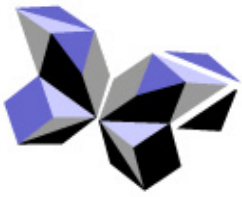

Digital Business Ecosystem

Contract n° 507953

WP17: Composer

D17.1: Recommender

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

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: D17.1

Due date: 31/1/2005

Delivery Date: 15/02/2005

Short Description:

This document describes the mechanisms developed in order to support knowledge access in DBE. The knowledge access mechanisms provide the underlined functionality for querying metamodels, models and instances in the DBE environment. It also forms the basis for supporting recommendation (i.e. expressing and matching preferences of users), with the knowledge base Query Metamodel Language (QML), which is based on OCL2.0 and which has been properly adapted to MOF1.4. Extensions to support fuzzy queries are also described. The Query Metamodel Language also allows the users to express preferences and as such it can form the basis for representing user profiles in DBE. This document accompanies the current implementation of the Recommender that was demonstrated in the review of DBE of January 2005 as part of the integrated first year prototype of DBE.

Partners owning: TUC

Partners contributed: TUC

Made available to: All project partners and the EC

Versioning

Version	Date	Author, Organization	Description
0.1	21/12/2004	FOTIS G. KAZASIS – TUC GEORGE KOTOPOULOS – TUC YIANNIS KOTOPOULOS - TUC	Initial Creation
0.2	17/1/2005	PROF. STAVROS CHRISTODOULAKIS - TUC	Revisions/Additions
0.3	14/2/2005	NEKTARIOS GIOLDASIS – TUC NIKOS PAPPAS - TUC	Revisions/Additions
0.4	22/03/2005	FOTIS G. KAZASIS - TUC GEORGE KOTOPOULOS – TUC NIKOS PAPPAS – TUC NEKTARIOS GIOLDASIS – TUC	Revisions/Additions according to the Comments of the Internal Reviewers

Quality check

1st Internal Reviewer: Pierfranco Ferronato – SOLUTA.net

2nd Internal Reviewer: Jim Dowling - TCD

Recommender

Table of Contents:

List of Figures:

List of Tables:

Executive Summary

This document describes the mechanisms developed in order to support knowledge access in DBE. This implementation was demonstrated on January 2005, during the DBE review, as part of the integrated first year prototype of DBE.

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. Personalization is supported with profiles of users and filtering mechanisms. The DBE knowledge is managed by the Knowledge Base (KB). The Knowledge Base is compatible with the OMG's MOF metadata framework therefore each segment of information stored in the KB has been produced and placed as instance of a model at a higher layer of the MOF meta-data architecture. The KB manages MOF metamodels, models, and instances providing the full functionality of a MOF repository, and uses XMI documents for metadata and data interchange.

The knowledge access mechanisms that we describe in this document provide the underlined mechanism for querying metamodels, models and instances of DBE. It also forms the basis for supporting recommendations (i.e. expressing and matching preferences of users). The core of the knowledge access functionality is the Query Metamodel Language (QML) which is a language based on OCL2.0 [2], 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 recommender in its final form will encompass the knowledge access module augmented with user profiling mechanisms 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.

The knowledge access module uses the DBE knowledge base implementation, which is based on the MDR [3] repository and a relational database to store and manage DBE models and metamodels. It is used by the exposed services of the DBE Knowledge Base according to the requirements of the particular architectural environment in which it is employed. That is, when this environment is the DBE Service Factory, it is used for accessing models available in Knowledge Base's repository[1]. On the other hand, when the environment is the DBE Execution Environment, it is used for accessing service descriptions published in Knowledge Base's semantic registry. In a distributed environment with many Knowledge Base instances, the module can be used at each instance (node) to allow local knowledge access (query processing).

[1] It has to be noted that different KB instances may serve the Service Factory and the Execution environments.

1. Introduction

This document describes the mechanisms developed in order to support knowledge access in DBE. DBE information is hosted in the Knowledge Base (KB). The DBE KB provides a common and consistent description of the DBE world and its dynamics, as well as the external factors of the biosphere affecting it. Its content includes:

- Representations of domain specific ontologies (common conceptualization in a particular domain);
- Semantic Descriptions of the SMEs themselves in terms of business models, business rules, policies, strategies, views etc.;
- Semantic Description of the SME value offerings (description on how the services may be called) and the achieved solutions (service chains/compositions) to particular SME needs.
- Models for gathering usage 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*" [14]. 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 Relational Data Management System. Other implementations could be also possible.

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

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

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

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

Furthermore, other metamodels like those of Business Modeling Language (BML) will be represented into the DBE Knowledge Base, as they will become available in the future. Thus, the exploitable knowledge spectrum in DBE will range from ontologies, to business models, to semantic and technical service descriptions, to user profiles, to usage data, etc. Each one of these knowledge segments will be represented using a different metamodel. In order to support efficient knowledge access over all these metamodels there is a need for a query mechanism that will be quite generic so that it can specify, in a uniform way, knowledge access requests over all types of knowledge (both data and meta-data) that are kept in the DBE KB. Such kind of functionality is a prerequisite for implementing explicit querying of DBE Knowledge (information retrieval / pull-mode) as well as knowledge personalization (information filtering / push-mode) [1] functionality of the recommender component.

To this end, in this document we describe a query mechanism, which is based on a query metamodel that is quite generic so that the expressions (query models) that form the instances of this metamodel are capable to query all types of knowledge (models and corresponding data), which are available in the Knowledge base in a uniform way.

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 database technologies provide 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. SQL-99 [15] 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. Since MOF models and instances can be mapped to XML documents (XMI [10]), an XML query mechanism can be integrated with MOF technology. XQuery [16] and Xpath [17], 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 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 [18]. However, the standard does not define the language's abstract syntax or formal semantics.

The approach that we 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 use a language based on the Object Constraint Language (OCL2.0), which has been also used as the formal basis of our query metamodel. The choice of OCL was also motivated from the fact that OMG advocates

in the core of its business architecture the use of MOF and on the top of it the use of Unified Modeling Language (UML), which contains OCL for specifying constraints in the models. Thus mechanisms that support OCL would be also useful for efficiently supporting UML in a Knowledge Base that would support also UML functionality.

In the past few years OCL has evolved from being merely an extension of the Unified Modeling Language (UML) to representing an integral part of it. The latest response to the UML 2.0 OCL request for proposal [2] 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[4]. 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 our 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.

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 would also allow 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 is considered as a prerequisite. Such a metamodel could be part of the work carried out in WP7 "User Profiling" (responsible partner Forschungszentrum Informatik - FZI).

The knowledge access mechanism utilizes the functionality for processing valid query expressions based on the QML (suitably formulated by the Query Formulator Tool[5]). Such functionality includes query parsing and analysis, query syntax tree construction and code

[4] 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.).

[5] The description of the Query Formulator Tool along with its supported functionality and GUI is out of the scope of this document. Nevertheless section 7 "Semantics of Query Expressions and Examples" presents some indicative screenshots of the tool for the outlined examples.

generation using the KB infrastructure. From a technical point of view, the KB infrastructure is based on a combination of a MOF/JMI-compliant repository and a data management system. Two alternatives are currently available: a) the data is queried in MOF object representation; b) the object-oriented queries are mapped to SQL statements and executed by the relational database management system. The current implementation of the code generator allows the generation of SQL statements (enhanced when needed with fuzzy extensions) that correspond to the submitted queries and can be executed by the relational data base management system.

Our work also aims at providing the functionality related to the evaluation of candidate services and candidate business partners in the process of creating composite services and establishing partnerships respectively. The major assumption behind the design and implementation of the Recommendation mechanisms is the existence of powerful business and service Ontologies that capture the semantics of business models and service descriptions. These Ontologies will be also used to define the corresponding preferences for businesses and services. The recommender exploits the relational database system to store preferences and all recommendation mechanisms use SQL statements to implement the necessary matching functions between preferences and business/service descriptions. In the current implementation basic recommendation mechanisms are being tested based on primitive preference structures since the SME preferences structures (part of the User Profiles) are not yet defined. As already mentioned the definition of the SME preferences structures (user profiles) will be part of the work carried out in WP7 “User Profiling” and will follow the MOF architecture; in such a case there will exist a User Profile Metamodel and the SME preferences (profiles) will be the instances (models) of this metamodel suitably stored in the relational database system.

The rest of the document is organized as follows: section 2 presents some preliminary issues concerning the technologies used. The section first outlines the MOF metadata architecture adopted and then the KB infrastructure. Section 3 presents the global communication diagram in the DBE as well as the overall knowledge base architecture of DBE in order to provide the interdependencies of the knowledge access module with the rest of the system. In section 4 we present the general architecture of the knowledge access module that supports the formulation and the evaluation of involved discovery requests to the KB. Next, in sections 5, 6 the QML along with representative examples is presented. Sections 7, 8 present the methodology used for the construction of execution plans of the query expressions and the code generation process. Section 9 presents the mathematical framework on which the implementation of the Recommendation Mechanisms is based. The last section 10 presents the conclusions and the issues that are under further consideration.

2. Overview and Background

This section presents some background issues related to the technologies used in the DBE Knowledge Base and discussed throughout the rest of this document. These technologies refer to the adopted MOF metadata architecture and the KB infrastructure. The DBE Knowledge Base follows the OMG's MOF metadata architectural framework.
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 QML Query expressions. 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.

As described in [5], there are various kinds of DBE Knowledge. Roughly speaking, we distinguish the following kinds of knowledge:

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 Modeling 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.

Our M2 Query Metamodel Language (QML) is also defined as a 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).

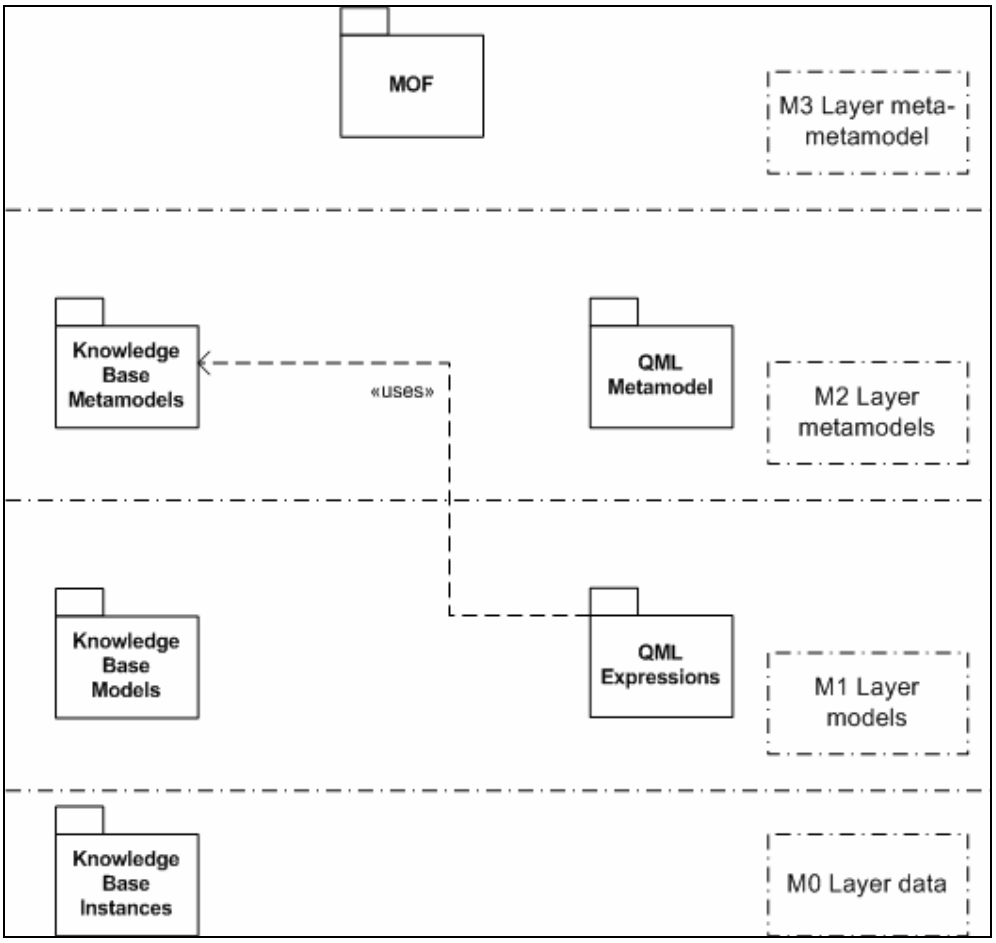

Figure 1: The DBE knowledge access process follows the MOF Metadata architecture. Valid QML expressions are instances of the QML Metamodel that refer to the semantics of the metamodels, which have been implemented in the knowledge base, using the definition of the metamodel in the M2 layer. This process allows the instantiation of valid query expressions for ODM models, SSL models, SDL models, and any other models that will be inserted later on in the knowledge base.

In addition (not depicted in Figure 1) 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).

The Knowledge Base uses XMI documents for information exchange with other DBE components. The KB is built on top of the NetBeans Metadata Repository[6] [3] and the Apache Derby Relational Database[7] [4]. The metadata repository offers the functionality of processing XMI documents and exporting in XMI documents pre-selected content that is

[6] 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

[7] The Apache Derby project develops open source database technology. Derby is an effort undergoing incubation at the Apache Software Foundation

stored in the repository. Appropriate middleware has been developed to provide a bi-directional bridge between the repository and the database system. The middleware utilizes SQL/JDBC for the communication with the database and 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 modeling tools (based on the OMG-UML) and metadata repositories (OMG-MOF based) in distributed heterogeneous environments.*" [10] 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.

3. 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 first year 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.
In short in its final form the recommender component will encompass 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. Therefore the full design and implementation of the Recommender component will be completed when these frameworks become available in the project.
Furthermore, since the next implementation of the DBE Knowledge Base will have a p2p nature, the recommender will be strongly affected by the p2p Knowledge Base framework that will be adopted. In particular the following KB related issues, will have to be taken into account for the support of the recommendation process in the p2p environment:

- Semantic-based indexing schemes that will use 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

For the purpose of this section we re-use here Figure 2, which was originally presented in the DBE deliverable “D18.1 Report on DBE-specific Use Cases” authored by another DBE partner, LSE (London School of Economics). This figure illustrates the global communication diagram of DBE. In this figure the Knowledge Management Infrastructure is used in both the Service Factory Environment (SFE) and the Execution Environment (ExE). 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[8]:

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

[8] 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.

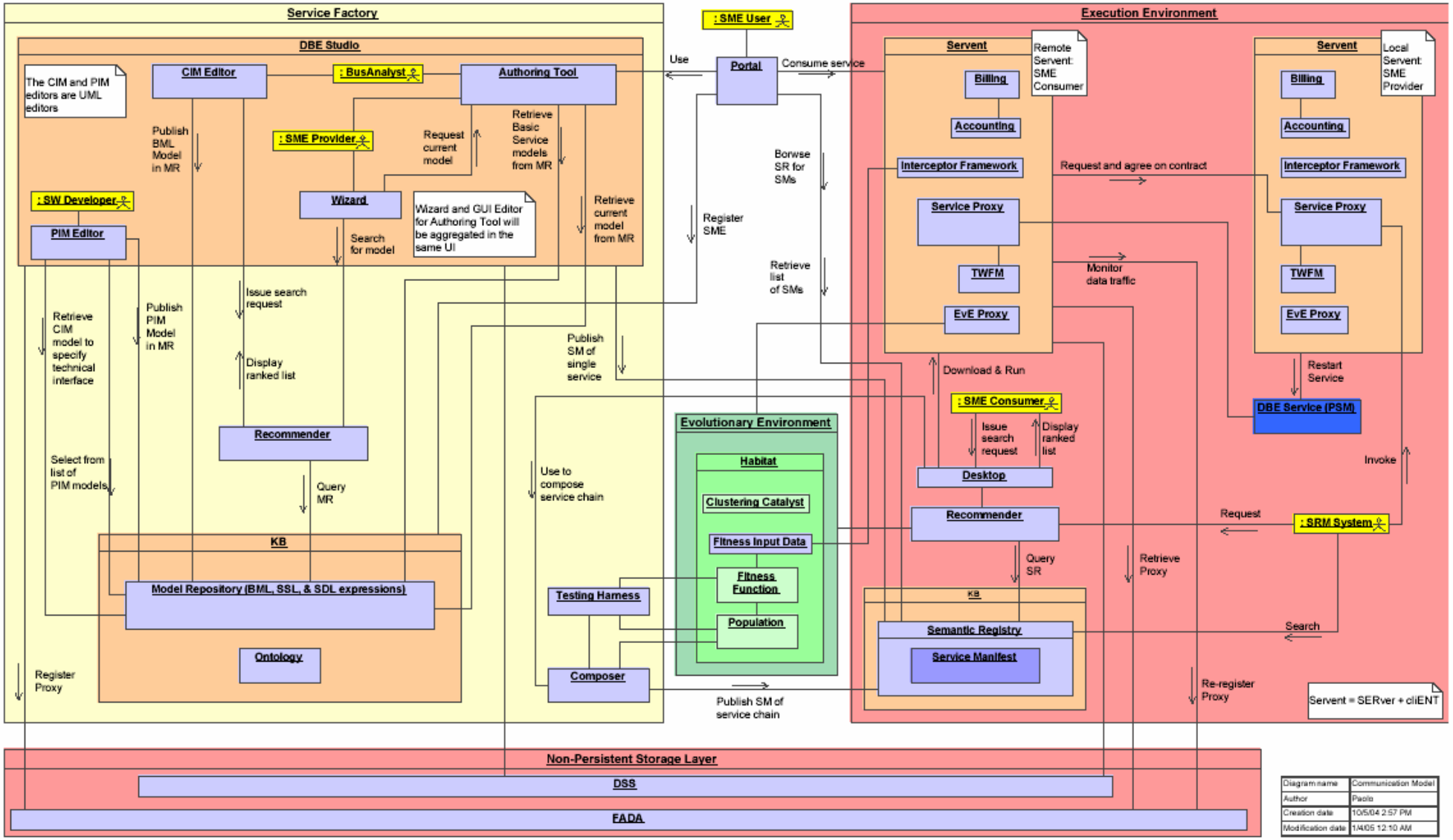

Figure 2: The Global Communication Diagram of the DBE (figure taken by Deliverable D18.1 “Report on DBE-specific Use Cases”)

Figure 3, 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 first year 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 it is employed. 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 3) serve the Service Factory and the Execution environments. For a detailed description of the functionality offered by the Query Interface the reader should refer to Appendix A: Query API.

