ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός και υλοποίηση γλώσσας ερωτήσεων και σημασιολογικού συστήματος ανάκτησης γνώσης για κατανεμημένη P2P Βάση Γνώσης.

Διατριβή για την ολοκλήρωση του μεταπτυχιακού προγράμματος ειδίκευσης ηλεκτρονικών μηχανικών & μηχανικών υπολογιστών

Γεώργιος Κοτόπουλος

Οκτώβριος 2010

SUMMARY

In the modern economy it is observed a high diversity in the data models among different companies. Each and every company expresses its business and service models in a different way and it cannot be foreseen that these models are going to be standardized (even for each domain) in the near future. In such environment querying data is a difficult task as the system has to create different queries for each different data structure.

Meta Object Facility (MOF) is a modelling framework by using which one can create and and manage metamodels, models and data. The work in this thesis is part of the Digital Business Ecosystem (DBE), an EU/IST project. DBE has a Knowledge Base where SMEs can define (using MOF) and store business and service models along with appropriate data. This information is distributed among peers in a P2P fashion. SMEs describe their data not only in terms of business and service models, but also with ontologies. Moreover, ontologies are not shared between companies but may differ. The reason is that SMEs may find that something is not contained in an ontology and add the required structures.

The aim of this thesis is to provide the mechanisms to query structured data (structured by metamodels and models) with fuzzy algorithms enhanced in such a way that data structured in different ways can be retrieved as well. In order to achieve this we designed and implemented the Query Metamodel Language (QML), a language which can pose queries against metamodels, models and data. The implementation uses the semantics of models and ontologies in order to provide ranked results. The ranking is done with a fuzzy information model provided. We implemented algorithms for calculating similarities between ontology elements. These similarities are used in order to find similar ontology paths to the path of the query terms. The query is expanded with these similar paths, and thus data structured in different ways can be retrieved by using the semantics of the structures used.

CONTENTS

LIST OF FIGURES

List of Figures

LIST OF TABLES

ACKNOLEDGEMENT

The author would like to thank Prof. Stavros Christodoulakis for supervision and guidance for the preparation of this thesis, and the important lessons offered him during his work at the Laboratory of Distributed Information Systems and Applications (MUSIC/TUC).

He would like also to thank the members of this thesis committee Associate Prof. E. Petrakis and Assistant Prof. V. Samoladas for the time they spent in reading it and for their valuable comments.

A special thank to his colleagues of MUSIC Fotis Kazasis, Nikos Pappas, George Anestis, Nektarios Gioldasis, John Kotopoulos, and Anastasia Karanastasi whose timely and sound work has led to a faster fulfillment of this master thesis and their suggestions and assistance impacted the design of this work.

This thesis was funded under the European Research Project Digital Business Ecosystems (DBE), on which participated the laboratory of Distributed Systems and Applications (MUSIC) of Technical University of Crete.

PUBLICATION

Part of the work of this thesis was published at the Conference Proceedings of IEEE “Digital Ecosystems and Technologies” as the main author under the title “Querying MOF Repositories: The Design and Implementation of the Query Metamodel Language (QML)”.

CHAPTER I – INTRODUCTION

This thesis describes the mechanisms developed in order to support knowledge access in a distributed and collaborative environment of SMEs called Digital Business Ecosystem (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 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*" (1). 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 XML Data Management System. Other implementations could be also possible.

In addition the Knowledge Base follows the OMG's Meta Object Facility (MOF) (2) approach for metadata and data[1] modelling and organization. The DBE Knowledge Base supports the four levels of the MOF architecture. The level M0 of the architecture consists of the data that we wish to describe; the level M1 comprises the metadata that describe the data and are informally aggregated into models; the M2 level consists of the descriptions that define the structure and semantics of the metadata and are informally aggregated into metamodels; and the M3 level consists of the description of the structure and semantics of the meta-metadata. Thus, each segment of information that is stored in the KB is placed as an instance of a modelling element of a higher layer of the MOF meta-data architecture. That is, MOF based languages or mechanisms should be used in the upper levels of the architecture for defining each segment of information. Different kinds of metamodels[2] have been already developed and represented in the KB:

- the metamodel for Ontology Definition (ODM) (3), which enables the representation and storage of existing OWL domain ontologies into the KB
- the metamodel for the Semantic Description of Services (SSL) (3), which enables the representation and storage for the semantics of the services offered by SMEs into DBE.
- the metamodel for the Business Modelling (BML), which enables the representation and storage of business models, business rules, policies, strategies, views etc by SMEs into DBE.
- The metamodel for User Profiles (UPM) (4), which enables representation and storage of user profiles.
- other metamodels for the technical description of single and composite services (SDL, BPEL)

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. Moreover, the DBE KB supports personalization of services by allowing business differentiation from the common standards and models. Thus, each SME may use these languages to express itself and its services but the models and data produced will differ.

[1] 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.

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

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) (5) functionality of the recommender component. Moreover, in order to address the differentiation in model level there is a need the knowledge access mechanism to use all semantic information available in the system.

To this end, in this thesis we describe a query mechanism, which is based on a query metamodel (language) 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.

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.

Many technologies have successful and powerful query mechanisms that are widely known, understood and used. The Structured Query Language (SQL), adopted as an industry standard in 1986, is a very successful language for relational databases. More recently, SQL-99 (6) has introduced object-oriented concepts into the language. However, there are significant differences between the object models of the MOF and of object relational databases (SQL-99 is also restricted to using only linear recursion). Since MOF models and instances can be mapped to XML documents (XMI (7)), an XML query mechanism can be easily integrated with MOF technology. XQuery (8) and XPath (9), standardized by the W3C, are some of the many query languages for XML documents. The problem of querying in the MOF can be reduced to an already-solved problem of how to query 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. 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 (10). However, the standard does not define the language's abstract syntax nor formal semantics.

The approach that was followed was to define an object-oriented knowledge access language, named Query Metamodel Language (QML), using the same meta-language (MOF) that was used to define the languages that represent the DBE Knowledge. To achieve as much compliance with the existing standards, we opted to leverage the Object Constraint Language (OCL2.0), which has been used as the formal basis of our query metamodel. The choice of OCL was also motivated from the fact that OMG advocates in the core of its business architecture the use of MOF and on the top of it the use of Unified Modelling Language (UML), which contains OCL for specifying constraints in the models. Thus mechanisms that support OCL would be also useful for efficiently supporting UML in a Knowledge Base that would support also UML functionality.

In the past few years OCL has evolved from being merely an extension of the Unified Modelling Language (UML) to representing an integral part of it. The latest response to the UML 2.0 OCL request for proposal (11) contains a completely reworked specification of the OCL which defines it as a general query language that can be used everywhere in UML models to express desired properties[3]. Shortly the OMG adopted OCL supports: Query expressions, Derived values, Conditions and Business rules. It should be noted that the current OCL is seen as equivalent to SQL when it comes to querying object models.

In this work the OCL2.0 metamodel has been suitably aligned (i.e. integrated) with the current adopted MOF version (1.4). In addition the metamodel has been refined in order to better suit to our needs by subtracting the metamodel's UML-specific parts (since currently the Knowledge Base infrastructure is based on MOF) and by enriching it with an appropriate helper meta-class in order to utilize the metamodel's internal query-specific elements in a more effective way (i.e. the usage of this helper is not required since it is transparent to the user). The developed metamodel is the Query Metamodel Language (QML).

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.

[3] OCL was originally designed specifically for expressing constraints or restrict values in a UML model. However, its ability to navigate the model and form collections of objects has also lead to attempts to use it as query language (Borland's ECO framework uses OCL for querying as well as constraints, derived values, etc.).

At a technical level (implementation and theoretical approach) the knowledge access approach is uniform for both desired functionalities. The implementation provides a coherent framework for QML processing that incorporates IR functionality and the theoretical approach is based on the Extended Boolean Model. For this reason, we have provided in QML the capability to specify ranking and fuzzy Boolean operators. 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 could form the basis of user profiles. For that the existence of a MOF User Profile Metamodel (UPM) is considered as a prerequisite. UPM imports QML and uses its expressions to form user preferences.

In an open environment such as DBE, where SMEs tend to express their business characteristics and services using different models (although following common metamodels), there is a need to bridge differentiated models when querying the KB. QML is a typed language depending on the structure of the queried terms, thus the solution cannot be part of QML but rather at the analysis level. The formulated query expressions need to be semantically exploited and expanded (if needed) as to retrieve relevant information expressed in different model structures. The semantic query exploitation and expansion process, although it is not an easy task, is possible to be modelled by using ontologies and the MOF architecture incorporated with IR framework already used.

The knowledge access mechanism utilizes the functionality for processing valid query expressions based on the QML (suitably formulated by the Query Formulator API). Such functionality includes query parsing and analysis, query syntax tree construction, semantic expansion and code generation using the KB infrastructure. From a technical point of view, the KB infrastructure is based on a combination of a MOF/JMI-compliant repository and a data management system. Two alternatives were available: a) the data is queried in MOF object representation; b) the object-oriented queries are mapped to XQuery statements and executed by the XML database management system. Although, on the fly execution of QML expressions is implemented through reflection mechanisms performance issues brought us (the knowledge base development team) to use as permanent storage system and query facilities an XML database 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.

This work also aims at providing the functionality of finding relevant ontology elements and ontology paths in the process of semantically exploiting and expanding query expressions. 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 ontologies to store mappings between their elements. During the query pre-execution process relevance paths are found and the original query is expanded.

The rest of the document is organized as follows: chapter 0 presents some preliminary issues concerning the technologies used. The chapter first outlines the MOF metadata architecture adopted and then the KB infrastructure. Chapter 0 presents the architectures of knowledge access module that supports the formulation and the evaluation of involved discovery requests to the KB. The chapter first outlines the overall knowledge base architecture of DBE in order to provide the interdependencies of the knowledge access module with the rest of the system and then. Next, in chapter 4 the QML along with representative examples is presented. Chapters 5, 6 present the methodology used for the query formulation process and the construction of execution plans of the query expressions. Chapter 7 presents the semantic exploitation mechanism for the semantic query expansion. The last chapter 8 presents the conclusions.

CHAPTER II – OVERVIEW AND BACKGROUND

INTRODUCTION

This chapter presents some background issues related to the technologies used in the DBE Knowledge Base (KB) and discussed throughout the rest of this document. These technologies refer to the adopted MOF metadata architecture and the Knowledge Access Module.

While MOF is a world standard to represent complex information in different layers, the DBE Knowledge Base is the P2P distributed implementation of it, and the Knowledge Access Module a way to query, recommend, and profile the knowledge stored.

The first section describes the MOF Metadata Architecture and the second one, where in the DBE Knowledge Base the Knowledge Access Module resides, which other module are using it and for what reason. Note that the concrete architecture of the Knowledge Access Module is presented in the next chapter.

THE MOF METADATA ARCHITECTURE

MOF is a framework for describing and defining metadata, which uses a layered metadata architecture with four different abstraction layers. The basis of this architecture is the MOF meta-model (also called meta-meta-model). Figure 1 shows the DBE Knowledge base architecture illustrating the metamodels for representing the Knowledge Base information and the BML models and data. The four layers of this architecture (numbered from M0 to M3) are:

1. The (M0) information layer. It consists of the data that we wish to describe.
2. The (M1) model layer. It comprises the metadata that describe data in the information layer. Metadata is informally aggregated into models.
3. The (M2) metamodel layer. It consists of the descriptions (i.e., meta-metadata) that define the structure and semantics of metadata. Meta-metadata constructs are informally aggregated into metamodels. A metamodel is essentially an “abstract language” for describing different kinds of data.
4. The (M3) meta-metamodel layer**.** It consists of the description of the structure and semantics of meta-metadata. In other words, it is the “abstract language” for defining different kinds of metadata organizations.

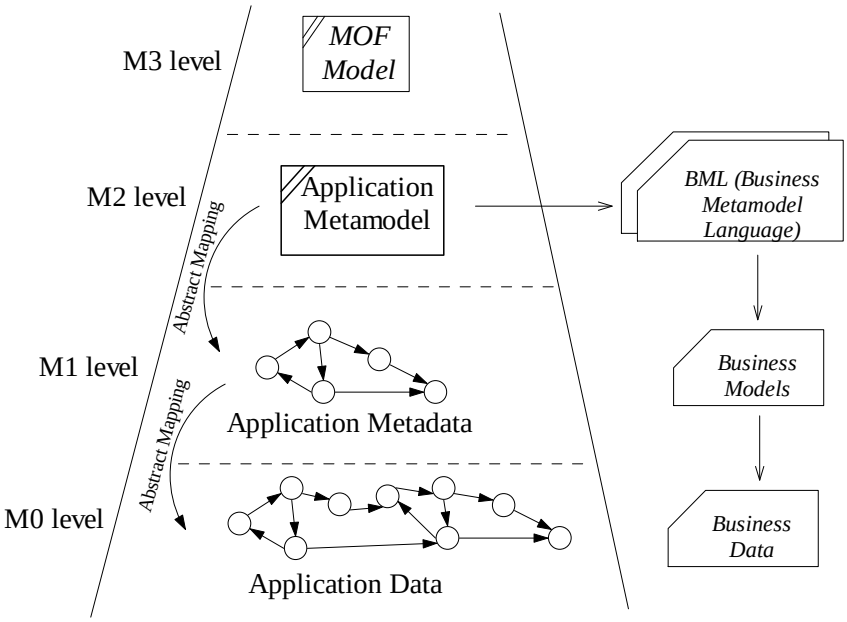

Figure 1: The DBE knowledge access process follows the four layer MOF Metadata architecture.

As described in the "DBE knowledge representation models" (3), there are various kinds of DBE Knowledge. Roughly speaking, we distinguish the following kinds:

a) Domain Specific Knowledge. It refers to common conceptualization (ontologies) that describe the semantics of specific business domains. The Ontology Definition Metamodel (ODM) is used for representing ontologies in DBE.
b) Organization Specific Knowledge. It refers to organization models, business processes, rules, etc. that describe the business model of a particular organization (SME) as a service provider. This kind of knowledge is captured with the use of the Business Modelling Language (BML).
c) Service Specific Knowledge. It refers to the knowledge about a specific value offering in DBE. Such knowledge refers to both business and technical level description of a service. For the business level description the Semantic Service Language (SSL) is used. The technical description of a service is provided by the Service Description Language (SDL), and the Business Process Execution Language (BPEL).

It is worthy to mention that the domain specific knowledge is particularly important in knowledge sharing since BML, SSL, and SDL are all referring to domain ontologies (described with ODM) for semantic enhancements with common understanding. All the mechanisms (languages) for representing these kinds of knowledge are defined in terms of the MOF (i.e. MOF metamodels) and constitute the basis for advanced semantic discovery of partners and services as well as effective development of recommendation mechanisms.

The Knowledge Base uses XMI documents for information exchange with other DBE components. The KB is built on top of the Net Beans Metadata Repository[4] (12) and the Berkeley Native XML Database[5] (13). The metadata repository offers the functionality of processing XMI documents and exporting in XMI documents pre-selected content that is stored in the repository. Appropriate middleware has been developed to provide storage, update, retrieval and load capabilities from the MDR repository to a stand-alone DBE XML server. The middleware utilizes the JMI-Java Metadata interfaces for communicating with the repository.

XMI is an implementation of the Stream-Based Model Interchange Format of the OMG Repository Architecture using XML. In particular, "*The main purpose of XMI is to enable easy interchange of metadata between modelling tools (based on the OMG-UML) and metadata repositories (OMG-MOF based) in distributed heterogeneous environments.*" (7) JMI technology enables the implementation of a dynamic, platform-independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata. Java interfaces are generated from arbitrary M2 layer metamodels, which are then used to access corresponding M1 instances or to perform necessary operations on them. The JMI 1.0 specification is the result of a Java Community Process (JCP) effort to develop a standard Java API for metadata access and management. The advantages to comply with JMI 1.0 are:

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

THE KNOWLEDGE ACCESS MODULE IN DBE

In this section we present the architectural position of the Knowledge Access Module in the DBE architecture as deployed in the integrated prototype of DBE. To do this, we firstly recall how the entire recommender component is positioned in the global DBE environment, and then, with some more details, how the recommender and its Knowledge Access Module are considered with respect to the DBE Knowledge Base.

[4] MDR is an open source metadata repository which implements the OMG's Meta Object Facility (MOF 1.4) specification and its interface is compliant with the JMI 1.0 specification

[5] The Berkeley XML database project develops open source database technology by Sleepycat Software which was recently by Oracle Co.

In short in its final form the recommender component encompasses the knowledge access module augmented with user profiling mechanism and with reasoning mechanisms that are based on the existing business and service Ontologies that capture the semantics of business models and service descriptions and on the DBE regulatory framework that determines the various contexts in which business and services operate.

Furthermore, the implementation of the DBE Knowledge Base has a p2p nature, the recommender is strongly affected by the p2p Knowledge Base framework that is adopted. In particular, the following KB related issues, are taken into account for the support of the recommendation process in the p2p environment:

- Semantic-based indexing schemes that will leverage information retrieval algorithms for the full indexing of content in order to facilitate enhanced semantic queries and effective measurement of the similarities between the queries and the underlying information
- Semantic-based knowledge distribution and mechanisms for knowledge replication that will be used in order to ensure high information availability
- Ontology mappings between domain and user (local) ontologies that will automate as much as possible information reasoning
- Storage and maintenance of critical information such as SME profiles and data

The Knowledge Management Infrastructure is used in both the Service Factory Environment (SFE) and the Execution Environment (ExE). The SFE is the environment where the actual companies built their business models and data to describe themselves and their implement their services (if available). The EXE is the environment where the services deployed and available for execution. The KB's Model Repository is used at the Service Factory Environment where existing models (BML, SSL, SDL, etc.) are stored by SMEs. In the ExE, the KB's Semantic Registry is deployed. This Semantic Registry keeps knowledge about available (published) SME services that can be searched, found, and executed in the DBE.

As already mentioned the knowledge access functionality is required for two purposes. It is needed for pure search functionality (discovery process), as well as for supporting recommendation mechanisms (recommendation process). Whereas the discovery process is used for answering the formulated query expressions based on the available metamodel and model specific information laid in the KB, the recommendation process is used for matching SME profiles (that include preferences - captured through the user profiling mechanism- on business and/or service semantics) and the underlying information. It has to be noted that at a

technical level the information filtering/retrieval approach is uniform for both desired functionalities. The two types of functionality provided by the Knowledge Access Module are exported in the form of separate services, namely the Knowledge Base/Semantic Registry services (to support discovery requests) and the Recommender Service (to support recommendations). All recommendations and discovery requests computed by the Knowledge Access Module could be considered as similarity based retrieval requests.

Regarding the Recommender Service three candidate use cases are currently foreseen[6]:

1) **Business Matching:** Based on the preferences of a specific SME *A* for possible partner SMEs, the Recommender Service may find SMEs that match the business preferences of *A*.
2) **Service Matching:** Based on the preferences of a specific SME *A* for possible services to be used on more complex services provided by *A*, the Recommender Service may find Service Descriptions that match the service preferences of *A*.
3) **Service Searching:** Based on the description of a Service *S*, the Recommender Service may find Service Descriptions that match the service description of *S*.

These use cases are applicable in both SFE and EXE environments. The preferences may refer to existing models (for helping an SME to describe itself or its services), or to existing services for service consumption (Service Manifests containing models and data). Thus, the Knowledge Access Module (through its corresponding services) is used in both environments for accessing the knowledge that is kept in each environment. Although the underlying information is conceptually different (models of business and service descriptions instead of available SME services) the Knowledge Access Module component is used in both environments since the information in both environments constitutes uniform knowledge that can be exploited in the DBE environment in the scope of the same query or user profile. For example, user preferences in the SFE may refer only to Models (of businesses and services), while in the ExE may refer to both models and data.

The Recommender Service acts as an autonomous process that manages SME preferences (either business preferences or service preferences) and matches theses preferences with available business descriptions and service descriptions and make recommendations to the DBE user in a push mode (information filtering). The user profiles may contain preferences regarding knowledge that is kept in either ExE or the SFE. Thus, the recommender service is the same in both environments. This Recommender Service is supported by the Recommender

[6] The term preferences could be seen as instances of a user (SME) profile or as instances of the QML Metamodel. In both cases it refers to preferences on business and/or service semantics.

Module, which also exploits the Knowledge Access Module for performing the appropriate matching of user preferences with the underlying knowledge (in both environments).

Figure 2, describes in more detail the Knowledge Access Module, which is a part of the recommender component, and a fundamental module of the entire DBE Knowledge Management Infrastructure as it is currently implemented in the integrated prototype of DBE. This module through its Query Interface (QI) is used by the exposed services of the DBE Knowledge Base according to the requirements of the particular architectural environment in which is used. That is, when this environment is the Service Factory, the KB Service exposes the provided functionality for accessing the DBE knowledge relying in a KB instance. On the other hand, when the environment is the Execution Environment, the SR Service exposes the desired functionality. Both services however utilize the Knowledge Access Module in order to query the underlying information. It has to be noted that different KB instances (that follow the KB infrastructure depicted in Figure 2) serve the Service Factory and the Execution environments.

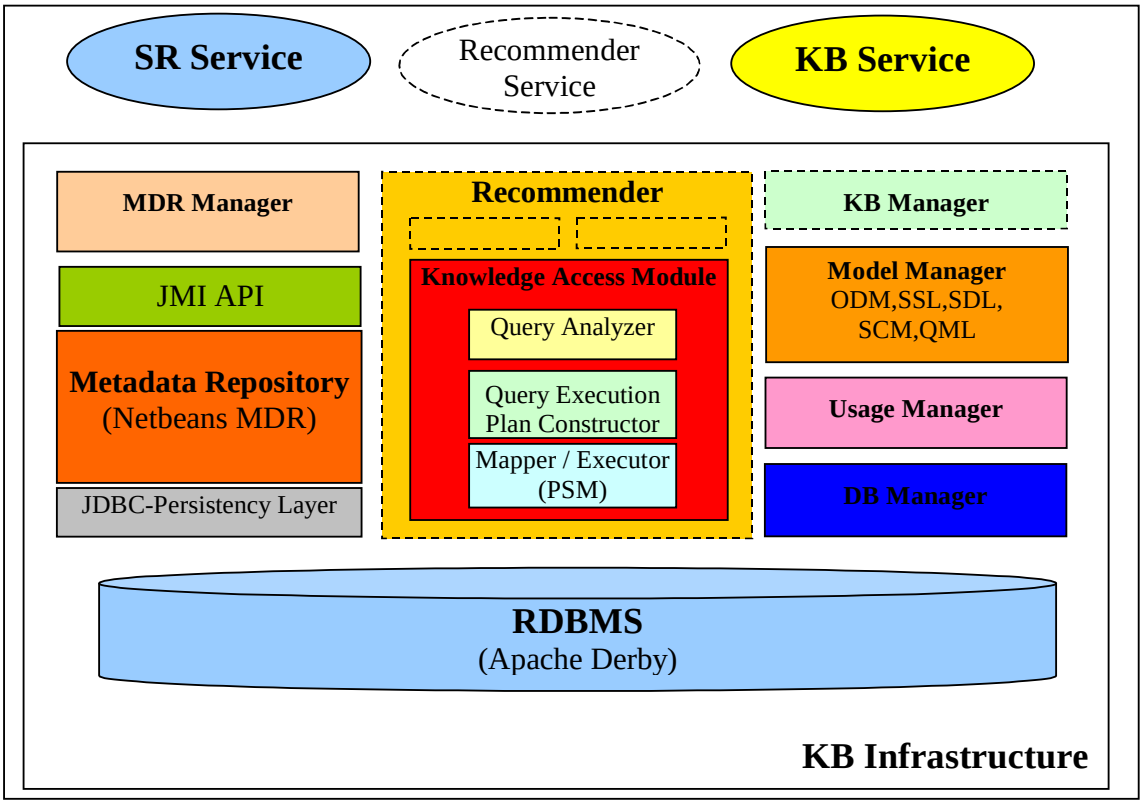

Figure 2: The Knowledge Access Module with respect to the DBE KB. The Knowledge Access module is a central component that is used by the exposed services of the DBE knowledge management infrastructure. The KB Service and SR Service are used in the Service Factory Environment or the Execution Environment respectively.

SUMMARY

In this chapter we discussed the basic frameworks this thesis is based on, i.e. the MOF Metamodel architecture and the DBE Knowledge Access Module.

The MOF Framework has a four layer architecture. On top resides the MOF Model, which is language designed to define application metamodels. The application metamodels are languages for a specific application (like business model language, ontology language, business service language, etc). Once you define an application metamodel you have a concrete way to define application models like business models (ex. Hotels, Software Companies, etc) or Ontologies for specific area of interest (as Hotels, etc). With this application models you can describe in a concrete way the actual data that differentiate each Hotel for example, called application (business, service, etc) descriptions.

Next we presented the general purpose and application of the Knowledge Access Module. The Knowledge Base Infrastructure is the implementation of the MOF architecture in DBE, along with a powerful recommender service, which exploits business and service ontologies in order to capture semantics of business and service descriptions. We described that the Knowledge Access Module is distributed in a P2P manner and thus all data is distributed as well. The information stored includes business models and data, service models and data, and ontologies. In order to provide knowledge instead of raw data we described that the recommender service is able to exploit the metadata (and meta-metadata), ontologies stored elaborated with information retrieval techniques. Special considerations where given to the P2P distributed manner of knowledge.

While in this chapter presented the general idea of the Knowledge Access Module, the following one describes in detail the Knowledge Access Module architecture; how the data are stored, and what mechanisms the recommender service exploits to provide knowledge.

CHAPTER III – KNOWLEDGE ACCESS MODULE ARCHITECTURE

INTRODUCTION

In chapter II we described both the MOF Metamodel Architecture and the how the Knowledge Model Access Module is used by other DBE components and models in general.

In order the metamodels, models, and data to be useful query mechanisms should exist. The purpose of this thesis is to provide advance query and recommendation mechanisms upon this knowledge. In this chapter we will describe the architecture of the recommender module, how it works, and which modules are implemented in order to provide these advance features, which will be explained in detail on the following chapters.

Thus, in this chapter, we present the general architecture of the recommender and knowledge access modules. Special consideration has been given on how the module will operate on the distributed p2p environment. Moreover, we present what each component is responsible for, in order to understand how all parts integrated together are able to provide advance knowledge recommendation services.

The first section describes the Knowledge Base Infrastructure. Internal part of KB Infrastructure is the Knowledge Access Module, which is presented along with a description for each sub module and how they cooperate in the second and last section of this chapter.

THE KNOWLEDGE BASE INFRASTRACTURE

The Knowledge Base Infrastructure is the implementation of the MOF Metamodel Architecture in DBE. It is decentralized in a P2P manner and provides stable storage for metamodels, models, and data.

Figure 3 shows the architecture of the Knowledge Base (KB) infrastructure. The KB infrastructure provides a common persistence and knowledge management layer in the digital ecosystem. It offers a set of APIs and tools for accessing the DBE Knowledge (M1 Model information and M0 data). The components that comprise the basic KB infrastructure are described in deliverable D14.3 (14).

