nodes to answer the forwarded query. The structure used for the index is of the following form:

Path | Ontology Concept | Rank | Node ID

The important thing at this point is the fact that the query kept in the index associated with the answering nodes, is the original query and not the reformulated query for each of the nodes, something that would turn the index into a caching mechanism. It must also be clear that the queries stored are the possible sub-queries resulting from analysis of the initial query and not the complex initial query in the common case that it refers to more than one concepts or data values. What is still under evaluation is the need for either keeping an association between sub-queries contained in the same query or the answering node having the ability to return additional information like a rank for each sub-query processed locally at his side. Another important part of the mechanism is the fact that each node for every received query, after forwarding it to the network collects back the results from the different nodes and is also responsible for sending them back to the node from which the query was received. 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 sort 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	147.27.3.143
ServiceFunctionality/Name/..	"Cancellation"	1.0	147.27.3.143
ServiceAttribute/..	"Address""	0.4	147.27.1.45
. . . . .			

Table 5: The contents of an index residing on a DBE node

Having a table with the above information any node can proceed in more efficient ways for the query routing. 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.

8.4 Using the Index and "Cold Start" Routing

In all this section 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 naive approach for actually advertising the nodes 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 bases for a more concrete and flexible facing of the above challenges.

Finally, assuming that a sort or a longer learning period has elapsed and the index has been trained the forwarding mechanism can make use of the index. More specifically after the analysis of each query (and query sub-queries) the node can find in its index

the nodes that best qualify for matching the required concepts. This list of nodes can then be used for forwarding the query avoiding the flooding to the network. In addition, the initial node can use a threshold for the minimum number of results so that in case the indexes do not suffice for providing this minimum number of results, further flooding of the query can take place. This may prove useful in the early stages of the index learning process.

To summarize, the framework for the construction of the DBE semantic overlay network has been described. This includes the combination of the query formulation, query forwarding and indexing mechanism. The structure and the current approach for the utilization of the semantic indexes, has also been described. The characteristics of the DBE P2P network have led to a completely decentralized semantic indexing mechanism exploiting fully the advantages of the semantic overlay network. The evolution of this network to a fuzzy semantic overlay network is the subject of the rest of this section.

8.5 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 decentralized 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 neighboring 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.

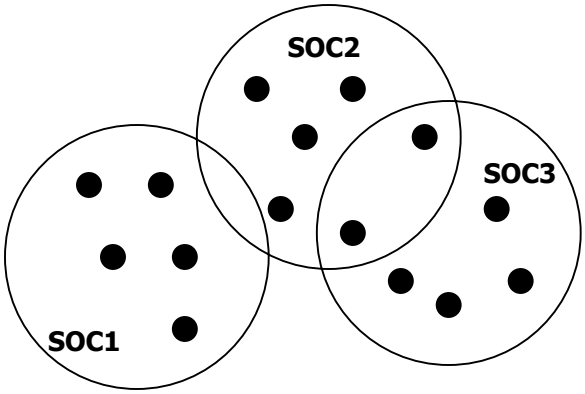


Figure 10 Formulation of semantic clusters of the nodes in the overlay network.

In Figure 8, 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 they belong. As this is a local procedure, its 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 decentralized 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 decentralized P2P environment of the DBE.

9 Conclusions

In the new release of the recommender, the most important issue was the extension of its mechanisms and algorithms over the fully decentralized 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. For this, QML 1.0 has evolved to QML 2.0 while the Query Formulator Module was created in order to facilitate the creation of more complex queries taking advantage of all the QML capabilities and serving the need for advanced information retrieval techniques with the use of fuzzy expressions.

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, imposed the utilization of an XML database. The code generation module was adapted for the generation of queries in the XQuery language while transparently recognizing the QML capabilities and the need for efficiently supporting fuzzy queries following the p-norm Extended Boolean Model.

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 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, decentralized 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 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

11 Appendix A - 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.

12Appendix B – 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.

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) 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

Method Summary

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

Method Summary

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)

Method Summary

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)