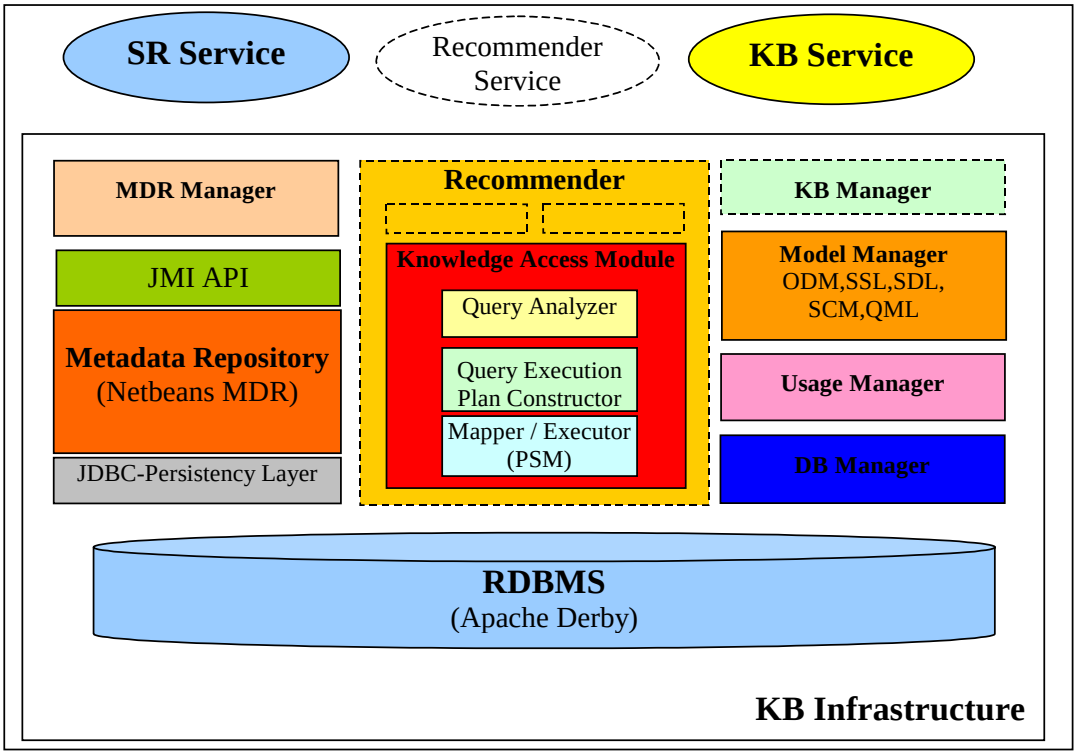

Figure 3: The Knowledge Access Module with Respect to the DBE KB as it is currently implemented in the integrated first year prototype of DBE. The Knowledge Access module is a central component that is used by the exposed services of the DBE knowledge management infrastructure. The KB Service and SR Service are used in the Service Factory Environment or the Execution Environment respectively.

4. The Knowledge Access Module Architecture

This section presents the component architecture, along with the established interactions of the Knowledge Access Module, that supports the formulation and the evaluation of involved access requests to the KB and that was deployed in the integrated first year prototype of DBE.

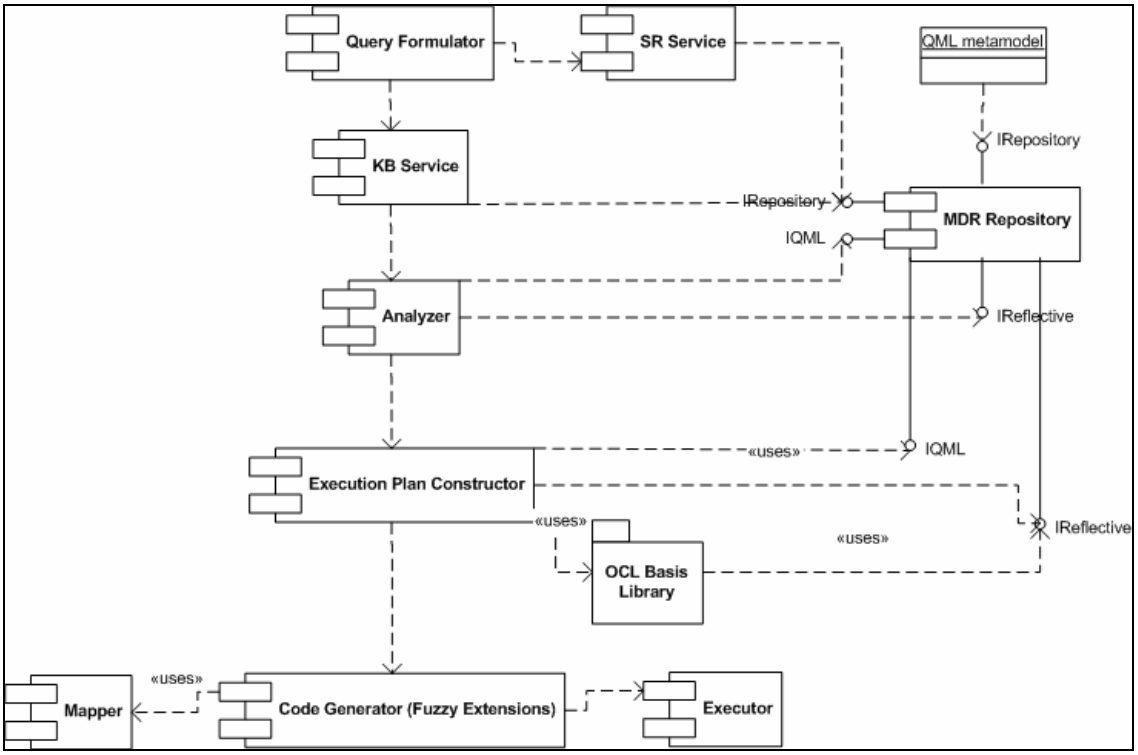

Figure 4: The architecture of the Knowledge Access Module as it was deployed in the first DBE integrated prototype.

The Knowledge Access Module uses the Query Metamodel Language (QML) to retrieve knowledge for the Knowledge Base. The module reuses components and interfaces deployed in other parts of the KB system as the IRepository, the IReflective and the MDR Repository (explained below). The Query Metamodel Language is based on OCL 2.0 and is appropriately adapted to MOF 1.4.

The Query Formulator component is a graphical front-end tool with tree view based browsing capabilities that allow the formulation of metamodel driven queries utilizing the JMI reflective functionality. These queries (i.e. instances/models of the QML metamodel) are then streamed to the Knowledge Base (KB) service or the Semantic Registry (SR) service (encoded as XMI 1.2 documents) depending on the architectural environment[9] and are stored into the metadata repository. Currently, the Knowledge Access module includes the Query Analyzer, the Query Execution Plan Constructor, and the Query Code Generator/Executor components. The Query Analyzer interprets and analyzes the query model using the JMI interfaces of the MDR Repository; the Query Execution Plan Constructor evaluates the query and constructs a valid query syntax tree. The Query Code

[9] KB service is used at the Service Factory Environment whereas the SR service is used during the Service Execution Environment.

Generator/Executor component accepts as input the query syntax tree and based on platform specific mapping rules generates the appropriate code and executes the query against the contents of the Knowledge. Figure 4 shows the architecture of the module.

Next we examine each component of the general architecture and the goals it fulfills separately.

4.1 MDR Repository

All the available knowledge base information represented by corresponding MOF metamodels and models is stored in the MDR Repository. The MOF QML Metamodel along with its instances (i.e. query or evaluation expressions) is also stored here. The MDR Repository provides interfaces for interacting with the other components of the system. The Knowledge Access module utilizes three interfaces:

- **IRepository** interface that includes operations for loading and saving models and metamodels via XMI and operations for generation of JMI interfaces from metamodels.
- **IQML** interface that is a set of JMI interfaces that enable access to instances of the QML metamodel in a transparent way.
- **IReflective** interface that is a set of reflective JMI interfaces that enable to access models without the generated metamodel-specific interfaces.

4.2 KB/SR Services

These services act as JMI-enabled metadata services exporting Knowledge Base functionality to the rest DBE components. This functionality provides support for storing, querying and retrieving DBE models and data. In the Service Factory Environment the Knowledge Base service is used whereas during the Service Execution Environment the Semantic Registry service is used.

4.3 Query Formulator

Query Formulator is the component that interfaces with the user and allows the formulation of user's queries into a QML representation[10]. In the current implementation, during this process the user browses the primitives of the metamodels that have been implemented into the Knowledge Base (ODM, SSL, SDL, BPEL) and poses constraints (criteria) on these primitives, which should be satisfied by the models or the instances that he demands.

4.4 Analyzer

This module analyzes a QML expression in order to interpret and validate it against the metamodel-specific semantics of the expression, which are stored in the MDR Repository. Analyzer uses the IQML interface to gain access to QML expressions. The IReflective interface is used to access the metamodel specific information stored in MDR in order to be able to validate the expression's semantics.

[10] The reader should refer to section 5 The Query Metamodel for a complete reference of the QML representation.

4.5 Execution Plan Constructor

This module evaluates the QML expressions already analyzed into a syntax tree representation. This module serves two main goals. The first is to hide from the execution engine (might be apart from the SQL Executor, an XQuery one, etc.) the complexity of the QML metamodel. The second one, which will be supported in the long run, is to perform optimizations on the query, simplify expressions if possible, etc. The evaluation process and the construction of the syntax tree are explained in detail in section 7 (Construction of Execution Plans of Query Expressions). Alternatively the Execution Plan Constructor can use the OCL Basis Library along with the IQML and the IReflective interfaces in order to perform evaluation of Well-Formedness Rules and invariant constraints for the information (metamodels, models) that reside in the MDR repository.

4.6 Code Generator

QML is used in order to pose a query against a specific metamodel. The instances of this metamodel could be kept in some persistence storage, which may be a repository, a relational database, an XML repository, etc for further processing. In order to be able to answer the question the formulated query must be translated into the appropriate language (code) for querying the stored data depending on the data management system (for instance, SQL for a DBMS, XQuery for an XML database, etc) that is used for the persistency requirements of the KB. The task of the code generator is to take as input the query syntax tree, produced by the query execution plan constructor and generate the code to be executed in the appropriate query language. In order for the code generator to be able to recognize the stored data, a mapping is needed (Mapper component) between the concepts in the metamodel and the elements of the schema used to capture this information for the specific data management system. In the current implementation the formulated queries are stored in the MDR repository and the generated code is SQL since the current persistency layer is a relational DBMS.

4.7 Executor

The statements produced by the Code Generator are forwarded to the executor component, which is responsible for their execution in the correct order. Such an executor maybe an SQL RDBMS, an XQuery Engine, etc.

The current implementation utilizes a relational database system and therefore the code generator supports the translation into SQL statements whereas the mapper is based on mapping the metamodel elements to tables of their corresponding database schema and the executor is the sql engine of the RDBMS.

5. The Query Metamodel Language

This section discusses in detail the terms of the Query Metamodel Language (QML) and its placement in the adopted MOF architecture.
As noted QML is an M2 MOF model and therefore it has been implemented as an instantiation of the MOF metamodel. Since QML aims at formulating queries and for expressing constraints on the M3 – M1 layers of the MOF architecture its metaclasses have been also implemented as specializations of appropriate MOF model elements[11] as it will be explained later on.
QML leverages the Object Constraint Language (OCL2.0), which has been used as the formal basis of its metamodel. OCL2.0 cannot be directly used for querying and/or applying constraints to MOF 1.4 models because its formal semantics are based on UML 1.4 Core package and there exist a number of differences between the UML 1.4 and the MOF 1.4 Core packages. In our work UML metaclasses referred by the OCL2.0 metamodel have been suitably aligned to MOF 1.4 metaclasses. The elaborated metamodel refers to these metaclasses. The differences and the alignment adopted can be seen in Table 1.
In addition, since we only aim at MOF, the OCL2.0 metamodel has been refined by subtracting the metamodel's UML-specific parts. The OCL2.0 metamodel does not provide a concrete or abstract syntax for representing queries, therefore we have also enriched it with an appropriate expression and a 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). In particular, we have refined the OCL's query-specific elements by formally treating a query as an OCL expression (constrain) on an object defined in a metamodel (context), returning objects of that kind or any other object of the metamodel that can be directly referenced from the context (result type)[12]. The elaborated OCL2.0 metamodel is the Query Metamodel Language (QML).
As already mentioned before one additional requirement that has to be met is to support Information Retrieval (IR)-style approximate matching and to allow the ordering of results by their relevance score. For that we have provided a framework that is based on the Extended Boolean Model. In order to support efficient QML processing in the specific framework we have extended it by incorporating fuzzy Boolean operators. The followed approach is presented in subsection 5.4 "QML Fuzzy Extensions".

Table 1 stands QML comparing with the complete OCL2.0 specification:

UML Metaclasses (referred from the OCL2.0 Abstract Syntax metamodel)	MOF1.4 Metaclasses (referred from the QML metamodel)
ModelElement	ModelElement
Classifier	Classifier
DataType	DataType
PrimitiveType	PrimitiveType
Attribute	Attribute
AssociationEnd	AssociationEnd

[11] For implementing the specialization of a MOF Model Element the intrinsic MOF repository functionality has been exploited

[12] this will been shown later in figure 12 with the QueryContextDecl class

Operation	Operation
EnumerationLiteral	EnumerationType (with multivalued attribute labels)
AssociationClass	*MOF does not support the UML metaclass AssociationClass. Therefore QML does not include AssociationCallExp*
Messages	*MOF does not support messages, therefore QML does not provide messages support.*

Table 1: Comparison of QML and OCL.

Table 1 depicts the UML metaclasses that have been eliminated (EnumerationLiteral, AssociationClass and Messages) as well as the alignment of UML metaclasses with MOF ones. QML also extends OCL2.0 with the QueryContextDecl Metaclass and fuzzy Boolean operators. The QueryContextDecl metaclass and its semantics are not part of the core QML metamodel and thus, there is no interference with OCL2.0. For the rest of its structure QML utilizes the concrete and abstract syntax of OCL 2.0.

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, 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 MOF). In the following sections these packages are described in detail.