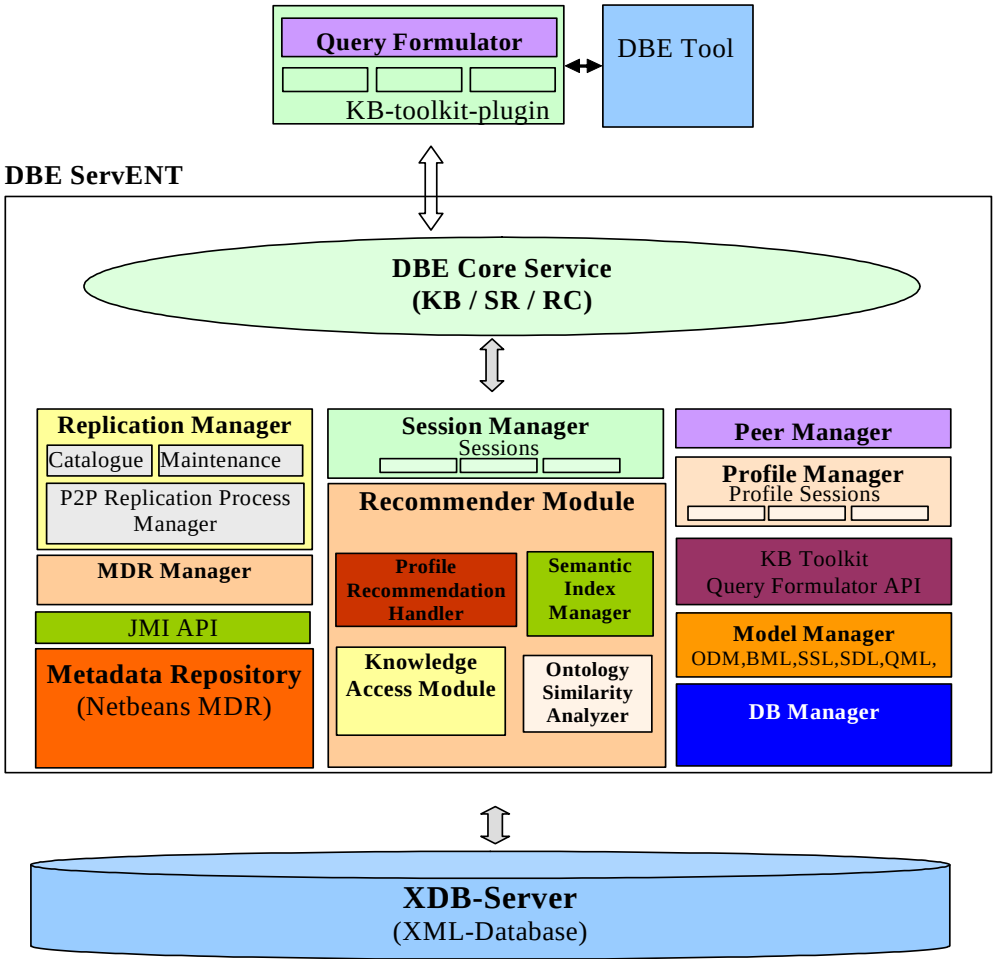

Figure 3: The architecture of the Knowledge Base Infrastructure.

THE RECOMMENDER MODULE ARCHITECTURE

Recommender Module implements the components related to the evaluation of candidate services and candidate business partners in the process of discovering or composing services and establishing partnerships respectively. The Recommender Module is responsible for matching preferences with business descriptions and service descriptions. The major assumption behind the design and implementation of the Recommendation mechanisms is the existence of powerful business and service Ontologies that capture the semantics of business models and service descriptions. These Ontologies are used to define the corresponding preferences for businesses and services. The recommender exploits Profile Manager to store preferences and all recommendation mechanisms operate on top of the native XML database to implement the necessary matching functions between preferences and business/service descriptions. Figure 4 shows the architecture of the Recommender Module in detail.

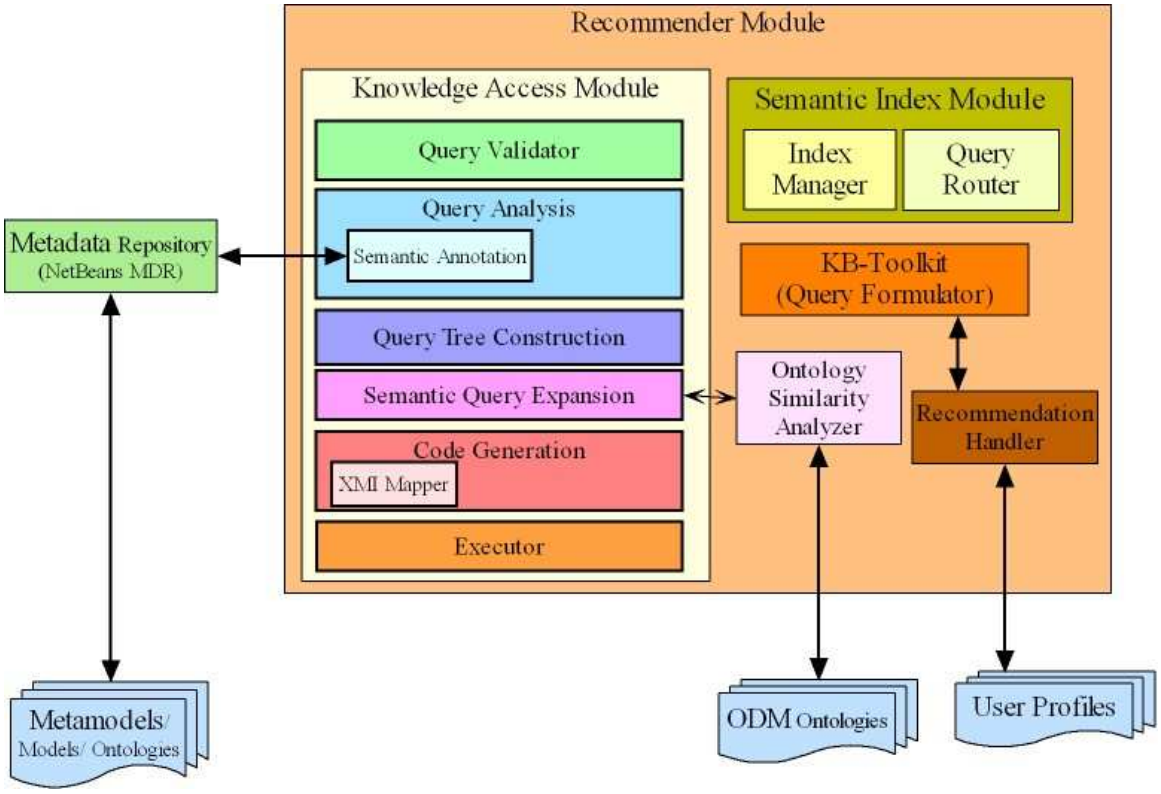

Figure 4: The architecture of the Recommender Module.

Next paragraphs describe what each sub module is responsible for. At the end of section we put all these modules together to understand how they operate in an integrated way, in order to provide advance knowledge access.

THE QUERY METAMODEL LANGUAGE

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. Queries can be formulated by using the Query Formulator module which automatically constructs fuzzy queries in terms of QML that use senses from metamodels, models, and ontologies. The query formulator module uses the JMI Reflective API from the MDR Manager in order to understand the context of the query terms.

THE QUERY ANALUSIS MODULE

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

THE QUERY TREE CONSTRUCTOR

The Query Tree Constructor is responsible for creating an execution tree of the query. This execution tree contains information regarding semantic annotation and fuzzy weights following the IR fuzzy model defined. This information is retrieved from the metamodels, models, and ontologies by using the JMI reflective API.

THE SEMANTIC QUERY EXPANDER

The Semantic Query Expander expands the query tree with new query branches which are semantically equivalent or semantically related to the original query. The information required for the expansion are provided from the semantic annotation process and the from ontology elements similarities provided by the Ontology Similarity Analyzer.

THE ONTLOGY SIMILARITY ANALYER

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

THE CODE GENERATOR

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

The XMI 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 QUERY EXECUTOR

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

THE QUERY FORMULATOR

The Query Formulator module is responsible for creating valid QML queries through an API. This module wraps the complexity of creating fuzzy QML expressions and using JMI. The Query Formulator comes into two versions; a simple and an advanced. In the advanced version reusable templates can be created. The Query Formulator is used by front-end tools, by the recommender to rewrite user profiles into QML queries, and by SMEs services that need to have query capabilities.

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

THE PROFILE MANAGER

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

THE OVERALL PROCESS

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

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

exploitation of the semantic overlay network and the available infrastructure for searching and querying with the use of ontologies and the MDA architecture over the P2P physical network.

The overall process consists of the following steps. First of all a query is formulated into QML with the Query Formulator. Both, a graphical user interface exists and a keyword extractions and formulation mechanism to formulate QML queries. At next the Recommender parses the query (it is forwarded to other peers as well) and analysis it. In this step each part of the QML query is semantically annotated with a term of a metamodel. What follows is query tree construction with the query branches to be evaluated and merged. Each branch at the next step is expanded by using semantics of the models. In order to do this information from ontologies is used. Now, the query pre-processing has finished and XQuery code is generated, which will be executed at the XML database. Finally, the queries are executed and the results are merged in a fuzzy model manner. The results are sent back to the originator along with weights which represent how well the result matched the conditions set. Each peer sends results asynchronously to the responder and no merging occurs in the Recommender module.

SUMMARY

In this chapter we focused on what each sub module of the Recommender is responsible for and how all of the work together in order to provide an advanced knowledge access service.

A language (QML) is developed to describe queries by using the MOF construct. An API (along with a GUI) and Keyword parser is provided to formulate queries into QML. Each QML expression is annotated using metadata information (semantic annotation process), query evaluation trees are constructed which are used to semantically expand queries with ontology terms and constructs from other models. The query tree is transformed into XQuery expressions and executed on the XML database. Finally, the results are merged using the evaluation tree.

On the next four chapters each of these sub modules is discussed thoroughly. The next chapter presents formally the Query Metamodel Language (QML) along with examples.

CHAPTER IV – THE QUERY METAMODEL LANGUAGE

INTRODUCTION

This chapter discusses in detail the terms of the Query Metamodel Language (QML) and its placement in the adopted MOF architecture. Along with the formal presentation of the language indicative examples are given in order to explore the capabilities and functionalities of QML. Finally in this chapter we discuss the query analysis mechanisms and an evaluation process developed apart from that inside the DBE Knowledge Base.

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

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

QML AS A MOF METAMODEL

SQL is a language commonly known and understood. We will try to describe where MOF for layer architecture stands and what we want QML to query with examples coming from relational databases domain in order the purpose and the needs of QML to be clarified.

A relational schema could be a MOF model (M1 layer) equivalent whereas the data is the MOF M0 layer. The relation schema obeys the relational model; senses like table, index, varchar, int exist in the relational model level and may vary between RDBMS implementations (mainly on supported functions). The relational model is the M2 layer of the MOF architecture. The actual MOF metamodel could be used to define the relational model constructs and their semantics (M3 layer). With SQL we can query M0 data using M1 constructs, but we can not query M1 schema information using M2 constructs (e.g. to retrieve the table names which have a column named "City": SELECT table.name FROM table, column WHERE table.id = column.table_id AND

column.name = 'City'). Querying relational schema is something that nobody needs up to now, while in the MOF environment this is something common (e.g. find the models which have an element named "Hotel"). Thus, QML should be able to query both M0, M1. Imagine now we could query the RDBMS relational model and if a function exists to use or if not to do something else! This would allow us to write cross platform SQL queries! In the same sense QML can be used to write queries with M3 constructs for M2 metamodels. Thus, with QML we can search for all the following examples:

- Hotel.City = "Athens" (M1 constructs searching for M0 data)
- BusinessEntity.Name = "Hotel" (M2 constructs searching for M1 models)
- Class.Name = "BusinessEntity" (M3 constructs searching for M2 metamodels)

As it should be clear by now QML should be able to search in all 3 layers. To go from one layer to the next we instantiate the meta-metamodel (MOF), metamodel (Business Language Metamodel), and model (Hotel Model) in the same sense we instantiate Objects from Classes in programming languages. Thus, QML query expressions (like SQL queries) are instances of the QML model. It might seem quite strange how "one language fits all" but this is possible (as it in OCL) because the main QML class is both an instance and a sub-class of the main MOF element called ModelElement. On the next sections this will be seen in more detail. In the MOF architecture QML stands in both M3 layer (as an extension of MOF) and in M2 layer (as a Metamodel).

OCL AND QML

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

UML Meta-classes (referred from the OCL2.0 Abstract Syntax metamodel)	**MOF1.4 Meta-classes (referred from the QML metamodel)**
ModelElement	ModelElement
Classifier	Classifier
DataType	DataType

PrimitiveType	PrimitiveType
Attribute	Attribute
AssociationEnd	AssociationEnd
Operation	Operation
EnumerationLiteral	EnumerationType (with multivalued attribute labels)
AssociationClass	*MOF does not support it and therefore QML does not include it.*
Messages	*MOF does not support messages, therefore QML does not provide messages support.*

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

The Query Metamodel Language (QML) is also defined as a M2 MOF metamodel. QML allows writing query expressions (M1 QML models) using the information provided by the M2 Knowledge Base metamodels in order to obtain M1 Knowledge Base models. To support this, QML elements are also directly related to MOF elements (through references and specializations). It should be noted that the granularity of the QML query expressiveness is not limited to only one metamodel (i.e. it is allowed to combine semantic information from more than one metamodels).

In addition QML also allows writing constraint expressions for the M3 layer, M2 layer and the M1 layer Knowledge Base metamodels and models respectively. Since QML is intended to be used for MOF models only, a MOF version of OCL has been produced in order to allow writing constraints for MOF models too. This essentially enriches the MOF language with a constraint language, which is compatible with MOF (and UML which can also be defined using MOF).

Although, OCL is quite powerful it cannot be used directly as a query language for a number of reasons. The OCL constraints do not have a mechanism for defining result types. Moreover, an OCL constraint refers to one and only class or object (the so called *context*), which is a drawback for complex queries. For these reasons we introduced the Query Context Declaration meta-class. This meta-class's semantics are somewhat equivalent to the SQL *select* statement. For the rest of its structure QML utilises the concrete and abstract syntax of OCL. By extending the OCL core QML supports highly expressive queries as it inherits a set of valuable characteristics from OCL; it is an object-oriented, strongly typed language able to navigate through not only metamodels (M2 layer) but also on models (M1 layer) – as explained in detail later on.

Therefore, a query expression can be semantically analyzed and be able to query both models and data.

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

In the following sections the QML abstract syntax is described in detail and a number of examples of simple and more complex queries are given as to demonstrate the capabilities of QML.

THE QML PACKAGE STRUCTURE

Figure 5 presents the QML package structure. The core QML metamodel consists of two packages; the Expressions package and the Types package, where the QML expressions and types are defined respectively. QML also contains the Context Declarations Package, which makes use of the Core QML package in order to express queries and constraints separate from the corps of a MOF model. Furthermore, the 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 (using the *Constraint* model element of defined in MOF).

The following QML query is used as an example to demonstrate these main principals (we want to find the Hilton hotels in Athens which have a room cheaper than 100 Euros):

```
1: Context A: HotelModel#Hotel instanceQuery
2: A.HotelName = “Hilton” and
3: A.HotelAddress.City = “Athens”
4: A.Rooms->exists(Price < 100)
5: out Hotels := A
```

Lines 1 and 5 define the context and the output (package Context Declarations); lines 2 to 4 define an expression (package Core). The types of each element used are defined in the Types package and might be either primitive (as “Hilton” is String) or come from a MOF model or metamodel (like HotelName is a BusinessAttribute (defined in BML metamodel of M2 level) of the HotelModel (defined at M1 level)). In more detail the example is explained on Context Declaration Package section.

We will use the same example as we go through the packages and elements presented. When the formal presentation ends we will present some examples and how they are formally represented on QML on both M1 and M2 layers.

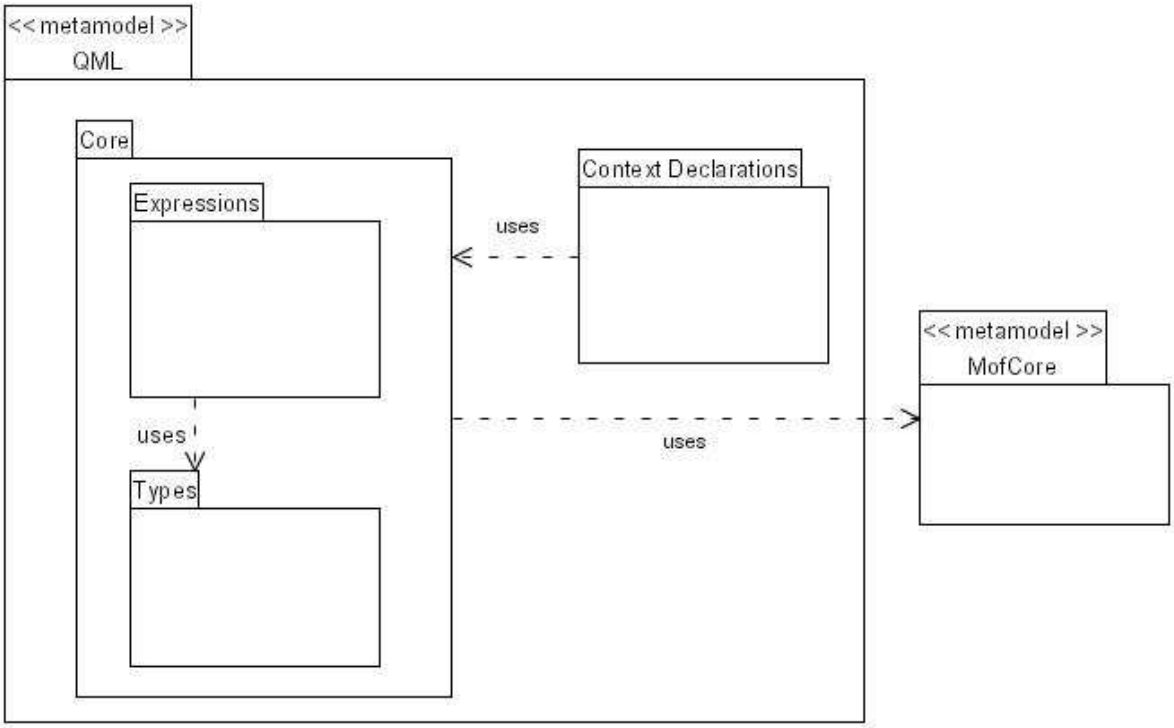

Figure 5: The QML package structure. The core QML metamodel consists of two packages; the Expressions package and the Types package, where the QML expressions and types are defined respectively. QML also consists of a Context Declarations Package which makes use of the Core QML package in order to express queries and constraints separate from the corps of a model.

THE MOF ELEMENTS

Some of the MOF elements will be used by QML. These are the *ModelElement,* the *Classifier*, the *AssociationEnd*, the *Operation*, and the *Attribute*.

ModelElement is a general MOF construct to represent all model elements. Instances of this class have a name, attributes with a type. The general MOF class to represent types is the *Classifier* (the instance of it is a Class or Primitive Type). Instances of MOF *Attribute* are what its name stands for: an *ModelElement* attribute. Instance of *ModelElements* can be associated with other instances. Each association has to ends (the two *ModelElements*) these are called *AssociationEnd* and can have a multiplicity argument (0 to many, many to many, etc). Last MOF element used here is the *Operation*; instances of this class are the actual operations (like "=", "<", etc).

THE EXPRESSIONS PACKAGE

Figure 6 shows the core part of the Expressions package. The basic structure in the package consists of the classes *OclExpression*, *PropertyCallExp* and *VariableExp*. An *OclExpression* always has a type, which is usually not explicitly modelled, but derived. Each *PropertyCallExp* has exactly one source, identified by an *OclExpression*. In order to be able to express constraints incorporated in a model, on a model's specific element, MOF structure forces us to specialize MOF's ModelElement class by OclExpression (since a MOF Constraint is also a specialization of the ModelElement). In this section we use the term 'property', which is a generalization of *Feature*, *AssociationEnd* and predefined iterating OCL collection operations.

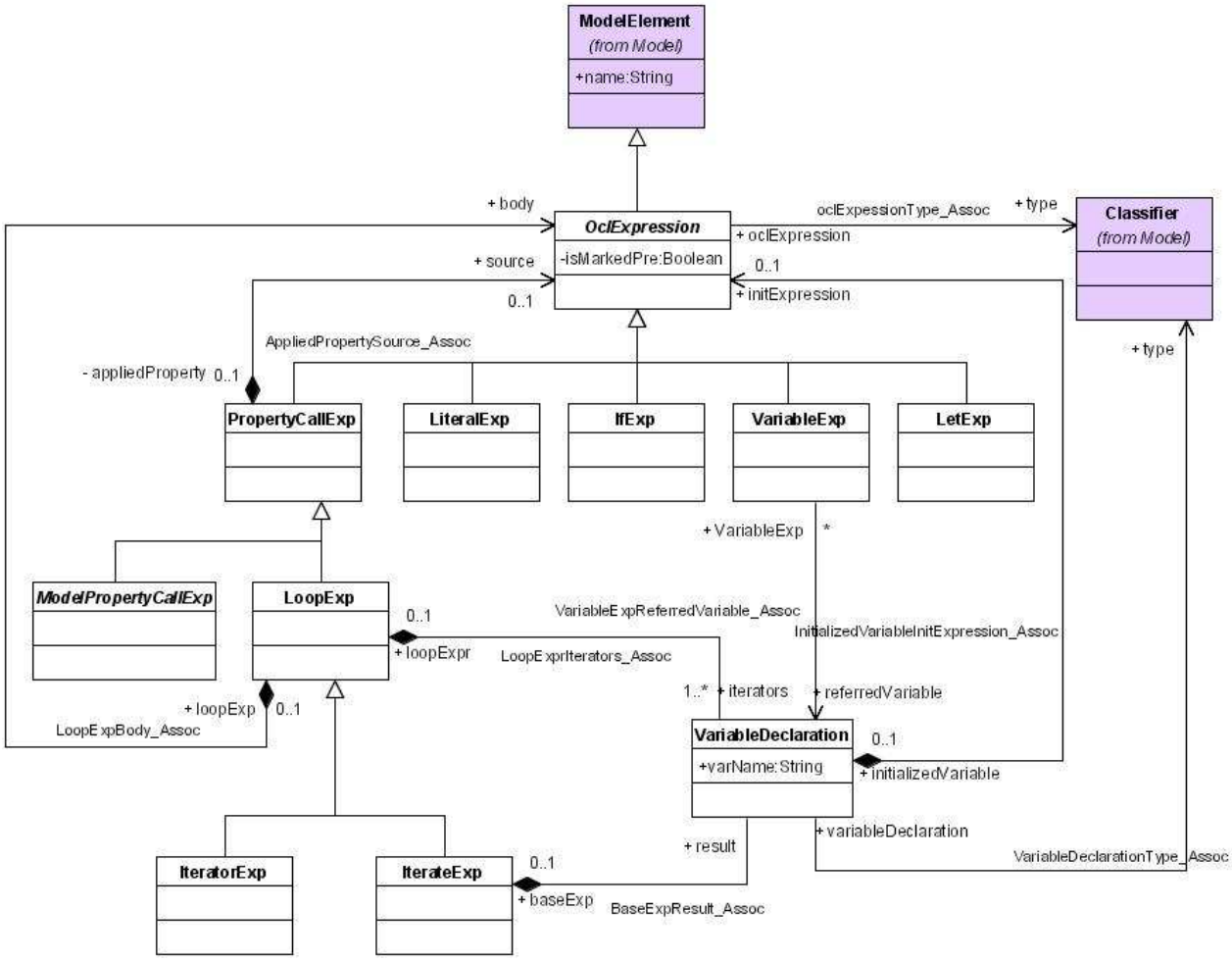

Figure 6: The basic structure of the core Query Metamodel for Expressions. The Query Metamodel Language (QML) is based on OCL 2.0 properly transformed to conform to MOF 1.4 and to effectively support queries in our DBE context. The basic structure in the package consists of the classes OclExpression, PropertyCallExp and VariableExp. An OclExpression always has a type, which is usually not explicitly modelled, but derived. Each PropertyCallExp has exactly one source, identified by an OclExpression. We use the term 'property', which is a generalization of Feature, AssociationEnd and

predefined iterating OCL collection operations. From the metamodel it can be deduced that an OCL expression always starts with a variable or literal, on which a property is recursively applied.

From the metamodel it can be deduced that a QML expression always starts with a variable or literal, on which a property is recursively applied.

OclExpression

An *OclExpression* is an expression that can be evaluated in a given environment. *OclExpression* is the abstract super-class of all other expressions in the metamodel. It is the top-level element of the QML Expressions package. Every *OclExpression* has a type that can be statically determined by analyzing the expression and its context. Evaluation of an expression results in a value. Expressions with Boolean result can be used as constraints and queries e.g. to specify an invariant of a class. Expressions of any type can be used to specify initial attribute values, target sets, etc.

The environment of an *OclExpression* defines what model elements are visible and can be referred to in an expression. At the topmost level the environment will be defined by the *ModelElement* to which the QML expression is attached, for example by a *Classifier* if the QML expression is used as an invariant. On a lower level, each iterator expression can also introduce one or more iterator variables into the environment. The environment is not modelled as a separate meta-class, because it can be completely derived using derivation rules. The complete derivation rules can be found in chapter 9 (“Concrete Syntax”) on OCL 2.0 specification.

In the example used before, lines 2-4 is a (IsA) *OCLExpression*. Moreover, each line is a (IsA) different *OCLExpression*.

PropertyCallExp

A *PropertyCallExp* is an expression that refers to a property (operation, attribute, association end, predefined iterator for collections). Its result value is the evaluation of the corresponding property. This is an abstract meta-class. The result value of the *source* expression is the instance that performs the property call.

In line 2 of the example used before (for convenience `A.HotelName = “Hilton”`) is a *PropertyCallExp*. But also “.HotelName” and “=” are *PropertyCallExp*. “A” and “Hilton” are not and we will see next what are they.

LoopExp

A *LoopExp* is an expression that represents a loop construct over a collection. It has an iterator variable that represents the elements of the collection during iteration. The body expression is evaluated for each element in the collection. The result of a loop expression depends on the specific kind and its name.

In the "HotelModel" model definition of the example, the "Rooms" "BusinessEntity" is associated with "Hotel" ("A" stands for "Hotel") "BusinessEntity" with 1 to many relationship. For that reason a "Hotel" may have many "Rooms". In order to loop between the Rooms we use the *LoopExp.*

IterateExp

An *IterateExp* is an expression, which evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. An iterate expression evaluates its *body* expression for each element of its *source* collection. The evaluated value of the *body* expression in each iteration step becomes the new value for the *result* variable for the succeeding iteration-step. The result can be of any type and is defined by the *result* association. The *IterateExp* is the most fundamental collection expression defined in the QML Expressions package.

IteratorExp

An *IteratorExp* is an expression, which evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. The type of the iterator expression depends on the name of the expression, and sometimes on the type of the associated *source* expression. The *IteratorExp* represents all other predefined collection operations that use an iterator. This includes select, collect, reject, forAll, exists, etc. The QML Standard Library defines a number of predefined iterator expressions. Their semantics is defined in terms of the iterate expression. Refer to the official adopted OCL 2.0 specification ("Mapping rules for predefined iterator expressions") for a complete reference on predefined iterator expressions.

In the example used before the line "A. Rooms->exists(Price < 100)", "exists" is an *IteratorExp,* which iterates between all "Rooms" of "A".

VariableExp

A *VariableExp* is an expression, which consists of a reference to a variable. References to the variables *self* and *result* or to variables defined by Let expressions are examples of such variable expressions.

In the example used before, a *VariableExp* is *"A".*

VariableDeclaration

A *VariableDeclaration* declares a variable name and binds it to a type. The variable can be used in expressions where the variable is in scope. This meta-class represents amongst others the variables *self* and *result* and the variables defined using the Let expression.

In the example used before, "A: HotelModel#Hotel" is the variable declaration (*VariableDeclaration* element).

ModelPropertyCallExp

A *ModelPropertyCall* expression is an expression that refers to a property that is defined for a *Classifier* in the MOF model to which this expression is attached. Its result value is the evaluation of the corresponding property. A *ModelPropertyCallExp* generalizes all property calls that refer to *Features* or *AssociationEnds* in the MOF metamodel. Figure 7 shows the three different subtypes of *ModelPropertyCallExp*, each of which is associated with its own type of *ModelElement*.

