An Object Query Language for Multimedia Federations

Damir Bećarević

Bachelor of Science in Electrical Engineering

A dissertation submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy

to the

DCU

Dublin City University

School of Computing

Supervisor: Dr. Mark Roantree

June, 2004

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed	[illegible]
Student ID	99141655
Date	June, 2004

Acknowledgments

I would like to thank all those people who made this thesis possible and enjoyable experience for me. First of all, I wish to express my sincere gratitude to my supervisor Dr. Mark Roantree for his patient guidance, encouragement and excellent advices thought this research. Without his guidance and availability this thesis would not have been possible.

A special note of thanks goes to Dr. Patricia Allen for her guidance and support on my initial entry into this research.

Thanks should also go to the Strategic Research Grant Scheme at Enterprise Ireland which supplied fundings towards this research. I am also thankful to School of Computing at Dublin City University and all its employees for their continues support.

I thank my colleagues from the Interoperable Systems Group for sharing experiences and knowledge during the time of study. Especially to Martin, Noel and Seamus for their help in editing this thesis and Dalen for his support in LaTeX/Lyx issues.

Special thanks go to my beloved parents Emina and Muhamed for their constant support, understanding and love.

Finally, I would like to express my deepest gratitude to my wife Sabina for all her love and encouragement during my work on this thesis.

This thesis is dedicated to my newborn son, my wife and my parents.

Abstract

The *Físchlár* system provides a large centralised repository of multimedia files. As expansion is difficult in centralised systems and as different user groups have a requirement to define their own schemas, the EGTV (**E**fficient **G**lobal **T**ransactions for **V**ideo) project was established to examine how the distribution of this database could be managed. The federated database approach is advocated where global schema is designed in a top-down approach, while all multimedia and textual data is stored in object-oriented (O-O) and object-relational (O-R) compliant databases.

This thesis investigates queries and updates on large multimedia collections organised in the database federation. The goal of this research is to provide a generic query language capable of interrogating global and local multimedia database schemas. Therefore, a new query language EQL is defined to facilitate the querying of object-oriented and object-relational database schemas in a database and platform independent manner, and acts as a canonical language for database federations. A new canonical language was required as the existing query language standards (SQL:1999 and OQL) are generally incompatible and translation between them is not trivial. EQL is supported with a formally defined object algebra and specified semantics for query evaluation.

The ability to capture and store metadata of multiple database schemas is essential when constructing and querying a federated schema. Therefore we also present a new platform independent metamodel for specifying multimedia schemas stored in both object-oriented and object-relational databases. This metadata information is later used for the construction of a global schemas, and during the evaluation of local and global queries.

Another important feature of any federated system is the ability to unambiguously define database schemas. The schema definition language for an EGTV database federation must be capable of specifying both object-oriented and object-relational schemas in the database independent format. As XML represents a standard for encoding and distributing data across various platforms, a language based upon XML has been developed as a part of our research. The ODL$_x$ (Object Definition Language XML) language specifies a set of XML-based structures for defining complex database schemas capable of representing different multimedia types. The language is fully integrated with the EGTV metamodel through which ODL$_x$ schemas can be mapped to O-O and O-R databases.

Contents

Declaration i

Acknowledgments ii

Abstract iii

Contents iv

List of Figures ix

List of Tables xi

Preface xii

1 Introduction 1

1.1 The Físchlár Digital Video Recording and Browsing System 2

1.2 Distributed Database Systems 3

1.2.1 Federated Database Systems 4

1.3 The Canonical Data Model 7

1.3.1 Object-Oriented Database Systems (OODBs) 8

1.3.2 Object-Relational Database Systems (ORDBs) 9

1.4 Multimedia Database Systems 11

1.5 Research Objectives 12

1.5.1 Motivating Area 13

1.6 Conclusions 14

2 Related Research 15

2.1 Introduction 15

2.1.1 Sample Multimedia Archive Schema 15

2.2 LOQIS 17

2.3 MOOD and MIND projects 20

2.4 The Garlic Project 24

2.5 The News-On-Demand System 27

2.6 IRO-DB 31

2.7 Hera Project 37

2.8 Conclusions 41

3 The EGTV Metamodel 44

3.1 Introduction 45

3.2 The EGTV Metamodel Specification 47

3.2.1 Defining Name Scopes 48

3.2.2 Defining Types 49

3.2.3 Defining Properties 51

3.2.4 Defining Inheritance 51

3.2.5 Defining Operations 52

3.2.6 Defining Schemas 53

3.2.7 Defining Views 53

3.2.8 Eliminated ODMG Metaclasses 54

3.2.9 A Meta-Metadata Level 55

3.3 Metamodel Mappings 56

3.3.1 EGTV To ODMG Mapping 57

3.3.2 EGTV To Object-Relational Mapping 63

3.4 Conclusions 68

4 Schema Definition 70

4.1 Introduction 70

4.2 The Structure of ODL$_x$ 72

4.2.1 Class Definition 73
4.2.2 Virtual Class Definition 75
4.2.3 Import Class Definition 75
4.3 The ODL$_x$ to Metamodel Mappings 76
4.3.1 Base Schema Mapping 76
4.3.2 Virtual Class Mapping 78
4.3.3 Import Class Mapping 79
4.4 Conclusions 79

5 The EQL Query Language **81**
5.1 Query Language Standards 82
5.2 The EGTV Data Model 83
5.2.1 Type System 84
5.3 Language Fundamentals 85
5.3.1 Query Input and Output 86
5.3.2 Query Structure 87
5.3.3 Deep and Shallow Extent 88
5.3.4 Operation Invocation 89
5.3.5 Aliases 90
5.3.6 Undefined Values 91
5.4 Query Language Operators 92
5.4.1 Type Operators 92
5.4.2 Language Operators 94
5.5 EQL Algebra 99
5.5.1 Algebraic Operators 101
5.5.2 General Operators 102
5.5.3 Set Operators 106
5.5.4 Mapping Operators 109
5.6 Conclusions 110

6 Implementation **112**

6.1 Deployment Architecture . . . 113

6.2 Schema Definition . . . 115

6.3 Query Processing . . . 118

6.3.1 Local Query Processing . . . 118

6.3.2 Global Query Processing . . . 120

6.4 EGTV Transaction Processing . . . 122

6.4.1 Transaction Model . . . 123

6.4.2 The EGTV Object Pool . . . 125

6.4.3 Requirements . . . 125

6.4.4 Lock Types . . . 126

6.4.5 Transaction Types . . . 126

6.4.6 Local Transactions . . . 127

6.4.7 Global Transactions . . . 131

6.5 Experiments . . . 134

6.5.1 Schema Definition Tests . . . 134

6.5.2 Query Processing Tests . . . 135

6.6 Conclusions . . . 138

7 Conclusions **139**

7.1 Thesis Summary . . . 139

7.2 Areas for Further Research . . . 142

7.3 Final Thoughts . . . 146

Bibliography **147**

A EGTV Metamodel UML Diagram **155**

B An XML Schema Definition of the ODL$_x$ Language **157**

C The ODL$_x$ Language Specification and Metamodel Mappings **166**

C.1 Schema Element 166

C.2 Class Element 167

C.3 Attribute Element 168

C.4 Inheritance Element 168

C.5 Relationship Element 169

C.6 Method Element 170

C.7 Operator Element 171

C.8 Return Value Element 171

C.9 Parameter Element 172

C.10 Type Element 173

C.11 Virtual Class Element 173

C.12 Extent Element 175

C.13 Import Class Element 175

D EQL Operator Classification **177**

E Sample Multimedia Database Schemas **179**

F The ODL$_x$ Definition of a Test Schema **183**

G EQL Grammar **192**

H Query Experiments **197**

H.1 Basic Local Queries 197

H.2 Join Queries 197

H.3 Navigational Queries 198

H.4 Exported Queries 198

H.5 Update Queries 198

List of Figures

1.1 The Five Level Architecture of a Federated Database System. 5

2.1 Multimedia Archive Schema. 16

2.2 LOQIS Metamodel Instance: A `Character` class example. 19

2.3 News-on-Demand schema generation. 28

3.1 Components of the EGTV project. 44

3.2 Metadata Definition of Name Scopes. 48

3.3 Metadata Definition of Data Types. 49

3.4 Metadata Definition of Properties. 51

3.5 Metadata Definition of Inheritance. 51

3.6 Metadata Definition of Operations. 52

3.7 Metadata Definition of Schema. 53

3.8 Meta-Metamodel Specification. 56

4.1 Film Archive Schema. 72

4.2 Metamodel Representation of Film Archive Schema. 77

4.3 Metamodel Representation of `RecentFilms` Virtual Class. 78

5.1 The EGTV Model. 83

5.2 Database Schema for Multimedia Recording System. 87

6.1 EGTV Deployment Architecture. 113

6.2 Schema Generation Process. 116

6.3 Syntactic tree example. 119

6.4 Execution tree example. 120

6.5 Global query processing. 124

6.6 EGTV Transaction Architecture. 124

E.1 Multimedia Recording System schema. 180

E.2 Multimedia Archive System schema. 181

E.3 Multimedia Editing System schema. 182

List of Tables

3.2 The EGTV to ODMG Mapping. 58
3.4 EGTV Type Mappings. 62
3.6 EGTV to Object-Relational Mapping. 63
3.8 EGTV to Extended Object-Relational Mapping. 68

6.1 Concurrent Transaction Schedule. 130
6.2 Schema Definition Performance 135
6.3 Query Processing Performance. 136

D.2 Comparison Operators 177
D.4 Arithmetic Operators 178
D.6 Logic Operators 178
D.8 Aggregate Operators. 178
D.10 Set Operators. 178

Preface

This thesis proposes that an object-oriented query language can be defined to facilitate efficient queries and updates on distributed multimedia objects. It also provides extensions to the ODMG metamodel and schema definition language to facilitate the integration of multiple multimedia repositories, and implements services to query and update federated schemas.

In chapter one, an introduction to the area of multimedia repositories and federated database systems is provided for the reader. An architecture which has been adopted by many researchers is described; the requirements for federated database models are listed; standard technologies required by this research are introduced; and a hypothesis for this research is offered.