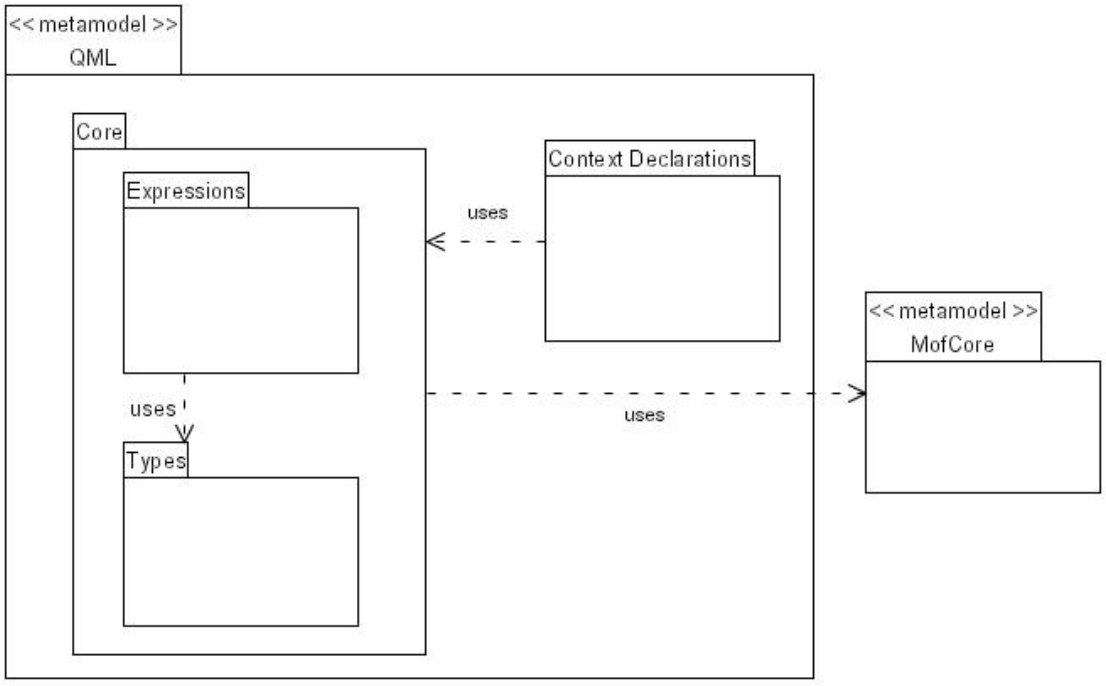

Figure 5: The QML package structure.

5.1 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 modeled, 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.

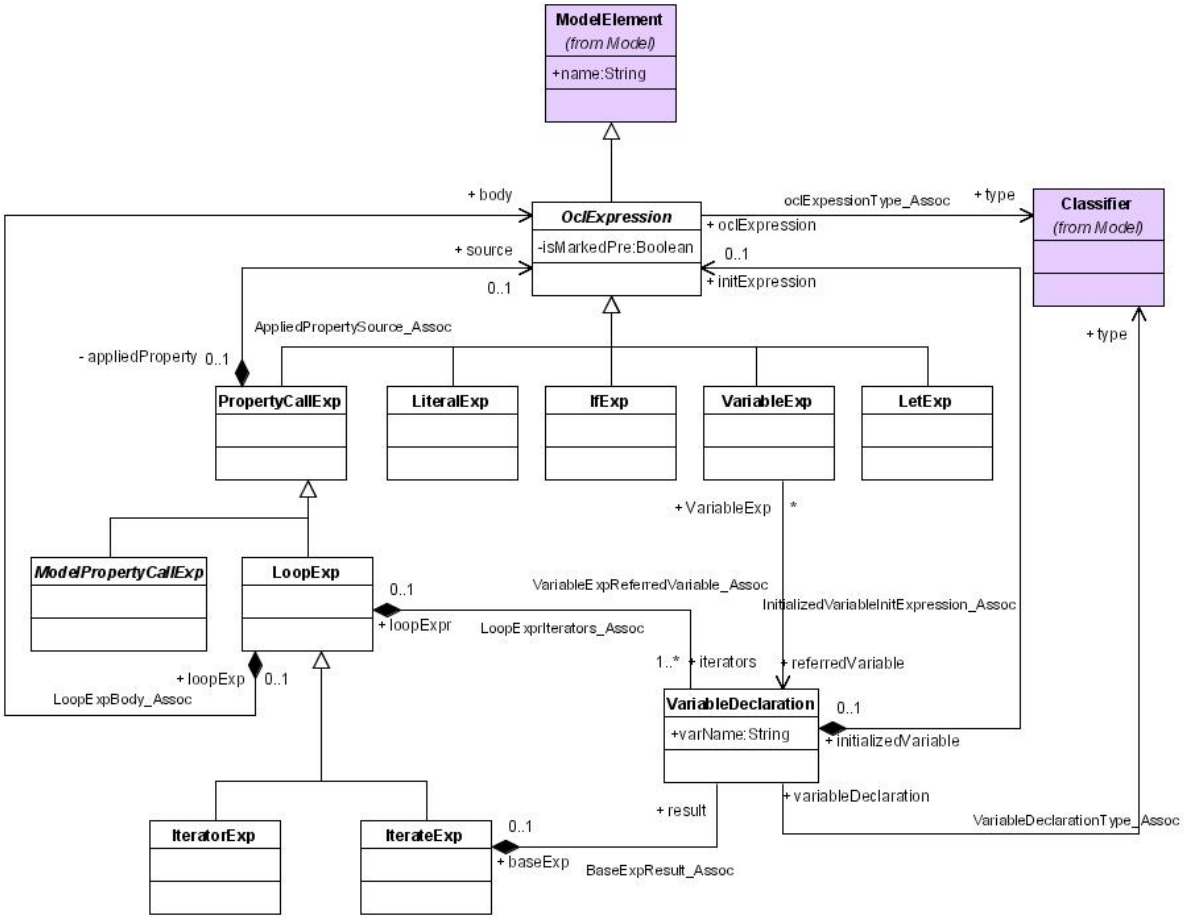

Figure 6: The QML Expressions package.

A *ModelPropertyCallExp* generalizes all property calls that refer to *Features* or *AssociationEnds* in the MOF metamodel. In Figure 7 the various subtypes of *ModelPropertyCallExp* are defined.

Most of the remainder of the expressions package consists of a specification of the different subclasses of *PropertyCallExp* and their specific structure. 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.

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 are defined in terms of the iterate expression. The reader should refer to the official adopted OCL 2.0 specification ("Mapping rules for predefined iterator expressions") for a complete reference on predefined iterator expressions.

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.

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 modeled as a separate metaclass, because it can be completely derived using derivation rules. The complete derivation rules can be found in chapter 9 ("Concrete Syntax") of OCL 2.0 specification.

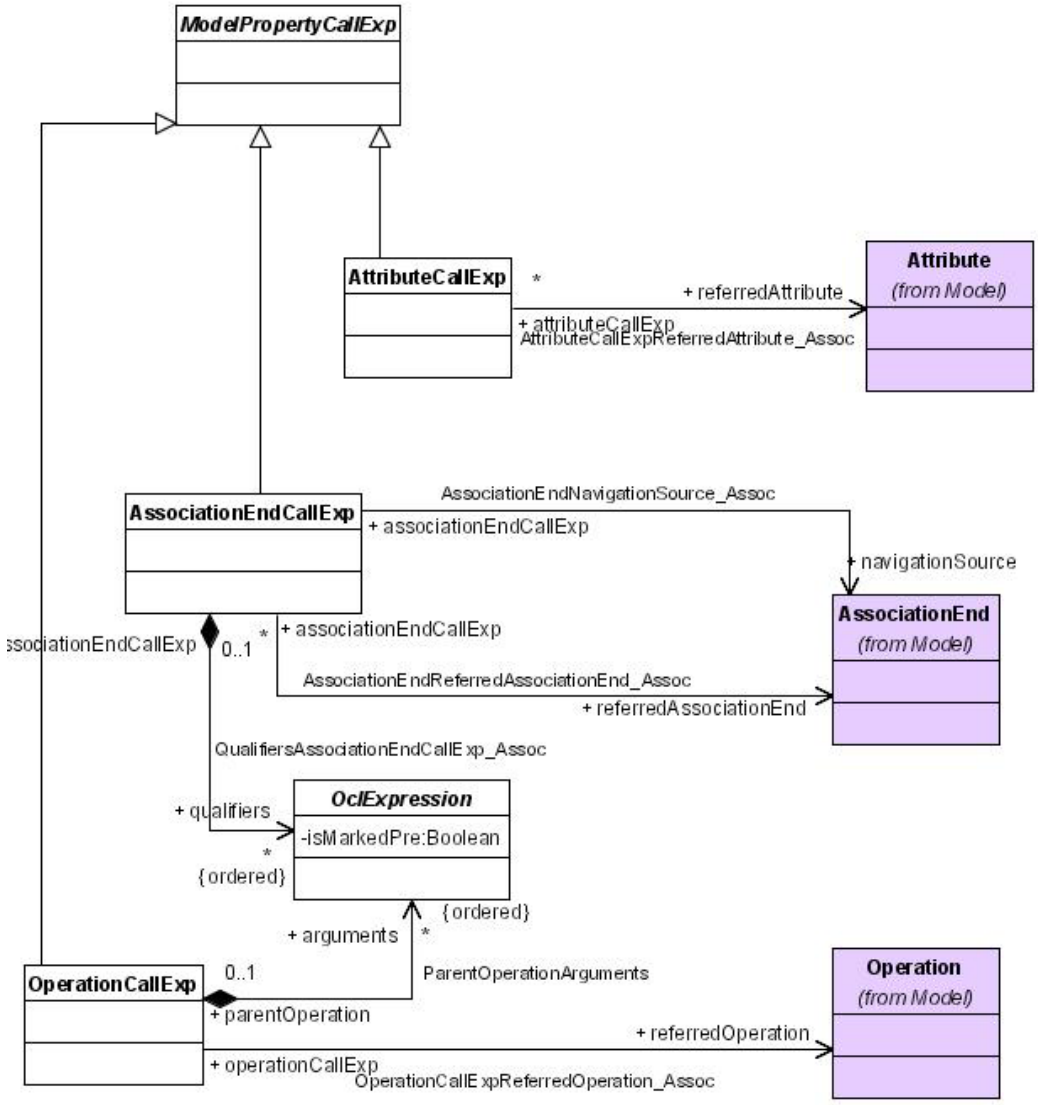

Figure 7: The ModelPropertyCallExp in the Expressions package.

PropertyCallExp

A *PropertyCallExp* is an expression that refers to a property (operation, attribute, association end, predefined iterator for collections) and it is referred by a *source* expression. Its result value is the evaluation of the corresponding property. *PropertyCallExp* is an abstract metaclass and its *source* expression is the instance that performs the property call.

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. *VariableDeclaration* metaclass represents amongst others the variables *self* and *result* and the variables defined using the Let expression.

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.

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. Figure 7 shows the three different subtypes of *ModelPropertyCall*, each of which is associated with its own type of *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.

AttributeCallExp

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

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.

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 *thenExpression* and the *elseExpression* are mandatory. The reason is that an if expression should always result in a value, something that cannot be guaranteed if the else part is left out.

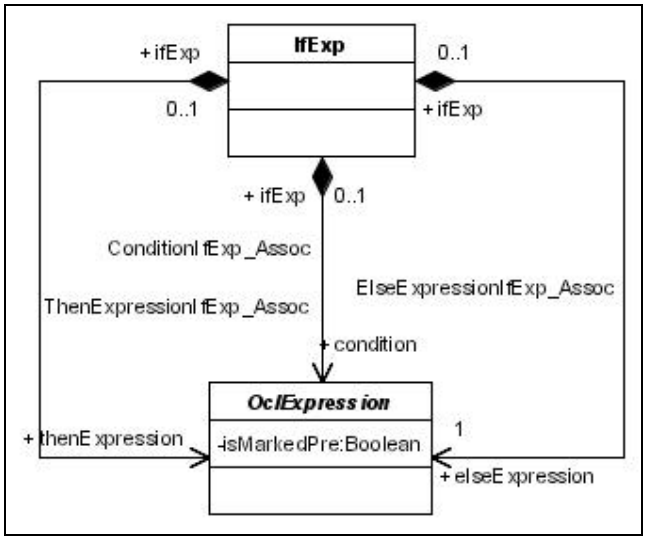

Figure 8: Definition of If expression.

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.

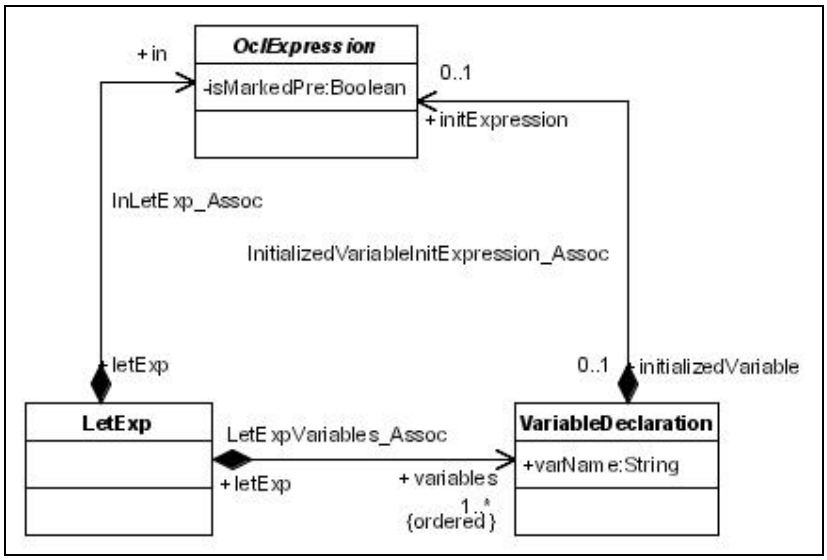

Figure 9: Definition of Let expression.

LiteralExp

A *LiteralExp* is an expression, which has no arguments that produce 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'. *LiteralExp* and its specializations are shown in figure 10.

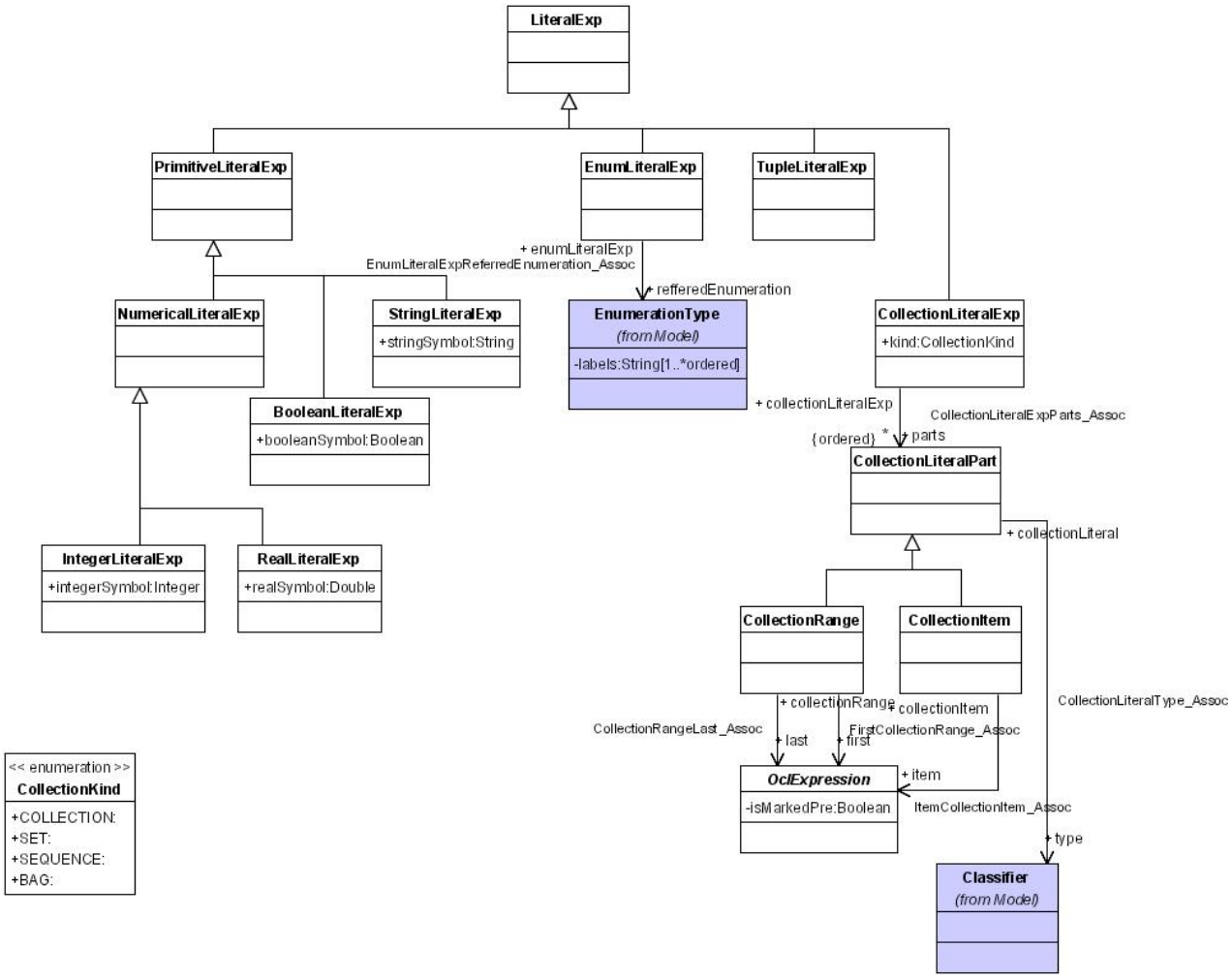

Figure 10: Definition of literal expressions.