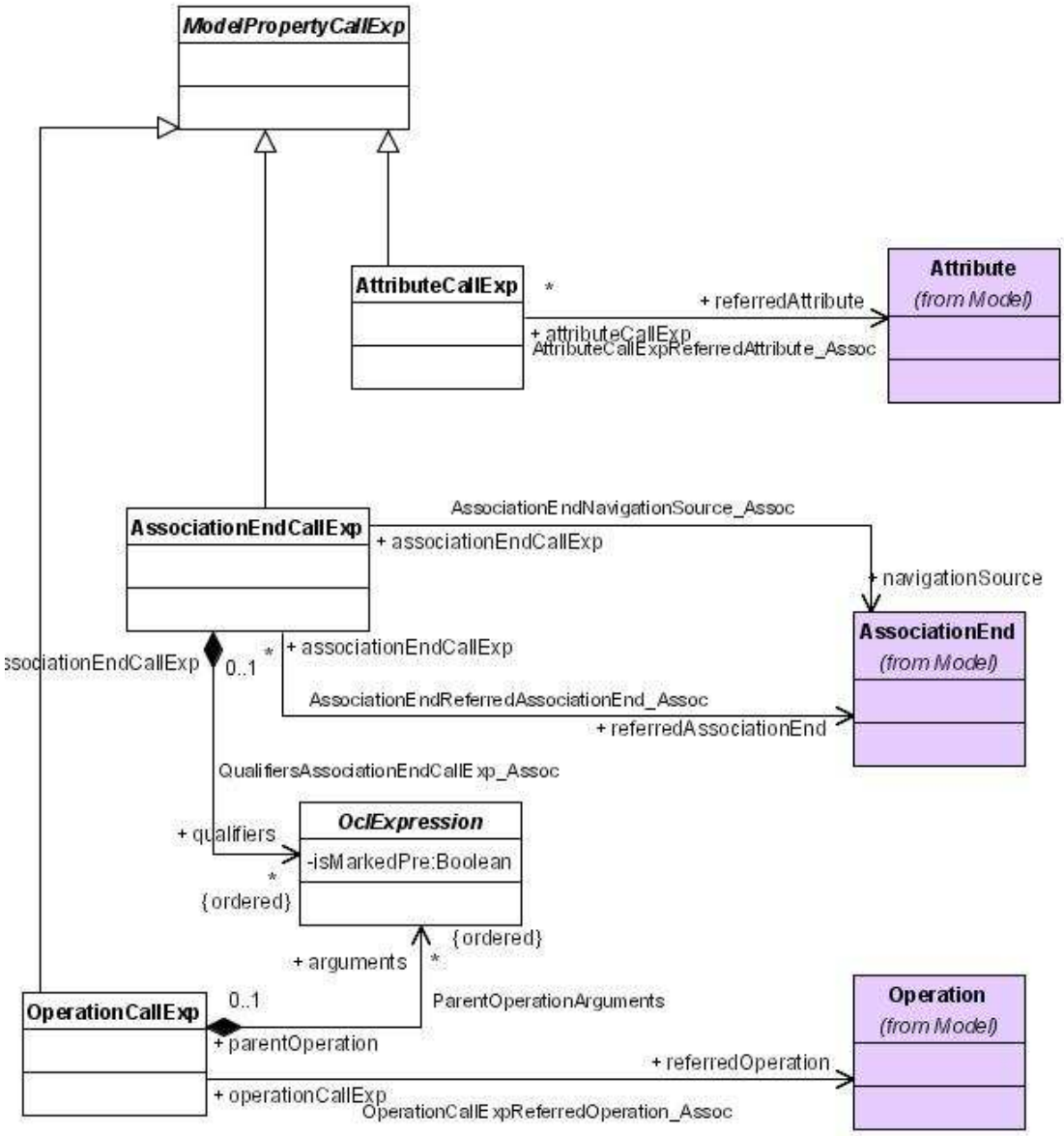

Figure 7: The ModelPropertyCallExp in the Expressions package. A ModelPropertyCallExp expression is an expression that refers to a property that is defined for a Classifier in the MOF model to which this expression is attached. Its result value is the evaluation of the corresponding property. There are three different subtypes of ModelPropertyCallExp AttributeCallExp, AssociationEndCallExp and OperationCallExp, each of which is associated with its own type of MOF's ModelElement.

AssociationEndCallExp

An *AssociationEndCallExp* is a reference to an *AssociationEnd* defined in a MOF model. It is used to determine objects linked to a target object by an association. The expression refers to these target objects by the role name of the association end connected to the target class.

In the example used before in the line "A.HotelAddress.City = 'Athens'" the "HotelAddress" is the name of a MOF *AssociationEnd* (i.e. how we go from the "Hotel" entity to the "Address" entity)

Note that the two entities defined in "HotelModel" are not "Hotel" and "HotelAddress" but rather "Hotel" and "Address". From "Address" to "Hotel" another *AssociationEnd* exist named for example "AddressHotel". The *source* OCLExpression of this *AssociationEndCallExp* is the *VariableExp* "A".

AttributeCallExp

An *AttributeCallExpression* is a reference to an *Attribute* of a *Classifier* defined in a MOF model. It evaluates to the value of the attribute.

In the same sense "HotelAddress" in the previous paragraph was the *AssociationEnd* in this paragraph "HotelName" is the *Attribute* of the entity "Hotel". Thus it depends how it is defined in the model in order to use the appropriate expression.

OperationCallExp

An *OperationCallExp* refers to an *Operation* defined in a *Classifier*. The expression may contain a list of argument expressions if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters.

In the example used before "=", "<", "and" are all MOF *Operation* instances and, thus, we use an *OperationCallExp.* Note here that each *OCLExpression* has a type (MOF *Classifier*). Thus, *OperationCallExp* should also have. The type is the result of the operation, i.e. in the example "A.HotelName = City" the result of "=" is Boolean.

IfExp

IfExp is shown in Figure 8. An *IfExp* results in one of two alternative expressions depending on the evaluated value of a *condition*. Note that both the *then Expression* and the *else Expression* are mandatory. The reason behind this is that an if expression should always result in a value, which cannot be guaranteed if the else part is left out.

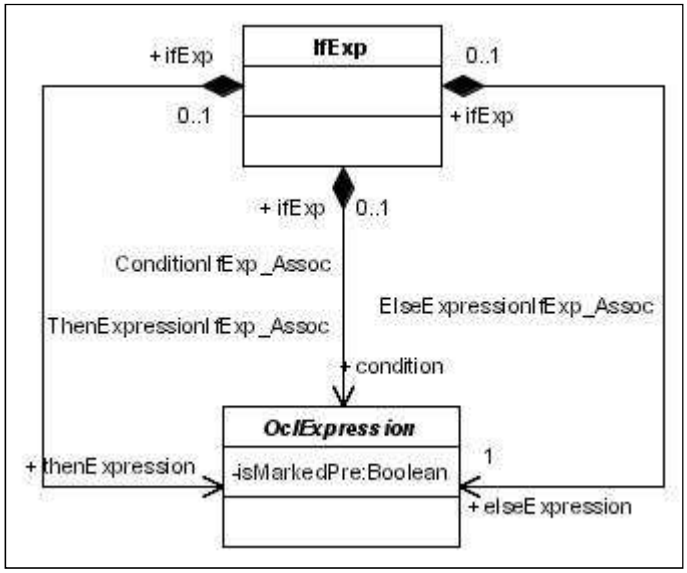

Figure 8: Definition of If expression. An IfExp results in one of two alternative expressions depending on the evaluated value of a condition. Note that both the thenExpression and the elseExpression are mandatory. The reason behind this is that an if expression should always result in a value, which cannot be guaranteed if the else part is left out.

LetExp

A *LetExp* is a special expression that defines a new variable with an initial value. A variable defined by a *LetExp* cannot change its value. The value is always the evaluated value of the initial expression. The variable is visible in the *in* expression. The *LetExp* is shown in Figure 9.

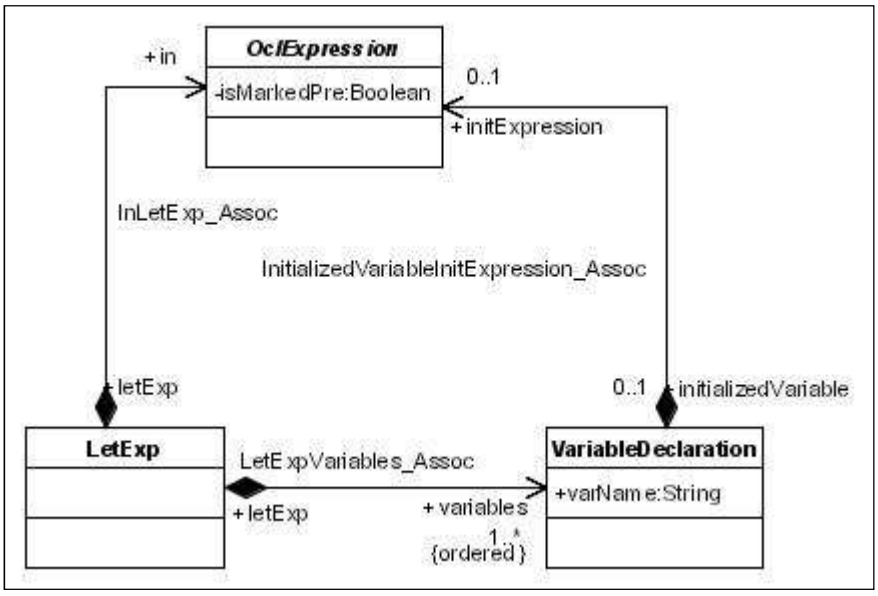

Figure 9: Definition of Let expression. A LetExp is a special expression that defines a new variable with an initial value. A variable defined by a LetExp cannot change its value. The value is always the evaluated value of the initial expression. The variable is visible in the in expression.

The example used so for does no have a *LetExp*. The following example demonstrates the use of it:

```
Let B := A.HotelName In
B = ‘Hilton’ and A.HotelAddress.City = ‘Athens’
```

“B := A.HotelName” is the *VariableDeclaration* and what follows “In” (i.e. “B = ‘Hilton’ and A.HotelAddress.City = ‘Athens’”) is the *OCLExpression* where the scope of *VariableExp* “B” is valid. The *VariableDeclaration* “B := A.HotelName” has two parts the *varName* “B” and the *initExpression* “A.HotelName”.

LiteralExp

A *LiteralExp* is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. This includes things like the integer 1 or literal strings like ’this is a LiteralExp’. They are shown in figure 10.

Examples of *LiteralExp* include “Athens”, “Hilton” and “100”.

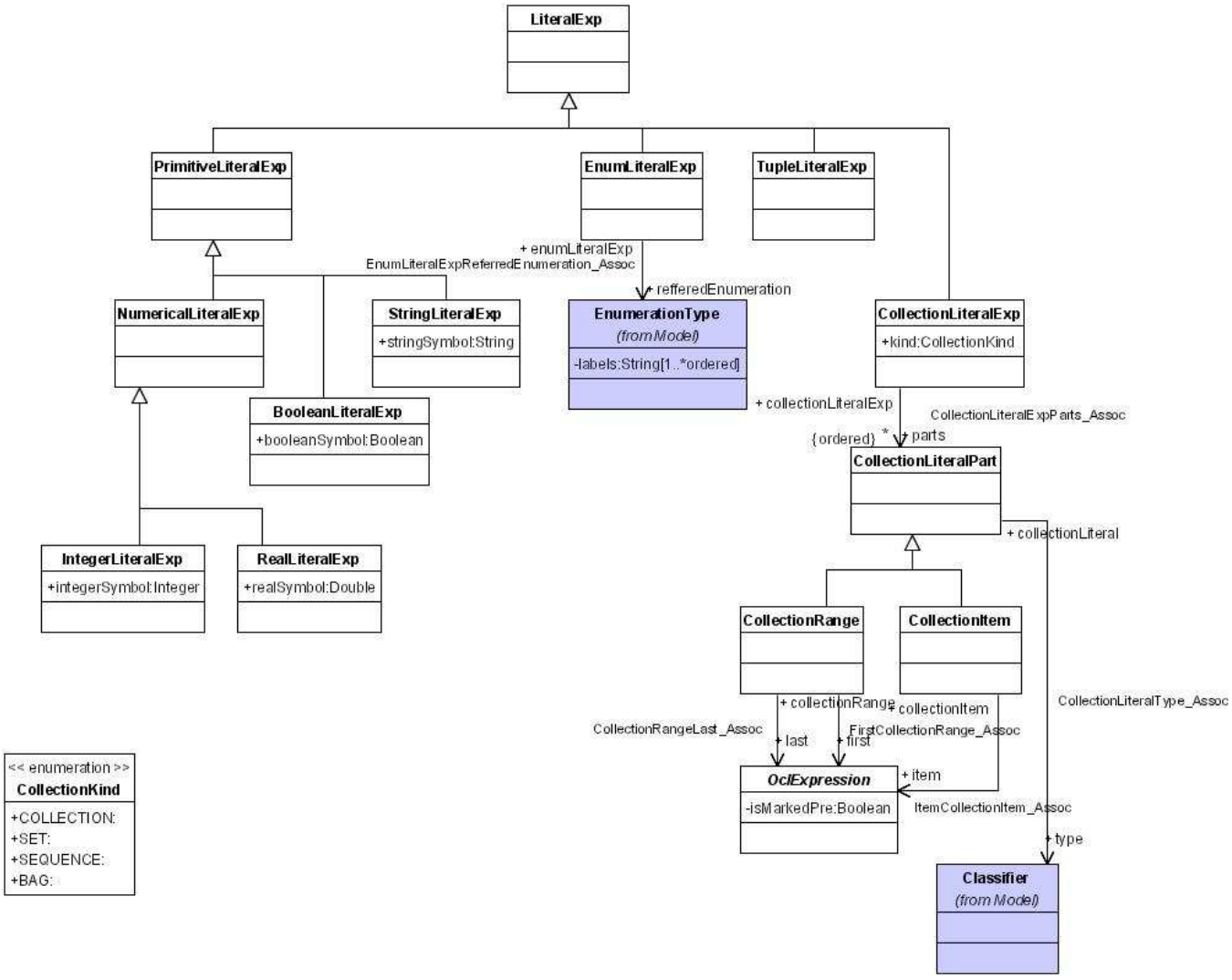

Figure 10: Definition of literal expressions. A LiteralExp is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. This includes things like the integer 1 or literal strings like 'this is a LiteralExp'.

THE TYPES PACKAGE

QML is a typed language. Each expression has a type which is either explicitly declared or can be statically derived. Evaluation of the expression yields a value of this type. A metamodel for QML types is shown in this section. Note that instances of the classes in the metamodel are the types themselves (e.g. Integer) not instances of the domain they represent (e.g. -15, 0, 2, 3).

The model depicted in Figure 11 shows the QML types. Note that the QML Types package is the same with the OCL Types package with the difference that UML Model Elements used in OCL (like UML Classifier) are aligned to the corresponding MOF elements. The basic type is the MOF *Classifier*, which includes all subtypes of Classifier from the MOF infrastructure. QML directly

specializes MOF types, as MOF *Classifier* and *DataType,* since it has to refer to OCL expression's types in a generic way, i.e. a type of an OCL Expression could be either a MOF Class or an OCL Tuple.

In the model the CollectionType and its subclasses as well as the TupleType are considered as special data types. One can never instantiate all collection types, because there is an infinite number, especially when nested collections are taken in account. Users will never instantiate these types explicitly. Conceptually all these types do exist, but such a type should be (lazily) instantiated by a tool, whenever it is needed in an expression.

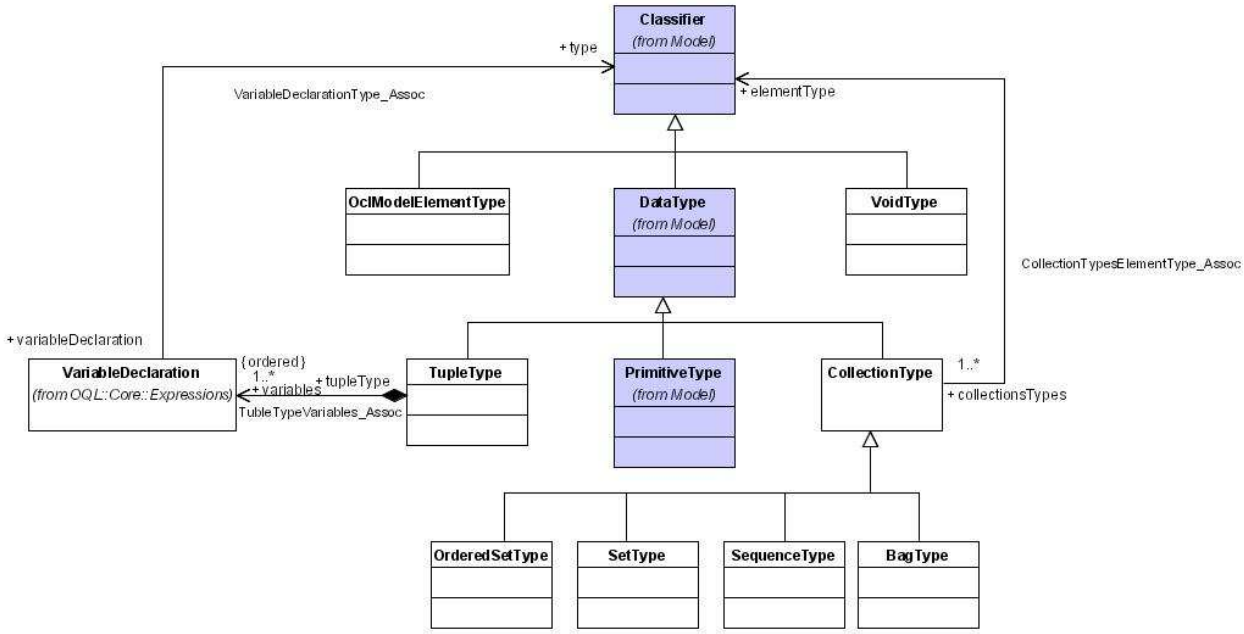

Figure 11: The core metamodel for QML Types. The basic type is the MOF Classifier, which includes all subtypes of Classifier from the MOF infrastructure. In the model the CollectionType and its subclasses as well as the TupleType are considered as special data types. Users will never instantiate these types explicitly. Conceptually all these types do exist, but such a type should be (lazily) instantiated by a tool, whenever it is needed in an expression.

OclModelElementType

OclModelElementType represents the types of elements that are *ModelElements* in the UML metamodel. It is used to be able to refer to states and classifiers in e.g. *oclInState*(...) and *oclIsKindOf*(...)

CollectionType

CollectionType describes a list of elements of a particular given type. *CollectionType* is an abstract class. Its concrete subclasses are *SetType*, *SequenceType* and *BagType* types. Part of every collection type is the declaration of the type of its elements, i.e. a collection type is *parameterized* with an element type. In the metamodel, this is shown as an association from *CollectionType* to *Classifier*. Note that there is no restriction on the element type of a collection type. This means in particular that a collection type may be parameterized with other collection types allowing collections to be nested arbitrarily deep.

BagType

BagType is a collection type, which describes a multiset of elements where each element may occur multiple times in the bag. The elements are unordered. Part of a *BagType* is the declaration of the type of its elements.

OrderedSetType

OrderedSetType is a collection type which describes a set of elements where each distinct element occurs only once in the set. The elements are ordered by their position in the sequence. Part of an *OrderedSetType* is the declaration of the type of its elements.

SequenceType

SequenceType is a collection type, which describes a list of elements where each element may occur multiple times in the sequence. The elements are ordered by their position in the sequence. Part of a *SequenceType* is the declaration of the type of its elements.

SetType

SetType is a collection type which describes a set of elements where each distinct element occurs only once in the set. The elements are not ordered. Part of a *SetType* is the declaration of the type of its elements.

TupleType

TupleType (informally known as record type or struct) combines different types into a single aggregate type. The parts of a *TupleType* are described by its attributes, each having a name and a type. There is no restriction on the kind of types that can be used as part of a tuple. In particular, a *TupleType* may contain other tuple types and collection types. Each attribute of a

TupleType represents a single feature of a *TupleType*. Each part is to uniquely identified by its name.

VoidType

VoidType represents a type that conforms to all types. The only instance of *VoidType* is *OclVoid*, which is further defined in the standard library. Furthermore *OclVoid* has exactly one instance called *OclUndefined*.

As example of *CollectionType* in expression "A. Rooms->exists(Price <100)"is the result of the *AssociationEnd "*Rooms", or the result of the "select" *IteratorExp* in the expression "A. Rooms -> select(Beds = 2)-> exist(Price < 100)". The last expression first selects all the rooms with 2 beds and then searches on them for one with price less that 100.

THE CONTEXT DECLARATIONS PACKAGE

Context declarations are not needed in OCL, because OCL constraints meant to be directly attached to the model elements they refer to. Nevertheless, a concrete syntax of them is given in the OCL2.0 specification [12] 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 where an OclExpression refers to, the kind of it (invariant, operation, definition and attribute) and any other specific information needed for each kind. For example if we want to create a query we need to take *QueryContextDeclaration* or if want to create a constraint on a model element we create an *InvariantContextDecl*. The adopted OCL2.0 specification explains in detail the concrete syntax of Context Declarations (Section 12.13). To express the idea of query as a constraint on a model element resulting a set of values with a specific type we have added the QueryContextDecl meta-class. Figure 12 shows the context declarations package with the QueryContextDecl meta-class.

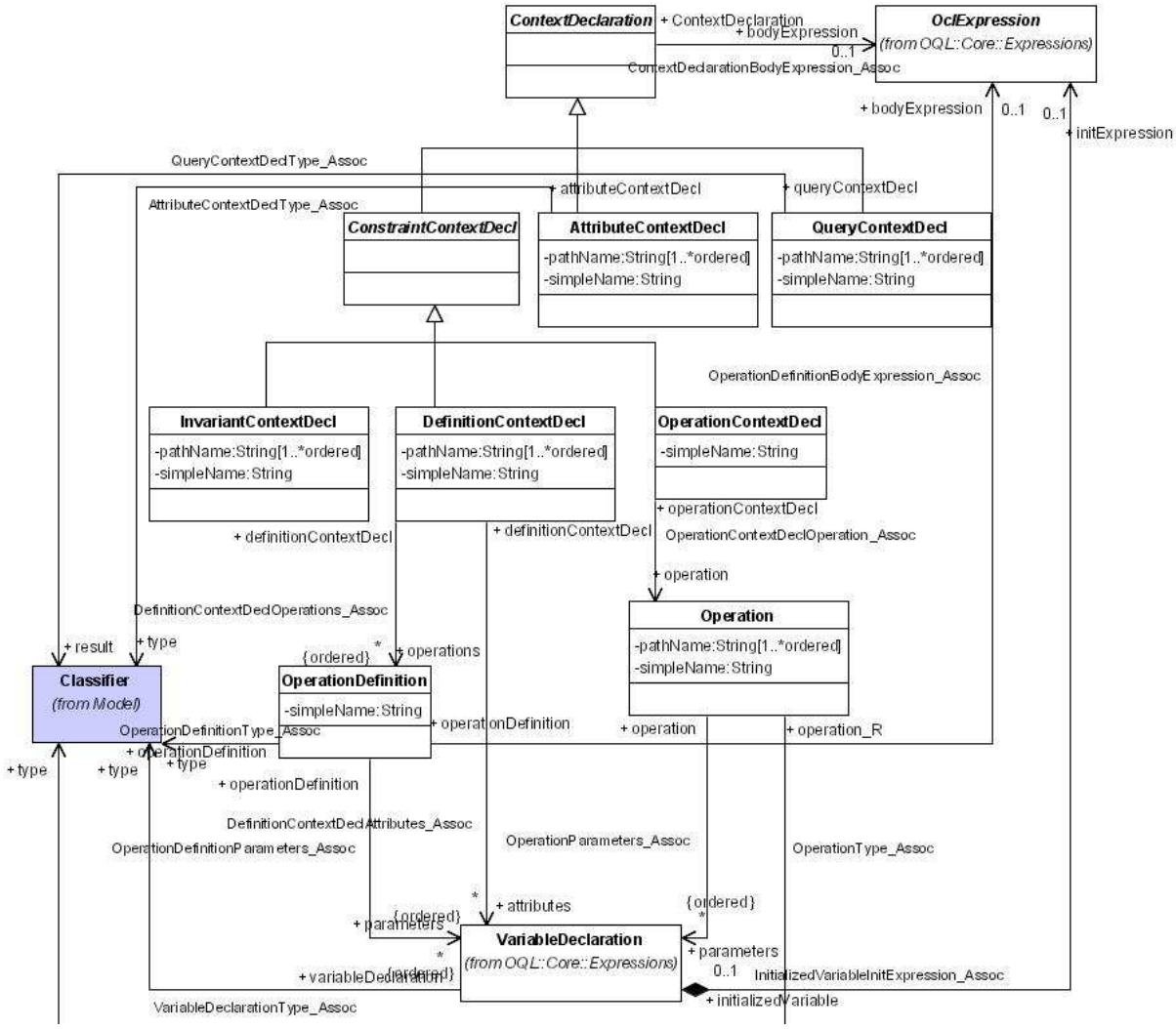

Figure 12: The Context Declaration Package. It does not belong to the core part of QML but is rather a set of helper meta-classes. These helper meta-classes are used to express where an OclExpression refers to, the kind of it (query, invariant, operation, definition and attribute) and any other specific information needed for each kind. The QueryContextDecl meta-class treats a query as a constraint on a model element resulting a set of values with a specific type.

THE QUERY CONTEXT DECLARATION METACLASS.

Figure 13 shows only a part of the context declarations package with the QueryContextDecl meta-class's syntax which is explained in detail here. It has to be noted that this extension approach stands outside the QML core metamodel and therefore it does not affect the compatibility of QML with OCL.

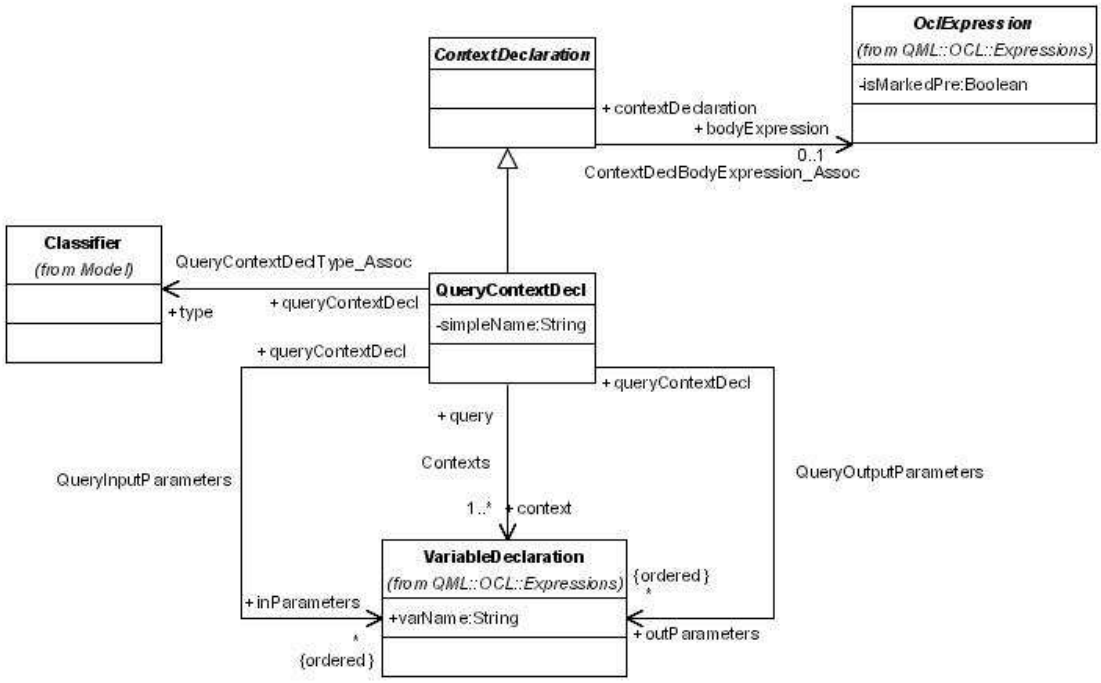

Figure 13: Part of the Context Declaration Package, showing the Query Context Declarations meta-class and its associations.