In chapter two, related research into query languages, metamodels, federated databases and database representation for multimedia is covered to outline the scope of this thesis. The early form of this appraisal was first discussed in [Bec02b]. An examination and comparison of some of existing global query languages is used to determine how query definitions and their metadata representations were specified in each project, and to assess their execution capabilities. The output of this analysis provides requirements for the design of a query language for this research.

A metamodel is an important prerequisite for the definition of the query language as it models metadata required for construction of federated schemas and for generic querying. Chapter three presents a new object-oriented metamodel developed to capture and represent metadata for multiple database schemas. It is based upon the ODMG metamodel, but improved with a more simplified design, the ability to represent multimedia data types and with an extended support for object views. The later is crucial when defining federated schemas. The meta-metamodel is also specified to represent different metamodel versions. The specification of the metamodel design was first presented in [RB02]. Rules are specified to facilitate mapping of our metamodel to object-oriented and object-relational schema repositories. This work was published in [BR04a].

Chapter four presents a schema definition language for our multimedia database federation. This language is designed to facilitate an object schema definition in an implementation independent format. Thus, it is based upon the XML as it represents a standard for encoding and distributing data across various platforms and the Internet. The language is also capable of defining simple object views that are used to restructure data and to construct global schemas.

In chapter five, a new query language is presented to facilitate querying of object-oriented and object-relational schemas in a database and a platform independent manner. This language (presented in [KBR03]) provides orthogonal query input and output; defines a clear semantics for the updatability of query results; facilitates primitives for object creation, update and deletion; includes operation invocation support; and provides ability of querying multimedia data types. Furthermore, operators of the query language are not hard-wired to the language itself, but defined in data types. Thus, the language is fully flexible and can be easily adapted to support different application domains, including multimedia. The query language is also supported with a formal algebraic representation, presented in [BR04b].

Chapter six discusses the implementation of the query system. Firstly, the deployment architecture for multimedia federations is presented, and then followed by a description of the prototype system. This includes the canonical schema definition and local and global query processing algorithms. A transaction control system developed to support global updates is also discussed in this chapter. An early version of this system was presented in [BR01]. This chapter concludes with a discussion of experiments performed on a small test system. This system was constructed using multimedia data extracted from the Físchlár video repository. Three separate multimedia schemas are defined and queried to test the performance of different query types.

Finally in chapter seven, a summary of the work carried out for this thesis is presented. Suggestions as to possible improvements are given, together with a discussion as to how this work might continue.

Chapter 1

Introduction

The need to efficiently query all available information has driven the development of database systems. Therefore, the amount of data electronically represented and stored in different kinds of queryable repositories is constantly increasing. Traditionally, the majority of this data was character based, thus consuming relatively small amounts of storage space per information unit. However, the demand for electronically represented multimedia contents has increased significantly in the last decade. This is mainly due to the proliferation of web-based technologies and increased processing and storage capabilities of mainstream personal computers. Despite the fact that most of the available multimedia related information is stored outside of, what we would consider conventional database systems (for instance, micro-filmed material, the Web etc.), there is growing need to store this material in a format that can be easily queried. As others pointed out, without this the information becomes useless data as *digging* for it is as simple as *finding a needle in a haystack*.

This leads to the problem of storing and querying large quantities of multimedia data. Contrary to traditional character data, multimedia consumes much more storage space per information unit and requires more complex searching and manipulation algorithms. The initial solution to this problem was to store multimedia files in file-systems, but it soon became apparent that this approach has several flaws. Firstly, standard file systems are not optimised for fast data retrieval. Secondly, their querying and searching capabilities are very limited, and thirdly only sequential read and write access is possible. For these reasons, multimedia repositories are stored in specialised data stores. These include specialised file servers optimised for fast data streaming (i.e. Oracle Video Server, Real Server) or standard databases which provide capabilities such as querying, indexing, updating and transaction control.

Since multimedia data is constantly increasing in both volume and size, centralised storage systems can not suffice as their processing power and storage capacity are limited. One solution is to distribute multimedia storage and manipulation resources to a set of inexpensive data stores. This however raises other issues such as data partitioning, communication protocols and platform interoperability. In the ideal scenario, distributed systems would always be built from scratch, avoiding the problems of data and platform incompatibilities. These systems would be modelled using a single data model and thus, be fully compatible. However, the vast majority of existing multimedia repositories are mutually incompatible and cannot be easily migrated to new platforms. Therefore, a different approach that builds distributed systems by integrating existing repositories must be taken.

1.1 The Físchlár Digital Video Recording and Browsing System

Físchlár [LSO+00, OMM+01, RS02] is a multimedia system developed at Dublin City University to facilitate digital video recording, browsing, indexing and playback. It enables clients to digitally record television programmes and watch previously recorded videos through a web interface. Selected TV programs are first recorded in the MPEG-1 format and stored in a regular file system. Recorded videos are then processed by video indexing software which performs shot and scene boundary detection and places analysed programs into a video library. The client interface is dynamically generated as a set of web pages with thumbnail images representing the beginning of each scene within the analysed video. Clients can select any of these images to begin video playback starting from that particular scene. Real-time video streams are then broadcast over the TCP/IP network from the video-streaming server to clients. A streaming server (Oracle Video Server) stores MPEG-1 video files in a specialised file system optimised for fast retrieval, while scene and shot indexing information are fully contained in the web interface as MPEG-7 content metadata.

The video indexing capability represents the most important feature of the Físchlár system. Automated shot and scene detection processes provide fast indexing and enable easy browsing of large video files. However, the main problem with this type of system is the lack of an advanced query interface for dynamic interrogation and updating of recorded video data. The existing system can only store MPEG videos in a proprietary video repository that is optimised for video streaming, but does not provide any database capabilities. Furthermore, video indexing information is stored in a separate system, which can be difficult to correlate with the main video repository. Existing behaviour for video indexing cannot be integrated with the repository as its interface is limited only to file read and write operations. Therefore, providing a database storage for both videos and indexing information would benefit in queryability and updatability of recorded multimedia. For example, a typical query in this system would retrieve all video clips made between years 2002 and 2003 where a white horse is featured (pattern recognition) and camera movement is detected. Results of this query can be then aggregated into one video clip which is saved to a database, thus facilitating updatability.

A further problem is the lack of support for unlimited expansion since the present storage system can only store video files in a single centralised repository. Since MPEG-1 files consume large amounts of space, problems like storage capacity, fast retrieval and real-time streaming can be expected with the growth of the system. Distribution of video data over multiple repositories provides one viable solution for this problem. However, the existing video repository cannot be distributed, nor it can be integrated into an existing distributed system, as its design is proprietary. One viable approach is to use multiple inexpensive databases to provide both data distribution and query support.

In this research we will explore the possibilities of providing data distribution and an advanced query interface for the Físchlár system based on standard database technologies. The strategy proposed here is to provide a distribution plan to cater for unlimited expansion, while also providing a federated-style [SL90] interface so that multiple autonomous video stores can be subsequently rejoined if required by certain user groups. The remainder of this chapter is organised as follows: an overview of distributed and federated databases is provided, followed by a discussion on common data models for autonomous data sources, and then an overview of multimedia databases. The chapter will conclude with a precise motivation for the research undertaken.

1.2 Distributed Database Systems

Database systems vary: starting from the platform (hardware and operating system) to the model being used to structure and store the information in the database. The information, independent of this, can be structured differently even if the same database is used, as different designers perceive structuring differently [CBS96]. This brings us to the need for sharing the information between different information sources, for which different models have been introduced.

A distributed database system is any collection of interconnected databases that facilitate information sharing and exchange. Usually they employ a global interface as a single access point from which data stored in any database in the system can be reached. Distributed systems can be classified into different categories based on the method of integration and the level of global control over the local data [BHP92]. The term multidatabase was employed by researchers [OV91] to the complete family of distributed database systems, while a general classification is given in [BHP92]. Distributed database systems are characterised by tight integration between individual databases in the system which commonly employ the same data model. Thus, these systems are homogeneous in terms of data model, and tightly coupled. A global schema is created in a top-down approach [OV99] and the global layer fully controls all databases in the system and data they locally store. As a result, distributed database systems have good global performance, but at the cost of significant loss of autonomy of its participating databases. Global queries are processed by the global layer which fully controls data in the local databases affected by the query.

However, this approach cannot be applied when participating databases are heterogeneous, and when local autonomy must be preserved. The only viable solution is to create an integration layer that will overcome platform heterogeneities and data incompatibilities. This is commonly achieved by transforming the data representation of each proprietary data repository to a common data model, which can then be globally queried and manipulated. This approach is known as a Federated Database System, and has already been proven in integration of non-multimedia database systems. Thus, the same can be applied to multimedia repositories.

1.2.1 Federated Database Systems

A federated database system is a distributed system that provides a global interface to heterogeneous local DBMSs [HB96]. All databases in this system are autonomous and have full control over the local data they manage. As databases participating in the federation need not be homogeneous, various types of existing database systems can participate. A global schema is created in a bottom-up approach by joining schemas (or some parts of them) from local databases exported to federation level. Global queries are processed at the federated level. Processing includes subdividing a global query into a set of local queries, and passing them to local databases for execution. Result sets retrieved from local databases are combined at the federated (global) level and delivered to the user. Users do not need to be aware of the internal organisation of local databases. Multiple federated database architectures have been proposed [PBE95, BE96], and they all share a common bottom-up design process.

Sheth and Larsen have defined a generic architecture for federated databases called a five-layer schema architecture [SL90]. Each layer is represented with a schema that defines data content and structure. Schemas are constructed by processors, an application-independent software modules of a federated DBMS. The five-layer schema architecture illustrated in *figure 1.1* consists of following layers and processors:

Local Schema. A local schema is the conceptual schema of a participating database which defines all local data. Databases at this layer are autonomous and unaware of existence of federated database system. Federated operations facilitated through the component schema are regarded as standard applications that cannot be differentiated from the any other application local to this database. Because a local schema is database specific, different local schemas may be expressed using different data models.