5.2 The Types Package

QML is a typed language. Each expression has a type which is either explicitly declared or can be statically derived and its evaluation yields to a value of this type. A metamodel for the QML types is shown in Figure 11. Note that the instances of the classes in the metamodel are the types themselves (e.g. Integer) and not instances of the domain they represent (e.g. -15, 0, 2, 3).

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 ones. 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 could never instantiate all collection types, because there is an infinite number, especially when nested collections are taken into 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.

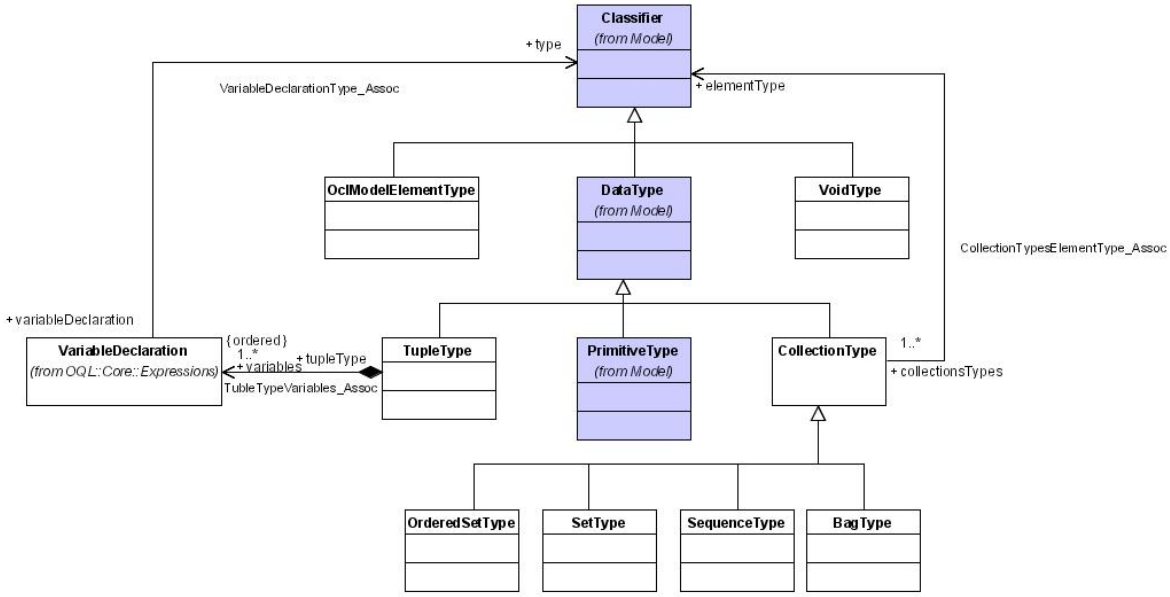

Figure 11: The QML Types package.

OclModelElementType

OclModelElementType represents the types of elements that are *ModelElements* in the MOF metamodel.

CollectionType

CollectionType describes a list of elements of a particular given type. *CollectionType* is an abstract class and 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, which means 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 (informaly 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 it and is 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*.

5.3 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 [2] in order to facilitate the declaration of the OCL expressions in separate text files. Based on the concrete syntax we developed the Context Declarations package, which does not belong to the Core part of QML but is rather a set of helper meta-classes. These helper meta-classes are used to express where an OclExpression refers to, the kind of it (invariant, operation, definition and attribute) and any other specific information needed for each kind. The 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 metaclass. Figure 12 shows the context declarations package with the QueryContextDecl metaclass.

QueryContextDecl

QueryContextDecl represents a query identified by a name *simpleName*. The query returns a set of objects of type *result* (may be any *Classifier*[13]) for which the constraint *bodyExpression* holds. *QueryContextDecl* has a *pathName* which is a String Sequence representing the context of the query, which is the object type where the constraint

[13] Note that it could also be a complex TupleType

bodyExpression is referring. There is a requirement that a *QueryContextDecl* may be associated to more than one context. This is a very useful requirement in order to allow queries to combine semantic information from more than one metamodels. This requirement is currently fulfilled by implicitly declaring the complete path for locating the desired semantic information.

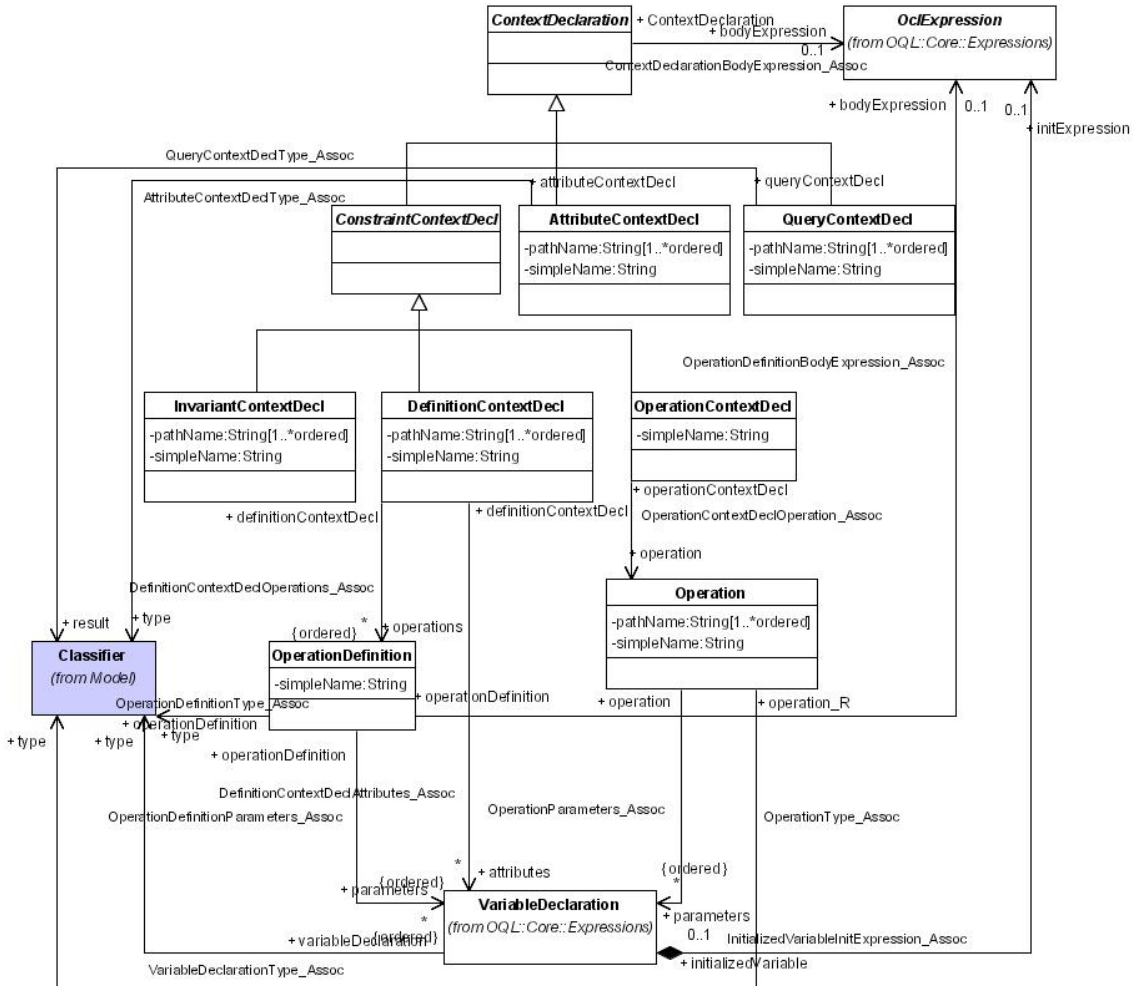

Figure 12: The QML Context Declaration package.

5.4 QML Fuzzy Extensions

Typical implementations of information retrieval systems model both information items and queries as vectors of features. The set of information items satisfying a query vector includes those items that contain any feature present in the query vector. In contrast, traditional Boolean logic considers a query as a Boolean expression that consists of features as atoms and the three Boolean operators (AND, OR, NOT) as connectives. The answer set of a query is the set of information items that satisfy the Boolean expression. The drawback of the Boolean approach is that an information item is considered either as relevant or non-relevant to a query. There is no means of ranking the information items with respect to a degree of similarity to the query. To bypass this drawback, the Extended Boolean Model has been introduced in the information retrieval literature. This model is a generalization of the

Boolean logic based on the fuzzy set theory. It provides formulas 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.

We have developed a framework for QML processing that incorporates Information Retrieval functionality and that is based on the Extended Boolean Model[14]. In order to support more efficient QML query processing in the specific framework, we have extended it by incorporating fuzzy Boolean operators.

In particular we have extended the OCLBoolean package of the OCL Basis Library with three new fuzzy Boolean operators, namely fAND, fOR and fNOT that are to be used for IR-style conjunction, disjunction and negation operations respectively. The corresponding evaluation fuzzy functions f_{NOT}, f_{AND}, and f_{OR} have been realized in the Code Generator Module as SQL statements (properly extended) in order to exploit the fact that the current implementation of the knowledge base is on top of a relational database management system. It has to be noted that this extension approach stands outside the QML metamodel and therefore it does not affect the compatibility of QML with OCL.

[14] for a detailed description of the framework the reader should refer to section 9 "Theoretical and Technical framework of the Knowledge Access Mechanisms"

6. 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 and how those expressions could be presented by graphical user interfaces. 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. 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. In the future the Query Formulator Tool will be extended to support appropriate user interfaces (there is a task in the new DBE workplan for that purpose).

The following example queries are driven by the Semantic Service Metamodel (SSL) [5] 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:

a) SemanticPackage: It refers to the set of the semantic descriptions of a given service.

b) 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).

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

d) Functionality: This primitive is used to describe what the Service Consumer can do when interacting with the Service Provider SME in the scope of particular service consumption.

6.1 A Simple QML Expression

We present in this subsection an example to give an idea of the semantics of the functionality offered by QML.

Consider the statement:

```
Context SSL::ServiceProfile simpleQuery : SemanticPackage
query:
Attribute.name = “HotelName”
```

The above statement defines a query with name “simpleQuery”. This is a query that is posed against the Metamodel of the Semantic Service Metamodel (SSL) [5] which is one of the metamodels imported and supported in the DBE knowledge base. It allows the semantic description of services. The query retrieves all the services that appear to have an attribute with name “HotelName”.

Context of the query is the sequence “SSL”, “ServiceProfile”. A sequence is needed to define where “ServiceProfile” (i.e. the last element of the sequence) is located. In this example it is located in the “SSL” metamodel, which is the context of this query. The query also defines the result type of the query, which is “SemanticPackage”. Note that, in the metamodel, to which the query applies to (in the example SSL), the SemanticPackage class must contain (in some way) the ServiceProfile class.

The body expression follows the “query” term. The evaluating expression, which must conform to a ServiceProfile instantiation in order to be in the result set, is the operation “=”. This operation is defined in the basic OCL library as an operation of the primitive type String. This operation has two arguments or more precisely a calling object, the “Attribute.name”, and an argument, the “HotelName”. The latter is a string literal. The first activates a navigation process through the ServiceProfile class, which is the environment of the query. Thus, starting with ServiceProfile class, we find that “Attribute” is an AssociationEnd of the association between ServiceProfile and ServiceAttribute classes. Now the current environment of the query is the ServiceAttribute class[15]. Following the navigation path, we find that “name” is an attribute of the ServiceAttribute class. Now the environment is the type of name, which is the String class. In this environment we find the operation “=”.

[15] Note that this is derived and does not need to be explicitly stated. However, there are cases where it must be defined.

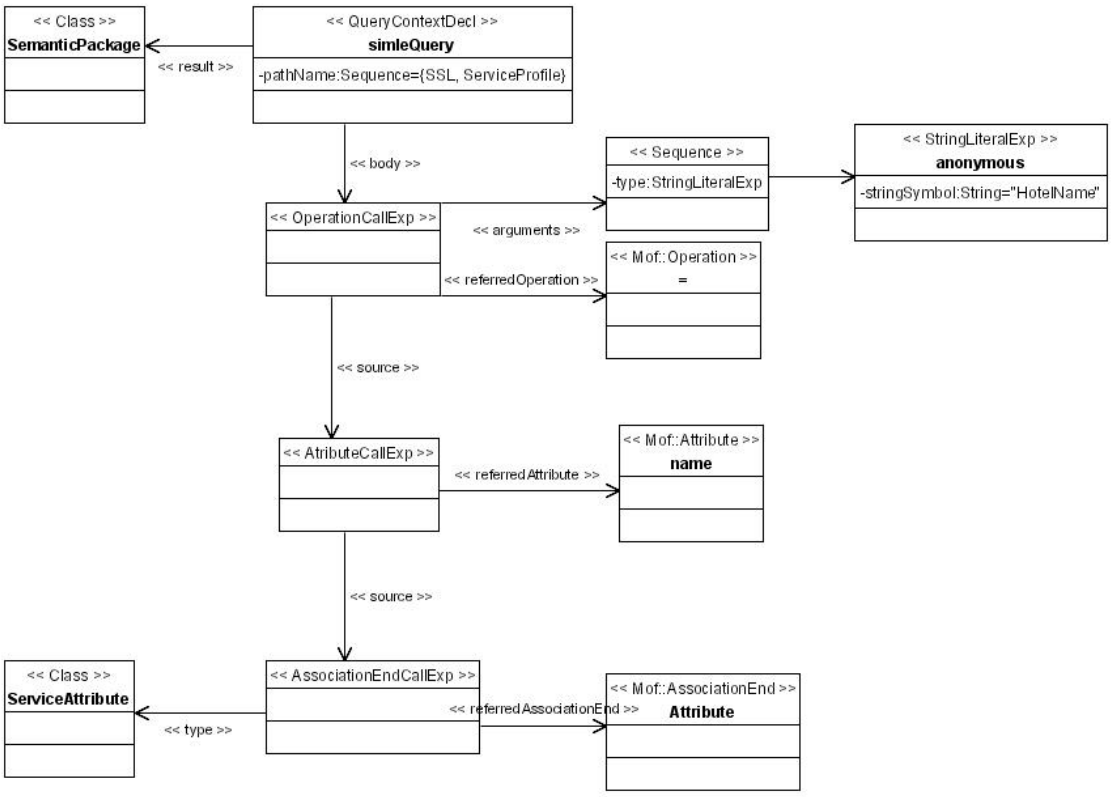

Figure 13: The QML representation of a query that retrieves all the services that appear to have an attribute with name “HotelName”.

The above statement is depicted in Figure 13 in QML metamodel elements. In terms of the QML metamodel the abovementioned query is as follows: There is a query defined in a QueryContextDecl object with the name “simpleQuery”. It has as result type the class SemanticPackage and as context the ServiceProfile class of the SSL metamodel. The body of the expression is the OperationCallExp referring to the operation “=” of the primitive type String. This OperationCallExp has an argument: the StringLiteralExp “HotelName”. Moreover, the OperationCallExp has a navigation source where it must apply. The source is the AttributeCallExp referring to the Attribute “name” of the ServiceAttribute class. The ServiceAttribute class is derived from the navigation source of this AttributeCallExpression. The source is an AssociationEndCallExp referring to the AssociationEnd “Attribute” of the class ServiceProfile, which is the context of the query.