QueryContextDecl

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

Lets recall the example used throughout this section:

```
1: Context A: HotelModel#Hotel instanceQuery
2: A.HotelName = “Hilton” and
3: A.HotelAddress.City = “Athens”
4: A.Rooms->exists(Price < 100)
5: out Hotels := A
```

“instanceQuery” is the simpleName of the *QueryContextDecl* object, “A: HotelModel#Hotel” is the context *VariableDeclaration,* “Hotels := A” is the *out VariableDeclaration.* Lines 2 to 4 are the *body OCLExpression.*

In the next and in the following subsections we present representative query expressions formulated with QML. The objective is to give an informal presentation of the semantics of the QML expressions. The first simple example explains the use and the semantics of the QueryContextDecl meta-class. The second example makes use of the let expression of OCL to show how aggregation can be performed. When query expressions refer to M2 (i.e. available M2 metamodels) they obtain, as a result, qualified M1 models. These examples demonstrate also how to query for models and the last example shows how a query for data can be expressed.

SEMANTICS OF QUERY EXPRESSIONS AND EXAMPLES

This section presents representative query expressions formulated with the QML Query Metamodel Language. 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. In case that an M2 Metamodel also contains an instantiation metamodel in order to define elements for M0 instances then the query expressions could also obtain, as a result M0 instances. We will first examine a simple example that demonstrates the usage of the QML metamodel. In that simple example the constructs of QML explained so far are used and demonstrated. Next, we will explore, through more complex examples, the expressiveness of QML and its support for similarity ranking.

In order to better clarify the query formulation process and the outlined QML examples we will present some indicative screenshots of the Query Formulator Tool (developed by TUC/MUSIC). This tool offers an intuitive GUI that facilitates the query formulation allowing the user to browse/navigate through M2 knowledge base metamodels, choose the desired terms, assign values and constraints and have a view of his/her query as a tree with filled values and also as a valid QML expression. The Query Formulator Tool is an initial attempt to transparently expose the QML metamodel semantics to the user. In its current implementation it is only a tool that helps to demonstrate the QML functionality (it was used in the 1st review of DBE for that

purpose). Therefore one should consider the query expressiveness of QML (as outlined below through the examples) and the query formulation capabilities provided by the tool, as two separate things.

The following example queries are driven by the Semantic Service Metamodel (SSL) [8] which is one of the metamodels imported and supported in the DBE knowledge base that allows the semantic description of services. The following SSL primitives are used to for the formulation of the example queries and are illustrated at figure 14:

1) ServiceProfile: A service profile is a model according to which a service will be semantically described. A semantic package may have more than one service profiles (e.g. for describing the service into different user groups).
2) Attribute: An attribute (of a service profile) defines a slot of semantic information for a particular profile.

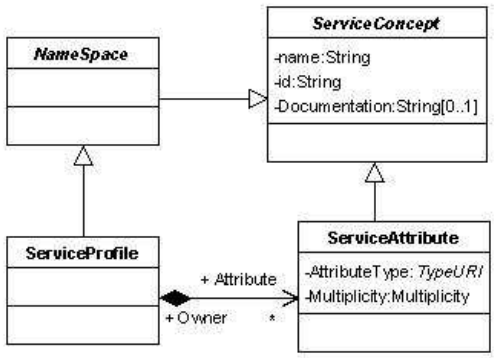

Figure 14: A part of the Service Semantics Language (SSL) metamodel

A SIMPLE QML QUERY

We will first examine a simple example that demonstrates the usage of the QML metamodel. In next sections, we will explore, through more complex examples, the expressiveness of QML and its support for similarity ranking.

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 15 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 VariableDeclaration* 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 (it is initiated as “A” and the type of “A” is the “ServiceProfile” class defined in the “SSL” metamodel). Moreover, in the case we had multiple result arguments the type of the query would be implied as TupleType. The body of the expression is an operation (the “A.name = ‘Hotel’”) on a property (here “A”) of the context. The context’s *body* is the expression “A.name = ‘Hotel’”, this expression is further analyzed on an OperationCallExp (with MOF *Operation* “=”) which has a *source* OclExpression (“A.name”) and a sequence of arguments. Here the sequence contains only one argument the *StringLiteralExp* “Hotel”. The *source* OCLExpression (“A.name”) can be further analyzed into “name” which is an *AttributeCallExp* which references the “name” *Attribute* defined in SSL metamodel. The *source* OCLExpression of this *AttributeCallExp* is “A” which is a *VariableExp.* “A” *VariableExp* refers to the “A” Variable declared at context level.

The following is a QML notation to express all the sentences of the previous paragraph into the specific QML constructs. It is equivalent to figure 15:

```
Queried(name: “SimpleQuery”, context:
                    VarDecl(name = “A”, type: “ServiceProfile”),
                    body:
          OperCE(oper:“=”, source: AttrCE(attrib: “name”,
                                source: VarExp(referVar:A)),
                  arg: StrLitE(val: “Hotel”)
                  )
                    out: VarDecl(name: “RServiceProfile”,
                                 init: VarExp(referVar:A)
                                 )
       )
```

The notation is Queried stands for QueryContextDeclaration, all attributes inside (i.e. name, context, body, out) are the attributes or associations of the QueryContextDeclaration model element presented earlier. VarDecl stands for the VariableDeclaration model element, the OperCE for OperationCallExp, the AttrCE for AttributeCallExpression, the AssocEndCE for AssocationEndCallExp, the StrLitExp for StringLiteralExpression, the VarExp for VariableExpression, the IterExp for IteratorExp.

It maybe useful at this point before reading the next examples which demonstrate capabilities and functionalities of QML to jump to the next section Evaluation Engine and Query Analysis. Next section explains how the QML queries are semantically annotated, validated, and evaluated against specific models or data.

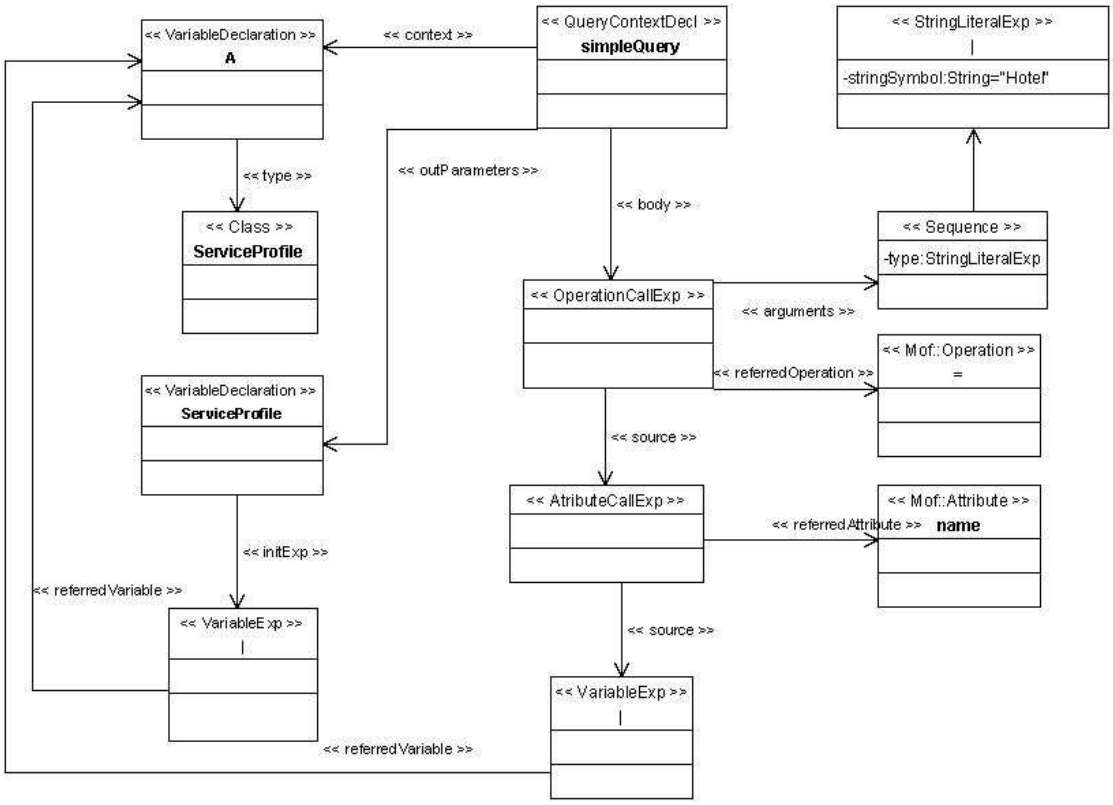

Figure 15: QML representation of the query: Context A: SSL::ServiceProfile simpleQuery A.name = "Hotel" out ServiceProfile: = A.

AGGREGATING OBJECTS

In this and the following examples the aim is not to present formally how the QML constructs work and how they are used, but rather to investigate capabilities, functionalities, and limitations of QML.

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 more than 2 ServiceAttributes 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”. “B” is bound by a let clause and represents a set of “ServiceAttribute” items. “Attribute” is the MOF AssociationEnd connecting the class “ServiceProfile” to the class “ServiceAttribute” with a cardinality 0 to many. 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, instead of a single “ServiceAttribute”, to the set of “ServiceAttribute” Model Elements that belong to “A”. We generally treat the MOF Associations as join conditions of classes. While the iterator “exists” iterates on the collection of ServiceAttributes, the “count” OperationCallExp is a method of the CollectionType class. This is the way one can use aggregation functions in QML.

For grouping objects we follow the same approach with XQuery. SQL like grouping is not available. 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 the next chapter about fuzzy queries 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.

QUERYING INSTANCES

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

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

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

The difficulty in this case is that this query is expressed in terms of a specific model (the “Hotel” model) and may not be able to retrieve the Athens Hilton hotel if it is expressed in terms of another model, for example “MyHotelModel”, that has a different structure. This is not a critical problem when searching for models because metamodels are not considered to change. In order to address this we used the semantic expansion of the query. The query during that process is expanded with terms of different models and maybe ontologies. The results that match other models will be ranked lower by using the fuzzy information model explained on the next chapter.

A MORE COMPLEX QML EXPRESSION

The next example tries to demonstrate some of the powerful capabilities of QML utilizing more complicated functions. What the next query does is to query both M1 layer model information and M0 layer data by using models coming from two metamodels: SSL and ODM. ODM is a metamodel for Ontologies. The example is as follows:

```
Context SSL:ServiceProfile complexQuery: SemanticPackage
Functionality->select(name=”CreditCardPayment”)->
exist(input.name=”CreditCardNumber”)
and
Attribute->select(name=”Address”)->
exists(type=”ODM::HotelDomain::Address” and getTypeClassInstance()->
select(type.name=”City”)->exists(TheDTPRange.lexicalForm=“Chania”))
```

This query is again posed against the SSL metamodel and retrieves all the services of the Hotel domain that are located in Chania and offer functionality for payments with credit card.

More precisely the query could expressed as (please recall SSL): Find the “SemanticPackages” that have at least one “ServiceProfile” which has at least one “Functionality” named “CreditCardPayment” with at least one “input” argument named “CreditCardNumber”. The “ServiceProfile” should also have an “Attribute” named “Address” with “type” the Ontology Class of the HotelDomain named “Address” and the instance of that “ServiceAttribute” class has an attribute named “City” which value is “Chania”.

This example demonstrates how one can formulate a query that refers to elements of an M2 metamodel that also contains an instantiation metamodel in order to define M0 knowledge base information. This query obtains as a result qualifying M0 instances. The expression *Functionality->select(name=”CreditCardPayment”)-> exist(input.name=”CreditCardNumber”)*

demands that a ServiceFunctionality (through the AssociationEnd Functionality) exists with a name "*CreditCardPayment"* and an input name *"CreditCardNumber"* (through navigation from the AssociationEnd "input" and an Attribute "name"). The latter sentence demands that a ServiceAttribute exists with name "Address", type "ODM::HotelDomain::Address" and the instance of this address has a property with name "City" and value "Chania". One should note here that the type of the address is obtained by a different context; in particular an ontology context offered by another M2 knowledge base information metamodel, named ODM[7],[8] [10].

Another interesting part of the query is the function *getTypeClassInstance* which is a function of OclModelElementType (the general type which refers to) and results to retrieving the instances of the type. In this example it retrieves the instances of "ServiceAttribute" Model Elements which are named "Address", etc). With this method it is possible to express a query which can be posed concurrently on two MOF layers. It is important though to state that this kind of queries are not supported by the DBE KB, but rather only from the standalone evaluation engine described on the next section.

A representative part of the above statement is depicted in Figure 16 in QML metamodel elements.

---

[7] For a detailed description of the Ontology Definition Metamodel (ODM) the reader should refer to the DBE document "Knowledge Base Design and Implementation Status" authored by TUC.

[8] It should be noted that the granularity of the QML query expressiveness is not limited to only one metamodel (i.e. it is allowed to combine semantic information from more than one metamodels).

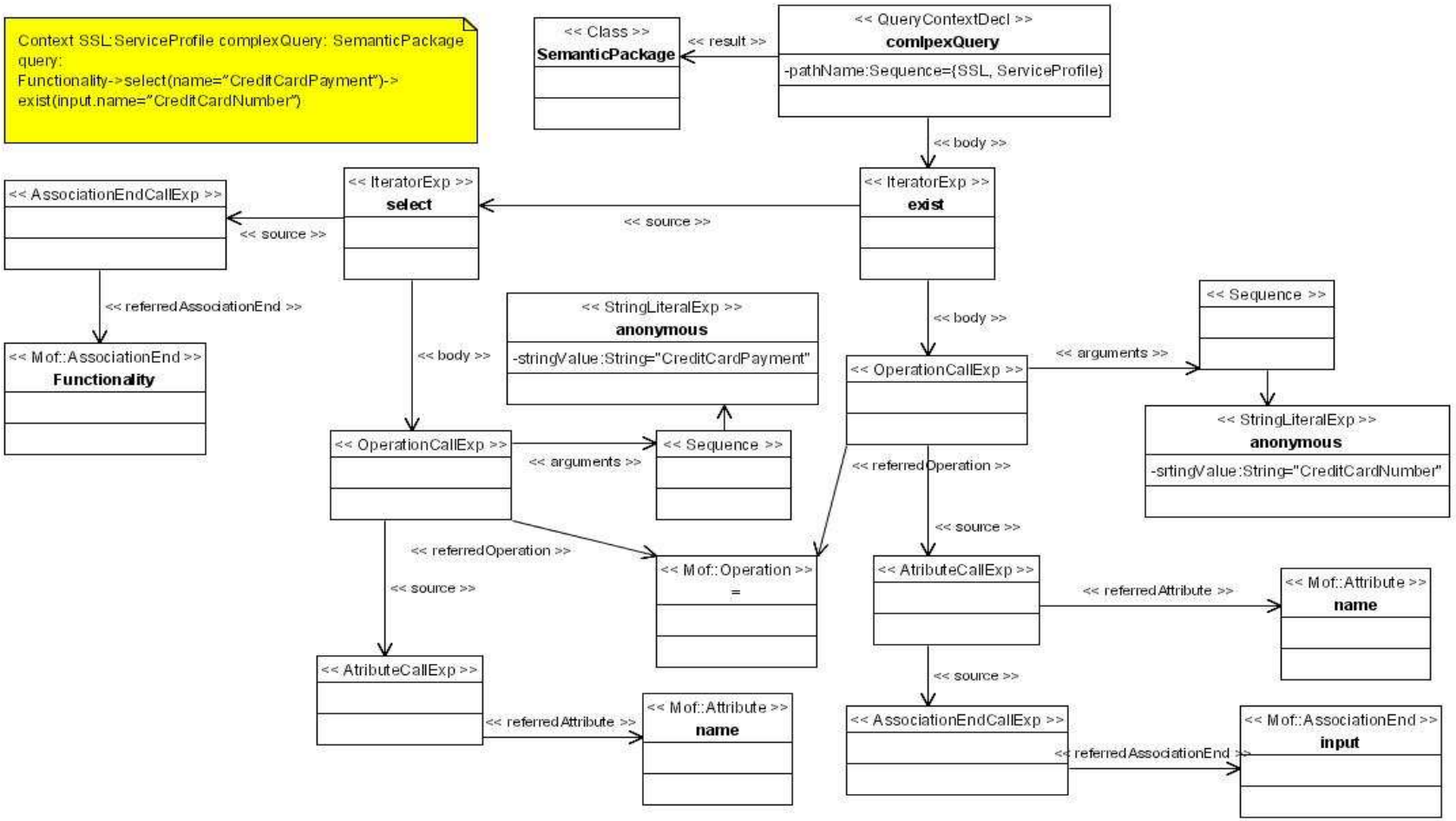

Figure 16: QML representation of the query: Context SSL:ServiceProfile complexQuery: SemanticPackage query: Functionality-> select(name="CreditCardPayment")-> exist(input.name="CreditCardNumber"). This figure presents the use of Iterators in QML. This query returns the service models that have a Functionality both named “CreditCardPayment” and having an input named “CreditCardNumber”.

EVALUATION ENGINE AND QUERY ANALYSIS

Two evaluation engines of QML were built. The first engine evaluates queries on top of Metadata Repositories (MDR). Thus, all query processing is done in Java. Indexing MDRs is not able so far, and thus, in the DBE Knowledge Base implementation we could not follow this approach, as the database was expected to have some thousands of models and data. In the DBE Knowledge Base we followed the architecture presented on Chapter IIV. In DBE KB in order to evaluate the QML queries against models and data the Recommender analyzes them, semantically annotates the terms, constructs an evaluation tree, semantically expands them with ontology terms, and finally, transforms them into XQuery queries, which are evaluated against an XML database. As the complexity of building XQueries from any QML query is very high we had to limit the QML queries to conform to a certain template. The Query Formulator is responsible for that. This template supports fuzzy queries and it is presented along with the Query Formulator in the next chapter. Note that in both implementations the query analysis and semantic annotation process is common.

The remaining section will describe the query analysis process and how the evaluation engine of QML against the MDR works. First of all we need to clear what is the Metadata Repository (MDR) and how it works. The MDR is an implementation of MOF architecture by Netbeans, where metamodels, models, and data resides. When a metamodel, model or data document comes into an XMI format (this is an XML document following the XMI XML Schema proposed by MOF) MDR has the API to parse it, validate it against its parent, and create the appropriate Java objects for it. When parsing models it creates Java objects that represent Java Classes, and the data (M0 level) document is represented as Java Objects which are instances of the model's Java Classes. All these java objects are stored in an OO database. The functionality of MDR is based on Refection mechanisms. More information is available at [12].

Something else to point out before going to the Query Analysis and Evaluation process is that each OCL expression evaluates to its type. For example *OperationCallExp* of the operation "=" evaluates to *Boolean*, while *AssociationEndCallExp* of the AssociationEnd Attribute evaluates to a *CollectionType* of *ServiceAttribute* elements. This is because in the SSL metamodel ServiceProfile and ServiceAttribute were connected with an association. In order to "go" from the ServiceProfile to the ServiceAttribute one has to "take" the AssociationEnd named "Attribute" which has a cardinality of 0 to many. Because the cardinality is 0 to many the type of the evaluation is a *CollectionType.*

Moreover, because AssociationEnd "ends" to the ServiceAttribute model element the type of the evaluation of the *AssociationEndCallExp* will be a *CollectionType* of ServiceAttribute elements.

As a guide through both the analysis and the evaluation steps we will use the following simple query:

```
Context A:  SSL::ServiceProfile
A.name = "Hotel"
and
A.Attribute->exists(name="Pool")
Out sProf := A
```

First of all variable A is evaluated, which means we take each ServiceProfile found in the MDR and put it in an object A (A has the Type ServiceProfile). Note here that the *body* OCLExpression of the Query Context is the OperationCallExp "and" this is how the *body* expression will look like in terms of QML:

```
OperCE(oper: "and", source:
          OperCE(oper:"=", source: AttrCE(attrib: "name",
                                source: VariableE(referVar:A)),
                  arg: StringLiterE(val: "Hotel")
                  )
          arg: IteratorE(name="exist",
                source: AssocEndCE(assocEnd="Attribute",
                                  source:VariableE(referVar:A)),
                body: OperCE(oper: "=",
                          source: AttributeCE(attrib: "name"),
                          arg: StringLiterE(val: "Pool")
                            )
                )
       )
```

QUERY ANALYSIS

The QML query first of all is analyzed. During analysis the query is parsed, it is validated against the metamodel (or model), and it is semantically annotated. All these three happen together. The analysis process is top down. For the previous example this would mean that the process starts from the OperCE "and" and goes through all other OCL expressions until the Literal expressions are reached. In this process the result types of each evaluation are calculated, and the QML query is annotated with the actual MOF Model Elements, Operations, Attributes and AssociationEnds. The result is the above query but semantically annotated with the information coming from the SSL Metamodel. Everything is a specific model element rather than a string. Query validation against the SSL metamodel occurs during this process. If when searching for an attribute named

"name" under the model element of SSL ServiceProfile and it is not found the processing ends. The query is not valid it tries to search something that is not modelled. Another validation check is the type checking. The operation "=" for example expects the source and the argument to have the same type. If the type of the of the expression "A.name" (i.e. the type of Attribute name of the ServiceProfile) is String, the analyzer expects the argument to also be of type String or a Sub-class of String. StringLiteralExp has as type String thus this type checking is correct. If someone tries to apply an iterator expression to a non Collection type there is a violation. Another validation check is if an operation is supported by an object. If we wrote "A.name < 3" this would not be valid. Operations and types are found dynamically by using the java reflection.

The query analysis can be seen as an evaluation process against metamodel. Evaluates that all the model elements exist, finds them, and annotates the query tree. This process is not necessary for the stand alone evaluation engine as it can run "on the fly" (while processing the query against actual data). For the DBE KB engine, on the other hand, it is vital as the semantic information will be needed on the next steps (evaluation tree and semantic expansion) and in no other step we can see if the query is valid. This can be done only with QML against the MDR where the metamodel (or model if the query is for M0 data) is loaded.

EVALUATION PROCESS

When the query analysis process ends we have the query of the example above validated and semantically annotated. Evaluation process will run for each model element defined in the context. Again a top down approach is used starting from the OperCE "and" until the Literal Expressions are evaluated into their values (quite simple). Note the actual evaluation OperCE "and" into true or false will occur after the literal evaluation. If the evaluation result of the body expression is true then the output parameters are evaluated with the same process. If it is false, that ServiceProfile will not be in the result set.

Now we will go through the evaluation of the body expression step by step. We start from the OperCE "and". In order to evaluate this expression we evaluate first the source expression which is the OperCE("=", A.name, "Hotel"). Again in order to evaluate it we try to evaluate the source of it, which is the AttrCE(name, A). Again we go deeper until the variable A is reached. A evaluates to one ServiceProfile and now we can evaluate the AttrCE (name, A). This evaluates to the value of attribute name (for example

"CarCompany"). Because name attribute has the type String, the type of the evaluation is String. Now the argument of OperCE("=", A.Name, "Hotel") should be evaluated. It is the String Literal Expression StrLitE(val: "Hotel") which a evaluates to the String "Hotel". Now the operation "=" of the type String should be evaluated. The evaluator searches on the java String for a method name "equals" with one argument of type "String". If found (it should be found because it passed validation) it is called and the result of it is the result of the evaluation of the OperCE("=", A.name, "Hotel"). Note here that the type String mentioned so far is not the Java Lang String but rather the MOF Primitive Type String.

After the OperCE("=", A.name, "Hotel") evaluates to false the OperCE "and" of type evaluates again to false. For another ServiceProfile which will have the attribute "name" valued as "Hotel" the body expression of the OperCE "and" will be true and the evaluation process will try to evaluate the argument of "and" the IterExp "exists". In order to do that the evaluation process evaluates the source expression of this IteratorExp (i.e. the "A.Attribute") with similar way like it did with "A.name" before. This evaluation results into a Collection (CollectionType) of ServiceAttribute model elements. Now the iteration between all ServiceAttributes found starts and for each of them the *body* OCL expression is evaluated (i.e. name="Pool"). The type of the body expression should be Boolean (it is in this example). When the first evaluates to true IterExp "exists" evaluates to true. If the body of no ServiceAttribute is evaluated to true the IterExp evaluates to false. The way the body is evaluated is similar the OperCE("=", A.name, "Hotel"), but there is a vital difference; this source of OperCE("=", name, "Pool"), i.e. the AttrCE(name) does not have a variable expression as a source like the other. AttrCE(name) does not have a body expression at all, against which object will be evaluated? The answer is the current context. In this example the current context of the body expression of "exists" is not the global one, but rather it is each object the iteration occurs (i.e. the first ServiceAttribute, then the second, etc).

Generally when evaluating an we have a Global Context (or many but with variable names) and each expression has its own context. For example of "City" in the expression "A.Address.City" is an address. Some contexts are specifically mentioned and are represented in QML as the body expressions and evaluation is against them. Some other contexts are implied as in "exists(name="Pool")" and if no special exists the expression is evaluated against the global context. We could rewrite the above query like this:

```
Context A:  SSL::ServiceProfile
name = “Hotel”
and
Attribute->exists(name=”Pool”)
Out sProf := A
```

“A” variable is implied as body of AttrCE(name). No other context exists.

SUMMARY

In this chapter we presented the formal declaration of QML along with a set of examples to show how it is used.

The Query Metamodel Language (QML) is actually an alignment of OCL 2.0 to MOF (OCL is bounded to UML). It is defined in terms of the MOF architecture as a M2 layer Metamodel and can query data on M2, M1, and M0 layer. It can also express constraints for M3 MOF Meta-metamodel. QML is very powerful as it is not bound neither to specific models or metamodels but has a generic way to express queries.

We discussed how the queries are semantically annotated and validated against the model or metamodel used into the query during the Query Analysis process. We explored the abilities and the limitations of each the two evaluation engines proposed in this thesis. The first one is the DBE KB where QML is used in a strict way (because XQueries need to be automatically generated by the QML queries), but the implementation is scalable and fast because data is stored in an XML repository. The second implementation, not used in DBE, is an evaluation engine on top of the MDR. While the second implementation makes full use of QML, the MDR is not scalable and does support indexes. Thus, in DBE we formalized the kind of queries to be supported. These query templates are constructed using the Query Formulator in order to support fuzzy information retrieval techniques. They are discussed in the next chapter.

In the next chapters we will examine how QML is used to express fuzzy queries (queries with weighted terms), how can one create QML queries with an API (the Query Formulator), how the QML queries are transformed to query evaluation trees and how exactly the queries are executed. On chapter VII, we will see how the MOF and QML terms are used in order to semantically expand the query.

CHAPTER V – THE FUZZY MODEL AND THE QUERY FORMULATOR

INTRODUCTION

In the last chapter we presented QML and should be clear to the reader that while QML offers many capabilities it is a complex language with complex constructs. For that reason we developed a set of formulation APIs (the Query Formulator, the Advanced Query Formulator and the Keyword Formulator denoted all together as Query Formulator). These APIs can be used from legacy systems or GUIs to query the DBE Knowledge Base. These APIs are responsible to create QML queries with respect to an information retrieval model, where each query term may have a weight.

This chapter presents the information retrieval techniques (the fuzzy model) elaborated and how QML queries are formulated using the Query Formulator, note that QML queries can be directly created and executed, but we offer this module to make the process simpler as QML is quite complex. This chapter also presents how fuzzy queries are formulated and how keyword queries are analyzed. The advanced query formulator module used to offer simpler and more advanced capabilities to users, is also presented here.

THE QUERY FORMULATOR

The Query Formulator module is responsible for producing fuzzy QML queries based on a set of weighted criteria. The criteria may be structured, semi-structured, or unstructured (as in keyword-based search). A criterion is described by five parameters, which are shown in Table 2. Context is the model element (in either M1 or M2 level) that must exist and conform to the criterion. Path is the path from the context to an attribute (i.e. name, price, etc). We do not want for example any price to be less than 70 but only the room price of a Hotel element. The operation is the operation of the criterion. The supported operations vary depending on the type of attribute (numeric types support “<”, “>”, etc and string type support “=”, “like”, etc); for a complete reference of available operations please refer to OCL2.0 specification (11). Value refers to the value that the attribute must have exact, greater than, etc. Finally, weight refers to how important this criterion is for the general query. The results of a query come to a ranked order of

relevance, user may define which criteria are more important and which not. The weight value ranges from 0 to 1.

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

Table 2: Examples of criteria for the query formulator API

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

When value is of string type it can either be a single word or a phrase. The value "Los Angeles" will formulate a query searching for the phrase "Los Angeles" and not for the two words "Los" and "Angeles". If users require the second option they can create queries with two criteria. The case of values is preserved during the formulation process and it is up to the execution engine on how to handle it.