Component Schema. A component schema represents the local schema translated to the common (canonical) data model chosen for the federation. The main reason for creating a component schema is the ability to describe the divergent local schemas using a single representation. Also, semantics that are missing in a local schema can be added

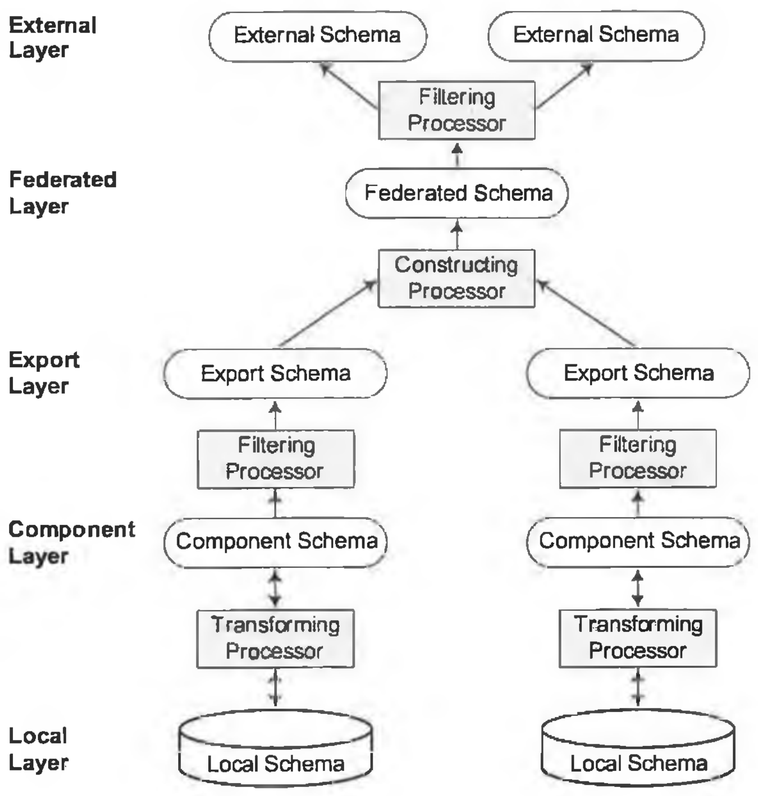


Figure 1.1: The Five Level Architecture of a Federated Database System.

to its component schema. Mapping rules are defined to translate local schema objects to corresponding objects in the component schema. These mappings are also used to transform commands on a component schema into an equivalent commands on the local schema.

Export Schema. An export schema is that portion of the component schema to be shared with other nodes in the federated database. Multiple export schemas can be defined above one component schema. An export schema may also include access control information regarding its use by specific federation users. The purpose of defining export schemas is to facilitate control and management of local autonomy by allowing local administrators to define the pieces of component schema to be shared with all federated users.

Federated Schema. A federated (global) schema is an integration of multiple export schemas and represents a logical global schema of the federated system. Prior to federated schema construction, export schemas can be further restructured to facilitate semantic matching and integration of export schema objects. A federated schema also includes information on data distribution that is generated when integrating export schemas.

External Schema. An external schema defines a subset of a federated schema customised for specific client or a group of clients of a federated system. The data model for an external schema may be different than that of the federated schema. This is because each client group can require a different data model as its interface to the federation (i.e. a client-server or web interface). An external schema can also implement an access control system and additional integrity constraints.

Transforming Processor. A transforming processor constructs an interface between the common form and the local representation of data. This is achieved by translating the commands from the representation of the higher level schema to an equivalent set of commands in the language of the lower level schema. Similarly, the result data is converted in the opposite direction, from the lower level to higher level schema representations. Therefore, this processor provides a data model transparency in which the data structures and commands used by one database schema are hidden from the rest of the system. Data model transparency hides the differences in query languages and data formats. In the Sheth and Larson model, a transforming processor constructs component schemas by fully encapsulating the command and data interface of heterogeneous local schemas.

Filtering Processor. A filtering processor resides between a component schema and its export schema. Its role is to limit data and commands passed between these two layers, thus acting as a security mechanism. Therefore, the filtering processor is used to place a constraint on the size of data to be exported to federated layer. Exported data can be then horizontally partitioned, where only a subset of object attributes are projected into the export schema. The filtering processor can be also used for constructing external schemas, where it restricts access to specific segments of the federated schema for different groups of global users. This is illustrated in *figure 1.1*.

Constructing Processor. This processor resides between the export and federated layer and is responsible for the construction of the federated schema. Its main role is to merge data from multiple export schemas into a single data set which can be then further restructured to form federated schema. The constructing processor also decomposes global queries into multiple sub-queries for component schemas and provides query optimisation. Its final role is global transaction management where its responsibility is to ensure concurrency and atomicity for transactions.

The architecture specified in [SL90] serves only as a framework architecture and does not address implementation. Therefore, many research projects that employ this architecture provide extensions where required. The EGTV multimedia architecture described in chapter six is also based upon the federated database concepts, but it is modified to include global querying, multimedia support and updatability at the global layer.

1.3 The Canonical Data Model

When constructing a federated database system, all local database models must be converted to a common representation (component schema) to facilitate schema integration. Thus, a data model of the component schema must be capable of capturing data structures and semantics of all local schemas in the database federation. This data model is commonly referred to as a canonical data model (CDM) and its proper selection is crucial for the construction of a federated database system. As the relational data model is used in the majority of existing database systems, it is the most common candidate for the canonical data model. When all participating local databases use a relational model, then it can be easily employed as a canonical model. However, it has been shown [SCGS91] that an object-oriented data model is the most suitable as a canonical model for database federations. This is especially so when local databases in the federation are heterogeneous, thus implementing different data models.

In [SCGS91] they measure the suitability of a number of data models including the relational, object-oriented, functional and entity-relationship, as a canonical model for federated database architectures. The principal idea is that the canonical data model must have an expressive power that is greater than or equal to data models of all local databases. Otherwise, the canonical data model would not be able to capture and represent the semantics of local database schemas. Therefore, each model is assessed using characteristics such as *expressiveness* and *semantic relativism*. Expressiveness is described through common properties such as classification, generalisation and specialisation, aggregation and decomposition and extendibility of behaviour. All these are common properties of object-oriented models and are described in software development books such as [Boo94, BRJ99]. The semantic relativism properties advocate a rich integration algebra such as that found in [Mot87], and the power to define views. Two additional recommendations are made for a suitable CDM. The first is a concept of *one basic structure*, which advocates usage of only a single structure for representing the entity to be modelled. This eliminates the possibility of modelling the same entity in different ways and rule out the ER model as a CDM as it has two modelling constructs which are entities and relationships. The second is a concept of *multiple semantics* that means the ability of defining multiple views of the same semantic entity. For example, one user would identify a shoe colour as 'tan' while the other would prefer the colour term 'cream' [SCGS91]. This feature is important when integrating semantically heterogeneous database schemas in the database federations.

When these metrics are applied to data models commonly used in the database world, only the object-oriented model satisfies all categories, whereas the relational model fails to provide generalisation and specialisation, aggregation and decomposition, and extensibility of behaviour. It also fails to provide a rich integration algebra required for semantic relativism. Based on this arguments, there is a clear motivation for using an object-based canonical model. Presently, only two database standards are capable of representing all the features of an object-oriented data model. These are object-oriented and more recently introduced object-relational databases.

1.3.1 Object-Oriented Database Systems (OODBs)

Data in object-oriented database systems are organised into classes [BP97]. Contrary to flat relational tuples, classes can model very complex data structures, thus providing developers with more flexibility in representing real world entities and relationships. Classes in object databases are very similar in structure to classes in object oriented programming languages. The main difference is that objects instantiated from database classes are persistent in the database and object data persists when an application terminates [ER98]. Object databases provide features such as encapsulation, inheritance and polymorphism and this makes them a natural persistent storage for object-oriented applications.

The major standard for object-oriented database systems is maintained by the Object Data Management Group (ODMG) [CB99]. The ODMG standard defines rules for describing and manipulating persistent objects stored in the database and represents the superset of the CORBA OMG [OH98] model. Most of the commercial object databases have some level of compliance with the ODMG standard, but none of them are totally compliant with the standard. The final version of the ODMG standard is 3.0, and it consists of the following parts:

- **Object Model.**
  It determines the meaning of basic concepts of object oriented data structures such as: objects, attributes, relationships, collections, classes, interfaces, operations, inheritance and encapsulation. Metadata is represented in the ODL Schema Repository metamodel. Metamodel specification consists of a set of ODL interface definitions where each interface defines one database construct. The actual implementation of metamodel classes is not provided.

- **Object Definition Language (ODL).**
  Both the object model and ODL are extensions of the corresponding parts of the OMG CORBA standard. ODL is an extension of CORBA IDL [Sie96] and it is used to determine the structure of a database, i.e., a database schema. The schema is necessary to understand what the database contains and how it is organised.

- **Object Interchange Format (OIF).**
  ODMG has defined an Object Interchange Format (OIF) as a specification language used to dump or load objects to or from a file. OIF can also be used for object exchange between different ODMG compliant object databases where it acts as a transport protocol for persistent object encapsulation.

- **Object Query Language (OQL).**
  The language has a SQL-like syntax, but semantically OQL is very different from SQL, mainly because it follows the ODMG object model, which is essentially incompatible with the relational model. OQL is intended to retrieve data from an object base. It does not deal with updating and does not define SQL-like abstractions such as views, constraints and stored procedures.

- **Bindings to programming languages C++, Smalltalk and Java.**
  The bindings determine how to include ODL and OQL statements as constructs of these programming languages. Furthermore, the bindings define many classes (written in the syntax of a particular programming language), allowing access and processing of an object base directly from an application. Behaviour is not stored in the database, but it is linked with the client applications.

The ODMG model is currently the main standard for object oriented databases and it is supported by the majority of object database vendors. It provides a standardised object interface to the database from multiple object-oriented programming languages. The ODMG model is convenient for the definition of a component schema for federated database systems because of its ability to represent complex data structures and easy integration with object-oriented languages. However, it lacks the ability of defining object views required for the construction of an export schema. The OQL language does not provide any updating capabilities, nor it can support server side behaviour. The later is of particular importance when constructing multimedia federations, as complex operations (indexing, editing, pattern recognition) on large multimedia collections can only be supported at the server side.

1.3.2 Object-Relational Database Systems (ORDBs)

Object-relational (O-R) databases are a hybrid between object databases and relational databases. They provide an object-oriented interface built upon the relational database engine. The object-relational data model is standardised in the SQL:1999 specification [GP99, Mel03] where O-R data structures are defined.