Figure 14 shows the simple QML expression, described so far, in a graphical way (screenshot depicted by the Query Formulator Tool). Users need to browse/navigate (on a tree-view basis) through the desired knowledge base model up to the term where they want to add a constraint (hard/soft). Each hard constraint is handled as a Boolean AND (i.e. the result must always conform to it) while the soft constraints are handled as a fuzzy fOR. To produce the results of the initial expression any soft constraints are used to rank the results of the queries while any hard constraints are used to filter the final result set

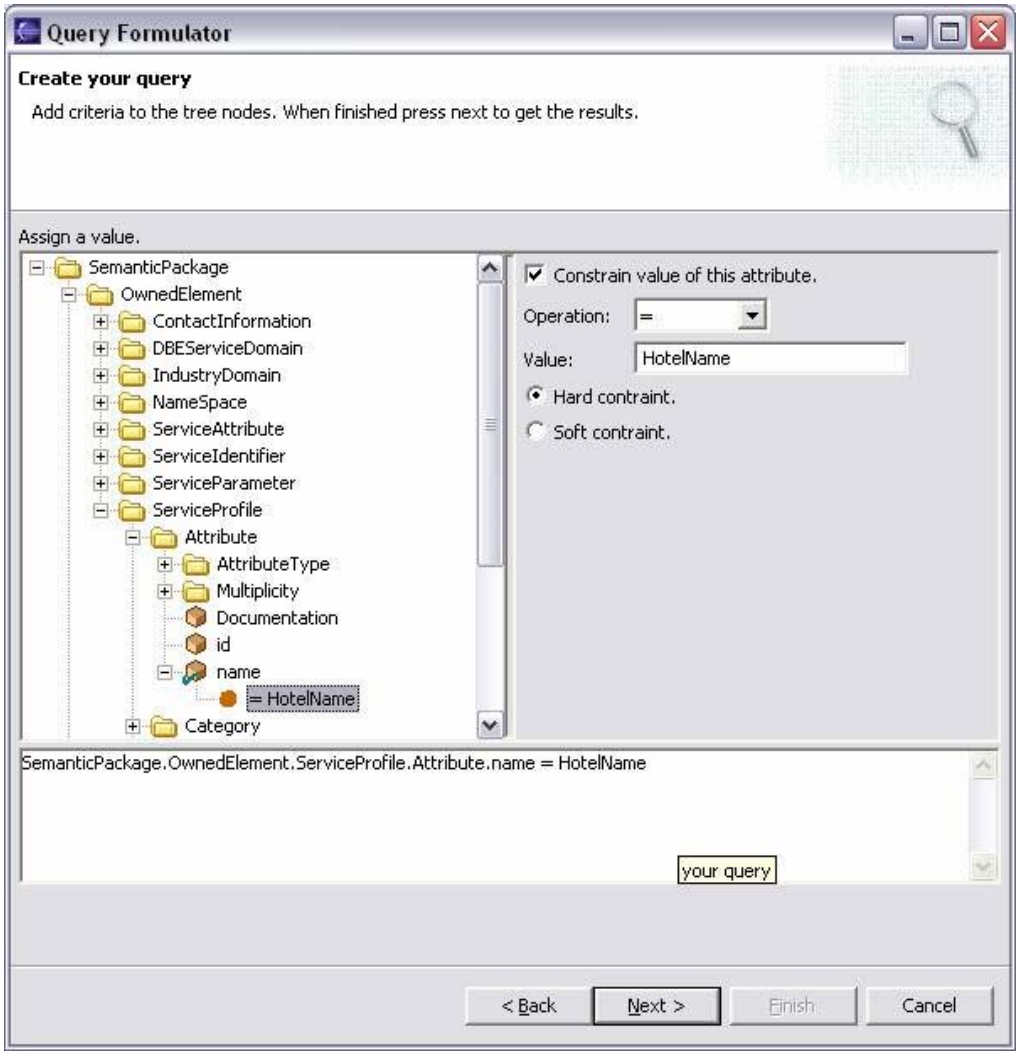

Figure 14: The Query Formulator tool, showing the simple QML expression (Attribute.name = "HotelName").

6.2 Some More Complex QML Expressions

The next example tries to demonstrate some of the powerful capabilities of QML utilizing more complicated functions:

```
Context SSL:ServiceProfile complexQuery: SemanticPackage
query:
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.

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[16],[17] [5]. Finally, a conjunction between these two sentences exists.

A representative part of the above statement is depicted in Figure 15 in QML metamodel elements.

[16] 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.

[17] 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).

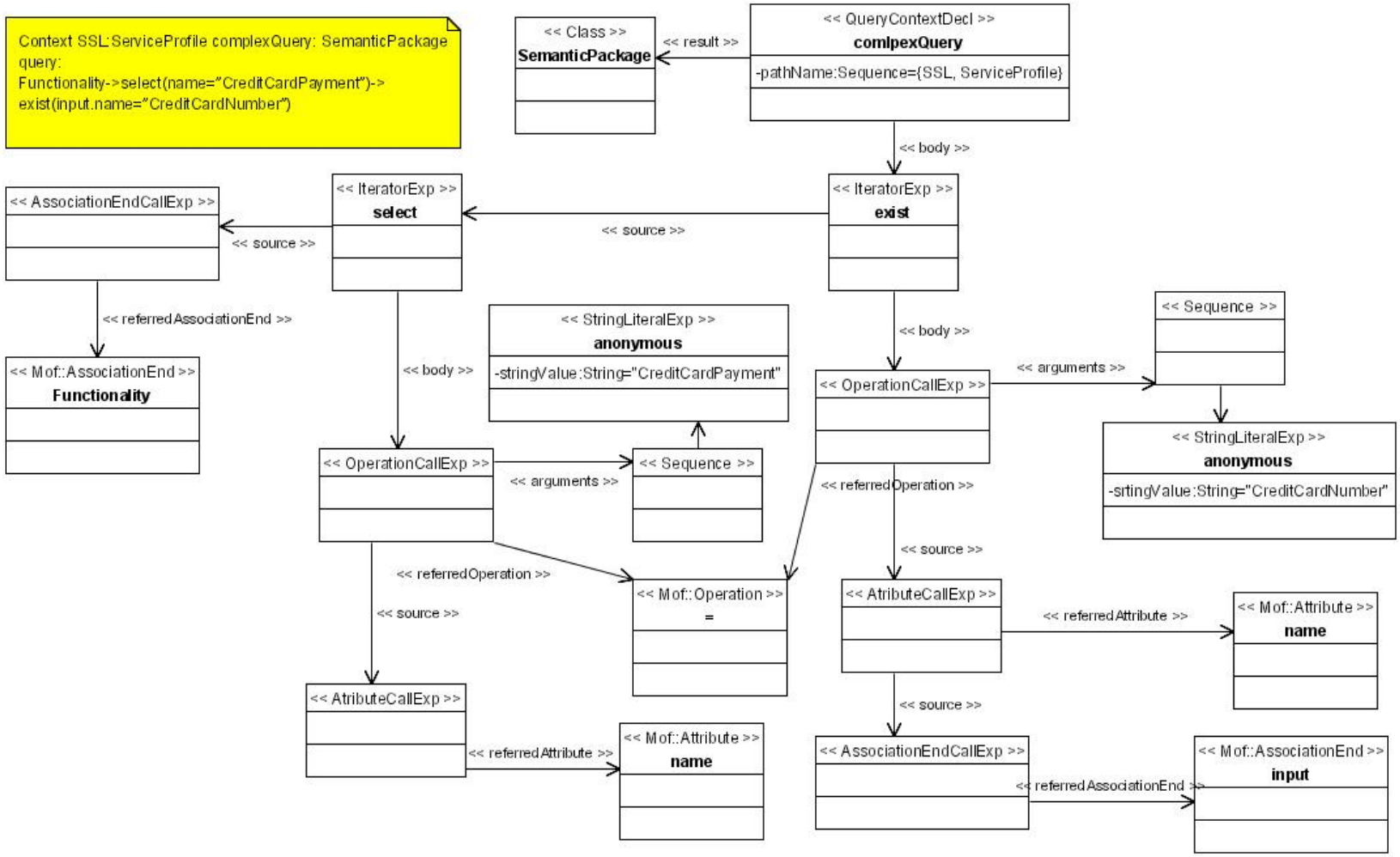

Figure 15: QML representation of a query that presents the use of Iterators in QML and returns the service models that have a Functionality both named "CreditCardPayment" and having an input named "CreditCardNumber".

6.3 A QML Fuzzy Expression

The next example demonstrates a QML expression that utilizes the added fuzzy operators[18].

```
Context SSL::ServiceProfile fuzzyQuery : SemanticPackage
query:
Attribute.name = “HotelName”
fOR
(Functionality.name = “CancelReservation”
fAND
Functionality.name = “MakeReservation”)
```

This query is posed against the SSL metamodel and retrieves all the services that have a ServiceProfile Attribute with name “HotelName” or Functionality with name “CancelReservation” and another one with name “MakeReservation”. The difference from the previous examples is that the resulting services may have any number of the three aforementioned constraints. The more constraints they have the better rank they get.

The Query Formulator tool also offers users the capability of expressing fuzzy query expressions. Users can apply both soft and hard constraints in their queries. Each hard constraint is handled as a Boolean AND (i.e. the result must always conform to it) while the soft constraints are handled as a fuzzy fOR. To produce the results of the initial expression any soft constraints are used to rank the results of the queries while any hard constraints are used to filter the final result set. Figure 16 shows a fuzzy query expression, which asks for services that must have an Attribute named “HotelName” (hard constraint), they may have another Attribute named “Address” (soft constraint) and they may also offer functionality for reservation canceling (soft constraint).

[18] The reader should refer to subsection 5.4 QML Fuzzy Extensions for a more detailed description.

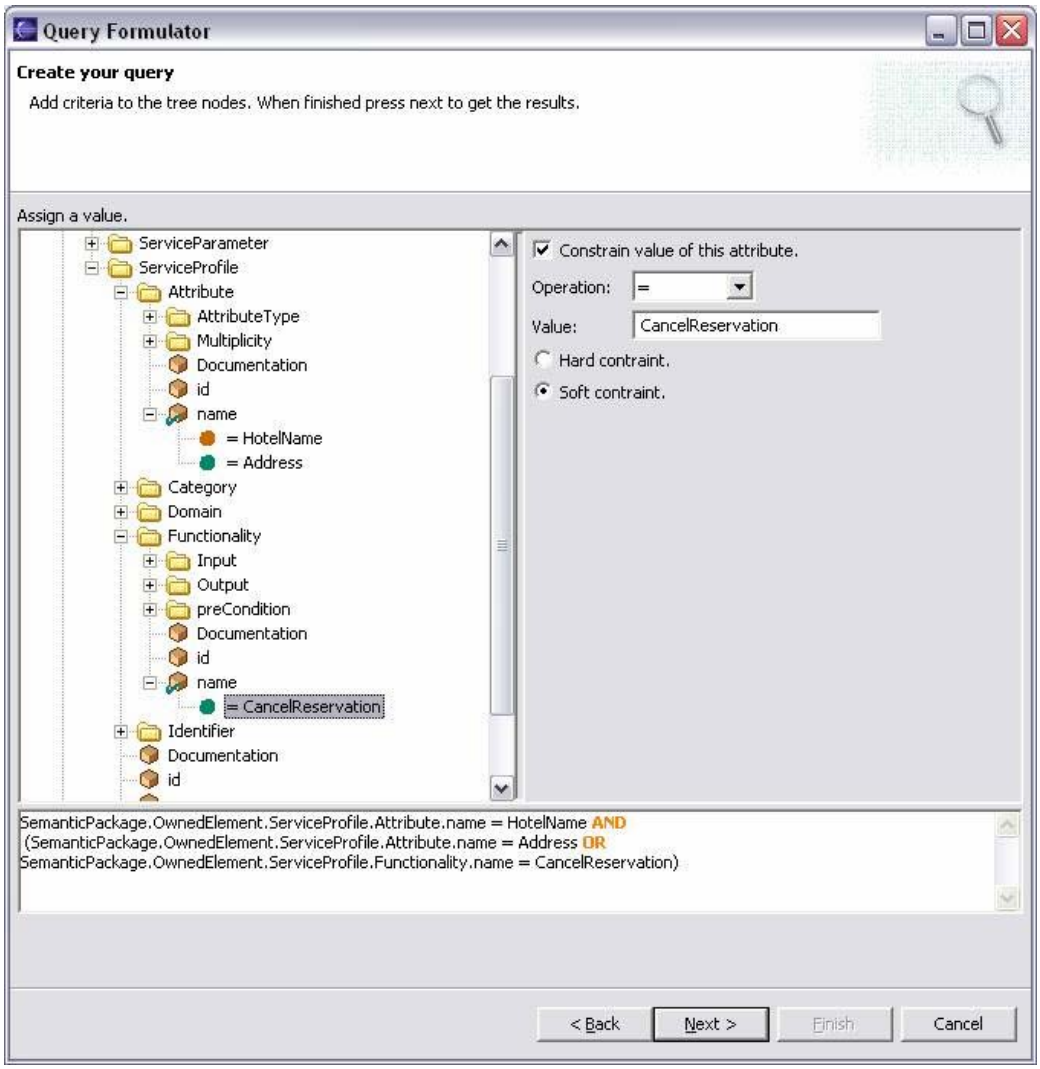

Figure 16: The Query Formulator tool showing a fuzzy query expression, which asks for services that must have an Attribute named “HotelName” (hard constraint), they may have another Attribute named “Address” (soft constraint) and they may also offer functionality for reservation canceling (soft constraint).

The query formulator undertakes to formulate this statement into a valid QML expression:

```
Context SSL::ServiceProfile fuzzyQuery : SemanticPackage
query:
Attribute.name = “HotelName”
AND
(Attribute.name = “Address”
fOR
Functionality.name = “CancelReservation”)
```

The above statement is depicted in Figure 17 in QML metamodel elements.

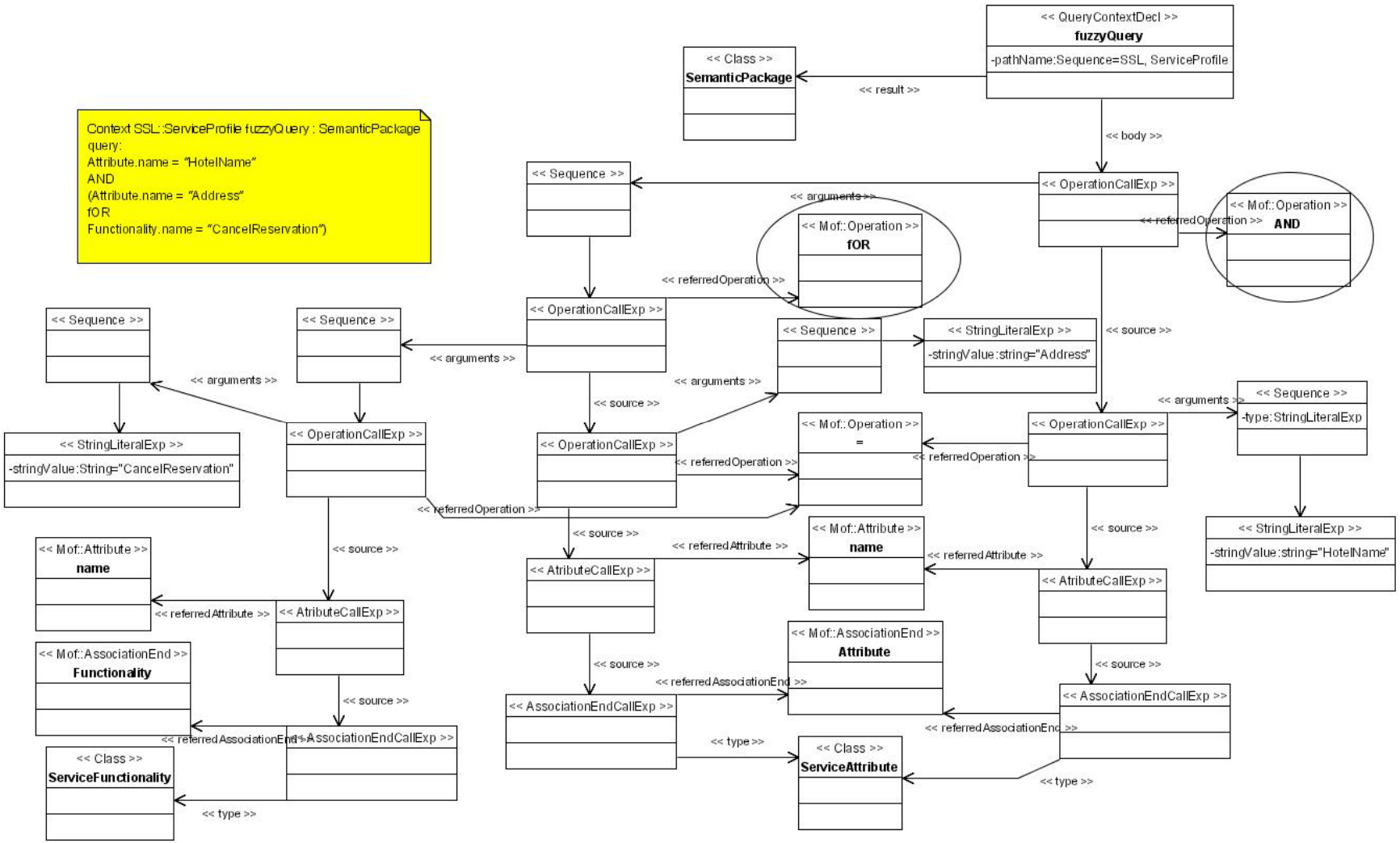

Figure 17: The QML representation of the fuzzy query which asks for services that must have an Attribute named “HotelName” (hard constraint), they may have another Attribute named “Address” (soft constraint) and they may also offer functionality for reservation canceling (soft constraint)

7. Construction of Execution Plans of Query Expressions

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

The syntax tree consists 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 and the result type of the result set. The branches of the syntax tree are *OperationNodes* that provide information about the operation, which must be evaluated, and its parameters. The parameters of *OperationsNodes* can be either *OperationNodes*, or *NavigationNodes* or *LiteralNodes.* Only *NavigationNodes* and *LiteralNodes* can be leaves of the syntax tree. *NavigationNodes* provide information about a navigation path from the context of the query through the specific metamodel's elements[19]. 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. The type of the path elements is known in the query model and is therefore found when needed. *LiteralNodes* provide information about explicit literal values used in the query model, e.g. 1, 'string literal' etc.

The construction process consists of several rules, which when applied to a query model a syntax 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 children might be any of the *NavigationNodes*, *OperationNodes* and *LiteralNodes*.
- 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*[20] 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

[19] it refers to the knowledge base metamodel against of which the specific query is posed

[20] in the current implementation only *exists* is supported

`Functionality->exists(name = “a”) and Functionality->exists(input.name = “b”)`. Such transformations may also exist for the rest of the Iterators (forall etc).

It is interesting to notice that a single constraint may be formulated in two (or more) ways. This observation will be taken into account in the long run in the optimization phase that will be applied when constructing the syntax tree.

7.1 Syntax Tree Construction Example

This example demonstrates how the simple QML expression, that was presented in the previous section, is formulated into a syntax tree. Figure 18 shows the syntax tree that is produced. The ContextNode is the root element. It contains information for the query context, the result type and the query name. The ContextNode has as a child the OperationNode with attribute “=”. This OperationNode has two children; a NavigationNode and a StringLiteralNode. The NavigationNode contains the elements that participate to the navigation process that starts from the context (i.e. ServiceProfile) element; it may also contain the types of the elements. The StringLiteralNode contains the string value “HotelName”.

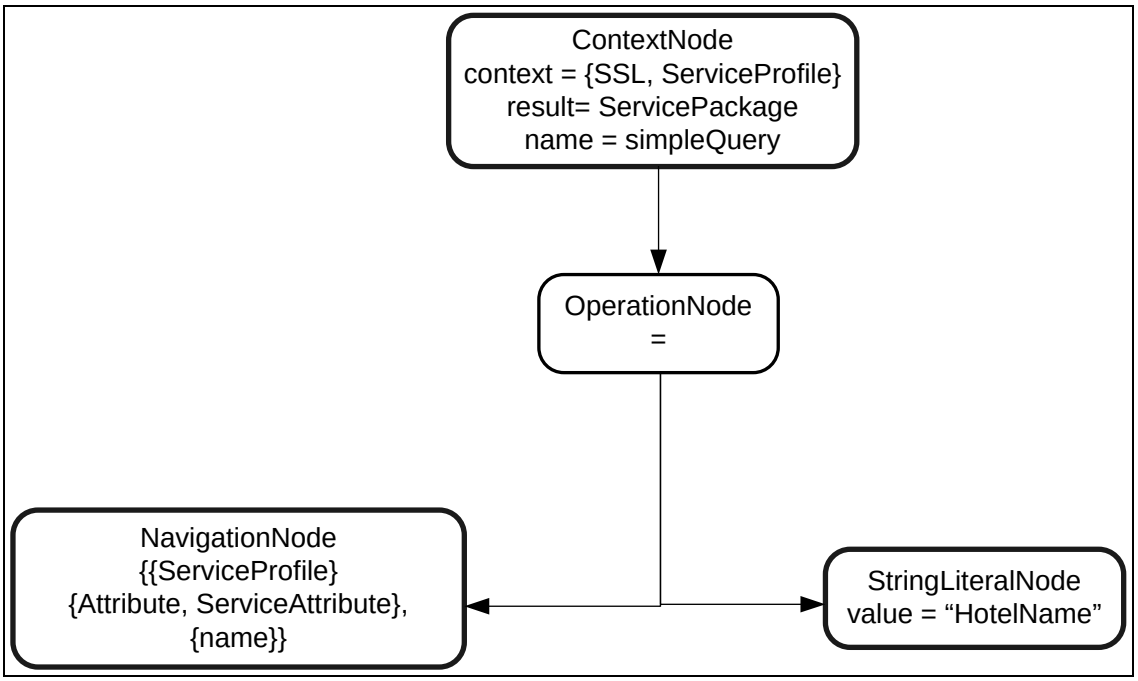

Figure 18: The syntax tree that corresponds to the simple QML expression presented in section 7.

8. Code Generation

The QML Query Metamodel Language, as described in the previous sections, is used in order to pose a query against a specific metamodel. The instances of the metamodel are supposed to be kept in persistent storage managed by a data management system, which can range from a repository or a relational database to an XML repository. In order to be able to answer the question the formulated query must be translated into the appropriate language (code) for querying the stored data depending on the storing system (for instance, SQL for a DBMS, XQuery for an XML database, etc). The task of the code generator is to take as input the syntax tree, produced by the query execution plan constructor and generate the code to be executed in the appropriate query language.

8.1 The Mapper

The syntax tree produced by the query execution plan constructor is based on the queried metamodel. The instances of the metamodel, as described above, can be stored in a number of ways. In order for the code generator to be able to recognize the stored data, a mapping is needed between the concepts in the metamodel and the elements of the schema used to capture this information for the specific data management system. In the current implementation a relational database management system is used. In this case the role of the mapper is to provide a mapping of the elements and relations in the metamodel to the schema of the database. This mapping is not always straightforward and aims to provide the code generator with the table or attribute name corresponding to the element or element attribute, met in the syntax tree. Finally associations in the metamodel can map to relations between tables based on their foreign keys. The structure of the mapper is a list of all tables in the database, their attributes and all the relations in which they participate. For each table, attribute or relation the corresponding element in the metamodel is provided. In the case of a non-straightforward mapping, the administrator of the mapper is responsible for providing the appropriate mappings to the data base schema in order to ensure the resolution of the query.

8.2 The Code Generator

In the case described above, a relational database, the generated code must actually be SQL queries. To produce the SQL query, the Code Generator parses the syntax tree created by the query execution plan constructor. During the parsing, an SQL query is produced for every operation node in the syntax tree. These queries are following the syntax pattern of a simple SQL query of the form[21]:

```
select *
from list of tables
where list of conditions
```

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

The generator must dynamically complete the query with the involved tables, joining conditions and/or constraints. For this purpose the generator has to recognize the following:

- the tables or relations, which correspond to the elements of the metamodel, that appear in the syntax tree. Here, the generator looks up every element in the mapper
- the attribute taking part in each operation and the constraining value, if it exists, of the operation
- the operation itself. It can be either a `simple` operation or a `Boolean` operation.

The `simple` operations are all the operations supported for constraining the value of an attribute (=, <, >=, like etc). In this case the query is constructed as above with the completion of the SQL pattern. The `Boolean` operations refer to the Boolean AND, OR operators. Subsection 5.4 QML Fuzzy Extensions presented the QML extensions that have been considered in order to support QML processing that incorporates IR functionality.

We are using the Extended Boolean Model that is a model that offers 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. In order to actually evaluate the queries one should provide the evaluation fuzzy functions f_{NOT}, f_{AND}, and f_{OR} that correspond to the IR-style Boolean operators mentioned above. There are numerous possible definitions of these functions. We utilize the definitions that correspond the p-Norm Extended Boolean Model, which is the most general one. For the possible specifications (conjunction, disjunction, negation) appropriate SQL queries are produced in order to join the results of the queries in either side of the operation. The results of the constituent queries are stored in intermediate tables as the expressions are evaluated bottom-up. A detailed description of the followed approach and its possible variations is provided in the next section.

As already discussed we have extended the OCLBoolean package of the OCL Basis Library with three new fuzzy Boolean operators, namely fAND, fOR and fNOT that are to be used for IR-style conjunction, disjunction and negation operations respectively. The corresponding evaluation fuzzy functions f_{NOT}, f_{AND}, and f_{OR} have been realized as SQL statements (properly extended) in order to exploit the fact that the current implementation of the knowledge base is on top of a relational database management system. From the end-user's point of view each query condition can be set either as a 'hard' constraint or as a 'soft' one. Each hard constraint is handled as a Boolean AND while the soft constraints are handled as a fOR. To produce the results of the initial expression any soft constraints are used to rank the results of the queries while any hard constraints are used to filter the final result set. As already mentioned the ranking with the fuzzy OR is based on the p-norm Extended Boolean Model. The part of the model used, is implemented to support matching between the elements in the queried metamodel and the features in the query expression. Finally, according to the model, weights are supported for both the expressions and the features participating in the matching.

Next we present the SQL statements that have been produced for the example queries of section 6 Semantics of Query Expressions and Examples.

8.2.1 Generated code example

For the simple QML expression of subsection 6.1 the generated SQL code is as follows:

```
select t0.SemanticPackageID
from DBESSL.SemanticPackage as t0,
```

```
        DBESSL.ServiceProfile as t1,
        DBESSL.ServiceAttribute as t2
    Where t1.SemanticPackageID=t0.SemanticPackageID
    AND t2.ServiceProfileID=t1.ServiceProfileID
    AND t2.ServiceAttributeName = 'HotelName';
```

Since the query contains one and simple operation (=), only one SQL query has been generated. The Code Generator had to recognize the whole path of the expression in the model and then to use the mapper in order to find the corresponding tables in the database. The tables are joined on their foreign keys and finally the requested attribute is added as a condition at the end of the query.

Next is the code generated for the fuzzy query of 6.3:

Query B1 (for the hard constraint):

```
    insert into DBE_SYS.temp1
    select t0.SemanticPackageID
    from DBESSL.SemanticPackage as t0,DBESSL.ServiceProfile as t1,
        DBESSL.ServiceAttribute as t2
    Where t1.SemanticPackageID=t0.SemanticPackageID
    AND t2.ServiceProfileID=t1.ServiceProfileID
    AND t2.ServiceAttributeName = 'HotelName';
```

Query F1 (for the soft constraint):

```
    insert into DBE_SYS.temp2
    select t0.SemanticPackageID
    from DBESSL.SemanticPackage as t0,DBESSL.ServiceProfile as t1,
        DBESSL.ServiceFunctionality as t2
    Where t1.SemanticPackageID=t0.SemanticPackageID
    AND t2.ServiceProfileID=t1.ServiceProfileID
    AND t2.ServiceFunctionalityName = 'CancelReservation';
```

Query F2 (for the soft constraint):

```
    insert into DBE_SYS.temp3
    select t0.SemanticPackageID
    from DBESSL.SemanticPackage as t0,DBESSL.ServiceProfile as t1,
        DBESSL.ServiceAttribute as t2
    Where t1.SemanticPackageID=t0.SemanticPackageID
    AND t2.ServiceProfileID=t1.ServiceProfileID
    AND t2.ServiceAttributeName = 'Address';
```

The expression has resulted into three queries. The first is to be treated as a hard constraint, that means a Boolean AND with the rest of the results. The other two queries

are two soft constraints and their results will be combined in a fuzzy OR using the formula for Extended Boolean model. Notice that the results of the above queries are temporally stored in the tables temp1, temp2 and temp3. The next step is to prepare the data for the fuzzy query. For this purpose all the results of the 'soft' queries (*F1, F2*) are collected in tables fTerms and fWeight. These tables hold all the terms with their weights and all the operations with their weights respectively[22].

```
insert into DBE_SYS.fTerms select id, 8,1 from DBE_SYS.temp3;
insert into DBE_SYS.fWeight(tid, weight) values(8,1);
insert into DBE_SYS.fTerms select id, 9,1 from DBE_SYS.temp2;
insert into DBE_SYS.fWeight(tid, weight) values(9,1);
```

Finally the fuzzy operation is applied as following:

Fuzzy OR query