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

THE INFORMATION RETRIEVAL TECHNIQUES

In this section are presented the information retrieval techniques that were used to support fuzzy queries and a demonstrative example of how queries are formulated into QML. Note that this framework is independent of metamodels. Thus, if DBE evolves and

uses new metamodels, this framework is not needed to change on both design and implementation levels.

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

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

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

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

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

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

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

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(15) prove its superior performance in comparison with the traditional information retrieval models.

In order to actually evaluate the queries, one should give the evaluation functions f_{NOT}, f_{AND}, and f_{OR}. There are numerous possible definitions of these functions. Study (15) presents the definitions that correspond to the p-Norm Extended Boolean Model, which is the most general one. Note that the functions f_{AND}, and f_{OR} are n-ary instead of binary. This is due to the fact that these evaluation functions are not commutative as their Boolean counterparts.

There are numerous strategies for the implementation of Extended Boolean Model on top of a RDBMS (16), (17). But we need an implementation that is platform independent, and as such we need to implement it on top of QML. However, there is a straightforward implementation of the p-Norm by using QML in case that the queries that are accepted by the system have a simple form (either conjunctive or disjunctive queries).

To demonstrate the technique, let us assume that the queries accepted by the system are simple disjunctive queries. Let us further assume that Items are the Model elements or Objects depending on the level the query refers to (i.e. M0 or M1). Features are the elements connected with the Items. Note that a Model Element may be a Feature for another Model Element or an Item. Depending on the query context the Items are resolved. Every connection between an Item and a Feature is a path from the Item to Feature and the Feature's value. This path can be defined by a QML path expression and has a weight denoting how relevant the Feature is for the Item. Every query consists of

query terms (t_{qi}) which are QML path expressions and a query weight (w_i) denoting how important the term is for the overall query. Thus, every term must match a feature item path. In other words, if an Item (i.e. Model Element) is connected to a Feature with the query term (i.e. the QML path expression) a weight (a_i) is returned on how "strong" this connection is. For example if an Item has the same path to the Feature as the query term the weight 1 is returned; if the path is "alike" the query term an intermediate weight is returned denoting how relevant the two paths are; and if the Item-Feature path has nothing to do with the Query path zero is returned. This weight is the Item-Feature weight. Equation 4-1 is the f_{OR} evaluation function of the p-norm model.

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

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

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

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

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

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

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 is done in order to improve the recall of the system.

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

Generally, the argument *a* represents how relevant is the 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 it exists. In our application, this value can vary from 0 to 1 depending on a set of criteria which are shown in table 4. The weight of qualifying operations is always 1.0 and of non-qualifying operations is always 0. Note that if a model has more than one features that qualify or semi-qualify – for example two room prices – then if between them a qualifying feature exists *a* is assigned the value 1 and the semi-qualifying features are ignored. If, on the other hand, all the features returned are semi-qualifying the weight is assigned a value depending on all of them.

The string operators either finds an exact match or a sub-string match assigning to *a* the values 1 and 0,5 respectively. For numerical operations the semi-qualifying value of *a* is closer to 1 as closer to user's best value the found value(s) is. The formulae which results the weight of the numeric semi-qualifying operation is:

$$a_{numeric} = \begin{cases} \dfrac{1 - |\, best - avgSemiValues \,|}{best}, if\ 1 - |\, best - avgSemiValues \,| > 0 \\ 0, \text{otherwise} \end{cases}, \qquad \textbf{(V-2)}$$

where *best* is the user's numerical value (either upper or lower bound) and *avgSemiValues* is the average of the values resulted from the semi-qualifying operation.

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

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

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

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 the contexts and the attributes of the query, result types and join conditions are declared. It is common that most queries use the same main attributes of the many offered by a model or a metamodel and the template realises this idea.

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

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

```
Template template = new Template();

//Adds a template element named "location" searching

//on the BML path "Hotel/Locality" of type String

template.addTemplateElement(new TemplateElement("location",

                            "Hotel::Locality", "String"));

template.addTemplateElement(new TemplateElement("country",

                            "Hotel::Country", "String"));

template.addTemplateElement(new TemplateElement("starCategory",

                            "Hotel::StarCategory", "Integer"));

template.addTemplateElement(new TemplateElement("roomPrice",

                            "Hotel::Rooms::Price", "Integer"));

…

//Create a Query formulator

AdvancedQueryFormulator form = new AdvancedQueryFormulator(

       (QmlPackage)qmlTool.getModelPackage(),
       AdvancedQueryFormulator.INSTANCE_QUERY);


//Initialise the formulator with the static query Template created earlier.
form.setTemplate(template);

//Create a query expression for each criteria (if exists) and added to a

Vector exprs = new Vector();

//Note the last number is a weight of how important query expression is

QueryExpr expr = new QueryExpr("=", "location", "Tampere", 1.0);

exprs.add(expr);


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

exprs.add(expr);


//make an array of query expressions

QueryExpr[] queryExprs = new QueryExpr[exprs.size()];

…

//put the expressions inside the formulator

form.getQuery(queryExprs);
```

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

FORMULATING KEYWORD EXPRESSIONS

This section discusses how keyword expressions can be formulated into valid queries. Keyword expressions consist of query terms which can either be unstructured (ex. just “Athens” and not Hotel.City=”Athens”) or semi-structured (ex. City=”Athens”) as was shown at Table 2.

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

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

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

The parsing of the text into a query expression is done by a java parser produced by JavaCC application based on the grammar shown in Appendix C.

This keyword query is also exploited and expanded using ontology information explained in Chapter VII – Semantic Exploitation.

SUMMARY

In this chapter we presented the Query Formulator and information retrieval techniques. Query Formulator is developed as an API for creating QML expressions; a GUI used this API to create QML expressions. On top of the Query Formulator an Advance version of it was created to offer in an easy to use manner complex constructs. Query templates can be created and reused to create queries. This module offers the

ability to create template for example for querying hotel data based on a model by an expert. This template can then be distributed and be used by many either by legacy systems to query DBE Knowledge Base or by simpler user interfaces. For example with the Query Formulator you have to select the term Hotel.City and then enter values like “Athens” and “Volos”, while with the Advanced Query Formulator someone has selected the main parts of querying hotels and created a template. Then the user enter just “Athens” on the City field.

While both Query Formulator and Advanced Query Formulator are used to construct structured QML queries the Keyword Formulator is used to parse semi structured (ex. room.price < 70) and unstructured (ex. “Hotel” or “Finland”) query expressions and formulate them into QML queries.

All the query terms are expressed inside QML with weights. The final results are ranked with information retrieval techniques explained in this chapter. Specifically the p-norm model was used to rank the results. An information retrieval model for the specific application was given. This model expands the result set with close values ranked with smaller values. For example for the query Hotel.Room.Price < 70 the result set will be expanded with Rooms with price greater than 70 with less rank.

Note that the queries will be further expanded semantically (i.e. with similar terms of other (or the same) models) by using ontologies. For example Hotel.Room.Price < 70 will be expanded with Motel.Room.Price < 70 coming from another model with smaller weight. This is discussed on Chapter VII – Semantic Exploitation.

Next chapter discusses how this QML formulated queries will be transformed into query evaluation trees. These trees are the heart of the recommender module. By using these constructs we can break down query terms, semantically exploit them, and merge result sets with information retrieval techniques discussed here.

CHAPTER VI – QUERY EVALUATION TREES

INTRODUCTION

This chapter presents the methodology used for the construction of evaluation trees of QML query expressions. These expressions refer to M2 or M1 knowledge base information (i.e. available M2 metamodels, M1 models). The evaluation process has the goal of extracting the semantic information residing in the query model (an instantiation of the Query Metamodel specifying a query) into an evaluation tree. The evaluation tree can be easily parsed later on by execution engines, or other modules (e.g. the XQuery code generator).

Next sections describe the query evaluation model (the object model in the object oriented sense), with which rules from a QML expression an evaluation tree is constructed, and, finally, an example of an evaluation tree for a QML expression.

THE QUERY EVALUATION TREE MODEL

A QML query consists of QML expressions with fuzzy operations. The evaluation process extracts semantic information of the model elements used in the query expressions and constructs a hierarchy of operations to be executed into the evaluation tree. The process of extracting semantic information is called semantic annotation and refers to defining the model (or metamodel) elements of the knowledge base a query expression is using. The process of placing the operations used in QML query into a hierarchy refers to defining the order in which the operations of the query should be executed and join the partial results in order to produce the final results. Keep in mind that each item evaluated as relevant result is assigned a weight and when joining a standard procedure should be followed on assigning a final weight.

The evaluation tree consists of nodes each one denoting different functions that should be applied when evaluating the whole query. A query, as discussed already in the previous chapter, is composed by query terms. A *query term* is a basic *query expression*, a criterion to be evaluated. For example the query *"Hotel.City='Athens' and Hotel.Room.Price<100"* consists of two query terms: *"Hotel.City='Athens'"* and *"Hotel.Room.Price<100"*, while the "*and"* is the operation between them, although it is a QML query expression, it is not referred as a *query term* but just as an *operation*.

Thus, each query term *qt* consists of several characteristics taking into account that it is a fuzzy criterion. Namely a query term qt_i consists of a path expression (p_i), an operation (op_i), a literal value (v_i) and a weight (w_i) denoting the user importance on this term. The following formula describes this:

$$qt_i = (p_i, op_i, v_i, w_i) \quad \text{(VI-1)}$$

Moreover, queries are composed by query terms using operations which do not have fuzzy weights but are responsible for joining the ranked results from query terms assigning new ranks on the results following the p-norm fuzzy model explained in the previous chapter.

To model these needs an evaluation tree model was proposed consisting of nodes of the following types: *ContextNode*, *OperationNodes*, *NavigationNodes* and *LiteralNodes*. The root node of the syntax tree is the *ContextNode,* which provides the information about the context of the query (ex. "HotelBusinessModel") and the result type of the result set. The query terms of the evaluation tree are *OperationNodes* that provide information about the operation to be evaluated (ex. "<") and its parameters (ex. "Hotel.Rooms.Price", "100", and the weight "0,5"). But not all *OperationNodes* are query terms; for example *OperationNode* can be the "and" operation between two query terms. The parameters of *OperationNodes* can be either *OperationNodes*, *NavigationNodes,* or *LiteralNodes.* Only *NavigationNodes* and *LiteralNodes* can be leaves of the evaluation tree. *NavigationNodes* provide information about a navigation path from the context of the query through the specific metamodel or model elements[9] (ex. "Hotel.Rooms.Price" where "Hotel" is the specific model element of the Context model etc.). In case that a metamodel element is abstract (e.g. it generalizes various model elements' types), then the type of the specialization element is also needed to define the navigation path in a concrete way. An example is if "Accommodation" is an abstract element which has as sub-elements "Hotel" and "Motel" then two different *NavigationNodes* will be created for the term "Accommodation.City" one as "Accommodation[Hotel].City" and another as "Accommodation[Model].City". The type of the path elements is known in the QML query and is therefore determined and assigned as the semantic annotation process. *LiteralNodes* provide information about explicit literal values used in the query model, e.g. 100, 'Athens' etc.

[9] it refers to the knowledge base metamodel against which the specific query is posed.

THE EVALUATION TREE CONSTRUCTION

The construction process consists of several rules, which when applied to a query model an evaluation tree is generated. The rules are summarized below for each of the main expressions of the query metamodel:

- For each *OperationCallExp* found add an *OperationNode* in the current node with children the source (expression that supports this method) and the arguments. The weight is found by extracting the appropriate information. The children might be any of:
  - A *NavigationNodes* and a *LiteralNode* if it is a query term or
  - *OperationNodes* if it is a complex query expression (e.g. *and*).
- For each *LiteralExp* (*StringLiteralExp*, etc) create a *LiteralNode* containing the specified value.
- For a navigation path of *PropertyCallExp* (apart from *IteratorExp* and *OperationCallExp*) construct a single *NavigationNode*. For example `Attribute.name` is translated to a single NavigationNode with two elements: `Attribute` and `name`.
- The *IteratorExp* is transformed to a complex representation of Operation, Navigation and Literal nodes of the Syntax Tree. In particular *exists* is added to the current NavigationNode with no other special meaning. For example *Attribute-> exists(name = "Something")* is transformed into an OperationNode (for "=") with two children: A StringLiteralNode (for "Something") and a NavigationNode (with two elements: Attribute and name). Other Iterator expressions as *select* or *forAll* may be transformed in similar ways. For the case of *select* it may be transformed in an AND *OperationNode*. For example *Functionality-> select(name = "a")-> exists(input.name = "b")* is the same as *Functionality-> exists(name = "a") and Functionality-> exists(input.name = "b").* Such transformations may also exist for the rest of the Iterators (forAll etc).

Another type of node is the query branch which is a specialization of the *OperationNode* and is used when semantically expanding the query terms. It will be discussed in the following chapter with the query expansion.

EVALUATION TREE EXAMPLE

In this section a usage example is demonstrated on how the simple QML expression that was presented in the previous chapter is formulated into an evaluation tree. The following figure illustrates the evaluation tree produced. The query without the information of weights and context in order to be easy to read is *"Hotel.City=Athens and Hotel.Room.Price<100".* The ContextNode is the root element. It contains information for the query context (a hotel business model on which the following model elements apply), the result type and the query name. The ContextNode has as a child the OperationNode with attribute "and". This OperationNode has two children; the query terms. The first one has the operation "=" and weight value one and two children a NavigationNode with the path "Hotel.City" and a StringLiteralNode with value "Athens". The other OperationNode has the operation "less than" and has two children; a NavigationNode and a RealLiteralNode. The NavigationNode contains the elements that participate to the navigation process that starts from the context element. Note that the elements inside the Navigation Nodes are semantically annotated with the appropriate model terms.

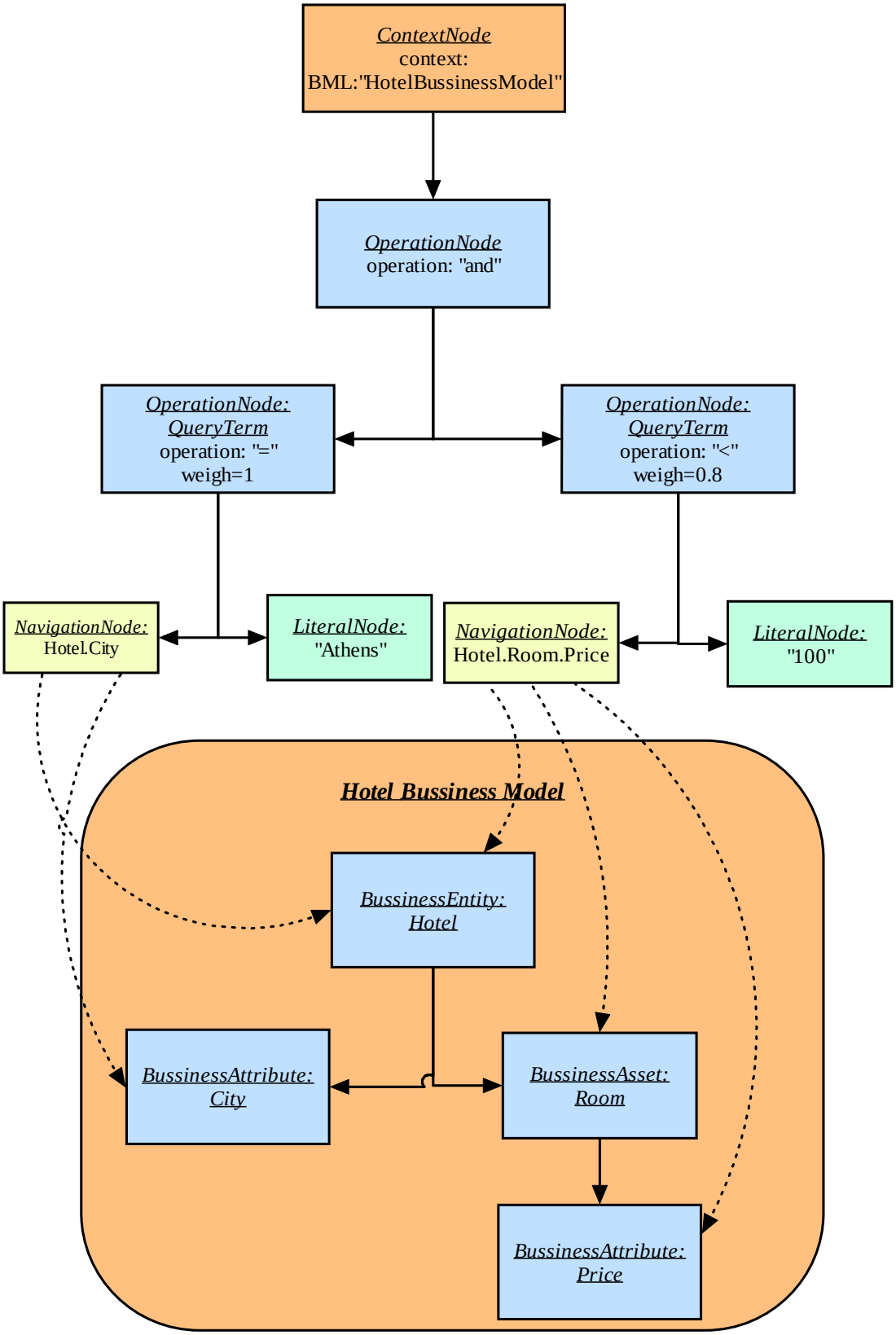

Figure 17: The evaluation tree for the query "Hotel.City=Athens and Hotel.Room.Price<100". In the figure the semantic annotation is also illustrated from the Navigation nodes to the actual business elements.

SUMMARY

This chapter discusses the Query Evaluation Tree Model and how these trees are constructed from a QML expression.

The Evaluation Trees are used to contain all the information about how a query will be processed and the results will be merged with the appropriate ranks. The evaluation trees contain information about each query term, their weights with respect to the fuzzy model, how query term results will be merged and ranked. The evaluation trees can easily be expanded with similar terms from other models (using the semantic exploitation discussed in the next chapter). SQL is the formal language to express queries for RDBMS and the evaluation plan is what operation should be performed and in which order. On the same sense, evaluation trees in this thesis are the operations to be performed (easily managed and transformed into XQueries) and how result sets will be merged with respect to the information retrieval model used.

In the next chapter we will discuss how the evaluation trees will be expanded with semantically similar terms of other models, metamodels, and ontologies by using information from ontologies. Moreover, as each business domain (egg. the hotel industry) in DBE may use more than one ontology to express similar ideas (egg. “Hotels” and their data structure), an algorithm is proposed, implemented and tested to find similarities between ontologies.

CHAPTER VII – SEMANTIC EXPLOITATION

INTRODUCTION

This chapter 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 follow different models (structures) from the model the query was expressed. This is desirable since, the different models refer to the same kind of service. For example two hotels use two different models two express their services. When someone makes a query for hotels, he doesn't care in which terms the data is modelled (structured), he just wants to find a hotel.

At chapter five on section for the fuzzy model a problem was described where a service could not be retrieved because it was expressed on a model having the model path *Hotel.Address.City* and the query searched for the path *Hotel.City.* This problem is addressed if we reformulate the starting query in terms of the second model adapting properly (downscale) the weights of each term. In our example we want from *Hotel.Address.City* to go to *Hotel.City* stating that the first has weigh for example 1, while the second has the weight 0.8. The reformulation process is not an easy task in the relational world of different schemas for each database and different approaches exist like (18) and (19). That is another reason we selected a query language that is expressed in semantic terms instead of directly using a language as XQuery, SQL or SPARQL. With these languages semantic information is lost (in fact was never present). Someone who knows the semantics of a schema (either XML or Relational) poses the query. The query does not carry the information about the semantics and thus you can only work lexicographically if you want to reformulate the query.

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 meta-class and have similar properties - have both in common City) or if the metamodel is expressed in terms of ontologies, we can use equivalent relationships from ontologies and reformulate the query by the means of the ontologies.

DEFINING THE PROBLEM

We consider a P2P environment where any SME can describe its services (using SSL metamodel) and its business (using BML metamodel) 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 scenario is that each domain has a small number of models which are reused from the large variety of SMEs with few or no 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 (for description of ODM please see (20) and (3)). We suppose that each domain is described not necessarily by one but possibly by several ontologies.

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

If models were just different structures of 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 then executed. That is; create 800 queries, execute them, and finally, join the results. Thus, even though the system can explore all information to enhance the recall and the precision of the system it is prohibited by performance costs.

The main function which calculates similarities in the fuzzy model was described in chapter V. As the following work uses this function and expands it, for convenience it is repeated here:

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

In order to include all the above statements in a general case, we specify the problem as follows: The a_i term of equation (V-1) is a weight which denotes the association of the feature (model element path) for the query term *i*. w_i is the weight of the query term *i*. The matching factor is decomposed of two factors c_i and v_i as expressed by the equation:

$$a_i = c_i * v_i, \qquad \textbf{(VII-2)}$$

where c_i is the weight of the feature's concept to the query term's concept and v_i is the weight of the feature's value to the query term's value.

Remember from chapter V that a_i took a value in the interval [0, 1] depending only on the value of the query term. For example "Athens Hilton" matched "Hilton" query term value with weigh 0.5 – it is not equal but it is very close. Note that the model elements should be equal (here "Hotel.Name"). If they weren't equal a_i is 0 (i.e. for "Hotel.Name" and "Hotel.hotelName". With equation (VII-1) this can change. Now v_i is the weight denoting how similar is the value of the feature with the value of the query and is described in table 4 of chapter V and c_i is the weight denoting the similarity of the two concepts. For example if the query term is "Hotel.Name = 'Hilton'" and the feature is "Hotel.hotelName = 'Athens Hilton'" c_i could be 0.7 and v_i 0.5. Thus, the resulting a_i is 0.35. While for the v_i the algorithm to produce the weight value is described on chapter V, for the c_i the algorithm will be described in this chapter.

The approach we follow to resolve related paths and their corresponding c_i*'s* and the model for creating an expanded query which makes use of this information to improve the system's recall is explained in this chapter. First we discover the related paths for each query term of the query by making use of the semantic information of the query and the ontology concept similarities found at an earlier stage (see the following "Ontology Similarity Analyzer" section). Along with the related paths we compute their corresponding weight, which denotes their relevance to the original query term. Finally we reformulate the query by adding new query terms. The final query can be then

processed to produce a valid XQuery expression, which will be executed by the PSM execution engine (the XML database).

ONTOLOGY SIMILARITY ANALYZER

In this section we discuss how the ontology mappings are created based on a set of rules. Although the techniques discussed in this section apply for ODM ontologies keep in mind that are also valid for OWL/RDF ontologies as there is a one to one mapping between ODM and OWL as explained at (3). In order to formally define these rules we need first to define the ontology space.

DEFINITION OF ONTOLOGY SPACE

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

Inside *O* we define a number of functions as follows:

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

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

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

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

- $\mathbf{o_1 \text{ -> } p_1 \Rightarrow p_1}$ **domain** $\mathbf{o_1}$
- $\mathbf{p_1 \text{ -> } o_1 \Rightarrow p_1}$ **range** $\mathbf{o_1}$
- $\mathbf{p_1 \text{ -> } d_1 \Rightarrow p_1}$ **range** $\mathbf{d_1}$

DEFINITION OF ONTOLOGY SIMILARITY RULES

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

Definition 1. A similarity *s: O x O Є* [T_{sim}, 1] is a function from a pair of entities to a real number expressing the similarity between two ontology elements such that
s(a, b) =1 *iff a = b* (definiteness)

The following rules define how similarities are created:

Rule 1. if $\mathbf{e_1 \sim e_2}$ then $\mathbf{s(e_1, e_2) = re}$ and $\mathbf{s(e_2, e_1) = re}$ (equivalence rule)
Rule 2. if $\mathbf{e_1}$ **IsA** $\mathbf{e_2}$ then $\mathbf{s(e_1, e_2) = rb}$ and $\mathbf{s(e_2, e_1) = rp}$ (IsA rule)
Rule 3. if $\mathbf{e_1}$ **IsA** **e** and $\mathbf{e_2}$ **IsA** **e** then $\mathbf{s(e_1, e_2) = rb}$ and $\mathbf{s(e_2, e_1) = rb}$ (sibling rule)
Rule 4. if $\mathbf{op_1}$ **range** $\mathbf{\{o_i\}}$ and $\mathbf{op_2}$ **range** $\mathbf{\{o_i\}}$ then $\mathbf{s(op_1, op_2)=rt}$ and $\mathbf{s(op_2, op_1)=rt}$ (type rule)
Rule 5. if $\mathbf{op_1}$ **range** $\mathbf{\{o_i\}}$ and $\mathbf{op_2}$ **range** $\mathbf{\{o_i'\}}$ and $\mathbf{s(\{o_i\}, \{o_i'\})=r}$ then $\mathbf{s(op_1, op_2)=rt*r}$ (type rule 2)

Rules 3 and 4 are indirect as we assume that properties with the same range (i.e. type) have some equivalence. This is needed because most of the times an ontology creator might define that two classes are equivalent but drop out that two properties are equivalent. Four similarity parameters are defined: *re* for equivalence, *rb* for subclasses and siblings, *rp* for superclasses, and *rt* for types. All these parameters' values range in the interval [0,1]. Some indicative values are: *re* = 0,9, *rp* = 0,8, *rb* = 0,9 and *rt*=0,8. In a later section we will discuss how these values are estimated.

Imagine now three classes: Accommodation, Motel, and Hotel. Motel IsA Accommodation, while Hotel and Motel are equivalent (Hotel ~ Motel). If we apply the abovementioned rules and parameters we end up with four similarities: s(Accommodation, Motel) = 0.9, s(Motel , Accommodation) = 0.8, s(Hotel , Accommodation) = 0.9, and s(Accommodation, Hotel) = 0.9. As you can see no similarity exists between Hotel and Motel because there is no direct connection between them. We would like though the similarity function to be transitive and, thus, we applied the following rule:

Rule 6. if $\mathbf{s(e_1, e_2)=r1}$ and $\mathbf{s(e_2, e_3)=r2}$ and $\mathbf{e_1 \neq e_3}$ and $\mathbf{r_1 * r_2 > T_{sim}}$ then $\mathbf{s(e_1, e_3)=r1*r2}$ (transitivity rule)

Imagine now when we want to figure out if two elements are similar and with what similarity. We would apply first the rules 1 to 4 and if we don't end up with a similarity we will try to apply the transitivity rule. The first time the rule 6 is applied some new similarities are found. This means that if we apply again this rule we will end up with some more similarities. But how many are we going to apply this rule? We will apply this rule until no other similarities can be created (note the criterion $r1*r2 > T_{sim}$). Thus, the number of times (denoted as *n*) the transitivity rule should run depends on the

maximum of *re*, *rb*, *rp*, and *rt* and the threshold T_{sim}. In Appendix B we proved that the following equation which calculates *n* holds:

$$n = \left\lceil \log(T_{sim}) \Big/ \log(\max(re, rb, rp, rt)) \right\rceil \text{ if } \max(re, rb, rp, rt) \neq 1. \qquad \textbf{(VII-3)}$$

For a threshold T_{sim} = 0,5 and the abovementioned weights the number of transitions applied is *n* = 7. Thus, if we apply the transitivity rule more that 7 times no new similarities will be produced.

The abovementioned rules will produce similarities in a ontology, but what happens for different ontologies. In the DBE environment two different ontologies may exist for the same domain (e.g. for Hotels) and if elements of different ontologies are not connected between them no similarities will exist even for identical elements. For example in ontologies two identical classes exist named both Hotel. These two classes are not marked explicitly as equivalent and thus no similarities will be produced by our rules. Although all ontologies belong on the same space and equivalences and sub classing may exist between elements of different ontologies, the general rule in our environment (DBE) does not incorporate these practices. Thus, the abovementioned rules (i.e. (1) to (6)) will not perform well when searching for similarities between different ontologies.

In order to overcome this problem we introduced the following rule to allow automatic extraction of similarities when information about equivalences and sub classing is not present. The rule is based on similarity between strings and is often described in bibliography as the edit distance (also called the Levenshtein edit distance defined at (21)), that is, the minimum number of changes necessary to turn one string into another.