- **Data Model.**
  The object-relational model supports features commonly associated with object models such as complex user-defined types, generalisation and association relationships, polymorphism, and encapsulation of behaviour [Sto96]. The main modelling entity is a user-defined object type which corresponds to a class type in the object-oriented databases, while its extent is instantiated as a typed object table. Contrary to the relational tables, object tables can define complex domains and operations [GP99], and each object in the table has unique system assigned object identifier.

A metamodel for the O-R databases is defined in the form of an *Information Schema.* The Information Schema is a special database schema that defines a set of relational views and tables for representing both relational and object-relational metadata. However, this standard has not been widely accepted, and no object-relational database has implemented it yet.

- **Data definition language (DDL).**
  An object-relational DDL is defined in a form of SQL. Its syntax consists of a series of SQL CREATE statements for specifying the structure of user-defined types, object tables, and other O-R model elements.

- **Data manipulation language (DML).**
  The O-R data manipulation language extends the relational SQL language with object retrieval and update features. Thus, contrary to the OQL, objects in the O-R database are directly updatable from the query language. Object views in the O-R model can be defined as named SQL queries. However, these views can only be based upon a single base table. Joins and other advanced schema transformations are not supported in the object-relational view definitions. Object-relational views employ a strict object-preserving semantics [KK95] by deriving their OIDs from the base table objects.

- **Database behaviour.**
  Contrary to ODMG, the object-relational model supports server side behaviour. Behaviour can be defined in the native SQL procedural language or as an external routine specified in a standard programming language such as COBOL, C++, or Java. Native SQL procedural languages (e.g. Oracle's PL/SQL and Sybase's Transact-SQL) are fully integrated with the database schema and share the same address space as the SQL code. However, these languages are platform specific and not portable across different systems. Operations written in SQL procedural languages are also much slower then the equivalent external routines, as the code is interpreted and not compiled. External operations have better performance, but they lack full integration with the SQL and the O-R data model. Moving data between an external routine and SQL code involves the impedance mismatch where object-relational data must be mapped to different types in the external programming languages [Mel03].

The SQL:1999 standard provides a complete definition of the object-relational data model. However, the majority of existing commercial databases does not support object-relational features and only a few incomplete implementations of the O-R standard currently exist. Although the O-R model has limited schema restructuring capabilities, its object views are very simple and insufficient for complex federated schema integration. Server side behaviour supported in the O-R model lacks some important features required for multimedia processing: native SQL procedures are too slow and inefficient; while external routines are not well integrated with the model. Therefore, both ODMG and O-R models are not an ideal choice as canonical data models for multimedia database federations. However, the O-R model has more potential for future improvements as it is standardised in SQL:1999 standard, and is expected to be implemented by all major relational database vendors. Thus, it is a preferred platform to the ODMG which is not widely accepted and does not have a strong a user base.

1.4 Multimedia Database Systems

Although there is no official standard for multimedia databases, all commercial multimedia repositories fall into two general categories. The first category is dominated by specialised file-based repositories optimised for large storage, fast data retrieval and streaming of multimedia data. Commercial products such as *Oracle Video Server*, *QuickTime Streaming Server*, and *RealNetworks Helix Universal Server* belong to this group. Their main role is to provide fast on-demand multimedia streaming to multiple clients. However, all these architectures lack querying and data manipulation capabilities, while data distribution although supported at the file level, benefits only in improved scalability and server-level fault tolerance [Lee98].

The other approach to building multimedia repositories is extending standard database systems with multimedia storage and manipulation capabilities.This is supported in the SQL Multimedia and Application Packages (SQL/MM) standard [Mel03]. The SQL/MM is a multi-part standard that spans several domains including textual, spatial, and image data. Each domain provides a set of SQL and O-R data model extensions for storing and manipulating specific categories of multimedia data (textual, image or spatial). Multimedia data is stored in the object-relational user-defined types (UDTs), for which type specific behaviour is provided. For example a `SI_StillImage` UDT is defined for database storage of still images. `SI_StillImage` stores image data as BLOBs (Binary Large Object), while image parameters (height, width, etc.) are stored as text values. Each `SI_StillImage` object type defines constructors and methods for basic image manipulation. Additional methods for context querying based on average colour, colour histogram, positional colour, and texture are also defined in the standard. Similarly, a `FullText` UDT supports the construction and storage of large textual data blocks. It also contain methods for text retrieval, indexing and linguistic searching. The complete specification of Image, Full-Text and Spatial SQL/MM standards are available in [ISO02c], [ISO02a], and [ISO02b] respectively. Commercial products such as Oracle InterMedia [Ora02a] and IBM DataBlade [IBM01] provide a partial implementation of the SQL/MM standard.

The main advantage of SQL/MM as a query language for multimedia is that it provides multimedia extensions for existing SQL. Since SQL is established as a standard for relational and object-relational database systems, developers do not need to learn a new language for multimedia. The disadvantage of this approach is that SQL/MM is limited only to relational and object-relational databases that support user-defined types. Also, the SQL/MM Image standard supports still images only and cannot represent video contents. Data distribution features and multimedia metaschema are not included in this standard.

1.5 Research Objectives

The *Físchlár* system [LSO+00] provides a large centralised repository of multimedia files. As expansion is very difficult and different user groups often have a requirement to define their own schemas, the EGTV (**E**fficient **G**lobal **T**ransactions for **V**ideo) project [RS02] was established to examine how the distribution of this database could be managed. Currently, multimedia data is mainly stored in proprietary repositories that are vendor specific and mutually incompatible. Any data interchange between these multimedia systems is difficult to implement and provide many challenges for researchers. One solution is to use standard object-oriented and object-relational databases for distributed storage of large multimedia data in the form of objects. The individual databases are independently designed and supplied by different vendors, thus heterogeneous in terms of data model and schema design. This assumes a federated database approach [SL90], although it is unusual in that it takes a top-down strategy for design. The advantages of using a federated architecture are its ability to distribute large amounts of multimedia data across multiple databases and to provide interoperability between proprietary multimedia data stores. Thus, object-oriented and object-relational databases in the EGTV federation physically store multimedia or act as object wrappers for proprietary data stores.

Our research is primarily aimed at providing efficient query and update capabilities for this potentially large distributed repository of multimedia objects. It builds upon services provided by other independent researches within the EGTV project, such as federated data model, server side behaviour and remote object access. The hypothesis put forward in this research is that an OQL-like object-oriented query language can be dynamically extended to efficiently facilitate multimedia queries and updates in a distributed environment. Furthermore, an architecture and metadata services should be specified and implemented to support global queries in this federated architecture. Therefore, issues that need to be addressed in building a global query system for the EGTV multimedia federation can be classified as follows:

Identification of requirements for global query interface. This task should identify and evaluate the existing research in the area of federated and multimedia systems. A special emphasis should be placed on projects that use standard technologies and common databases to create a global query interface.

Common metadata interface to O-O and O-R multimedia databases. Metadata information is crucial for generic querying and for construction of global schemas in the federated architecture. Therefore, a common metadata representation is required when capturing schema definitions from multiple heterogeneous databases at the local layer. As all databases participating in the federation are capable of storing objects, a common metamodel for the EGTV federation must be capable of supporting the full set of object-oriented modelling paradigms. The metamodel must also facilitate the definition of federated schemas and multimedia types. Furthermore, metadata mappings must be specified to translate between the canonical EGTV representation of metadata and proprietary metamodels of O-O and O-R local databases.

Federated (global) schema definition. Local database schemas must be integrated to enable global querying. Therefore, a schema definition language capable of representing multimedia schemas, both at local and global layers should be investigated. This language must be able to fully capture both O-O and O-R multimedia schemas and map them to the EGTV metamodel representation. Furthermore, the schema definition language should facilitate the construction of federated schemas by providing a means for integrating and restructuring multiple local database schemas.

Global query language. A query language must be developed to facilitate queries at both local and global layers of the EGTV federation. This language must be specifically optimised for multimedia manipulation and updatability at the global layer. Furthermore, a transaction control interface should be implemented at the federated layer to allow for updatable global queries.

Prototype. The prototype should demonstrate the workability of research by implementing an appropriate number of services, by constructing a metadata repository for a global schema and analysing the results of the queries specified at both the local and global layers of the architecture. Furthermore, performance characteristics of the prototype must be measured and analysed to evaluate implementability of the full scale system.

1.5.1 Motivating Area

One application of querying a federated multimedia system can be illustrated by the following example. Consider the scenario where a previously recorded video is edited in the video studio. The video recording is stored in the specialised video store in the recording studio. The director of the video wants to add some special effects to the recorded video. Special effect clips are stored and managed by different database systems within the recording studio that may be incompatible with video recording system. The director needs the ability to access the special effects data store, browse available clips and retrieve the selected ones. The problem becomes even more complicated if data from the films archive is required for further video editing. The film archive is located outside the recording studio and provides a browsing and retrieval interface to its clients. The film archive is a completely autonomous data source with a data model and query interface that can not be changed. Autonomous data sources from this example can be incorporated in the federated database system that will be specifically constructed to support operations on multimedia data. This requires a solution whereby generic querying and updating will be facilitated across a system of federated multimedia data stores.

1.6 Conclusions

In this chapter, a general introduction to federated database systems was provided, together with the motivation and objectives of this research. It has also been argued that an object-oriented model is the best suited as the canonical data model for federated database systems. However, two existing standards, ODMG [CB99] and SQL:1999 [GP99] lack some features required for efficient multimedia manipulation and global schema construction. Existing proprietary multimedia repositories do not employ standard data models, thus their distribution is difficult. Furthermore, they provide insufficient query features and do not facilitate generic schema interrogation. Therefore a new federated architecture requires additional levels of functionality, a canonical data model which provides a common representation for each local schema joined to the federation, and the services to support each layer of functionality.

The underlining hypothesis of this thesis is that standard object-oriented and object-relational databases can be used for distributing large repositories of multimedia data. Although existing standards should be used where possible, extensions are necessary to provide an efficient global query interface and a federated data and metadata repository. In chapter two, related work in the area of federated databases, metamodels, query and schema definition languages is discussed. In chapter three, a novel metamodel for database federations is presented, and in chapter four an XML-based object definition language is discussed. A new query language and its algebraic definitions are discussed in chapter five, while the deployment architecture and prototype details are presented in chapter six. In chapter seven we conclude our work and offer suggestions for future research topics.