```
insert into DBE_SYS.results
select p.id, power(sum((power(t.weight, 2) *
       power(w.weight, 2)) / sum(power(w.weight,2))),0.5)
from DBE_SYS.fTerms as t,
 DBE_SYS.fWeight as w,
 DBE_SYS.temp1 as p
where t.tid = w.tid
and t.id = p.id
group by p.id;
```

The formula is based on the p-Norm Extended Boolean Model as presented in Table 3: Weighted n-ary basic evaluation functions for the p-Norm Extended Boolean Model. of subsection 9.1 (Defining a General Weighted Fuzzy Information Retrieval System (GWFIRS)). It is evaluated on the results of F1, F2 as stored in the tables fWeight and fTerms described above. The results of the query B1 (in table temp1) are used in a Boolean AND to filter the final results.

[22] The reader can find a more general description of the process in subsection 9.3 Relational DBMS Implementation of the p-Norm Extended Boolean Model

9. Theoretical and Technical framework of the Knowledge Access Mechanisms

This section is centered on the theoretical framework and the respective implementation of the knowledge access mechanisms using the Knoweldge Base.

As already mentioned the knowledge access context that we consider is twofold. It is related to pure search functionality as well as recommendation mechanisms. It has to be noted that at a technical level the information filtering/retrieval approach is uniform for both desired functionalities. However, whereas the discovery process is based on answering the formulated query expressions based on the available metamodel and model specific information laid in the KB, the recommendation process is based on matching SME profiles (that include preferences on business and/or service semantics) and the underlying information. The two types of functionality provided by the Knowledge Access Module are exported in the form of separate services, namely the KB/SR 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[23]:

4) **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*.
5) **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*.
6) **Service Searching:** Based on the description of a Service *S*, the Recommender Service may find Service Descriptions that match the service description of *S*.

All different kinds of recommendations encapsulated in the three candidate use cases can be modeled using the same general mathematical framework based on information retrieval theory. This framework is summarized here and the implementation of this framework on top of a relational database management system is given. Essentially what we describe is the first approach to the implementation of the Knowledge Access Mechanisms. Next steps include the consideration of SME user profiles and the extension of the mechanisms in order to exploit powerful business and service ontologies that capture the semantics of business models and service descriptions.

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

Universe *I*	Feature space *F*	Queries *Q*	Interpretation
SMEs	SME business description features (like category, type of products, etc).	Preferences of the SME in terms of possible partners	Retrieve SMEs that are similar (i.e. compatible in business terms) to a given SME

[23] 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.

Service descriptions	Service features	Desired service description features	Retrieve Service descriptions that are similar with the service preferences of a given SME
Service descriptions	Service features	Service features that are similar to the features of a give service	Retrieve Service descriptions that could be combined with a given service.

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

The objective now is to define a generalized information retrieval framework that could be used in all of the above scenarios. Moreover, taking into account that the correspondence between information items and features, as well as queries and features, could be implemented in a relational databases management system (RDBMS), we should extend the generalized framework to work on such a system and develop mechanisms that use pure SQL in order to support all kind of recommendation functionalities.

9.1 Defining a General Weighted Fuzzy Information Retrieval System (GWFIRS)

Classical information retrieval systems model both information items and queries as vectors of features [6], [13]. The set of information items satisfying a query vector includes those items that contain any feature present in the query vector.

In contrast, traditional Boolean logic considers a query as a Boolean expression that consists of features as atoms and the three Boolean operators (AND, OR, NOT) as connectives. The answer set of a query is the set of information items that satisfy the Boolean expression. The drawback of the Boolean approach is that an information item is considered either as relevant or non-relevant to a query. There is no means of ranking the information items with respect to a degree of similarity to the query.

To bypass this drawback, the Extended Boolean Model has been introduced in the information retrieval literature. This 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 [7], [8] prove its superior performance in comparison with the traditional information retrieval models. To make things more concrete, we present here a definition, properly adapted to the needs of the Recommender:

Definition: *Weighted Fuzzy Information Retrieval System (WFIRS):*

A WFIRS **C** is defined as a quadruple $\mathbf{C} = \langle F, I, Q, E \rangle$ where:

F is the set of *features* used to describe information items and to formulate queries.

I is a set of *information items*. Each information item corresponds to a function from *F* to the interval [0,1]. This is equivalent to say that $I \subseteq [0,1]^F$.

Q is a set of queries that are identified by the system. They correspond to Boolean expressions using features as atoms and operators AND, OR, NOT as connectives. In addition each sub-query is associated with a relative weight from [0,1] that specifies its relative importance with respect to the rest of the sub-queries. Examples of valid queries are: $\langle q_1, w_1 \rangle AND \langle q_2, w_2 \rangle$ and $\langle q_1, w_1 \rangle OR \langle q_2, w_2 \rangle$.

E is an evaluation function $E : Q \times I \rightarrow [0,1]$ that gives a value from [0,1] to any valid query from *Q* with respect to each information item. That is, $E(q,i)$ is the similarity ranking of information item *i* with respect to the query *q*. *E* is then defined recursively based on the evaluation functions of the logical operators:

$$E(\langle q_1, w_1 \rangle AND \langle q_2, w_2 \rangle, i) = f_{AND}(\langle E(q_1, i), w_1 \rangle, \langle E(q_2, i), w_2 \rangle)$$

$$E(\langle q_1, w_1 \rangle OR \langle q_2, w_2 \rangle, i) = f_{OR}(\langle E(q_1, i), w_1 \rangle, \langle E(q_2, i), w_2 \rangle)$$

$$E(NOT q_1, i) = f_{NOT}(E(q_1, i))$$

$E(f, i) = i(f)$ (note that *i* is a function from *F* to [0,1]).

For a query $q \in Q$, the subset *ANS(q)* of *I* containing all the elements of *I* for which the evaluation function F has a positive (>0) value, is the answer to q. This subset is defined as:

$$ANS(q) = \{\langle i, E(q,i) \rangle \in I \times (0,1] \mid E(q,i) > 0\}.$$

Note that in practice we might get too many possible answers, most with too small values. In this case we may consider only the values with $E(q,I) > E_{min}$, for some application appropriate E_{min}, or take the first *N* highest ranked entries, again for some *N* appropriate for the application.

In order to actually evaluate the queries recognized by a WFIRS, one should give the evaluation functions f_{NOT}, f_{AND}, and f_{OR}. There are numerous possible definitions of these functions. The following table presents the definitions that correspond 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.

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

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

9.2 Incorporating Negative Query Weights into a WFIRS

It is also possible that the Recommender could use negative preference values to model dislikes. To handle this situation, the definition of a WFIRS should be adapted to allow for negative query weights. We assume that a term with a negative query weight is equivalent to the same term with a positive weight with the NOT operator applied to it. In consequence, a query of the form $q = \langle f_1, 1 \rangle AND \langle f_2, 0.5 \rangle AND \langle f_3, -0.3 \rangle AND \langle f_4, -1 \rangle$ means that the qualifying items should have features f_1, f_2 and not have f_3 nor f_4.

One straightforward way to integrate such weights in a WFIRS is to adopt the following definition:

Definition: Generalized WFIRS (GWFIRS):

> A Generalized Weighted Fuzzy IRS **c** is defined as a Weighted Fuzzy IRS **c** = $\langle F, I, Q, E \rangle$ where the query weights in the set Q are real numbers in the interval [-1,1].

In addition, the evaluation formulae should be revised so that negative query weights are incorporated. The revision is based on the following substitution of a_i's in the formulae of Table 3:

$$a_i \leftarrow \frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\} \cdot a_i$$

$$1 - a_i \leftarrow \frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\} \cdot a_i$$

where {.} is defined as follows:

$$\{x\} = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ +1, & x > 0 \end{cases}$$

It is straightforward to see that the above definitions yield:

$$\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\} \cdot a_i = \begin{cases} 1 - a_i, & w_i < 0 \\ 0, & w_i = 0 \\ a_i, & w_i > 0 \end{cases}$$

$$\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\} \cdot a_i = \begin{cases} a_i, & w_i < 0 \\ 0, & w_i = 0 \\ 1 - a_i, & w_i > 0 \end{cases}$$

Based on the above equations, the evaluation functions of the p-Norm model are revised as follows:

$f_{AND}((a_1,w_1),\ldots,(a_n,w_n))$	$f_{OR}((a_1,w_1),\ldots,(a_n,w_n))$	$f_{NOT}(a)$
$1-\left(\frac{\sum_{i=1}^{n}\left(\frac{\lvert\{w_i\}\rvert+\{w_i\}}{2}-\{w_i\}\cdot a_i\right)^p\lvert w_i\rvert^p}{\sum_{i=1}^{n}\lvert w_i\rvert^p}\right)^{\frac{1}{p}}$	$\left(\frac{\sum_{i=1}^{n}\left(\frac{\lvert\{w_i\}\rvert-\{w_i\}}{2}+\{w_i\}\cdot a_i\right)^p\lvert w_i\rvert^p}{\sum_{i=1}^{n}\lvert w_i\rvert^p}\right)^{\frac{1}{p}}$	$1-a$

Table 4: Evaluation functions for p-Norm GWFIRS.

9.3 Relational DBMS Implementation of the p-Norm Extended Boolean Model

There are numerous strategies for the implementation of a GWFIRS on top of a RDBMS [9], [11], [12]. However, there is a straightforward implementation of a p-Norm GWFIRS on top of a RDBMS 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 of the form:

$$q = \langle t_{q1}, w_{q1} \rangle \, OR \ldots OR \, \langle t_{qn}, w_{qn} \rangle, \quad -1 \le w_{qi} \le 1, i = 1, \ldots, n \ .$$

Let us further assume, that we have the following relational database schema (primary keys are in bold face, foreign keys are underlined) to store the information handled by the GWFIRS:

- FEATURES(**FeatureID**, FeatureName) is used to store the feature space of the system.
- ITEMS(**ItemID**, ItemName) stores the set of information items.
- ITEMS_FEATURES(**ItemID**, **FeatureID**, Weight) stores the association between items and features (ie. the functions i:F→[0,1]).
- QUERIES(**QueryID**, **FeatureID**, QueryWeight) stores the queries that are recognized by the system.

In our implementation the tables above are used to store all the intermediate results during the execution of the SQL statements generated by the code generator (discussed in the previous section). The results are retrieved based on the primary key of the tables participating in the query and corresponding to the root element of the metamodel tree or subtree used as a context in the initial QML query expression.

To following SQL statement can be used to compute the result set for any disjunctive query recognized by the system (refer to table 4 for the definition of the f_{OR} evaluation function of a GWFIRS following the p-Norm model):

SQL[1]:

```
SELECT Q.QueryID, IF.ItemID,
    POWER(SUM(POWER((ABS(SGN(Q.QueryWeight))-
    SGN(Q.QueryWeight))/2+SGN(Q.QueryWeight)*IF.Weight,p)*
    POWER(ABS(Q.QueryWeight),p))
```

```
        /SUM(POWER(ABS(Q.QueryWeight),p)),1/p)
    FROM ITEMS_FEATURES AS IF, QUERIES AS Q
    WHERE IF.FeatureID=Q.FeatureID AND Q.QueryID=@qid
    GROUP BY Q.QueryID, IF.ItemID
```

It is quite easy to develop a similar SQL query in case that the queries recognized by the system are simple conjunctive queries of the form:

$$q = \langle t_{q1}, w_{q1} \rangle \; AND \ldots AND \; \langle t_{qn}, w_{qn} \rangle, \quad -1 \le w_{qi} \le 1, i = 1, \ldots, n$$

The corresponding SQL statement is the following (refer to table 2 for the definition of the f_{AND} evaluation function for the p-Norm model):

SQL[2]:

```
    SELECT Q.QueryID, IF.ItemID,
        1 - POWER(SUM(POWER((ABS(SGN(Q.QueryWeight))+
        SGN(Q.QueryWeight))/2-SGN(Q.QueryWeight)*IF.Weight,p)*
        POWER(ABS(Q.QueryWeight),p))
        /SUM(POWER(ABS(Q.QueryWeight),p)),1/p)
    FROM ITEMS_FEATURES AS IF, QUERIES AS Q
    WHERE IF.FeatureID=Q.FeatureID AND Q.QueryID=@qid
    GROUP BY Q.QueryID, IF.ItemID
```

The careful reader will notice that the statement *SQL[2]* differs from *SQL[1]* in the third field of the **SELECT** statement, the one that computes the weights of the result set.

9.4 Reducing the Space Overhead in the database

It is worth noting that all useful applications of a GWFIRS correspond to environments where the majority of Item-Feature associations correspond to zero weights (i.e. non-cares). The technique presented so far, relies on storing all Item-Feature associations (both zero and non-zero ones) in the ITEMS_FEATURES table. One way to reduce this huge space overhead is to store only non-zero associations. In this case, there is an assumption introduced in the design of the database that reads: "All Item-Feature associations not explicitly stored in the database are zero-valued". However, this method has a serious implication in the computation of the result set: The SQL statements above give wrong results if the query terms that correspond to zero-valued features are omitted from the ITEMS_FEATURES table. This is the so-called "*zero dependency problem*". It is almost trivial to see that the evaluation formulae that suffer from this problem are the ones that do not satisfy the following property: $f(\langle 0, w \rangle, \langle a_1, w_1 \rangle) = f(\langle a_1, w_1 \rangle)$. In a GWFIRS following the p-Norm model both f_{AND} and f_{OR} suffer from the zero dependency problem as one can confirm from the formulae of table 4.

There is however a technique to overcome the zero dependency problem by re-writing each problematic evaluation formula: Start from the initial formula and split the sum in the numerator in two parts: the first sums over the zero-valued a_i's and the second over the non-zero ones. Then make all necessary algebraic manipulations so that in the final form of the formula only non-zero a_i's are present. Note also that:

$$\left(\frac{|\{w_i\}| \pm \{w_i\}}{2}\right)^p = \left(\frac{|\{w_i\}| \pm \{w_i\}}{2}\right),\ p > 0$$

that will prove to be useful for simplification in the formula.

Let us first apply this technique for the evaluation function f_{AND} of table 4:

$$f_{AND}\left(\langle a_1, w_1\rangle,\ldots,\langle a_n, w_n\rangle\right) = 1 - \left(\frac{\sum_{i=1}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\}\cdot a_i\right)^p |w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}} =$$

$$= 1 - \left(\frac{\sum_{\substack{i=1\\a_i=0}}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2}\right)\cdot|w_i|^p + \sum_{\substack{i=1\\a_i\neq 0}}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\}\cdot a_i\right)^p |w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}} =$$

$$= 1 - \left(\frac{\sum_{i=1}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2}\right)\cdot|w_i|^p - \sum_{\substack{i=1\\a_i\neq 0}}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2}\right)\cdot|w_i|^p + \sum_{\substack{i=1\\a_i\neq 0}}^{n}\left(\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\}\cdot a_i\right)^p |w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}} =$$

$$= 1 - \left(\frac{\sum_{i=1}^{n}\frac{|\{w_i\}| + \{w_i\}}{2}|w_i|^p + \sum_{\substack{i=1\\a_i\neq 0}}^{n}\left(\left(\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\}\cdot a_i\right)^p - \frac{|\{w_i\}| + \{w_i\}}{2}\right)\cdot|w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}}$$

The evaluation function f_{OR} is revised as follows:

$$f_{OR}\left(\langle a_1, w_1\rangle,\ldots,\langle a_n, w_n\rangle\right) = \left(\frac{\sum_{i=1}^{n}\left(\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\}\cdot a_i\right)^p |w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}} =$$

$$= \left(\frac{\sum_{\substack{i=1\\a_i=0}}^{n}\left(\frac{|\{w_i\}| - \{w_i\}}{2}\right)\cdot|w_i|^p + \sum_{\substack{i=1\\a_i\neq 0}}^{n}\left(\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\}\cdot a_i\right)^p |w_i|^p}{\sum_{i=1}^{n}|w_i|^p}\right)^{\frac{1}{p}} =$$

$$= \left(\frac{\sum_{i=1}^{n} \left(\frac{|\{w_i\}| - \{w_i\}}{2} \right) \cdot |w_i|^p - \sum_{\substack{i=1 \\ a_i \neq 0}}^{n} \left(\frac{|\{w_i\}| - \{w_i\}}{2} \right) \cdot |w_i|^p + \sum_{\substack{i=1 \\ a_i \neq 0}}^{n} \left(\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\} \cdot a_i \right)^p |w_i|^p}{\sum_{i=1}^{n} |w_i|^p} \right)^{\frac{1}{p}} =$$

$$= \left(\frac{\sum_{i=1}^{n} \frac{|\{w_i\}| - \{w_i\}}{2} |w_i|^p + \sum_{\substack{i=1 \\ a_i \neq 0}}^{n} \left(\left(\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\} \cdot a_i \right)^p - \frac{|\{w_i\}| - \{w_i\}}{2} \right) \cdot |w_i|^p}{\sum_{i=1}^{n} |w_i|^p} \right)^{\frac{1}{p}}$$

Let us now summarize the results:

$f_{AND}((a_1,w_1),\ldots,(a_n,w_n))$	$f_{OR}((a_1,w_1),\ldots,(a_n,w_n))$
$1 - \left(\frac{\sum_{i=1}^{n} \frac{|\{w_i\}| + \{w_i\}}{2} |w_i|^p + \sum_{\substack{i=1 \\ a_i \neq 0}}^{n} \left(\left(\frac{|\{w_i\}| + \{w_i\}}{2} - \{w_i\} \cdot a_i \right)^p - \frac{|\{w_i\}| + \{w_i\}}{2} \right) \cdot |w_i|^p}{\sum_{i=1}^{n} |w_i|^p} \right)^{\frac{1}{p}}$	$\left(\frac{\sum_{i=1}^{n} \frac{|\{w_i\}| - \{w_i\}}{2} |w_i|^p + \sum_{\substack{i=1 \\ a_i \neq 0}}^{n} \left(\left(\frac{|\{w_i\}| - \{w_i\}}{2} + \{w_i\} \cdot a_i \right)^p - \frac{|\{w_i\}| - \{w_i\}}{2} \right) \cdot |w_i|^p}{\sum_{i=1}^{n} |w_i|^p} \right)^{\frac{1}{p}}$

Table 5: Revised evaluation functions for p-Norm GWFIRS to overcome the zero dependency problem.

The implementation in SQL needs some attention. Notice that the denominator of the revised f_{OR} and f_{AND} formulae sum over all query terms. The numerator has one part that sums over all query terms and another part that sums over non-zero item weights. To handle this issue, the revised SQL statements for computing the answer to simple disjunctive (or conjunctive) queries, are written as a series of two SQL statements: The first iterates over all the query terms to compute the denominator and the first part of the numerator. The second performs the join between the ITEMS_FEATURES and the QUERIES tables and computes the final result.

SQL[3]:

```
SELECT @denominator=sum(POWER(ABS(QueryWeight),p)),
       @numerator1=sum((ABS(SGN(QueryWeight))-SGN(QueryWeight))/
       2*POWER(ABS(QueryWeight),p))
FROM QUERIES
WHERE Q.QueryID=@qid;


SELECT Q.QueryID, IF.ItemID,
       (POWER(@numerator1+
       SUM(POWER((ABS(SGN(Q.QueryWeight))-
       SGN(Q.QueryWeight))/2+SGN(Q.QueryWeight)*IF.Weight,p)-
       (ABS(SGN(Q.QueryWeight))-SGN(Q.QueryWeight))/2)*
       POWER(ABS(Q.QueryWeight),p))
```

```
        /@denominator,1/p)
    FROM ITEMS_FEATURES AS IF, QUERIES AS Q
    WHERE IF.FeatureID=Q.FeatureID AND Q.QueryID=@qid
    GROUP BY Q.QueryID, IF.ItemID
```

SQL[4]:

```
    SELECT @denominator=sum(POWER(ABS(QueryWeight),p)),
        @numerator1=sum((ABS(SGN(QueryWeight))+SGN(QueryWeight))/
        2*POWER(ABS(QueryWeight),p))
    FROM QUERIES
    WHERE Q.QueryID=@qid;