Rule 7. if ***levenshtein*(e$_1$, e$_2$)=r** and **r>T$_{leven}$** and **e$_1$ ≠ e$_2$** and **¬s(e$_1$, e$_2$)** then **s(e$_1$, e$_2$) = f_{leven}*r**

where T_{leven} is a threshold for the levenshtein distance and f_{leven} is a factor in the interval (0, 1].

For example two classes named both Hotel belonging in different ontologies that have no other connection between them will have a Levenshtein distance 1 (their strings are identical). If f_{leven} is 0.7 the two classes will have a similarity s(Hotel, Hotel) = 0.7.

For the string based Levenshtein search we used the Apache Lucene (22) implementation, which is a high-performance, full-featured text search engine library written entirely in Java. It is distributed under the Apache Licence version 2.

ONTOLOGY LINKS

In previous paragraph the similarity function was defined in order to use it for finding similar paths. In this paragraph we will define another function to keep the links between the ontology elements indexed for fast reference. Moreover, these links may be used in future stages for more powerful expansions.

Each time we want to expand a query we don't want to parse and traverse the ontologies repeatedly, because this is time consuming. We want to pre-process each ontology and store the relations of each element into an indexed database in order to have fast access to this information. We call the relations of elements ontology links. For example the class Hotel has a property named AddressPr which has as range the class Address. After the pre-processing two ontology links will be created: Hotel to AddressPr and AddressPr to Address.

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

*Rule 1.*if $\mathbf{p_1}$ **domain** $\mathbf{o_i}$ then $\mathbf{ol(o_i, p_i) = 1}$
*Rule 2.*if $\mathbf{p_1}$ **range** $\mathbf{o_i}$ then $\mathbf{ol(p_i, o_i) = 1}$
*Rule 3.*if $\mathbf{p_1}$ **range** $\mathbf{d_i}$ then $\mathbf{ol(p_i, d_i) = 1}$

Note that these ontology links have the value one.

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

RETRIEVING RELATED PATHS

In previous sections we defined and created similarities in order to be able to find related paths to a given ontology path.

Before continuing to formally define the algorithm for retrieving related paths we will examine an example of what we want the algorithm to find and what should not. In Figure 18 four ontology paths and the similarities between the elements are depicted. The black lines are the ontology links while the red dashed lines are the similarities between the elements. Hotel, Address, City, Hotel2, Address2, City2, Hotel3, City3, Restaurant, and City4 are the ontology objects defined in an ontology. The names are not the best (i.e. none would name an ontology object Hotel2), but can be seen as identifiers of the ontology objects. AddressPr, CityPr, AddressPr2, CityPr2, CityPr3, and CityPr4 are the object properties linking two objects. The datatype properties of City are not shown.

As one can observe a similarity is found (red dashed line) between Hotel and Hotel2 etc. We don't need at this point the actual similarity value, we just need to know that they are similar.

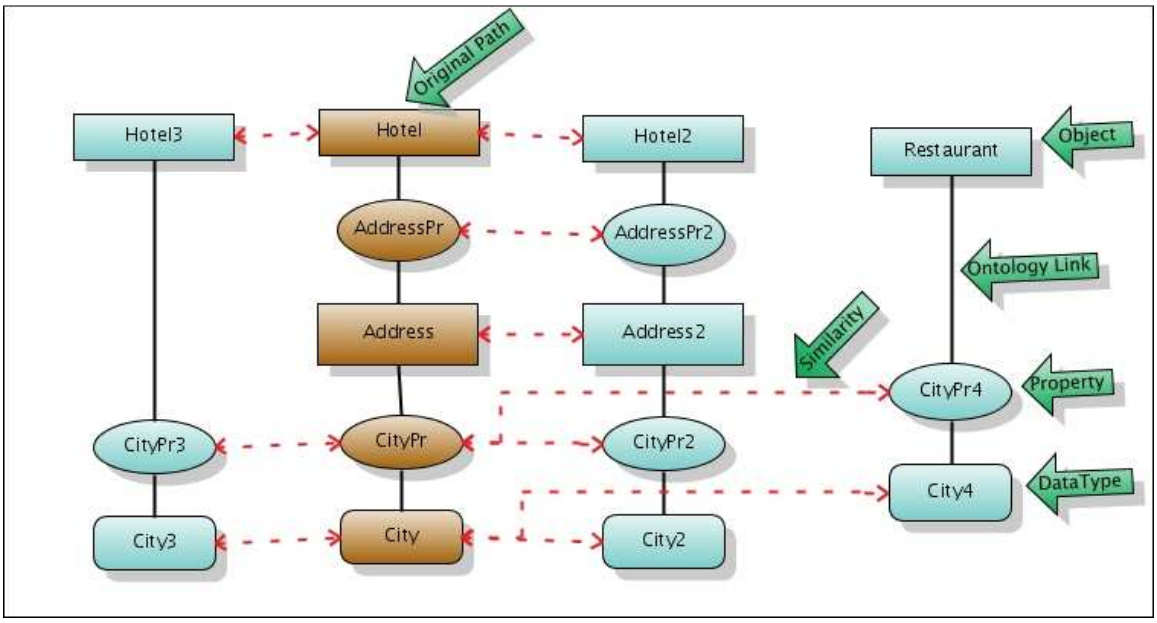

Figure 18: The ontology mappings stored for the main path.

We want to find related paths to Hotel->AddressPr->Address->CityPr->City. Ideally the algorithm should result only these two paths: Hotel2->AddressPr2->Address2->CityPr2 ->City2 and Hotel3->CityPr3->City3. Observe here that Hotel3->CityPr3->City3 misses the Address part, thus the algorithm we need to develop should take these cases into account. Moreover, the algorithm should result a value denoting how related two paths are. This value will be used in equation (VII-1) as the c_i parameter.

In order to continue with the actual process of retrieving relevant paths the following definitions must apply.

Definition 3. We define a path in the ontology space as <>: O^n -> O^n denoting a sequence of ontology elements starting with a class and ending to a datatype:
<e> or **$\{o_i \text{ -> } p_i\}^n$ -> d_i.**

Definition 4. We define the length of a path as the function $l: O^n \text{->} N$ denoting the number of elements which make up the path.

Definition 5. We define the distance between two elements as $\delta: O \times O \text{ -> } [0, 1]$. The complement of the distance (δc) is their similarity value defined earlier:
$\delta c(e, e') = 1 - \delta(e, e') = s(e, e')$ **(VII-4)**

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

Definition 6. We define the distance between two paths as *d: O^n x O^m -> [0, 1]* denoting the distance between the two paths. We are interested in the complement of the distance which is:
***dc* = 1-*d*.** (VII-5)

Definition 7. The distance complement between two paths with only one element is their similarity rate (if exists):
if ***l*(<e>) = *l*(<e'>) = 1** then ***dc*(<e>, <e'>) = s(e, e')** (VII-6)

Definition 8. Two paths as **related** if their distance complement is greater than zero (0):
dc(<e>, <e'>) > 0 then **<e>** and **<e'>** are **related** (VII-7)

Before continuing with calculation of the distance complement, the algorithm for retrieving the related paths will be presented. It is based on the similarities of ontology elements and the ontology links starting from the datatype and looping till the root class is reached. It will first be presented formally and then an example will be given.

FORMAL DEFINITION OF RELATED PATHS DISCOVERER ALGORITHM

Problem: Find the paths (called candidate paths) that are related to a given ontology path called the original path and how much they are related to the original path expressed by a weight in the interval [0, 1]. From the set of candidate paths select those that their weight is greater from a threshold T_{path}.

Before formally describing the algorithm we will examine the previous example of Figure 18 and have an idea of how it is going to produce the desired results. For the example's economy the object properties (AddressPr, CityPr, etc) will be omitted. The original path is <Hotel, Address, City> and we start from the lowest element (originally the datatype but here City). We find the similar elements of City and put them in a set (called frontier). Now the frontier contains {<City2>, <City3>, <City4>}. We now search for similar elements of Address that are linked to any of the paths in the frontier. We end up with just one the <Address2, City2> and we put it the frontier without removing anything (not even City2). Now the frontier contains {<Address2, City2>, <City2>, <City3>, <City4>}. We do the same search we did with Address for Hotel and end up to the final frontier which contains the candidate paths. After we formally present the algorithm we will revisit this example step by step.

The Related Ontology Paths Discoverer (ROPD) algorithm loops in a bottom up fashion the elements of the original path. We call a frontier ***F*** a set of paths related to the original

path. Each loop results in a new frontier ***F*** of the related paths found so far, which will be used for the next steps.

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

Step 1: Create the frontier ***F*** containing the similar datatypes of the datatype **d**.

$F(d) \leftarrow getSimilarDatatypes(d)$

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

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

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

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

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

Step 4: All paths of the last frontier are the candidate paths of the original.

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

The ROPD Algorithm Complexity

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

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

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

The exponential complexity of the algorithm does not pose a problem since the path length is not supposed to take large values.

Note that if step 2c was $F(e_i) \leftarrow getSimilarPaths(F, e_i)$ the complexity would not be exponential but some paths could not be found as will be demonstrated in the following example.

RELEVANT PATH RETRIEVING EXAMPLE

Recall again the example of Figure 18. The original path is: Hotel->AddressPr->Address->CityPr ->City and again we will omit the object properties and the datatypes for simplicity reasons. Thus, the original path is Hotel->Address->City. Note that the same rules apply for the whole path.

In order to construct the first frontier ***F*** (Step 1) we find the similar datatypes of the datatype City. The datatypes are City2, City3, and City4.

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

Next we have the element Address (Step 2a). We search for related elements Address that are associated with City2, City3, or City4 (Step 2b). We come up with only one the Address2 which is related to City2. We add the path <City2, Address2> to the frontier (Step 2c):

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

In the second loop we have the element Hotel (Step 2a).We search for related elements of Hotel that are associated with either path in the frontier (Step 2b). We find two: <City2, Address2, Hotel2> and <City3, Hotel3>. We add them to the frontier:

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

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

Root element reached and the process stops. The frontier of loop 2 has the **candidate paths**. Note that if we didn't add the frontier in each step (which is responsible for the exponential complexity of the algorithm) we couldn't retrieve the path <City3 , Hotel3>.

The last step of the ROPD algorithm is to calculate the weights (path distance) for each path (how similar they are to the original one) and select those which are greater than a threshold T_{path}. In the next section the algorithm for calculating the path distance is discussed.

CALCULATING PATH DISTANCE

In this paragraph we will describe a function for calculating the path distance between each candidate path with the original one. At least two different functions for calculating the path distance exist in the bibliography. The first is from Do at al (23) (f_{Do}) while the second is from Valtchev, (24) (f_{Val}). Let us have a closer look at each of them:

$$f_{Do} = d\left(< e_i >_{i=1}^{n}, < e_j >_{j=1}^{m}\right) = \lambda.\delta(e_n, e_m) + (1-\lambda).d\left(< e_i >_{i=1}^{n-1}, < e_j >_{j=1}^{m-1}\right)$$

which can be written also as:

$$f_{Do} = d\left(< e_i >_{i=1}^{n}, < e_j >_{j=1}^{m}\right) = \sum_{i=1,\, j=1}^{n,m} \lambda.(1-\lambda)^{n-i}.\delta(e_i, e_i) \quad \text{with } \lambda \in [0,1] \qquad \text{(VII-8)}$$

Where $\delta(e_i, e_j)$ is the distance of the two elements (as in Definition 5 of this chapter). The factor $\lambda\ (1-\lambda)^{n-i}$ reduces the impact to the final result of the distance closer to the root element of the path (in our example Hotel). For large values of λ (like 0.9) the differences of the root element to the leaf is large, while for small values (like 0.1) the differences are small. In order to compare the algorithms we used λ = 0.5.

The second function of Valtchev is:

$$f_{Val} = d\left(< e >, < e >'\right) = \frac{\sum \delta(e, e') + |\, l - l'\,|}{l} \qquad \text{(VII-9)}$$

where l and l' are the lengths of the <e> and <e>' respectively and $\delta(e, e')$ as in the previous function. This function operates differently than f_{Do} as it doesn't differentiate the distances of each element. Moreover, the factor $|l-l'|/l$ evaluates as more important the paths with closer lengths.

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

l = 3, δ = 0,1 (for all mapping elements) and λ=0,5.			
#	*Candidate Path*	f_{Do}	f_{Val}
1	**<City2>**	0.05	0.70
2	**<City3>**	0.05	0.70
3	**<City4>**	0.05	0.70
4	**<City2, Address2>**	0.075	0.40
5	**<City2, Address2, Hotel2>**	0.0875	0.10
6	**<City3, Hotel3>**	0.0625	0.40

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

The function f_{Do} cannot be used because is very much dependent on the similarity of the last element of each path, which is not correct in our case because we need a more balanced function. As one can see from the previous table Candidate path 1 (<City2>) has distance from the original 0.05 whereas path 5 (<City2, Address2, Hotel2>) which is much better for our environment has larger distance (0.0875) – which is worse – than path 1. Differences are very small to safely use any threshold to cut of irrelevant paths. This is not acceptable in our environment. We want the distance of candidate paths 5 and 6 to be much better that the rest.

The second function, $f_{Val;}$, does not suffer from the previous problem, but as it can be seen from the previous table the cases 4 and 6 do not produce different similarity values although they are completely different. Case 6 is a complete path whereas case 4 is not. In our environment cases 4 and 6 are completely different and this should be encapsulated in the ideal path distance function.

Thus, in our environment where the similar paths will be used for query expansion we need the path distance function to produce results (difference) that:

1) Candidate paths that have equal length to the original should have smallest differences than others with different lengths. The more close the length of the original path the smaller the differences.
2) Candidate paths that are not complete (the root element is missing – Hotel in the previous example) they should be rated less than complete paths (the difference should be larger). Even more the closer to the root element the candidate paths is the smaller the difference should be.
3) The final difference value of the path should be relative to the similarities of each element participating in the original path. In other words, complete candidate paths

with the same length should have differences to the original path depending on how similar (similarity value) their elements are.

f_{Do} supports only the third criterion while f_{Val} supports criteria 1 and 3. Thus, the function to be used should express in algebra the second criterion. We will be based on the f_{Val} function, but we need to add another factor to encapsulate the second criterion. The new factor will denote how close the last element of the candidate path to the original one is. It will be called the tenacious bondage of two related paths.

Definition 9. The tenacious bond *b: O^l x $O^{l'}$ -> [0, 1]* between a candidate path <e>' to an original one <e> is a real number in the interval [0, 1] denoting how close the last element of <e>' is to the last element of <e>. We calculate *b* as follows:

$$b(< e_i >_{i=1}^{l}, < e_j >_{j=1}^{l'}) = \frac{k}{l} \quad \text{where } k \text{ such that } s(e_k, e_{l'}) > 0 \qquad \text{(VII-10)}$$

where *l and l'* are the lengths of the original <e> and the candidate path <e>' respectively, *s* is the similarity function, and *k* is the largest number in [1, l] such that e_k of the original path is similar to $e_{l'}$ of the candidate path. $e_{l'}$ is the root element of the candidate path. Thus, the element of original path which is related (i.e. their similarity value is greater that zero) to the root element of the candidate path is the e_k.

Examples.

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

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

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

Definition 10. We define the evaluation function of path distance f_{pd} *: O^l x $O^{l'}$ -> [0, 1]* as follows:

$$f_{Pd} = d\left(< e_i >_{i=1}^{l}, < e_j >_{j=1}^{l'}\right) = \frac{\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i)}{l} + (1 - T_{Sim}) \frac{(l - l')}{l} + T_{Sim}(1 - b) \Leftrightarrow$$

$$f_{Pd} = \frac{\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i) + (1 - T_{Sim}) \cdot (l - l')}{l} + T_{Sim} \frac{l - k}{l} \Leftrightarrow$$

$$f_{Pd} = \frac{\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i) + l - l' - lT_{Sim} + l'T_{Sim} + lT_{Sim} - kT_{Sim}}{l} \Leftrightarrow$$

$$f_{Pd} = \frac{\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i) + l - l' - (k - l')T_{Sim}}{l}, \qquad \text{(VII-11)}$$

where *l', l* are the lengths of the original path and the candidate one respectively, *k* such that $s(e_k, e_{l'}) > 0$, T_{Sim} is the similarity threshold (defined in the previous section). *δ, s* are the difference and similarity functions respectively and *b* is the tenacious bond of the two paths. In Appendix B it is proven that this function belongs in the interval [0, 1].

Remember the criteria set to evaluate the difference functions. Criterion 1 demanded that the closer the length of the candidate path to the length of the original path the smaller the difference of the path distance value. This is denoted in f_{Pd} with the following term:

$$(1 - T_{Sim})\frac{(l - l')}{l}$$

Note that in f_{Val} the exact difference was used (*l-l'*), thus adding $1/l$ for each element not present. But we didn't want this factor to be so important tin the final result and thus we add $(1\text{-}T_{Sim})/l$ for each element not present. As T_{Sim} is the minimum allowed similarity, $(1\text{-}T_{Sim})$ is the maximum difference of two elements that are similar.

For the second criterion (the closer the root element of the candidate path to the root element of the original path the smaller the difference should be) the following factor was added:

$$T_{Sim}(1 - b) = T_{Sim}(1 - \frac{k}{l}) = T_{Sim}\frac{l - k}{l}$$

By this factor the difference is larger by T_{Sim}/l for each element that we have to make up in order to reach the root element (context) of the original path. For the paths <Hotel, Address, City> and <City3> we have to make up two elements Address and City in order to say that the paths denote the same context of data.

For the third criterion (to encapsulate the difference of each element participating in the path) we used the same factor as f_{Val} and f_{Do} the:

$$\frac{\sum_{i=1,\, j=1}^{l,l'} \delta(e_i, e_i)}{l}$$

f_{Pd} evaluation function is somewhat similar with f_{Val}. Because our algorithm produces candidate paths with the same or smaller lengths than the original one, but never larger (i.e. $l' \leq l$), equation VII-10 can be rewritten in terms of f_{Val} as:

$$f_{Pd} = f_{Val} - \frac{(k - l')}{l} T_{Sim}$$

Thus, the result of f_{val} in order to encapsulate criterion 2 (remember that we could not use f_{Val} is our environment because it did not satisfy criterion 2) reduces the difference by the factor:

$$\frac{(k - l')}{l} T_{Sim}$$

The following table shows the results of the three difference functions presented in this paragraph. We used $T_{Sim} = 0.6$

***l* = 3, δ = 0.1 (for all similar elements), λ=0.5 and T_{Sim} = 0.6.**						
#	***Related Path***	f_{Do}	f_{Val}	f_{Pd}	***k***	***l'***
1	**<City2>**	0.05	0.70	**0.70**	1	1
2	**<City3>**	0.05	0.70	**0.70**	1	1
3	**<City4>**	0.05	0.70	**0. 70**	1	1
4	**<City2, Address2>**	0.075	0.40	**0.40**	2	2
5	**<City2, Address2, Hotel2>**	0.0875	0.10	**0.10**	3	3
6	**<City3, Hotel3>**	0.0625	0.40	**0.20**	3	2

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

It is clear from the table that our function satisfies all three criteria set. Cases 4 and 6 are differentiated and the rank produced (case 5, case 6, case 4, 3, 2 and 1) is the one inferred from the criteria. Some of the related paths having very large distances can be omitted by using a threshold (the T_{Path} discussed in the previous paragraph). The threshold of 0.3 would result in selecting as similar paths the cases 5 and 6. These will be used later on to semantically expand the query.

SEMANTIC QUERY EXPANSION

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

A query consists of a number of query terms (for example the query "*Hotel.Address.City=Athens and Hotel.Room.Price <100*" consists of two query terms "*Hotel.Address.City=Athens*" and "*Hotel.Room.Price<100*"). The algorithm for expanding queries takes each query term and finds its relevant paths. Each of the relevant paths is assigned a weight with respect to the path distance of the related path. All the relevant paths along with the original query term are joined through an OR node in the query tree named as query branch. Thus, each query term is replaced with a query branch. The query branch has as weight the weight of the original query term. For example for the query term "*Hotel.Address.City=Athens*" we find the relevant path "*Hotel.City=Athens*" and these two form a query branch. The expanded query would look like "(*Hotel.Address.City=Athens OR Hotel.City=Athens) and Hotel.Room.Price <100*".

Although, the weights were not shown in the above example both in the original and the expanded query, assume that the query term "*Hotel.Address.City=Athens*" participated in the original query with the weight 0.9 and the query term "*Hotel.Room.Price<100*" with 0.7. This means that we prefer the result to contain hotels in Athens even if the price of the rooms exceed 100 Euros rather that the opposite (cheap rooms located elsewhere). The weights of the expanded query (assuming that the path "*Hotel.City*" has a path distance of 0.2 and, thus, a similarity of 0.8) would be as follows: the term "*Hotel.City=Athens*" would have 0.8 (the similarity of the path), the "*Hotel.Address.City=Athens*" would have 1 (it is the original term), the "Hotel.Address.City=Athens OR Hotel.City=Athens" would have 0.9 (the weight of the original query term "*Hotel.Address.City=Athens*"), and "*Hotel.Room.Price<100*" would have 0.7 (as in the original query). This will be further demonstrated after the formal presentation of the algorithm for expanding queries.

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

$$qt_i = (p_i, op_i, v_i, w_i) \quad \text{(VII-12)}$$

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

$p = \{modelPath, ontoPath\}$ (VII-13)

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

$qb_i = (w, \{qt_{i1},..., qt_{in}\})$ (VII-14)

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

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

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

The algorithm takes as input the query term to be expanded. It creates a new query branch (line 4). The weight of the branch is the weight of the original query term (line 5). A new query term is created for the original query term with weight 1 (line 6) and it is added to the query branch (line 7). A new query term is added with weight 1 containing only the ontology part (i.e. not the model part) of the original path (lines 8 and 9). The related paths of the ontology part of the original paths are found (line 10) and for each of them a query term is created with weight the path similarity (line 12) and added to the query branch (line 13).

In order to demonstrate the algorithm for constructing query branches we will use the query term:

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

The above query term participates to a larger query but for simplicity reasons it is not mentioned here. It participates to the original query with the weight 0.8 (this weight is assigned by the user). In simple English the query is to find the hotels in Athens. The path "*Hotel.Address.City*" is expressed in ontology terms (i.e. Hotel etc are defined in an ontology rather than a model). The path "*Hotel.Address.City*" is found to have as related paths (recall the previous section): "Hotel2.Address2.City2" with path distance 0.1 (and therefore similarity 0.9) and "Hotel3.City3" with path distance 0.2 (similarity is 0.8).

Figure 19 shows diagrammatically the reformulation (expansion) of the query term. A query branch is created with weight 0.8 containing three query terms: the "*Hotel.Address.City=Athens*" with weight 1 (the original term), the "*Hotel3.City3=Athens*" with weight 0.8, and "*Hotel2.Address2.City2=Athens*" with weight 0.9.

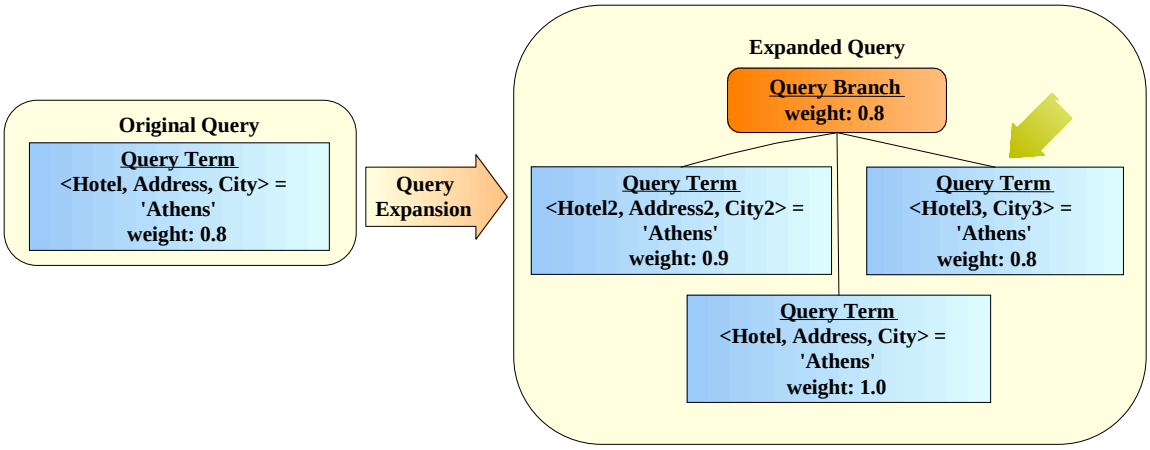

Figure 19: Query expansion example of the query term with path <Hotel, Address, City> with its related paths <Hotel2, Address2, City2> and <Hotel3, City3>.

The use of weights in the query expansion process is strongly associated with the fuzzy information retrieval techniques introduced in chapter "Fuzzy model and query formulation". In that chapter we described the fuzzy framework used to process queries and we reproduce the evaluating formula of the query terms here for convenience:

$$\left(\frac{\sum_{i=1}^{n} (a_i . w_i)^p}{\sum_{i=1}^{n} w_i^{\,p}} \right)^{1/p} \qquad \text{(VII-15)}$$

where w_i is the user defined weight for the query node i and a_i is a weight denoting how relevant is the data source to the query node. Before the query expansion a_i was

calculated using the weights of Table 4 of chapter V depending only on the data value and w_i was a user weight for the query node *i*. Remember here that a query was composed of flat query terms.

After the query expansion the query is composed of query branches which are composed of query terms. Each query term of a query branch will be evaluated separately and the query branch should join these results in terms of the fuzzy information model. In the abovementioned example each of "*Hotel.Address.City=Athens*", "*Hotel3.City3=Athens*", and "*Hotel2.Address2.City2=Athens*" will be evaluated separately against the dataset. If only one evaluates to true the result weight should denote the how relevant the query term was (from the query term weights) with respect to the original user weight (now the weight of the query branch). If though in a dataset more than query terms are evaluated the result weight instead of being a composition of the query terms evaluated to true should only be the best of it. If for example both "Hotel.Address.City=Athens" and "Hotel3.City3=Athens" exist in the dataset we want the result to express only that the original query term was matched (the user does not care that "Hotel3.Address3=Athens" was matched). Thus, only the best result of the query terms should be evaluated. In other terms in the fuzzy model the query branch does not operate as a fuzzy OR but rather as a Boolean OR.

In order to calculate the query branch a_i parameter of formula VII-14 we introduce the following formula.

$$a_i = \max_{j=1}^{k}\left(ad_{ij} * ap_{ij}\right) \qquad \text{(VII-16)}$$

where ad_{ij} is the factor of query term *j* of branch *i* depending on the data value given by Table 4 while ap_{ij} is the weight of the query term *j* which equals to the relevance of the path to the original one (distance complement). Note that by selecting the maximum of all these values we actually use a Boolean OR function between the query terms of each branch.

PARAMETER ESTIMATION

Throughout the semantic expansion process presented earlier in this chapter we used a number of parameters the values of which are crucial for the algorithms to produce reasonable results. These parameters cannot take any random value but should be in some range or have specific values. Some of them actually depend on the value of others.

In this section the mechanism for estimating the parameters' values, the best values and the relationship (correlation) between them is presented.

The methodology for estimating the parameters is to specify some ontologies, give the parameters a large variation of values, calculate the matching elements and based on a query find the equivalent paths. From the paths found we calculate an error metric based on precision and recall. It is quite clear that for the results giving the smallest error we can select the best values for the parameters. We performed three successive experiments, each one refining the previous results.

The first experiment was executed on two very simple ontologies in order to calculate a first range of the parameters. On these ranges and for more parameters but with better refinement a second experiment took place on top of two broader ontologies (still simple). From the second experiment some parameters were calculated when for others correlation between them and the error were found. The last experiment had to do with the same ontologies but with greater refinement and addition of two more parameters. Finally, we executed the algorithm with the estimated values for two given ontologies of the Framework for Ontology Alignment and Mapping[10] (FOAM)(25) for tourism in Russia. The results were good taking into account that the ontology alignment algorithms used on FOAM have different orientation for the results (i.e. the results produced by the algorithm described in this thesis need to be good for query expansion and not to just find good alignments).