Chapter 2

Related Research

In this chapter several research projects covering object query languages, metamodels, federated database architectures and multimedia databases are described. Existing research into global query languages was studied in order to determine how query definitions and their metadata representations were specified in each project, and to assess their execution capabilities. We also examine mechanisms used for global schema construction and transaction control. There is also an interest in any multimedia database system that could be applied in a federated architecture rather than simply operating as a single database. The chapter is structured as follows: in §2.1 a brief introduction is provided, followed by a discussion on a number of different research projects presented in §2.2 to §2.7; the output of this discussion is a set of necessary and useful properties for our own query language and metamodel; and in §2.8 some conclusions are presented.

2.1 Introduction

Since the work presented in this thesis is focused on building a query language for a multimedia federation, it was necessary to investigate the range of research projects that cover areas of federated systems and multimedia databases. When examining these projects, the emphasis was on their data distribution and querying facilities. Metadata representation was also studied, as an efficient metamodel is an important perquisite for building any dynamic query service.

Sample queries and schema definitions provided for some projects, are based on an object-oriented multimedia schema defined in the EGTV project. In each case, an example is presented in a query or schema definition language of the discussed project.

2.1.1 Sample Multimedia Archive Schema

To illustrate differences between research projects covered in this chapter, we use a simple multimedia schema illustrated in *figure 2.1.* This object-oriented schema represents an

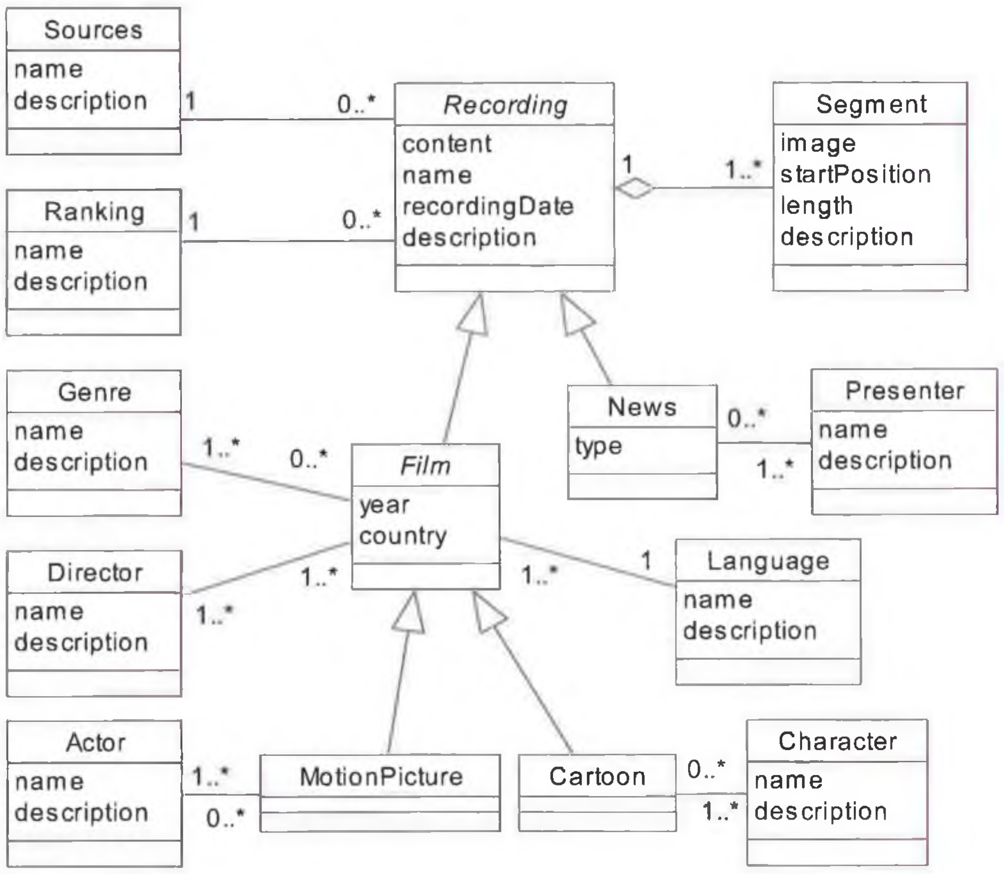


Figure 2.1: Multimedia Archive Schema.

EGTV multimedia archive database for storing video clips extracted from the Físchlár system. Database storage for recorded videos is required to facilitate dynamic querying and integration with other data sources in multimedia database federation. All stored multimedia is represented using subclasses of the `Recording` abstract class. This class contains the video `content`, its `name`, `recordingDate` and textual `description`. The source of the multimedia material is determined from relationship to `Source` class, where relationship to `Ranking` determines its rating. Each recording is *segmented* into multiple shots for navigational purposes. Each of the segments is an object of the `Segment` class, it is associated with the `Recording`, and described in terms of `startPosition`, `length`, `image`, and textual `description`. The `Recording` class is specialised in the abstract class `Film`, and in the class `News`. The class `Film` contains `year` and `country` attributes and it has associations to `Genre`, `Director` and `Language` classes. All of these are in common for further specialised `MotionPicture` and `Cartoon` classes. Class `MotionPicture` has an association with `Actor` class, where the class `Cartoon` has an association with `Character` class. The class `News` has an association with class `Presenter`. The schema was modelled using UML [BRJ99] and the Rational Rose case tool [EP97].

2.2 LOQIS

In the LOQIS database system [SBMS94], a database is modelled as a set of triplets $< i, n, \nu >$, whereas i is an unique *object reference*, n is the *external name* of the object and finally, ν is the *contents* of the object, which can be either an object-reference, a value, or a set of object-references. External names are used for fetching objects, while object references are considered internal to the data model. This is illustrated in *Example 2.1* where an object of a class `Cartoon` is defined in a LOQIS representation. This object contains attribute sub-objects `name` and `description`, and a relationship `appears_-in`. The relationship takes a reference to an object i_6 as its value. Thus, unique identifiers are given to both objects and their properties. An object of a subclass is created with all the properties of its superclasses, i.e. no special notation is introduced. LOQIS does not differentiate persistent and transient objects, thus both are treated equally within the queries and behaviour.

```
<i5,CHARACTER,
     { <i51, NAME, Goofy>,
       <i52, DESCRIPTION, Cartoon character>,
       <i53, APPEARS_IN, i6>}>
```

Example 2.1: LOQIS object representation.

Query Language. A single query, programming and view definition language named the *Stack-Based Query Language* (SBQL) is used to manipulate database objects. Each expression is considered a query and result of any query is an object reference (either to a set of objects, or a to single object) which enables orthogonal subquerying. A simple query in the SBQL language is illustrated in *Example 2.2*. This query first selects an object of a `Character` class using the `where` condition, and then follows the `appears_in` relationship to retrieve all related objects of a `Cartoon` class. Finally, a `name` sub-object of each `Cartoon` object is retrieved as the end result. Thus, the result of a query is always an object reference, as all object properties are objects themselves. The semantics of the query execution is defined in terms of operations on two stacks. The *environment stack* (ES) deals with the scope control and binding names, while the *result stack* (QRES) stores intermediate and final query results. The environment stack is similar to procedure call stacks in the programming languages, and contains references to objects used during the evaluation of a query. The query result is always generated as a top element of a result stack [SKL95]. Persistent objects can be inserted to the data store by invoking the `create persistent` command of the SBQL language, while `create local` command is used to define transient objects. The result of both of these create commands is always a set of references to created objects. Any persistent object can be removed by an explicit delete command, while transient object are automatically deallocated upon the termination of a query or procedure call in which they are defined.

```
(CHARACTER where NAME = "Goofy").APPEARS_IN.CARTOON.NAME
```

Example 2.2: SBQL query example.

Behaviour is implemented as a set of procedures [SKL95], where a procedure is a named sequence of queries and basic control statements such as `for` loops and `if` clauses, returning an expression. The main objective of this research is to provide orthogonality between query and procedural languages. The authors argue that one language can be used both for queries and procedural code definitions. Therefore, LOQIS procedures are orthogonal to queries as both languages consist of the same expression types and return an object references as a result. Thus, queries can be easily embedded within the procedural code and act as a procedure parameters, while procedures can be invoked from within the query definition. Procedures are permanently stored within the database.

Views are seen as a result of procedure executions and they always evaluate to a single virtual class. The result of view materialisation is a reference to an existing persistent object or a newly generated temporary object, thus all views are directly updatable. The original semantics of LOQIS views is extended in the recent paper [KLS03] where view updates are further discussed. An extended view consists of an object extent and a set of operations that define update semantics. This is illustrated in Example *2.2* where a view is defined to retrieve all Goofy's cartoons. A virtual object clause is an SBQL query that materialises a view extent in a form of LOQIS objects. However, these objects are not directly updatable, as this may introduce unwanted side effects. For example, inserts and deletes into a virtual class created from a join are ambiguous, as it is unclear how base objects are affected by this operation. Therefore, each updating operations on a view is defined as a SBQL procedure `on_update`, `on_delete` and `on_retrieve`. The execution of these procedures is triggered when a corresponding update operation on a view extent is invoked.

```
create view GoofyFilmsDef {
virtual object GoofyFilms
  {return (CHARACTER where NAME =
          "Goofy").APPEARS_IN.CARTOON.NAME as c}
on_retrieve do {...}
on_update rvalue do {...}
on_delete do {...}
on_insert objecref do {...}
```

Example 2.3: LOQIS view definition.