    SELECT Q.QueryID, IF.ItemID,
        1 - (POWER(@numerator1+
        SUM(POWER((ABS(SGN(Q.QueryWeight))+
        SGN(Q.QueryWeight))/2-SGN(Q.QueryWeight)*IF.Weight,p)-
        (ABS(SGN(Q.QueryWeight))+SGN(Q.QueryWeight))/2)*
        POWER(ABS(Q.QueryWeight),p))
        /@denominator,1/p)
    FROM ITEMS_FEATURES AS IF, QUERIES AS Q
    WHERE IF.FeatureID=Q.FeatureID AND Q.QueryID=@qid
    GROUP BY Q.QueryID, IF.ItemID
```

9.5 Handling Arbitrary Complex Queries

What happens if the queries to be evaluated are arbitrary complex queries instead of simple conjunctive or disjunctive queries? This is a critical question in order to make the above-described techniques applicable in our context. The solution is based on two steps. The first one is the decomposition of the complex queries that should be evaluated (e.g. complex preferences of an SME) into simple (either conjunctive or disjunctive) hierarchical components that takes place during the construction of the query's execution plan. The second step refers to evaluation of these components following a bottom up approach, that is taking place during the code generation phase. The intermediate results should be stored in temporary tables that may need appropriate indices to ensure efficient evaluation of the corresponding SQL queries. We are currently investigating these issues to provide a more generic framework in the next release of the recommender.

10. Conclusions

In this document we have described the mechanisms implemented in order to support knowledge access in DBE. This implementation was demonstrated on January 2005, during the DBE review, as part of the integrated first year prototype of DBE.

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. Personalization 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 document 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 current implementation of the recommender has been also described. The Recommender is using the DBE knowledge base implementation, which is based on the MDR repository and a relational database to store and manage DBE models and metamodels. Next steps include further testing, refinement and optimization of the implementation, the consideration of SME user profiles and the extension of the mechanisms in order to exploit powerful business and service ontologies that capture the semantics of business models and service descriptions. Furthermore, since the next implementation of the DBE Knowledge Base will have a p2p nature, the recommender will be strongly affected by the p2p Knowledge Base framework that will be adopted. The KB related issues that will be taken into account for the support of the recommendation process in the p2p environment are related to semantic- based indexing schemes, semantic-based knowledge distribution and mechanisms for knowledge replication, ontology mappings between domain and local ontologies and storage and maintenance of critical information such as SME profiles and data.

Regarding QML, some further issues are under investigation. The first issue refers to the requirement that the scope of a query should be associated to more than one context. This is a very useful requirement in order to allow queries to combine semantic information from more than one metamodels. In the current implementation this requirement is fulfilled by implicitly declaring the complete path for locating the desired semantic information. We are considering other ways to allow *QueryContextDecl* to be associated to more than one context by providing a suitable environment to declare all the context-related information (e.g. paths, variables, etc.) and to ensure the more effective reference and utilization of this information in the query's body.

We are also considering a more generic support for QML fuzzy extensions together with the precise semantics of QML expressions with fuzzy operators. We are investigating the

approach to extend the QML metamodel with a new Boolean IR-style predicate $e \sim S$, where e is any query expression that returns true if at least one element from the sequence returned by e matches the IR search specification, S. This approach will actually overload the common Boolean operators of OCL with IR-style semantics in a transparent way. The operators will then be suitably interpreted based on the query predicate. However such an extension will be intrinsic to the QML metamodel.

11. 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 Modeling Language
DBMS	Data 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-formedness 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 modeled system
PSM	Platform Specific Model of a modeled 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 artifacts 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 Modeling Language: A method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development; as well as for business modeling.
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.

12. Bibliography

[1] Belkin N. J., Croft W. B. "Information Filtering and Information Retrieval: two sides of the same coin?" Communications of the ACM, 35(12), 29--38, 1992

[2] Boldsoft, International Business Machines Corporation, IONA and Adaptive Ltd. "OCL 2.0 OMG Final Adopted Specification", (OMG Document ptc/03-10-14), October 2003

[3] http://mdr.netbeans.org

[4] http://incubator.apache.org/derby

[5] DBE Internal Document: "Knowledge Base Design and Implementation Status" authored by TUC.

[6] Salton G., Buckley C.: "Introduction to Modern Information Retrieval", McGraw-Hill Book Company, New York, 1982

[7] Lee J. H., "Properties of Extended Boolean Models in Information Retrieval", In Proceedings of the 17th ACM SIGIR International Conference on Research and Development in Information Retrieval, 1994, 182-190.

[8] Lee J. H., Kim W. Y., Kim M. H., Lee Y. J. "On the evaluation of boolean operators in the extended boolean framework", In Proceedings of the 16th ACM SIGIR International Conference on Research and Development in Information Retrieval, 1993, 291-297.

[9] Raghavan S., Garcia-Molina H.: "Integrating Diverse Information Management Systems: A Brief Survey", IEEE Data Engineering Bulletin, Vol. 24, No. 4, pp. 44-52, December 2001.

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

[11] Rantzau R., Shapiro L.D., Mitschang B., Wang Q.: "Algorithms and applications for universal quantification in relational databases", Information Systems, Vol. 28, No. 1-2, pp. 3-32, 2003

[12] Ortega-Binderberger M., Chakrabarti K., Mehrotra S.: "An Approach to Integrating Query Refinement in SQL", In Proceedings of the 8th International Conference on Extending Database Technology, pp. 15-33, March 2002

[13] Baeza-Yates R., Ribeiro-Neto B.: "Modern Information Retrieval", ACM Press, New York, 1999

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

[15] "SQL", ISO/IEC 9075:1999.

[16] "XQuery 1.0: An XML Query Language", http://www.w3.org/TR/xquery, November 2002.

[17] "XML Path Language (XPath) Version 1.0", http://www.w3.org/TR/xpath, November 1999.

[18] T. Attwood et al. "The Object database standard /ODMG-93", Morgan-Kaufmann, San Mateo, 1994.

13. Appendix A: Query API

org.dbe.kb.proxy

Query Interface QI

Public interface **QI**

The KB and SR Proxy interfaces (KBI, SRI) contain the method getQueryProxy that returns an object of a class that implements the QI interface. Using this interface one can formulate queries based upon a metamodel and submit them to KB or SR.

Method Summary	
void	exportQueryModel(java.io.OutputStream outStream) Export the cached QML model (query) to an output stream
void	exportQueryModel(java.lang.String filename) Export the cached QML model (query) to a file in the local filesystem
void	generateQuerySpecificJMI(java.lang.String directory) Generate QML – specific interfaces that can be used to formulate queries. Specify the directory where the interface classes will be generated
java.lang.String[]	getMetamodelNames() Get the valid metamodels (names) that can be used for queries
javax.jmi.reflect.RefPackage	getModelPackage(java.lang.String modelName) Get the ModelPackage of a specific metamodel.
javax.jmi.reflect.RefPackage	getQMLPackage() Get The QMLPackage. One can use the JMI reflective APIs to create QML models, or alternative the more comprehensive APIs generated from *generateQuerySpecificJMI()*.
void	importQueryModel(java.io.InputStream inStream) Import an already existing query from an input stream in order to use it
void	importQueryModel(java.lang.String filename) Import an already existing stored query from a local file in order to use it
java.util.Collection	(KB/SR)submitQuery() Submit the formulated query to KB or SR. The XMI representation of the query is produced and is streamed to the KB or SR service. The result is a collection of qualified models or service manifests respectively.
void	newQueryModel(java.lang.String name) Create a new query and clear proxy's cache.

org.dbe.kb.metamodel.qml

Interface QmlPackage

Public interface **QmlPackage**

QmlPackage contains context declarations and core packages. The Context Declarations and Core Package comprise a collection of interface classes for constructing new or browsing

existing QML models (queries). These interfaces have been generated by using the method generateQuerySpecificJMI() of the QI interface. In the following we briefly present the QML-specific APIs. More information can be found at installed KB and SR service information pages.

Method Summary	
ContextDeclarationsPackage	getContextDeclarations() Returns nested package ContextDeclarations.
CorePackage	getCore() Returns nested package Core.
javax.jmi.model.ModelPackage	getModel()

org.dbe.kb.metamodel.qml.contextdeclarations

Interface ContextDeclarationsPackage

Public interface **ContextDeclarationsPackage**

Method Summary	
AttributeContextDeclClass	getAttributeContextDecl() Returns AttributeContextDecl class proxy object.
AttributeContextDeclTypeAssoc	getAttributeContextDeclTypeAssoc() Returns AttributeContextDeclTypeAssoc association proxy object.
ConstraintContextDeclClass	getConstraintContextDecl() Returns ConstraintContextDecl class proxy object.
ContextDeclarationClass	getContextDeclaration() Returns ContextDeclaration class proxy object.
ContextDeclarationBodyExpressionAssoc	getContextDeclarationBodyExpressionAssoc() Returns ContextDeclarationBodyExpressionAssoc association proxy object.
DefinitionContextDeclClass	getDefinitionContextDecl() Returns DefinitionContextDecl class proxy object.
DefinitionContextDeclAttributesAssoc	getDefinitionContextDeclAttributesAssoc() Returns DefinitionContextDeclAttributesAssoc association proxy object.
DefinitionContextDeclOperationsAssoc	getDefinitionContextDeclOperationsAssoc() Returns DefinitionContextDeclOperationsAssoc association proxy object.
InvariantContextDeclClass	getInvariantContextDecl() Returns InvariantContextDecl class proxy object.
OperationClass	getOperation() Returns Operation class proxy object.
OperationContextDeclClass	getOperationContextDecl()

	Returns OperationContextDecl class proxy object.
OperationContextDeclOperationAssoc	getOperationContextDeclOperationAssoc() Returns OperationContextDeclOperationAssoc association proxy object.
OperationDefinitionClass	getOperationDefinition() Returns OperationDefinition class proxy object.
OperationDefinitionBodyExpressionAssoc	getOperationDefinitionBodyExpressionAssoc() Returns OperationDefinitionBodyExpressionAssoc association proxy object.
OperationDefinitionParametersAssoc	getOperationDefinitionParametersAssoc() Returns OperationDefinitionParametersAssoc association proxy object.
OperationDefinitionTypeAssoc	getOperationDefinitionTypeAssoc() Returns OperationDefinitionTypeAssoc association proxy object.
OperationParametersAssoc	getOperationParametersAssoc() Returns OperationParametersAssoc association proxy object.
OperationTypeAssoc	getOperationTypeAssoc() Returns OperationTypeAssoc association proxy object.
QueryContextDeclClass	getQueryContextDecl() Returns QueryContextDecl class proxy object.
QueryContextDeclTypeAssoc	getQueryContextDeclTypeAssoc() Returns QueryContextDeclTypeAssoc association proxy object.

org.dbe.kb.metamodel.qml.core

Interface CorePackage

Public interface **CorePackage**

Method Summary	
ExpressionsPackage	getExpressions() Returns nested package Expressions.
TypesPackage	getTypes() Returns nested package Types.

org.dbe.kb.metamodel.qml.core.expressions

Interface ExpressionsPackage

Public interface **ExpressionsPackage**

Method Summary

AppliedPropertySourceAssoc	getAppliedPropertySourceAssoc() Returns AppliedPropertySourceAssoc association proxy object.
AssociationEndCallExpClass	getAssociationEndCallExp() Returns AssociationEndCallExp class proxy object.
AssociationEndNavigationSourceAssoc	getAssociationEndNavigationSourceAssoc() Returns AssociationEndNavigationSourceAssoc association proxy object.
AssociationEndReferredAssociationEndAssoc	getAssociationEndReferredAssociationEndAssoc() Returns AssociationEndReferredAssociationEndAssoc association proxy object.
AttributeCallExpClass	getAttributeCallExp() Returns AttributeCallExp class proxy object.
AttributeCallExpReferredAttributeAssoc	getAttributeCallExpReferredAttributeAssoc() Returns AttributeCallExpReferredAttributeAssoc association proxy object.
BaseExpResultAssoc	getBaseExpResultAssoc() Returns BaseExpResultAssoc association proxy object.
BooleanLiteralExpClass	getBooleanLiteralExp() Returns BooleanLiteralExp class proxy object.
CollectionItemClass	getCollectionItem() Returns CollectionItem class proxy object.
CollectionLiteralExpClass	getCollectionLiteralExp() Returns CollectionLiteralExp class proxy object.
CollectionLiteralExpPartsAssoc	getCollectionLiteralExpPartsAssoc() Returns CollectionLiteralExpPartsAssoc association proxy object.
CollectionLiteralPartClass	getCollectionLiteralPart() Returns CollectionLiteralPart class proxy object.
CollectionLiteralTypeAssoc	getCollectionLiteralTypeAssoc() Returns CollectionLiteralTypeAssoc association proxy object.
CollectionRangeClass	getCollectionRange() Returns CollectionRange class proxy object.
CollectionRangeLastAssoc	getCollectionRangeLastAssoc() Returns CollectionRangeLastAssoc association proxy object.
ConditionIfExpAssoc	getConditionIfExpAssoc() Returns ConditionIfExpAssoc association proxy object.
ElseExpressionIfExpAssoc	getElseExpressionIfExpAssoc() Returns ElseExpressionIfExpAssoc association proxy object.
EnumLiteralExpClass	getEnumLiteralExp() Returns EnumLiteralExp class proxy object.
EnumLiteralExpReferredEnumerationAssoc	getEnumLiteralExpReferredEnumerationAssoc() Returns EnumLiteralExpReferredEnumerationAssoc association proxy object.
FirstCollectionRangeAssoc	getFirstCollectionRangeAssoc()

	Returns FirstCollectionRangeAssoc association proxy object.
IfExpClass	getIfExp() Returns IfExp class proxy object.
InitializedVariableInitExpressionAssoc	getInitializedVariableInitExpressionAssoc() Returns InitializedVariableInitExpressionAssoc association proxy object.
InLetExpAssoc	getInLetExpAssoc() Returns InLetExpAssoc association proxy object.
IntegerLiteralExpClass	getIntegerLiteralExp() Returns IntegerLiteralExp class proxy object.
IsMarkedPreClass	getIsMarkedPre() Returns IsMarkedPre class proxy object.
ItemCollectionItemAssoc	getItemCollectionItemAssoc() Returns ItemCollectionItemAssoc association proxy object.
IterateExpClass	getIterateExp() Returns IterateExp class proxy object.
IteratorExpClass	getIteratorExp() Returns IteratorExp class proxy object.
LetExpClass	getLetExp() Returns LetExp class proxy object.
LetExpVariablesAssoc	getLetExpVariablesAssoc() Returns LetExpVariablesAssoc association proxy object.
LiteralExpClass	getLiteralExp() Returns LiteralExp class proxy object.
LoopExpClass	getLoopExp() Returns LoopExp class proxy object.
LoopExpBodyAssoc	getLoopExpBodyAssoc() Returns LoopExpBodyAssoc association proxy object.
LoopExprIteratorsAssoc	getLoopExprIteratorsAssoc() Returns LoopExprIteratorsAssoc association proxy object.
ModelPropertyCallExpClass	getModelPropertyCallExp() Returns ModelPropertyCallExp class proxy object.
NumericalLiteralExpClass	getNumericalLiteralExp() Returns NumericalLiteralExp class proxy object.
OclExpessionTypeAssoc	getOclExpessionTypeAssoc() Returns OclExpessionTypeAssoc association proxy object.
OclExpressionClass	getOclExpression() Returns OclExpression class proxy object.
OperationCallExpClass	getOperationCallExp() Returns OperationCallExp class proxy object.
OperationCallExpReferredOperationAssoc	getOperationCallExpReferredOperationAssoc() Returns OperationCallExpReferredOperationAssoc association proxy object.

ParentOperationArguments	getParentOperationArguments() Returns ParentOperationArguments association proxy object.
PrimitiveLiteralExpClass	getPrimitiveLiteralExp() Returns PrimitiveLiteralExp class proxy object.
PropertyCallExpClass	getPropertyCallExp() Returns PropertyCallExp class proxy object.
QualifiersAssociationEndCallExpAssoc	getQualifiersAssociationEndCallExpAssoc() Returns QualifiersAssociationEndCallExpAssoc association proxy object.
RealLiteralExpClass	getRealLiteralExp() Returns RealLiteralExp class proxy object.
StringLiteralExpClass	getStringLiteralExp() Returns StringLiteralExp class proxy object.
ThenExpressionIfExpAssoc	getThenExpressionIfExpAssoc() Returns ThenExpressionIfExpAssoc association proxy object.
TupleLiteralExpClass	getTupleLiteralExp() Returns TupleLiteralExp class proxy object.
TupleLiteralExpTuplePartAssoc	getTupleLiteralExpTuplePartAssoc() Returns TupleLiteralExpTuplePartAssoc association proxy object.
VariableDeclarationClass	getVariableDeclaration() Returns VariableDeclaration class proxy object.
VariableDeclarationTypeAssoc	getVariableDeclarationTypeAssoc() Returns VariableDeclarationTypeAssoc association proxy object.
VariableExpClass	getVariableExp() Returns VariableExp class proxy object.
VariableExpReferredVariableAssoc	getVariableExpReferredVariableAssoc() Returns VariableExpReferredVariableAssoc association proxy object.

org.dbe.kb.metamodel.qml.core.types

Interface TypesPackage

Public interface **TypesPackage**

Method Summary	
BagTypeClass	getBagType() Returns BagType class proxy object.
CollectionTypeClass	getCollectionType() Returns CollectionType class proxy object.
CollectionTypesElementTypeAssoc	getCollectionTypesElementTypeAssoc() Returns CollectionTypesElementTypeAssoc association proxy object.

OclModelElementTypeClass	getOclModelElementType() Returns OclModelElementType class proxy object.
OrderedSetTypeClass	getOrderedSetType() Returns OrderedSetType class proxy object.
SequenceTypeClass	getSequenceType() Returns SequenceType class proxy object.
SetTypeClass	getSetType() Returns SetType class proxy object.
TubleTypeVariablesAssoc	getTubleTypeVariablesAssoc() Returns TubleTypeVariablesAssoc association proxy object.
TupleTypeClass	getTupleType() Returns TupleType class proxy object.
VoidTypeClass	getVoidType() Returns VoidType class proxy object.