The nine parameters used in the ontology mapping and query expansion algorithms that will be estimated are shown in the next table:

Parameter	**Abbreviation**	**Description**
String based weight	SW	Used for the Levenshtein algorithm defined in similarity rule 7 (f_{Leven}).
Similarity threshold	TT	Is the minimum allowed similarity (T_{Sim}). Defined in similarity rule 6. Used by the path distance function (equation VII-10) .
String based threshold	ST	Used for the Levenshtein algorithm defined in similarity rule 7 (T_{Leven})

[10] FOAM is a framework having test ontologies with human mappings and alignments along with ontology alignment software.

Parameter	Abbreviation	Description
Equivalent weight	EQ	The similarity weight of two equivalent elements. Defined in similarity rule 1 (*re*)
Super-class weight	SP	The similarity weight of the parent element to the child element of an IsA relationship. Defined in similarity rule 2 (*rp*)
Sub-class weight	SB	The similarity weight of the child element to the parent element of an IsA relationship. Moreover, it is the similarity weight of two siblings. Defined in similarity rule 2 and 3 (*rb*)
Path threshold	PT	The threshold of distance complement below which a candidate path will not be selected in the set of the related paths. Defined in the ROPD algorithm (T_{Path})

Table 7: The nine parameters used in query expansion algorithms and their abbreviations.

We used Precision, Recall and a total error metric in order to choose best values or ranges. In detail: precision error is the number of paths found that should not be in the result set with such path threshold that that recall is 0 or the smallest possible. Recall error is the number of paths that should be in the result set and cannot be found with any path threshold. Note that in the first two experiments the path threshold is not estimated while it is calculated a min value in order the recall to be the smallest possible. The error metric is a number denoting the distance of the path weights from their desired weight.

THE FIRST EXPERIMENT

For the first experiment we wanted to calculate a first range of the main weights and thresholds. We used two simple ontologies created for the experiment. We wanted the complexity to be low and to test how our algorithm responds to this simple scenario. We want to figure out the ranges of the parameters outside which the algorithm does not work even for the simple ontologies used. The parameters under investigation for this first experiment are SW, TT, ST, EQ, SP, and SB. Moreover, we wanted to figure out which parameters are correlated (the one depends on the value of another). We run our algorithm for 15625 different combinations of parameter values. For each combination

we calculated the minimum path threshold (PT) such that the recall is the smallest possible.

We used the ontologies HotelOnto1 and HotelOnto2 shown on figures 21 and 22 respectively.

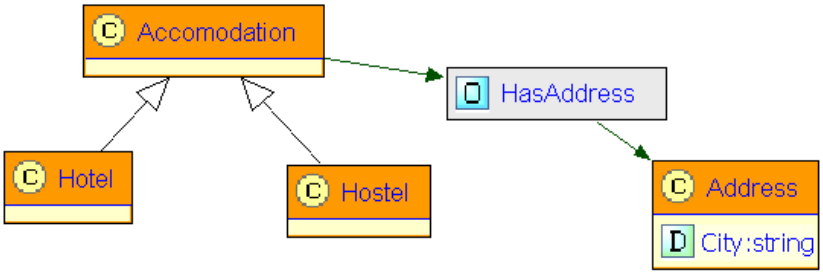

Figure 20: The ontology HotelOnto1 used in the first experiment.

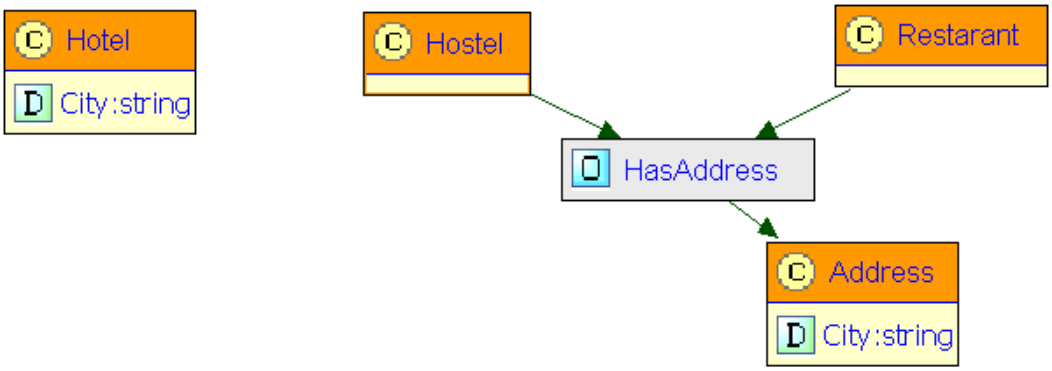

Figure 21: The ontology HotelOnto2 used in the first experiment. No equivalence exists between Hotel and Hostel

The ontologies, although simple, try to capture all kind of information that must be used by the algorithms. In ontology HotelOnto2 no equivalence exists between Hotel and Hostel. The mapping between them must be found using information through HotelOnto1 where Hotel and Hostel are mapped.

The parameters used along with the ranges of their values and the step are shown in table 8. The number of combinations that the experiment will run is 15625.

Parameter	**Abbr.**	**min**	**max**	**Step**
String based weight	SW	0.2	1	0.2
Transient threshold	TT	0.2	1	0.2

String based threshold	ST	0.2	1	0.2
Equivalent weight	EQ	0.2	1	0.2
Super-class weight	SP	0.2	1	0.2
Sub-class weight	SB	0.2	1	0.2

Table 8: The parameters used in the first experiment with their range and step.

The query path we used is of ontology HotelOnto1 and Hotel/HasAddress/Address/City and the desired result set is shown in the next table:

Ontology	**Path**
HotelOnto1	Hostel/ HasAddress/Address/City
HotelOnto1	Accommodation/ HasAddress/Address/City
HotelOnto2	Hotel /City
HotelOnto2	Hostel/ HasAddress/Address/City

Table 9: The good path results of the first experiment.

Experiment's Results

In order to produce the smallest precision and recall errors the following should hold:

- The smallest error (precision and recall) was 0 for a path threshold of 1,
- TT cannot have the value 1,
- When EQ = 1 and ST ≥ 0.8 then SB is irrelevant (this result is due to the very small ontology used),
- When ST is large we get less intermediate results and number of good combinations (of the rest parameters),
- When TT is large we get less intermediate results and number of good combinations, and
- Smallest errors when EQ, SW, SB, SP greater than 0.6.

Experiment's Conclusions

From this experiment we found out that our algorithm actually can work and produce the desired results for the right set of parameters values.

All the results with smallest precision and recall errors were observed when EQ, SW, SB, SP were greater than 0.6. The biggest values the two thresholds (ST and TT) have the better results were found. Finally when TT had value 1 error the algorithm did not produce the desired results for any combination.

In the next experiment we will use this information in order to narrow down the parameter's ranges and refine the values.

THE SECOND EXPERIMENT

For the second experiment we followed the same process with broader ontologies but for smaller parameter ranges. The ontologies used contained all different kinds of connections which can be found in real ontologies (as equivalences, IsA relationships, etc). We want to find the ranges of the parameters for which our algorithm performs best. Moreover, we want to understand how the parameters are correlated (their values depend on others) with others. Again the parameters under test are SW, TT, ST, EQ, SP, and SB.

The ontologies that were used capture all the kind of information need to be used by the algorithms and some traps that should be avoided. The ontologies HotelOnto3 and HotelOnto4 are shown in figures 23 and 24 respectively.

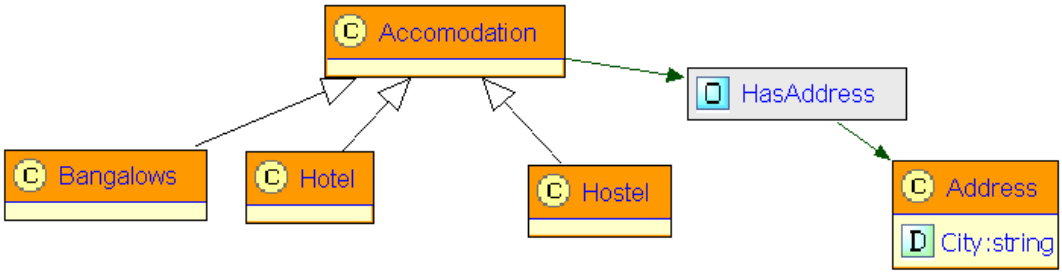

Figure 22: The HotelOnto3 ontology used for the second and third experiment.

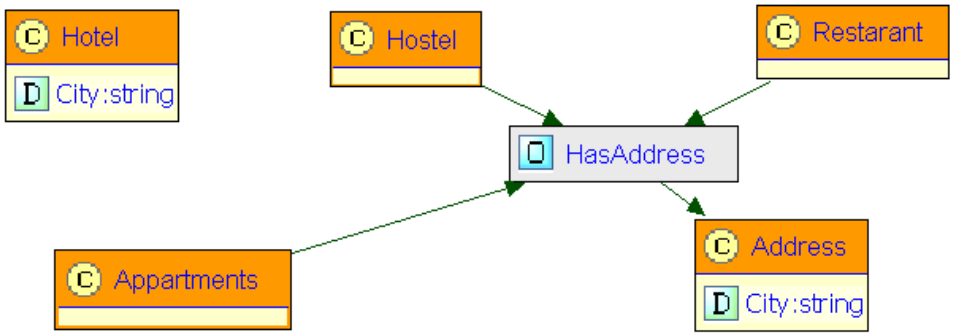

Figure 23: The HotelOnto4 ontology used for the second and third experiment. Hotel and Hostel do not have an equivalence whereas Apartments and Hostel do.

HotelOnto3 and HotelOnto4 are more complicated that HotelOnto1 and HotelOnto2. In HotelOnto4 the Hotel and Hostel do not have an equivalence whereas Apartments and Hostel do. The mappings are more difficult to be found and more easily errors may occur.

The parameters used along with the ranges of their values and the step are shown in table 8. The number of combinations that the experiment will run is 12500.

Parameter	**Abbr.**	**Min**	**max**	**Step**
String based weight	SW	0.6	1	0.1
Transient threshold	TT	0.5	0.9	0.1
String based threshold	ST	0.7	1	0.1
Equivalent weight	EQ	0.6	1	0.1
Super-class weight	SP	0.6	1	0.1
Sub-class weight	SB	0.6	1	0.1

Table 10: The parameters used in the second experiment with their range and step.

The query path we used is of ontology HotelOnto3 and Hotel/HasAddress/Address/City and the desired result set is shown in the next table:

Ontology	**Path**
HotelOnto3	Hostel/ HasAddress/Address/City
HotelOnto3	Accommodation/ HasAddress/Address/City
HotelOnto3	Bungalows/ HasAddress/Address/City
HotelOnto4	Hotel /City
HotelOnto4	Hostel/ HasAddress/Address/City
HotelOnto4	Apartments/ HasAddress/Address/City

Table 11: The good path results of the second experiment.

Experiment's Results

The results of the second experiment are more important and precise than those of the first one. We first have the results for zero precision and recall errors for which there were found 138 succeeding combinations. From these combinations we conclude:

- EQ must be larger than 0.8,
- SW must be larger than 0.8,
- When weights (SW, EQ, SB, and SP) are high then TT can be high (this was expected because the criterion for defining its weight was calculated as $maxWeights^n$, see equation VII-2)

- ST does not differentiate the final result and it does not correlate with any other parameter. The estimation of this parameter is irrelevant of this process with only criterion the good string results. Good values are from 0.6 to 0.8.
- SB and SP cannot be both small, and
- EQ and SW cannot be both small.

In order to observe how each parameter correlates with others and especially how they are correlated to precision recall error we had to calculate correlation matrixes. As bigger the absolute value of a correlation number is the larger the impact it has to the other parameter. Positive numbers denote that these parameters are analogical and negative that they are reverse analogical. The correlation matrixes that the abovementioned conclusions came from are shown to the following tables:

Correlation all data	*EQ*	*ST*	*SW*	*SB*	*SP*	*TT*	*Min PT*	*PrRecall*
Min PT	**-0,129**	1,58E-15	**0,702**	-0,032	0,0366	**0,413**	1	
PrRecall	**-0,276**	2,61E-16	**-0,318**	**-0,235**	**-0,2857**	**0,602**	**0,126**	1

Table 12: Correlation matrix of parameters with the minimum path threshold (PT) and precision recall error on all data.

In table 12 we can see how each parameter correlates to the minimum path threshold and to the precision recall error. From the small values of correlation of ST we come up to the conclusion that it is completely irrelevant to the PrRecall error (the correlation is very small). In bold they are shown the values bigger that 0.1. TT has the biggest impact on the precision recall error from all other parameters investigated.

Correlation good results	*EQ*	*ST*	*SW*	*SB*	*SP*	*TT*	*Min PT*
EQ	1						
ST	-1,78E-17	1					
SW	**-0,238**	-3,99E-18	1				
SB	0,0159	-4,28E-18	0,044879	1			
SP	0,1023	5,2186E-19	0,012689	**-0,20999**	1		
TT	**0,24997**	2,21831E-17	**0,112836**	**0,2759**	**0,253347**	1	
MIN PT	0,09456	1,25901E-17	**0,899565**	**0,1498**	**0,127403**	**0,28885**	1

Table 13: Correlation matrix on good results data only.

Table 13 shows the correlations of parameters for very low precision and recall errors. ST does not correlate to any other parameter. There is large correlation between SW

and min PT (almost 0.9) which means that in most cases with small precision recall error the largest the value of SW resulted in largest value in Min PT.

Correlation good results with max Of TT	*EQ*	*SW*	*SB*	*SP*	*Max Of TT*	*Avg Of MIN PT*
EQ	1					
SW	**-0,2361**	1				
SB	-0,0625	0,014	1			
SP	0,0443829	-0,0149	**-0,3274**	1		
Max Of TT	**0,3675**	**0,11224801**	**0,416655**	**0,369605**	1	
Avg Of MIN-PT	**0,10266**	**0,887718**	0,082826	0,072017	**0,403394**	1

Table 14: Correlation matrix on good results with max of TT. From this matrix is clear that as the greater the weights are the grater the TT can be.

Table 14 shows the correlation of parameters for a result set with small precision recall errors considering the maximum TT value for each combination of the rest parameters. For example for two parameter combinations that the only variance is TT we select the maximum (which produces small precision recall errors).

	SP	*Min Of SB*
SP	1	
Min Of SB	**-0,97014**	1

Table 15: Correlation matrix of SP and SB. From this matrix we can conclude that SP and SB cannot be both small.

Table 15 shows the correlation of two parameters SP and SB. It is clear from the very large negative number (-0.97) that these parameters are highly reverse analogical. If the one is big the other should be small in order to produce results with small precision recall errors.

	EQ	*Min Of SW*
EQ	1	
Min Of SW	**-0,86603**	1

Table 16: Correlation matrix of EQ and SW. From this matrix we can conclude that EQ and SW cannot be both small.

Table 16 demonstrates that EQ and SW are reverse analogical. This has the sense when we cannot find the similarity with equivalence (the parameter is very small) we have to use the string similarities.

The following table shows statistics for the results with precision recall error of zero. Here one can see the smallest and the largest values of the parameters estimated so far. Moreover, the medians and mean values are shown as well.

PrRecall = 0	*EQ*	*SW*	*SB*	*SP*	*Max of TT*	*Avg of min PT*
Mean	0,91087	0,913	0,8333	0,86087	0,596377	0,7614
Standard Error	0,00672	0,0066	0,0113	0,01041	0,008254	0,005
Median	**0,9**	**0,9**	**0,8**	**0,9**	**0,6**	**0,771125**
Standard Deviation	0,0789	0,078	0,13309	0,122	0,097	0,06
Sample Variance	0,00623	0,006106	0,0177	0,0149	0,0094	0,0036
Range	0,2	0,2	0,4	0,4	0,4	0,238
Count	**138**	138	138	138	138	138
Largest	**1**	**1**	**1**	**1**	**0,9**	**1**
Smallest	**0,8**	**0,8**	**0,6**	**0,6**	**0,5**	**0,662**

Table 17: The statistics table of the second experiment for precision recall error zero.

From table 17 we can see that for precision recall error equal to zero what are the smallest and largest values the parameters had. Apart from this, another interesting part is the median which can be seen as the most used value. For example, SP although the smallest and the largest values are 0.6 and 0.9 respectively the median is 0.9. Thus, in very few cases SP had the values 0.6 and 0.7.

If we narrow down the results from the precision recall error of zero (138 different combinations found) with even better criteria (to find the best different combinations) we come up with the previous conclusions but refined. The criteria used to select the best combinations are: the precision and recall error is zero, when the mean deviation of result set paths is 0.9 with error less than 0.05 and the path threshold (PT) is less than 0.9. The qualifying results with these criteria are 49. The main conclusions are:

- SB and SP cannot be large together,
- SB and SP cannot have the values 0.6, 0.7 and 1 together,
- EQ and SW cannot be large together,
- EQ and SW cannot have the values 0.8 and 1 together,
- In 88% of the cases SW has the value 0.9,

- In most cases SP ≤ EQ, and
- PT ranges between 0.65 and 0.9.

These conclusions were derived by the following correlation and statistical matrixes:

	EQ	*SW*	*SB*	*SP*	*Max Of TT*	*First Of MIN-PT*	*ERROR*
EQ	1						
SW	**-0,4419**	1					
SB	-0,0607	0,02435	1				
SP	**0,2026**	0,06679	**-0,66529**	1			
Max Of TT	**0,4392**	0,008471	**0,270501**	**0,3953**	1		
First Of MIN-PT	**0,3507**	**0,67559**	-0,01918	**0,289207**	**0,416214**	1	
ERROR	**-0,3278**	**-0,25798**	-0,19038	**0,399592**	**0,127151**	**-0,5326**	1

Table 18: The correlation matrix of the second experiment for precision recall error zero and mean deviation of results 0.9 with error less than 0.05.

Table 18 shows the correlation between the parameters. SB and SP are reverse analogical and the same holds for EQ and SW. EQ and TT are analogical.

	EQ	*SW*	*SB*	*SP*	*Max of TT*	*Min PT*	*Error*
Mean	**0,916326**	**0,88979**	**0,8142**	**0,7979**	**0,5632**	**0,752**	**0,0378**
Standard Error	0,01179	0,00667	0,0186	0,0181	0,0122	0,0058	0,001
Median	0,9	0,9	0,8	0,8	0,6	0,757	0,0383
Standard Deviation	0,08253	0,04675	0,1307	0,1266	0,0859	0,0405	0,0075
Sample Variance	0,0068	0,002185	0,017	0,01604	0,0074	0,0016	5,65E-05
Range	0,2	0,2	0,4	0,4	0,3	0,138	0,0302
Count	**49**	49	49	49	49	49	49
Largest	**1**	**1**	**1**	**1**	**0,8**	**0,9**	**0,0498**
Smallest	**0,8**	**0,8**	**0,6**	**0,6**	**0,5**	**0,662**	**0,0195**

Table 19: The statistics matrix of the second experiment for precision recall error zero and mean deviation of results 0.9 with error less than 0.05.

In table 19 a more refined range of the parameters can be found than table 17. The mean and median values of each parameter can be found.

Experiment's Conclusions

This experiment resulted for the most parameters to specific range of values that can be used. By the correlation between the parameters we can understand aspects of the algorithm.

First of all, equivalence weight (EQ) and string weight (SW) both vary between 0.8 and 1, while they cannot have both the values 1 and 0.8. For the algorithm and the specific ontologies used this means that when EQ is small (0.8) then alternative similarities can be found by using string matching (SW is high). But when both are small then the similar elements cannot be found. Thus, for both a value above 0.85 could be used. In most cases SW has the value 0.9.

SP and SB both vary between 0.6 and 1. They cannot be both small or large, and thus a value larger than 0.75 and smaller than 0.9 should be used for both. Note that the median and mean values for both are about 0.8.

The similarity threshold (TT) varies between 0.5 and 0.9 and it highly analogous to the precision recall error and the path threshold (PT). Most cases show that when TT is about 0.6 very could results are produced.

Although we didn't use a path threshold (PT) (in each case we measured the PT that contained all the desired results) we were able to find out that its range is between 0.65 and 0.9. Thus, in order to have recall zero the smallest threshold that should be used was 0.65. There is no need to be smaller. More details will be found on the next experiment where the PT will be used to cut of paths.

Finally, the string threshold (ST) is irrelevant to the outcome of the algorithm. The value assigned for it can be from 0.6 to 0.8 in order the string relationships to be good.

THE THIRD EXPERIMENT

In the previous experiments the path threshold (PT) parameter was calculated as to be the maximum value that recall was 0. In the third experiment its value will vary in the most common values it had during the last experiments (from table 19 can be seen to have values from 0.65 to 0.9).

Next table shows all the parameters and the range they vary along with the interval step. The variance comes from previous experiments and some refinement is needed on the step. Note that we use the ontologies of the second experiment (HotelOnto3 and HotelOnto4) with the same query and good results.

	min	max	Step
SW	0,85	0,95	0,05
TT	0,6	0,6	
ST	0,6	0,6	
EQ	0,8	0,95	0,05
SP	0,75	EQ	0,05
SB	0,75	0,9	0,05
PT	0,65	0,9	0,05

Table 20: The parameters of the third experiment and their variance.

There are 2100 different value combinations. From these and for correlation of the parameters with the precision recall error and the value error two important conclusions are made and the variance of the parameters is refined. We will see each of them with the correlation and statistics matrixes in the rest of this section.

Experiment's Results

In the next correlation matrix of all the cases we can observe the correlation of the parameters with precision recall error and mean deviation error:

All Data	*EQ*	*SW*	*SB*	*SP*	*PT*	*ERROR*	*PrRecall*
ERROR	-0,085	**-0,1798**	**0,24358**	**0,5915**	-0,05528	1	
PrRecall	-0,0439	0,05914	-0,01704	0,00586	**-0,27357**	0,00169	1

Table 21: The correlation matrix of all cases of the third experiment where we can see the high correlation of BW with precision recall error.

Form table 21 we can observe that the SP parameter is high analogous to the error, and its value should be kept small. PT is reverse analogical to the precision and recall error which is something we expected. As the threshold goes high the precision is better.

Next follow three statistical matrixes each one for different data sets according to criteria regarding precision recall error and mean deviation error.

precision recall error = 0	*EQ*	*SW*	*SB*	*SP*	*PT*	*ERROR*
Mean	0,8968	0,87658	0,823418	0,825	0,7	0,043273

precision recall error = 0	*EQ*	*SW*	*SB*	*SP*	*PT*	*ERROR*
Standard Error	0,00291676	0,0014	0,003525	0,00315	2,5E-16	0,000377
Median	0,9	0,9	0,8	0,825	0,7	0,042626
Standard Deviation	0,051849	0,02499	0,062658	0,05599	4,45E-15	0,006697
Sample Variance	0,00268	0,000624	0,003926	0,003135	1,98E-29	4,48E-05
Range	0,15	0,05	0,2	0,15	0	0,034754
Count	116	116	116	116	116	116
Largest	0,95	0,9	0,95	0,9	**0,7**	0,061931
Smallest	0,8	0,85	0,75	0,75	**0,7**	0,027177

Table 22: The statistical matrix for data where precision recall error equals to zero for the third experiment.

In table 22 116 combinations of parameters were found having precision recall error equal to zero. The most interesting part in this table is the range of PT; all the values of in the dataset had the PT equal to 0.7! (i.e. PT is calculated unambiguously). The rest parameters in the most cases take specific values (the mean values).

precision recall error = 0 & error < 0.04	*EQ*	*SW*	*SB*	*SP*	*ERROR*
Mean	**0,907798**	**0,8825688**	**0,81055**	**0,784404**	0,036307
Standard Error	0,004616	0,0022927	0,005684	0,003839	0,00028
Median	**0,9**	**0,9**	**0,8**	**0,8**	0,036975
Standard Deviation	0,0481963	0,023936787	0,059346	0,040079	0,002918
Sample Variance	0,0023229	0,00057297	0,003522	0,001606	8,52E-06
Range	0,15	0,05	0,2	0,15	0,012819
Count	42	42	42	42	42
Largest	0,95	0,9	0,95	0,9	0,039996
Smallest	0,8	0,85	0,75	0,75	0,027177

Table 23: The statistical matrix for data where precision recall error equals to zero and mean deviation error is less than 0.04 for the third experiment.

In table 23 the criteria of the dataset are stricter with precision recall error equal to zero and mean deviation error less than 0.04. The cases these two criteria apply are 42 and we can observe that minimum and maximum values do not change from the previous

case but the mean of each of the parameters is refined to a better value. As the dataset becomes smaller the parameters tend to have certain stable values.

In the last dataset there are the combinations where the precision recall error is zero and the mean deviation error is very small (less than 0.003) and has only 20 cases.

precision recall error = 0 & error <0.003	*EQ*	*SW*	*SB*	*SP*
Mean	0,925	0,8975	0,8125	0,775
Standard Error	0,007694838	0,0025	0,01399	0,007695
Median	0,95	0,9	0,8	0,75
Standard Deviation	0,03441236	0,01118034	0,062566	0,034412
Sample Variance	0,001184211	0,000125	0,003914	0,001184
Range	0,1	0,05	0,2	0,1
Count	20	20	20	20
Largest	0,95	0,9	0,95	0,85
Smallest	0,85	0,85	0,75	0,75

Table 24: The statistical matrix for data where precision recall error equals to zero and mean deviation error is less than 0.003 for the third experiment.

We can observe from the previous matrix that the mean is further refined for all parameters.

Experiment's Conclusions

The main conclusion of this experiment is the value of the path threshold (PT) parameter, which is 0.7.

Due to the mean and maximum and minimum values we have the next table where the actual best values are calculated.

	min	Max	Best
SW	0,85	0,9	**0,9**
EQ	0,8	0,95	0,95
SP	0,75	0,85	0,75
SB	0,75	0,95	0,8
PT	0,7		**0,7**
TT	0.6		**0.6**

	min	Max	Best
ST	0.6		**0.6**

Table 25: The calculated possible and best values of all parameters.

MAPPING ALGORITHM EVALUATION

In the previous section we tested our algorithm for finding similar paths by using two small ontologies created by us in order to define the ranges of the parameters used. Moreover we find out the algorithm produces results with small precision and recall errors. The ideal evaluation of our algorithm could be to use some real-world ontologies in order to find related paths.

Although, we were able to find real-world ontologies that had their elements were mapped semantically by humans (see next paragraphs), we were not able to find some work for retrieving related ontology paths. Thus, we could only evaluate the part for finding similar ontology elements. Note here that is not the best evaluation as our criteria to find similar elements was not only to find semantically similar elements but also to find similar paths for expanding queries. For example for in HotelOnto3 of the previous section depicted in Figure 22 element Hotel and its sibling Hostel by a human may be not denoted as semantically similar, while in our environment we need them to be related in order to expand the query with similar terms. Thus, as the applications of the algorithms differ we cannot measure effectively our algorithms. What is crucial and can be evaluated is the recall of our algorithm. Is our algorithm for finding similar ontology elements able to retrieve all the ontology elements mapped by humans as similar? This is the case we can evaluate. The precision of our algorithm cannot be evaluated as we need the broader range of elements in order to perform query expansions.

In order to evaluate the mapping algorithm against the first we used some test ontologies of the Framework for Ontology Alignment and Mapping (FOAM) (see [25]) which include human mappings and tested the algorithm in this thesis against them. The ontologies were RussiaA and RussiaB referring to tourist ontologies for Russia. RussiaA has 153 classes and RussiaB 470. Three datasets with mappings exist:

1. The human mappings
2. The mappings found from standard algorithms (KAON2)
3. The mapping produced by the algorithm in this thesis

Note that the human mappings and the mappings from the KAON2 algorithm include mappings of instances, instead of just classes. Our system does not work with ontology instances as they are model data, thus the mappings for instances were not compared. Another difference is that the other mappings do not have mappings between object and datatype properties, thus the comparison will be made only with classes. The last difference is that their mappings are between classes of different ontologies whereas our algorithm finds mappings between classes of the same ontology. Thus only common kinds of mappings will be compared, i.e. no instances, no object or datatype properties and no mappings between classes of the same ontology.

The next table shows the results of the three procedures and the comparison between them.