Metadata. The original LOQIS specification [SBMS94] does not discuss metadata representation for types, views and operations, nor it does specify a metamodel. How-

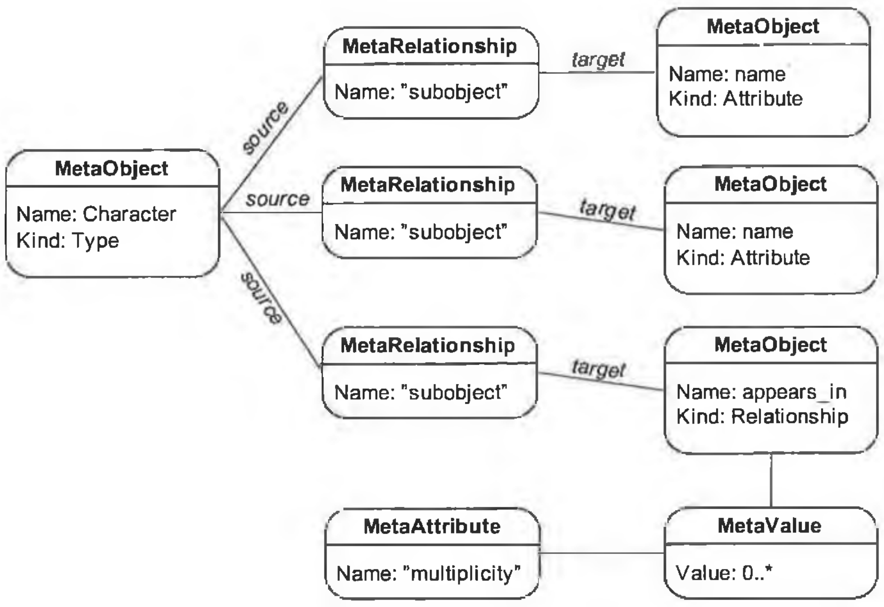


Figure 2.2: LOQIS Metamodel Instance: A `Character` class example.

ever, they introduce special *class*[1] *objects* embedded within the LOQIS data model. Class objects are used for mapping inheritance (by creating a triplet with external name `InheritsFrom` that connects subclass with the superclass) and for referencing procedures, constraints and properties belonging to a class. All objects of a class are considered to be connected to their class object. Details regarding creation, contents and usage of class objects are not published, but the published work clearly denotes that the class objects do not specify property types and names.

In their later work, they introduce a flattened metamodel [HRS02] that can be applied to LOQIS and other object-oriented database systems. This specification advocates a minimalistic metadata repository where the majority of meta-metadata is represented at the metadata level. This reduces the total number of classes in the metamodel, as several modelling entities can be represented using a single metaclass that takes multiple roles. Thus, a proposed metamodel contains only four metaclasses that represent types (`MetaObject`), their attributes (`MetaAttribute`) with values (`MetaValue`), and relationships (`MetaRelationship`) to other meta-objects. *Figure 2.2* illustrates an instance of class `Character` represented in the flattened metamodel representation. Each property of the `Character` class is represented as a separate meta-object connected to the root `Character` meta-object with the *sub-object* relationship. The relationship property `appears_in` defines cardinality meta-attribute with the `0..*` value.

[1]In this research, sometimes referred as to *master* object.

Limitations. The model contains no notion of scopes, and both base and virtual classes are stored together, resulting in naming restrictions (no two classes can have the same name). The SBQL language is not SQL compliant and uses syntax similar to low-level programming languages, therefore it is not intuitive to many database engineers. Furthermore, the authors use the same language for both queries and stored procedures. This can have a negative performance impact, as procedural code must be interpreted each time the procedure is invoked, thus resulting in a longer execution time. Therefore, complex multimedia operations written in this language cannot achieve optimum performance. Published work does not discuss any data distribution features, nor interoperability of LOQIS system in heterogeneous environments. Although the view system for SBQL is defined in the form of stored procedures, neither global views nor schema restructuring capabilities are discussed. The metadata side of the model is neglected: the class objects are too restrictive to be considered metadata as they do not describe attributes the objects have, nor their types, simply the methods. The published work does not discuss how database schemas are specified, nor how a schema definition language is provided. The flattened metamodel specification reduces the modelling complexity by using only a minimal set of metaclasses. This however results in the creation of multiple meta-objects for representing the metadata of a single class. Interrogation of this flattened structure is difficult, as it requires complex queries that span multiple objects in the schema repository. Furthermore, complex algorithms for maintaining the consistency of schema repository must be defined to allow updatability of the metamodel.

Summary. The reference-based model of the LOQIS system represents the central feature of this research. The concept of direct referencing of both object and its properties is reused in the EGTV project where it provides a basis for updatability of queries and views. This way, the query language does not need special syntax or keywords for updating objects. However, the proprietary syntax of SBQL is replaced with the more familiar SQL-like one. The query language must also be extended to address issues of interoperability of heterogeneous data sources and global schema construction. Metadata representation and schema definition language, neglected in LOQIS must be redefined and enhanced with the ability of representing multimedia metadata.

2.3 MOOD and MIND projects

The MOOD database system [DOAO94, DDK+96] is a proprietary object-oriented DBMS that derives its model from the C++ object model. The C++ model was chosen to avoid the impedance mismatch when converting objects between database and programming language representation. Therefore, types used for the definition of persistent classes are identical to the ones specified in the behaviour methods. The MOOD database kernel was built using the Exodus Storage Manager (ESM) to provide database kernel functions such as storage management, concurrency control, and backup and recovery. Persistent classes can be defined using MOOD SQL (an extended form of SQL-92 for object querying) or C++ class definitions.

METU INteroperable DBMS (MIND) [DDK$^+$96] is a multidatabase system that integrates multiple MOOD databases with other commercial databases (Oracle 7 and Sybase) into a multidatabase system. The integration is implemented using database wrappers specified in a form of CORBA objects. All command and data interactions between participating databases and global multidatabase services are facilitated through the CORBA architecture [HV99].

Federated Architecture. The MIND system is based upon a four layer multidatabase architecture similar to [SL90]. The architecture consists of Local Schema, Export Schema, Federated Schema and External Schema layers.

- **Local Schema.**
  A Local Schema is expressed in the native data model of the MOOD database or any commercial database for which an object wrapper is provided.
- **Export Schema.**
  An Export Schema defines a canonical data model in a form of ODMG [CB99] objects and corresponds to both Canonical and Export Schema layers in [SL90]. Data models of all databases at the local layer must be translated to the canonical ODMG model.
- **Federated Schema.**
  The individual Export Schemas are integrated into a Federated Schema using an extended form of the ODL language: the keyword `interface` is used to define the structure of the exported class (attributes and relationships), while the new keyword `mapping` is introduced to facilitate definition of the mapping rules, i.e. the class extent and how the attributes and relationships map to the source classes at the Export layer. Thus, a simple virtual class can be defined using multiple source classes located at different local database nodes. The virtual class extent is defined as a MOOD SQL query, while ODL extensions specify mapping rules to base classes. Inheritance of virtual classes can also be defined in the `interface` declaration of the class. A definition of a simple virtual class `GoofyFilm` is illustrated in *Example* 2.4, where a class interface is specified first, and then mappings to the source classes `Character` and `Cartoon` in the export schema are defined.
- **External Schema.**
  An External Schema is created to facilitate requirements of a specific application or a group of users. It represents a subset of the Federated schema and can contain additional integrity constraints and schema transformations.

```
interface GoofyFilm {
    extent GoofyFilms;

    attribute string name;
    attribute string description;
}


mapping GoffyFilm {
    origin MMArchive:Character Character,
           MMArchive:Cartoon Cartoon;

    def_extent GoffyFilms as
               select *
               from Cartoon, Character
               where Character.name = 'Goofy'
               and Character.appears_in = Cartoon;

    deff_att   name as Cartoon.name;
    deff_att   description as Cartoon.description;
}
```

Example 2.4: Virtual Class definition in MIND.

Query Language. The MOOD SQL [Alt94] is an "objectised" version of SQL-92 [MS92] that facilitates method invocation within queries and path navigation based on relationships defined between classes. The language is separated into DDL (Data Definition Language) and DML (Data Manipulation Language) segments. The DDL part of the language defines database schema as a collection of type and class definitions. The difference between MOOD types and classes is that types define only structure and no behaviour, while classes can have both structure and behaviour. Furthermore, only classes can be instantiated to objects, and define an object extent. Properties of both classes and types can have complex domains that include collection types, association and generalisation relationships, and other user-defined types. *Example 2.5* illustrates a MOOD DDL definition of a class `Character`. The data manipulation part of the MOOD SQL includes constructs for performing CRUD (Create, Retrieve, Update, Delete) operations on database objects. New objects are created by invoking an object constructor (defined as a C++ method) through the `NEW` command of MOOD SQL. Delete is based on a reference counting, where reference count is defined as the number of other objects in the database that reference the deleted object. Thus, an object can be permanently deleted from the database only when its reference count reaches zero. Otherwise, the reference count is only decreased each time an object is deleted. Behaviour is seen as a collection of persistent class methods defined in C++ [DAO$^+$95] which can be invoked within the MOOD SQL queries. However, virtual classes cannot define new behaviour.

```
CREATE CLASS Character
TUPLE(
       name STRING[12],
       description STRING[40],
       appears_in REF( Cartoon )
)
```

Example 2.5: MOOD definition of a Character class.

The MOOD SQL language is supported with an object algebra [Alt94]. Therefore, all queries can be represented in the algebraic form. The algebra defines three categories of operators: general, collection and conversion. General operators are applied to single objects to facilitate path navigation, identifiability and object dereferencing. Collection operators manipulate object collections that can be of a list, set or extent type. This group of algebraic operators include projection, selection and join operations. Conversion operators are used for conversion between different collection types.

The MIND multidatabase system supports execution of global queries specified against the federated and external schemas [NKOD96] of the architecture. Global queries are first decomposed to a multiple subqueries by the Global Query Manager (GQM) processor, implemented as a CORBA object. This decomposition process is based on the definition of the virtual class which specifies how its structure is derived from multiple base classes. Each subquery is targeted to one participating database node where it is translated to the proprietary query language of local layer database. Subquery results are converted back to the canonical data model representation and then merged at the federated layer by GQM to produce the final result. This merging is achieved through post-processing operations, namely join, the outer-join and union.

Transaction Model. Global transaction control is incorporated into the MIND architecture, where the ticketing method [GRS94] is used to enforce the stabilizability of global transactions in a multidatabase environment. However, this transaction method is implemented in a distributed manner, so that global transactions can be equally submitted to each participating site capable of coordinating their execution. A Global Transaction Manager (GTM) processor at the coordinating site employs an optimistic scheduling algorithm which assigns ticket values in a form of a global timestamp. Thus, global tickets are maintained distributively, and it is not necessary to obtain tickets from a specific central site. Global subtransactions are serialised in the timestamp order at all sites. Local conflicts between subtransactions are resolved by forcing each subtransaction to write and read ticket value in the local database. This guarantees the local serialisation order equivalent to the order of global transactions.

Limitations. MOOD is based on a proprietary object database model which restricts its usability. Since the data model is standard C++, the database is completely bound to this language and mappings to different programming languages are difficult to define. The integration approach chosen in MIND has multiple problems: firstly, it does not facilitate the definition of methods in views; secondly, although it uses ODMG as its canonical data model, virtual class extents are still defined in a proprietary MOOD SQL; and finally, MIND facilitates the definition of inheritance between unrelated virtual classes without offering any discussion on implications for querying and updating.

Details of defining External Schemas are not discussed, and no metamodel for MOOD was ever published, though in the MIND multidatabase system one of component databases is a MOOD database. Thus, it remains unclear how federated metadata is represented and stored. The MOOD SQL language allows invocation of custom methods from within queries, but operators (arithmetic, logical and comparison) are hard-coded in the language itself, and therefore not supported for user-defined types. Global query processing is complex and involves multiple stages of query transformation and rewriting. This requires complex query processors that translate queries from MOOD SQL to the proprietary language of local layer databases. However, specified transaction rules are still not able to support the full semantics of all local query languages used in the system. Updatability of global queries has not been discussed, while the MOOD SQL algebra is not fully orthogonal, as inputs and outputs to algebraic operators can be either, objects, identifiers or class names.

Summary. The most important feature of the MOOD object database system is a query language capable of invoking user-defined methods. Methods are specified in a standard (compiled) programming language, thus having minimal performance overhead. Performance is an important feature when processing large quantities of multimedia data stored in a database. Therefore, we reuse and further enhance this functionality in our query language developed within the EGTV project. The MIND multidatabase system provides a framework for integration of multiple local databases into a federation using CORBA services. CORBA solves platform dependencies by providing a layer of interoperability between different databases, thus simplifying the system. Global transaction management guarantees stabilizability of MIND transactions across multiple databases participating in the federated system, and provides a starting point when designing transaction management for the EGTV prototype.

2.4 The Garlic Project

The goal of the Garlic Project [CHS$^+$95] is to develop an architecture for a distributed database system that can store and manipulate heterogeneous multimedia data. This is achieved by constructing a distributed middleware capable of integrating multiple multimedia and traditional text-oriented data sources into a queryable multidatabase system. The middleware interacts with the heterogeneous databases and video repositories to facilitate global queries while preserving the autonomy of local data.

System Architecture. The architecture of Garlic system consists of three layers: Data Source, Wrapper and Integration Schema layer.

- **Data Source layer.**
  This layer provides storage for multimedia and textual data in the system. Different storage systems are supported ranging from simple file systems to specialised multimedia repositories and standard databases. The only requirement for an integration of a data repository to the Garlic system is that a wrapper has been provided for that specific repository.

- **Wrapper Layer.**
  Data stores are connected to the Garlic system by wrappers [HMN+99] that fully encapsulate their data retrieval interfaces and provide a common query language access to stored data. Wrappers also transform proprietary data models of underlining data stores to a canonical Garlic Data Model (GDM).

- **Integration Layer.**
  A global schema is created at this layer as a union of all wrapper schemas represented in Garlic data model. This schema can be further restructured to combine data from multiple data stores. All queries in the system are executed only against the global schema.

Garlic Data Model. The Garlic Data Model is effectively an ODMG model extended with weak object identity, type conformance and views. Garlic introduces a concept of *weak identity* which provides a unique, but not necessarily immutable identification of database objects. This approach enables an object encapsulation and referencing of multimedia data from repositories which do not support a strong notion of identity, such as relational databases. Therefore, object identifiers in the Garlic canonical schema are derived from the identifiers used by proprietary data stores. However, instantiated objects cannot maintain a permanent reference to the underlining data, and are materialised as snapshots only. Therefore, Garlic objects are not updatable. Another modification of the ODMG data model is a concept of flexible type system. It introduces a notion of interface conformance, where the *conforms* relationship is defined as a weaker form of inheritance in which one interface can be considered as a subtype of another even if the explicit inheritance relationship between them does no exist. This is exploited later when heterogeneous multimedia schemas are integrated in the federation.

Each canonical schema is defined as a set of interfaces specified in a Garlic Definition Language (GDL). The GDL extends the ODMG ODL schema definition language with the ability to rename types and attributes, change types and define relationships even if the underlining data source stores none. While the interface definition is platform independent, its implementation is bound to the repository where the actual data is stored. The interface implementation uses the proprietary interface of data store to retrieve data and wrap it into canonical layer objects. Therefore, Garlic data types are mapped to the native data types of the underlining data source, while methods defined in the interface declaration are just wrappers for the equivalent methods of the data source repository.

The global schema consists of multiple object views and is able to extend, simplify or reshape properties and methods defined in wrapper interfaces. A view is defined as a GDL interface, while its extent is generated as a SQL query. Any new method defined for a view is also implemented as a parametrised query. However, each view can be based upon a single base class. Objects originating from different data stores can be combined using *complex objects*. Complex objects are stored in the special Complex Object Repository and model relationships that exist between multiple multimedia objects. Complex objects are needed to integrate multimedia data with legacy data in situations where the legacy data cannot be changed, and as a place to attach methods to implement new behaviour. The existing behaviour in the source repositories can be invoked through the wrappers, but new methods can only be specified in the form of SQL queries.

Query Language. The query language for Garlic is based on SQL-92 and extended with object-oriented features such as references, collections and operations. Therefore, new operators are introduced to the Garlic SQL, such as `makeset`, `nest` and `unnest` for manipulating object collections and `lift` for generating virtual object OIDs. Virtual identifiers are always based upon the object identifier of underlining base objects and contain an identification of a wrapper and an OID of the object managed by the wrapper. Query language extensions also include predicates and operations for context querying of multimedia objects. A global query is decomposed to a set of subqueries that are executed on the wrapper databases. It is then the wrappers job to translate these subqueries into the repository's native query language (or its native search API, if it has no actual query language). All queries are read-only and data modifications are not possible.

Limitations. Garlic's object-oriented query language has advanced querying capabilities, but no transaction support is provided. Queries are read only and thus, data modification is impossible. Limited view capabilities are the other disadvantage of the architecture. The main issue with Garlic views is their inability to define relationships between classes in the global schema. Relationships are substituted by the concept of complex objects that must be stored in the special repository. A metamodel was not included in any of the literature covered, while multimedia querying requires an extension of an existing query language, thus limiting its generality. Our aim in EGTV is to preserve general SQL-like syntax and semantics of a language, but to provide a full integration of query language with the behaviour that implements multimedia operations. Furthermore, our approach extends ODMG OQL as it is more suited to querying an object-oriented data model than SQL. However, our EGTV data model requires a re-specification of OQL in order to exploit data distribution and updatability of views. Thus, while employing an OQL like syntax, the semantics are different.

Summary. The global schema and object based data model for multimedia are the most important features of the Garlic architecture. Different multimedia data stores can be joined to the Garlic system by using object wrappers that provide translation to the Garlic data model. We adopt the Garlic approach of extending the ODMG model, but our extensions are more comprehensive than those implemented in Garlic as they include a full specification of the metamodel and updatability at the global level. We regard a metamodel specification as a crucial feature in semantic integration.

2.5 The News-On-Demand System

The News-On-Demand System [OSEM+96, WLE+97] is a multimedia project developed at the University of Alberta in Canada. Its main objective is to provide database storage and advanced query interface for a collection of multimedia news articles. Each news document consists of textual and multimedia elements with spatial and temporal relationships between them. The spatial relationships between multimedia elements are represented in SGML, while the HyTime standard was used for temporal relationships. SGML (Standard Generalised Markup Language) [ISO86] is a predecessor of XML, and as such it provides markup representation of textual documents. The HyTime standard [ISO92] adds support for hypermedia and synchronised documents, thus enabling definition of complex documents containing hyperlinks and video.

Architecture. From a database aspect, the system provides an object-oriented representation for multimedia data. Non-continuous media (text and still images) are stored in the ObjectStore object-oriented database, while continuous media (audio and video) are stored in a specialised media file server. A wrapper has been provided to encapsulate this proprietary multimedia server into an ObjectStore database schema. The architecture does not provide heterogeneity, since all objects must be stored in these two data stores. However, data can be stored in a distributed manner, as multiple databases can be joined to the system. A global schema is a simple union of all local ObjectStore schemas, and is fully encapsulated within the client application. Thus, queries can operate on multiple databases, but due to the simplicity of schema integration only a simple global queries are possible.

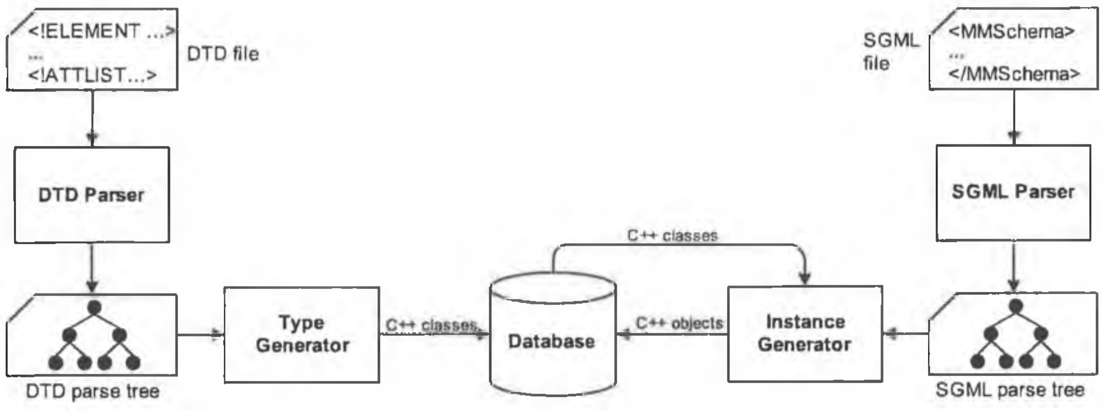


Figure 2.3: News-on-Demand schema generation.