	Number of mappings	**Common mappings with human**	**Lost mappings**	**Mappings not listed in the human set**	**Common mappings with KOAN2 algorithm**
human mappings:	63				
KOAN2 mappings:	69	48	15 (23,8%)	21	
This thesis algorithm mappings:	228	**59**	**4 (6.35%)**	169	46
This thesis algorithm best mappings:	87	**48**	**15 (23,68)**	39	38

Table 26: Comparison of ontology element mappings with manual mappings.

As one can observe our algorithm found more common mappings with the human mappings than the KOAN2 algorithm did. One can also observe that our algorithm found about three times more mappings (228) than both the human and the KOAN2. One could argue that for such a small improvement the redundant mappings are a large cost and disadvantage. But this evaluation was not performed for the precision because is designed not to find just similar elements but also elements that can replace other query terms (as previously mentioned Bungalows with Hostel). The human mappings were made with the rationale “what would be the best way to describe one class of the first ontology with another one of the second?” while in our algorithm the mappings are produced to answer a different question “with which classes is it reasonable to replace one class?”.

When comparing this thesis best matches results with the manual mappings the results are equivalent with the KOAN2 algorithm (a slightly worst precission).

The important conclusion of the experiment is that the human mappings were found with a reasonable error (6.35% elements lost). Another interesting point is that only two of the 48 of the KOAN2 mappings which were also in the human mappings list were not found by our algorithm, which means that our algorithm performs much better than the KOAN2 algorithm in this scope.

SUMMARY

In this chapter we presented the algorithms used for the semantic query expansion. In order to perform the semantic query expansion we introduced three algorithms: the ontology similarity analyzer, the algorithm for the retrieval of related ontology paths and, finally, the algorithm for expanding the queries with the related paths. Moreover, we performed three experiments to estimate the parameters used and an evaluation for the ontology similarity analyzer.

In order to find the similar ontology elements that can be used to semantically expand queries we introduced a set of rules (the similarity rules) which took advantage the equivalence, the IsA, the ontology types semantics defined in an ontology along with string matching in order to be able to find similar ontology elements defined in separate ontologies. Moreover, we introduced the transitivity rule which means that two elements that are similar to a third one are similar between them too. For all these rules parameters and thresholds were used in order to capture the similarity between two elements as real number in the interval [0, 1].

The similarities between ontology elements along with ontology links (capture the HAS relationship) are preprocessed for each ontology entered in the system and are stored in the XML database for fast querying and retrieval.

Next the algorithm for retrieving similar ontology paths to given one was introduced. This algorithm exploits the information of similar ontology elements produced before in order to step by step find all related paths. The interesting in this algorithm is that it can find similar paths that might miss some elements (for example “Hotel.Address.City” is similar to “Hostel.City”). A function to calculate the path distance was introduced which produces results which fulfill three criteria: (1) the more similar the elements of the two paths are, the less the path distance is, (2) the more close the lengths of the paths are the

smaller the distance is, and (3) the closer the contexts of the paths are (denoted by the root elements of the paths) the smaller the path distance is. The function was compared with two known path distance functions found in the bibliography.

The algorithm that expands a query term with similar query terms and the function that incorporates the similarities in the fuzzy information model introduced in chapter V was presented next. Each query term is replaced by a set of query terms (the original query term, along with similar ones). Between these query terms a Boolean OR is performed (instead of a fuzzy OR).

All the parameters and thresholds used in this chapter were estimated in ranges of values through three experiments held with test ontologies. We created four simple ontologies and tried for precision and recall error to find what the acceptable values these parameters can take are. For all parameters strict ranges where found and how they correlate with the precision and recall error and how they correlate between them. Another conclusion from the tree experiments is that the algorithms can produced the desired results (find the related paths and exclude unrelated).

Finally, we evaluated the ontology similarity analyzer against two known ontologies which human mappings between their elements were given. The algorithm finds the most of the human mappings (recall error is very small). Another algorithm tested with these ontologies had a very large recall error. The conclusion is that the algorithm for ontology similarities is able to find the human mappings.

CHAPTER VIII – CONCLUSIONS

In this thesis we have described the mechanisms and algorithms implemented in order to support knowledge access and discovery over the fully decentralised P2P network of the DBE. This includes the enhancement of the DBE framework with those mechanisms and characteristics that would allow the exploitation of all the advantages of the P2P network while facing the common problems and challenges of such networks. This implementation was demonstrated on January 2005, January 2006 and April 2007 during the DBE audits, as part of the integrated prototype of DBE. Moreover, it is used by a large number of SME across Europe which gave feedback during the design and implementation process for features and potential uses of the recommender.

Knowledge in DBE is coming from contextualization and personalization of information. 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. Personalisation is supported with profiles of users (WP7 "User Profiling") 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; it manages 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 that we described in this thesis provide the underlined mechanism for querying metamodels, models and instances of DBE. It also forms the basis for supporting recommendations. At a technical level the knowledge access approach is uniform for both desired functionalities. The core of the knowledge access functionality is the Query Metamodel Language (QML) which is a language based on OCL2.0, adapted for MOF1.4, and extended to allow similarity matching in order to accommodate user preferences. The implementation provides a framework for QML query processing that incorporates IR functionality and the theoretical approach is based on the Extended Boolean Model.

The querying mechanisms where improved in order to become more flexible and capable of fully supporting the query formulation needs on the one hand and the query answering mechanisms on the other hand. To these needs, the exploitation of rich domain specific ontologies was added. The query analysis had to be supported by an

ontology mapping mechanism and now, query expansion takes place to facilitate the information retrieval needs based again on the Extended Boolean Model. The algorithm for finding relevant ontology paths is one of the major contributions of this thesis.

Finally, the incorporation of the above mentioned querying mechanisms in the DBE services for each node and the deployment of the system in the P2P environment enhanced the P2P function of the Recommender. However, the need for an efficient query processing and routing mechanism was thereafter, imperative. The DBE framework, in accordance with the DBE P2P infrastructure, set the basis for a semantically strong manipulation of the semantic overlay DBE network. The common DBE metamodels and the capability to use rich ontologies for the description and management of knowledge in the DBE, plays a decisive role in the sharing and discovery of knowledge.

In detail the main contributions of this thesis are discussed on the following paragraphs.

Query Metamodel Language

The Query Metamodel Language (QML) was introduced in this thesis. QML is actually an alignment of OCL 2.0 to MOF (OCL is bounded to UML). It is defined in terms of the MOF architecture as a M2 layer Metamodel and can query data on M2, M1, and M0 layer. It can also express constraints for M3 MOF Meta-metamodel. QML is very powerful as it is not bound neither to specific models or metamodels but has a generic way to express queries.

QML queries are bound to meta-information and thus semantics of this meta-information is used to enhance semantically the query results.

Semantic Annotation & Query Analysis

The queries are semantically annotated and validated against the model or metamodel used into the query during the Query Analysis process. Two query evaluation engines were developed in this thesis. The first one is the DBE Recommender where QML is used in a strict way (because XQueries need to be automatically generated by the QML queries), but the implementation is scalable and fast because data is stored in an XML repository. The second implementation, not used in DBE, is an evaluation engine on top of the Metadata Repository (MDR). While the second implementation makes full use of QML, the MDR is not scalable and does support indexes. Thus, in DBE we formalized the

kind of queries to be supported. These query templates are constructed using the Query Formulator in order to support fuzzy information retrieval techniques.

Query Formulation

Query Formulator is developed as an API for creating QML expressions; a GUI used this API to create QML expressions. On top of the Query Formulator an Advance version of it was created to offer in an easy to use manner complex constructs. Query templates can be created and reused to create queries. This module offers the ability to create template for example for querying hotel data based on a model by an expert. This template can then be distributed and be used by many either by legacy systems to query DBE Knowledge Base or by simpler user interfaces. For example with the Query Formulator one has to select the term Hotel.City and then enter values like “Athens” and “Volos”, while with the Advanced Query Formulator someone has selected the main parts of querying hotels and created a template. Then the user enters just “Athens” on the City field. This automates the querying process a lot and simplifies the use of the language.

While both Query Formulator and Advanced Query Formulator are used to construct structured QML queries the Keyword Formulator is used to parse semi structured (ex. room.price < 70) and unstructured (ex. “Hotel” or “Finland”) query expressions and formulate them into QML queries. These semi structured QML queries will be enhanced with semantics (if possible) and processed as common QML queries.

The Fuzzy Model

All the query terms are expressed inside QML with weights. The final results are ranked with information retrieval techniques explained in this thesis. Specifically the p-norm model was used to rank the results. An information retrieval model for the specific application was given. This model expands the result set with data that are not exactly the same as the query suggested. For example for the query Hotel.Room.Price < 70 the result set will be expanded with Rooms with price greater than 70 with less rank.

Note that the queries will be further expanded semantically (i.e. with similar terms of other (or the same) models) by using ontologies. For example Hotel.Room.Price < 70 will be expanded with Motel.Room.Price < 70 coming from another model with smaller weight.

Ontology Similarity Analyzer

In order to find the similar ontology elements that can be used to semantically expand queries we introduced a set of rules (the similarity rules) which took advantage the equivalence, the IsA, the ontology types semantics defined in an ontology along with string matching in order to be able to find similar ontology elements defined in separate ontologies. Moreover, we introduced the transitivity rule which means that two elements that are similar to a third one are similar between them too. For all these rules parameters and thresholds were used in order to capture the similarity between two elements as real number in the interval [0, 1].

The similarities between ontology elements along with ontology links (capture the HAS relationship) are preprocessed for each ontology entered in the system and are stored in the XML database for fast querying and retrieval.

We evaluated the ontology similarity analyzer against two known ontologies which human mappings between their elements were given. The algorithm finds the most of the human mappings (recall error is very small). Another algorithm tested with these ontologies had a very large recall error. The conclusion is that the algorithm for ontology similarities is able to find the human mappings.

Retrieval of Related Ontology Paths

An algorithm for retrieving similar ontology paths to a given one was introduced. This algorithm exploits the information of similar ontology elements produced earlier in order to step by step find all related paths. Note that this algorithm can find similar paths that might miss some elements of the original one (for example “Hotel.Address.City” is similar to “Hostel.City”). A function to calculate the path distance was introduced which produces results which fulfill three criteria: (1) the more similar the elements of the two paths are, the less the path distance is, (2) the more close the lengths of the paths are the smaller the distance is, and (3) the closer the contexts of the paths are (denoted by the root elements of the paths) the smaller the path distance is. The function was compared with two known path distance functions found in the bibliography.

All the parameters and thresholds used were estimated in ranges of values through three experiments held with test ontologies. We created four simple ontologies and tried for precision and recall error to find what the acceptable values these parameters can take are. For all parameters strict ranges where found and how they correlate with the

precision and recall error and how they correlate between them. Another conclusion from the tree experiments is that the algorithms can produced the desired results (find the related paths and exclude unrelated).

Semantic Query Expansion

The algorithm that expands a query term with similar query terms and the function that incorporates the similarities in the fuzzy information model introduced in chapter V was presented next. Each query term is replaced by a set of query terms (the original query term, along with similar ones). Between these query terms a Boolean OR is performed (instead of a fuzzy OR).

APPENDIX A – THE QML FORMULATION API

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.context declarations.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 operation, e.g. "=", 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 operation, e.g. "=", and a String value.
org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateExpressions**(java.util.Collection expressions, int type) Formulates a conjunctive or disjunctive OCL expression

org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateFuzzyExpression**(java.util.Collection exprs, double[] weights, int type) Formulates a fuzzy conjunctive or disjunctive 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 expressions, int type) Formulates a conjunctive or disjunctive OCL expression

org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**formulateFuzzyExpression**(java.util.Collection expressions, double[] weights, int type) Formulates a fuzzy conjunctive or disjunctive OCL expression
org.dbe.kb.metamodel.qml.contextdeclarations.QueryContextDecl	**getQuery**(java.lang.String name, org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression exp, java.lang.String result) Constructs QML expressions for fuzzy query

Class ModelQueryFormulator

Title: QML Formulation API

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

Constructor Summary

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

Functionality

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

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

Class InstanceQueryFormulator

Title: QML Formulation API

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

Constructor Summary

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

Method Summary

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

org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression	**refineInstanceQuery**(org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression hard, org.dbe.kb.metamodel.qml.ocl.expressions.OclExpression soft)

Class AdvancedQueryFormulator

Title: Advanced QML Formulation API

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

Field Summary

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

Constructor Summary

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

Functionality

org.dbe.kb.metamodel.qml.contextdeclarations.QueryContextDecl	**getQuery**(QueryExpr[] expressions) Creates and returns a QueryContextDecl class.
Template	**getTemplate**() Gets the template of the formulator.

void	**setTemplate**(Template template) Sets a template to the formulator

Class QueryExpr

Title: Advanced QML Formulation API

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

The objects of this class are actual query expressions

Constructor Summary

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

Functionality

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

Class Template

Title: Advanced QML Formulation API

Description: An Advanced API for formulating QML queries and Expressions

(**org.dbe.kb.qi.adv**)

This class denotes a reusable query component.

Constructor Summary

Template()

Method Summary

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

Class TemplateElement

org.dbe.kb.qi.adv

Constructor Summary

TemplateElement()

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

Functionality

javax.jmi.model.MofClass	**getContext**()
java.lang.String	**getDelimeter**()
java.lang.String	**getId**()
java.lang.String	**getPath**()
java.lang.String	**getType**()
void	**setContext**(javax.jmi.model.MofClass context)
void	**setDelimeter**(java.lang.String delimiter)
void	**setId**(java.lang.String id)
void	**setPath**(java.lang.String path)
void	**setType**(java.lang.String type)

APPENDIX B – MATHEMATICAL PROOFS

In this appendix two proofs are demonstrated.

Proof 1. We will show that the function (originally presented in chapter VII as equation VII-10):

$$f_{Pd} = \frac{\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i) + l - l' - (k - l')T_{Sim}}{l}, \qquad \textbf{(B-1)}$$

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

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

$$f_{Pd} \leq \frac{l' + l - l' - (k - l')T_{Sim}}{l} = \frac{l - (k - l')T_{Sim}}{l} = 1 - \frac{(k - l')T_{Sim}}{l}, \qquad \textbf{(B-2)}$$

Moreover, remember the definition of *k* in chapter VII from which we occlude that the maximum value of *k* is *l* and the minimum is *l'*, i.e.:

$$l' \leq k \leq l, \qquad \text{(B-3)}$$

From (B-3) we have:

$$l' \leq k \Leftrightarrow 0 \leq k - l' \Leftrightarrow 0 \leq \frac{k - l'}{l} T_{Sim} \Leftrightarrow -\frac{k - l'}{l} T_{Sim} \leq 0 \Leftrightarrow 1 - \frac{k - l'}{l} T_{Sim} \leq 1 \Leftrightarrow f_{Pd} \leq 1$$

The first part is proven.

$$\sum_{i=1,j=1}^{l,l'} \delta(e_i, e_i) \geq 0, \qquad \text{(B-4)}$$

and $k \leq l$,

so we have:

$$f_{Pd} \geq \frac{0 + l - l' - (k - l')T_{Sim}}{l} \geq \frac{l - l' - (l - l')T_{Sim}}{l} = \frac{(l - l')(1 - T_{Sim})}{l}$$

And because (l-$l' \geq 0$) and $T_{Sim} \leq 1$ we have

$$f \geq 0$$

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

Proof 2. We will show the upper bound of the number of transitions required for rule 5 of section 0 is given by the formulae:

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

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

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

$$\mathbf{r_{final} = r_1 * r_2 * \ldots r_n = T} \qquad \textbf{(B-6)}$$

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

$$\mathbf{r_{max} = max(}\boldsymbol{re, rb, rp, rt}\mathbf{)} \qquad \textbf{(B-7)}$$

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

$$r_{final} = r_{\max}^{n} = T \qquad \textbf{(B-8)}$$

Finally if we use logarithms we get:

$$n = \left\lceil \log(T) \Big/ \log(r_{\max}) \right\rceil \qquad \textbf{(B-9)}$$

Q.E.D.

APPENDIX C – THE KEYWORD EXPRESSIONS PARSER GRAMMAR

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

The grammar of the keyword expressions parser is as follows:

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

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

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

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

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

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

      {
```

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

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

Object Factor() :
{
      java.util.Vector factorimage = new java.util.Vector();
      String s;
```

```
}
{
   s=Simple()
      {
        return s;
      }
|
<STR> s=Simple()
      {
        factorimage.add(s);
      }
( s=Simple()
      {
        factorimage.add(s);
      }
)* <STR>
      {
           return factorimage;
      }
}

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

```
  t=<NUM>
      {
        return t.image;
      }
|
  t=<FLOAT>
      {
        return t.image;
      }
}
```

GLOSSARY

Term	Description
API	Application Programming Interface: Is a technology that facilitates exchanging messages or data between two or more different software applications
BML	Business Modelling Language
DBMS	Database Management System: A software system that allows efficient manipulation (storage, organization, indexing, and querying) of large amounts of data.
EvE	Evolution Environment: It is where the services are evolved based in order to reach the best fitness point.
ExE	Execution Environment: It is where services live, where they are registered, deployed, searched, retrieved and consumed. This parallel word is sometimes referred to as the "runtime of the DBE".
IR	Information Retrieval: Technology for retrieving personalized information from large collections of unstructured, semi-structured, or structured data.
JCP	Java Community Process: The "home" of the international developer community whose charter it is to develop and evolve Java technology specifications, reference implementations, and technology compatibility kits
JDBC	Java Data Base Connectivity: A technology that provides cross-DBMS connectivity to a wide range of relational databases and access to other tabular data sources, such as spreadsheets or flat files
JMI	Java Metadata Interface: A Java Community Process (see JCP description) specification of a standard Java API (see description of API) for metadata access and management based on the MOF specification.
KB	Knowledge Base: Is the part of the DBE system where the DBE knowledge is stored and managed. Such knowledge refers to ontologies, business and service descriptions, etc.
KB Service	Knowledge Base Service: A Service on top of the DBE Knowledge Base that provides functionality for storing and retrieving models.

Knowledge Access Module	A component used to provide uniform access to the DBE Knowledge.
MDA	Model Driven Architecture: An approach (proposed by OMG) to IT system specification that separates the specification of system functionality for the specification of the implementation of that functionality on a specific technology.
MDR	Meta-Data Repository: MDR implements the OMG's MOF standard based metadata repository based on the JMI specification (see JMI description)
MOF	Meta Object Facility: A generalized facility and for specifying abstract information about very concrete object systems.
MOF Repository	A Repository for storing, managing and retrieving meta-data (models) and meta-meta-data (metamodels) that have been described with MOF.
OCL	Object Constraint Language: OMG's standard for expressing constraints and well-formness rules on object models. The last release is also considered suitable for querying object models.
ODM	Ontology Definition Metamodel: A MOF model (metamodel) developed in DBE for ontology representation.
OMG	Object Management Group: International standardization body
P2P	Peer-To-Peer
PIM	Platform Independent Model of a modelled system
PSM	Platform Specific Model of a modelled system
QML	Query Metamodel Language: It is a Knowledge Access Language developed in DBE in order to provide uniform access to the various kinds of DBE knowledge.
Query Analyzer	A component of the Knowledge Access Module that is used to analyze queries against the metamodel (used for knowledge representation) specific semantics.
Query Code Executor	A component used (by the Knowledge Access Module) to execute the generated query code
Query Code Generator	A component of the Knowledge Access Module that takes as input the query syntax tree and generates the code to be executed in the appropriate query language
Query Execution Plan Constructor	A component of the Knowledge Access Module that evaluates the QML expressions already analyzed into a syntax tree representation.

Query Formulator Tool	A front-end tool developed in DBE for allowing the user to formulate queries against the DBE knowledge using a tree-view representation of the Knowledge Structure.
RDBMS	Relational Data Management System: A DBMS (see DBMS description) based on the relational model.
Recommender	A DBE (autonomous) Core Service that will provide users (SMEs) with personalized knowledge by exploiting their profiles
SDL	Service Description Language: A MOF model (metamodel) that provides technical description of the programmatic interface of a service
Semantic Registry	The component of the DBE Knowledge Base that hosts the published services (in the form of Service Manifest Documents).
SFE	Service Factory Environment: Is devoted to service definition and development. Users of the DBE will utilize this environment to describe themselves, their services and to generate software artefacts for successive implementation, integration and use
SMIF	Stream-based Metadata Interchange Format: A general format to save and exchange data of programs that are implementations of expositions models.
SQL	Structured Query Language: A language for querying relational data
SR	Semantic Registry: It is the component of the Knowledge Base that hosts the service descriptions published in the DBE environment and available for discovery and consumption.
SSL	Semantic Service Language
UML	Unified Modelling Language: A method for specifying, visualizing, and documenting the artefacts of an object-oriented system under development; as well as for business modelling.
User Profiling Mechanism	A DBE mechanism used to trace user's actions (and transactions) in order to inspect his preferences on desirable services, and partners.
XMI	XML Metadata Interchange: An SMIF (see SMIF description) standard specification based on XML.
XQuery	A Query language by the W3C that is designed to query collections of XML data.

BIBLIOGRAPHY

1. *MDA Guide Version 1.0.1.* [Online] 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

2. *Meta Object Facility (MOF) Specification, version 1.4.* [Online] Object Managment Group (OMG), 2002. http://www.omg.org.

3. *DBE Knwoledge Representation Models.* **TUC/MUSIC**. May 2005. DBE Deliverable, D14.1.

4. *Initial Description of Profiling mechanism design and rationale with respect to one or two use cases.* **FZI.** October 2005. DBE Deliverable D7.2.

5. *Information Filtering and Information Retrieval: Two sides of the same coin?* **Belkin, J. N. and Croft, B. W.** 1992, Communications of the ACM, 35, pp. 29-38.

6. SQL. ISO/IEC 2075:1999.

7. *OMG XML Metadata Interchange (XMI) Specification v1.2.* [Online] 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-01-01.pdf.

8. *XQuery 1.0: An XML Query Language.* [Online] November 2002. http://www.w3.org/TR/xquery.

9. *XML Path Language (XPath) Version 1.0.* [Online] November 1999. http://www.w3.org/TR/xpath.

10. *The Object Data Management Standard: ODMG 3.0:* ***R. G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez*** Morgan Kaufmann, 2000**.**

11. *OCL 2.0 OMG Final Adopted Specification.* s.l. **Boldsoft, International Business Machines Corporation, IONA and Adaptive Ltd.** OMG Document, October 2003. ptc/03-10-14.

12. *Netbeans Metadata Repository (MDR).* **[Online]** http://mdr.netbean.org.

13. *Berkeley DB XML*. **[Online]** http://www.sleepycat.com/products/bdbxml.html.

14. *1st P2P Distributed Implementation of the DBE KB and SR.* **TUC/MUSIC**. December 2005. DBE Deliverable, D14.3.

15. *Properties of Extended Boolean Models in Information Retrieval.* **H., Lee J.** 1982. 17th ACM SIGIR International Conference on Research and Development. pp. 182-190.

16. *Integrating Diverce Information Managment Systems: A Brief Survey. IEEE Data Engineering Bulletin.* **Raghavan, S. and Garcia-Molina, H.** December 2001, Vol. 24, 4, pp. 44-52.

17. *Algorithms and Applications for universal quantification in relational databases. Information Systems.* **Rantzau, R., Leonard D. Shapiro, Bernhard Mitschang, Quan Wang.** 2003, Vol. 28, 1-2, pp. 3-32.

18. *Searching web databases by structuring keyword-based queries.* **Pável Calado, Altigran S. da Silva, Rodrigo C. Vieira, Alberto H. F. Laender, Berthier A.** 2002. 11th International Conference on Information and Knowledge Managment.

19. *Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching.* **Melnik, S., Garcia-Molina, H. and Rahm, E.** s.l. : 18th ICDE Conference , 2002.

20. *A P2P and SOA Infrastructure for Distributed Ontology-Based Knowledge Managment.* **Gioldasis, N., et al.** 6th Thematic Workshop of the EU Network of Excellence DELOS on Digital Library Infrastructures.

21. *Binary codes capable of correcting deletions, insertions and reversals.* **Levenshtein, I. V.** [ed.] Cybernetics and Control Theory. 1966.

22. *Lucene, Apache*. [Online] http://lucene.apache.org.

23. *Comparison of schema matching evaluations.* **Do, H., Melnik, S. and Rahm, E.** Erfurt (DE) : s.n., 2002. GI-Workshop "Web and Databases".

24. *Construction automatique de taxonomies pour l aide a la representation de connaissances par objects.* **Valtchev, P.** These d' Informatique, Universite Grenoble 1. 1999.

25. *Framework for Ontology Allignment and Mapping (FOAM).* **Institut AIFB, Universität Karlsruhe.** [Online] http://www.aifb.uni-karlsruhe.de/WBS/meh/foam/.

26. Apache Derby database. [Online] http://incubator.apache.org/derby.

27. *Knowledge Base Design and Implementation Status .* s.l. **MUSIC/TUC.**: DBE Internal Document.

28. *Introduction to Modern Information Retrieval.* **Salton, G. and Buckley, C.** New York : McGraw-Hill Book Company, 1982.

29. *On the evaluation of boolean operators in the extended boolean framework.* **Lee, J. H., et al.** 1993. 16th ACM SIGIR International Conference on Reasearch and Development in Information Retrieval. pp. 291-297.

30. *An Approach to Integrating Query Refinement in SQL.* **Ortega-Binderberger, M., Chakrabarti, K. and Mehrorta, Q.** March 2002. 8th International Conference on Extending Database Technology. pp. 15-33.

31. *Modern Information Retrieval.* **Baeza-Yates, R. and Ribeiro-Neto, B.** New York : ACM Press, 1999.

32. *The Object database standard /ODMG-93.* **Attword, T. et al.** San Mateo : Morgan-Kaufmann, 1994.

33. *MOF-based Knowledge Managment for a Digital Business Ecosystem.* **Kazasis, F. G., et al.** s.l. : 2nd IST Workshop on Metadata Managment in Grid and P2P systems (MMGPS), December 2004.

34. *XMOF Queries, Views and Transormations on Models using MOF, OCL and Patterns.* **Compuware Corporation and SUN Microsystems.** 2003. OMG Doc. ad/03-08-07.

35. *MQL: a Powerful Extension to OCL for MOF Queries.* **Heardean, D., Reymond, K. and Steel, J.** s.l. : EDOC, 2003. p. 264.

36. *Towards a Language for Querying OMG MOF-based Repository Systems.* **Petrov, I. and Jablonski, S.** Lisbon : Wisme Workshop UML , October 2004.

37. *Request for Proposal: MOF 2.0 Query/ Views / Transformations RFP*. October 2002. OMG Doc.: as/2002-04-10.

38. *A Metamodel-based OCL compiler for UML and MOF.* **Loecher, S. and Ocke, S.** s.l. : 6th International Conference on the UML and its Applications, UML 2003, October 2003. Vol. 154 of ENTCS.

39. *A Review of OMG MOF 2.0 Query / Views /Transformations Submissions and Recommendations towards the final standard.* **Gardner, J., et al.** York, England : s.n., 2003. Metamodelling for MDA workshop.

40. *Recommender.* **TUC/MUSIC.** March 2005. DBE Deliverable D17.1.

41. *A survey of approaches to automatic schema matching.* **Rahm, E. and Bernstein, P. A.** 10, s.l. : The VLDB Journal, 2001, pp. 334-350.

42. *H-Match: an algorithm for Dynamically Matching Ontologies in Peer-based Systems.* **Castano, S., Ferrara, A. and Montanelli, S.** Berlin, Germany : 1st SWDB VLDB Workshop, 2003.

43. *2nd Release of Recommender.* **TUC/MUSIC** December 2005. DBE Deliverable, D14.4.

44. *Final P2P Implementation of the DBE Knowledge Base and SR.* **TUC/MUSIC** November 2006. DBE Deliverable, D14.5.

45. *Final release of Recommender.* **TUC/MUSIC** November 2006. DBE Deliverable, D14.6.

46. *SparQL: The RDF query language.* [Online] http://www.w3.org/TR/rdf-sparql-query/.

47. *The EMF transformations language.* **IBM.** [Online] http://www.eclipse.org/emft/.