The system uses SGML related technologies for defining database schemas and multimedia documents. Therefore, multimedia schemas are first specified in the form of DTD (Data Type Definition) files. A DTD defines the blueprint for instantiation of SGML documents, and as such represents a metadata specification. Actual database schemas in the News-on-Demand system are generated from these DTD specifications as illustrated in *figure 2.3*. In the first step, a `DTD Parser` processes a DTD file, and creates its in-memory object tree representation where each DTD element is an object in a tree. A `Type Generator` uses this information to generate prototypes for C++ classes that are then loaded to an ObjectStore database, thus generating an object-oriented database schema. One class is created per element in the DTD specification, while element attributes are mapped to class attributes. For example if a subset of a Multimedia Archive schema, defined in the DTD in *Example* 2.6, is parsed, classes representing `Recording`, `Segment`, `Sources`, and `Ranking` elements would be created in the database.

```
<!ELEMENT Recording>
<!ELEMENT Segment>
<!ELEMENT Sources>
<!ELEMENT Ranking>
```

Example 2.6: News-on-Demand schema definition example.

Actual data is specified in a form of SGML/HyTime documents conforming to the previously specified DTD definitions. The process of document insertion to a database is also illustrated in *figure 2.3*. A document is firstly parsed by the `SGML Parser` which validates it and creates a parse tree. The `Instance Generator` processor traverses this parse tree and instantiates appropriate objects in a database corresponding to the elements in the document and previously created database classes. The database is then populated with persistent objects that can be accessed using the query interface.

In addition to user-defined types created from DTD definitions, some common multimedia types are already provided within the system. These types are defined as extensions to an existing ObjectStore type system, and are available to all users. System types are represented in a multimedia class hierarchy. The base multimedia type is an abstract class `Atomic` and all other types in the class hierarchy are descendants of `Atomic`. Its direct ancestors are non-continuous media class (further specialised as `Image` and `Text`) and continuous media class (further specialised as `Audio` and `Video`). Classes that represents single multimedia (like `jpg`, `avi`, `mpeg`) derive from leaf classes in this class hierarchy. Complex multimedia objects consist of interrelated single multimedia objects. Complex objects are represented as SGML and HyTime documents, where SGML describes spatial relationships between component single multimedia objects, while HyTime model temporal relationships.

Query Language. The query language for multimedia (MOQL) [LOSO97] was developed as a part of the News-on-Demand project. The language adds multimedia extensions to an existing ODMG OQL query language. These extensions represent the most important feature of the language as they include constructs for capturing the temporal and spatial relationships between multimedia objects in the database schema. All extensions are separated into two categories: predicate expressions and functional expressions. Predicate expressions extend the syntax of the language by introducing new predicates, while functional expressions are global functions defined specifically for manipulating different kinds of multimedia objects. For example, a `contains` predicate checks if a physical object is contained within some multimedia object (i.e. if a `videoClip` contains any persons). Each predicate or function expression can belong to either a spatial or temporal category.

Spatial expressions include predicates for querying spatial relationships between multimedia objects such as `nearest`, `farthest`, `inside`, `left`, `right`, `above` and `below`. By using these predicates, the position of an object and distance from other objects can be queried. Spatial functions compute attributes of an object or a set of spatial objects. For example, the function `distance` returns distance between two objects, while `length`, `area` and `perimeter` functions calculate the size of an object to which they are applied.

Temporal functions such as `before`, `after`, `overlap`, and `during` can compute temporal dependencies between objects in the MOQL query. Special category of temporal functions are those specifically constructed for querying video contents. These functions can manipulate video clips as a whole, or at the level of individual frames. Therefore, a query can ask for the previous frame to the current frame, or the last frame of a video. Functions belonging to this category include: `prior`, `next`, `firstFrame`, `lastFrame`, `firstClip`, and `lastClip`.

A sample MOQL query is illustrated in *Example 2.7.* This query selects the last frame in all video clips where cartoon character Goofy appears. It uses spatial predicate `contains` in the `where` clause to filter the result set, and the global function `lastFrame` to extract the last frame from all video clips returned as a result of a subquery.

```
select lastFrame(
        select  c.content
        from Cartoon c
        where c.content contains Character("Goofy") )
```

Example 2.7: MOQL query example.

Limitations. The main deficiency of the News-on-Demand system is its inability to provide a true heterogeneous multidatabase environment. The data distribution model is simple and supports only two proprietary data stores: the ObjectStore database and specialised media server. However, integration of the media server with the database schema is not discussed in published research. The global schema is constructed as a union of all local schemas and is fully contained within the client application. Therefore, schema restructuring is not possible as a single consistent schema instance must be maintained across multiple clients. Although, a schema definition process is fully specified, no metamodel is discussed in the published research.

MOQL is a read-only language and does not provide updatability or transaction features. Also, all MOQL multimedia functions are hard-coded to the query language itself. Thus, a language and existing type system cannot be extended with new multimedia operators and functions. Furthermore, each user-defined class must derive from an existing multimedia class hierarchy predefined in the data model. Behaviour methods and operators cannot be specified for these classes.

Summary. The main advantage of this system is its ability to define complex multimedia schemas using the standard markup language SGML. This enables the capturing of a complex spatial and temporal relationships between multimedia objects, while maintaining simple algorithms for parsing and constructing database schema. We use a similar approach in the EGTV project where a XML markup language (a successor to SGML) is used for defining both local and global database schemas.

The MOQL language adds advanced multimedia extensions to an existing ODMG OQL query language. Although these extensions enable querying of spatial and temporal relationships between database objects, their implementation is proprietary. Furthermore, MOQL multimedia functions cannot be extended or modified. Therefore, our query language for the EGTV project takes a different approach by providing flexible operators and methods whose behaviour can be easily modified. Thus, a query language can be customised for any specific domain, including multimedia.

2.6 IRO-DB

The IRO-DB (Interoperable Relational and Object-Oriented DataBases) project [GGF$^+$96] aims for the provision of appropriate integration tools to achieve interoperability between pre-existing object-oriented and relational databases. A global schema is constructed to enable queries evaluated against multiple databases, while the updatability is guaranteed by the global transaction mechanism.

Architecture. The system follows the general concept of federated database systems [SL90], which provide interoperability of autonomous data sources by adding multiple schema transformation layers. However, the IRO-DB architecture differs from the generic [SL90] model as it consists of only three layers: local layer, communication layer and interoperable layer.

- **Local Layer.**
  This layer encapsulates local databases and transform their proprietary data model into the canonical model of database federation. Only a subset of the local schema is represented in the canonical format, thus the canonical schema at this layer is effectively an export schema of the generic [SL90] federated architecture. The canonical model chosen for the IRO-DB project is the ODMG data model [CB99], therefore this layer is able to answer OQL queries and define export schemas in terms of IDL interfaces.

- **Communication Layer.**
  The communication layer implements remote object access services for both clients and servers. Thus, its role is to pass OQL queries and updates from the upper layer to the Local Layer, and return results in a virtual object representation.

- **Interoperable Layer.**
  The Interoperable Layer facilitates the definition of an integrated (federated) schema, thus providing a single access point to multiple local databases joined in a federation. The integrated schema is defined using global views. Views are specified as an OQL queries and are able to integrate and restructure interface definitions imported from multiple Local Layer schemas. Views and imported interface definitions are stored in the specially constructed data dictionary built upon the specialised ODMG compliant database at the Interoperable Layer. This database is called *home OODBMS* and its roles also include materialisation of virtual objects, transaction control and provision of data manipulation interface in a form of OQL embedded within the C++ mappings for ODMG.

Schema Definition. The IRO-DB project extends the ODMG data model with the ability to define federated schemas. Therefore, an IDL data definition language is extended with support for defining virtual classes in the integrated (federated) schema and with syntax for importing an existing class definition from multiple local nodes to integrated schema [BFN94]. These extensions are similar to ones defined in the MIND project. The main difference is that instead of proprietary query language, virtual class extents in IRO-DB are defined in the ODMG OQL language.

Each schema transformation step adds a new layer of virtual classes. Therefore, four different types of classes can be distinguished in the IRO-DB integrated and local schemas.

1. **External classes.**
   These classes are created at the Local Layer to provide an external schema that can be accessed and queried by the federated clients. External classes are simply ODMG wrappers for data in proprietary data sources. Their definition is specified as a set of interfaces in the ODMG IDL schema definition language, while their implementation is platform specific and defines mappings to the underlining proprietary data sources.

2. **Imported classes.**
   Imported classes are 1-to-1 copies of external classes imported to the federated schema. They provide only a means for hiding physical locations and for making the external classes accessible from the Interoperable Layer. Imported classes generate a global instance for each local object that needs to be accessed from the integrated schema and propagate accesses to the attributes of this global instance to the original local objects. Thus, they act as global proxies. A simple extension to the ODL language (keyword `imported`) is provided for specifying classes to be imported into the federated schema.

3. **Derived classes.**
   This is a further transformation layer in the federated schema that integrates imported classes originating from multiple local databases. Derived classes also provide schema restructuring capabilities, as they are able to hide attributes of the underlining imported classes or derive new ones. Thus, a derived class may be based upon multiple imported or other derived classes. The interface of a derived class is specified in the ODL language, while its implementation is defined as an OQL query embedded within a `mapping` declaration. This declaration is an IRO-DB extension to the ODL schema definition language for defining the extent and attributes of derived virtual classes. Thus, a derived class acts as a simple view in the federated schema.

4. **Standard classes.**
   These are stand-alone classes defined in the integrated schema and are not based on any imported class.

An extent of each virtual class contains surrogate objects only. This is an important feature of the IRO-DB federated schema as surrogate objects do not contain any data, but provide references to other real or surrogate objects. Thus, surrogate objects instantiated from an imported class are simply references to corresponding objects in the Local Layer databases. Surrogates instantiated from a derived class reference other derived or imported surrogate objects in the integrated schema. This can create multiple referencing levels until each surrogate object is ultimately resolved to a real object in a local database.

Surrogate objects are important for updatability of virtual classes, as each update on the surrogate object is directly propagated to objects in local databases it is based upon. Therefore, IRO-DB views are updatable.

Metamodel. The metamodel specification for IRO-DB is discussed in [BFN94]. This metamodel stores virtual class metadata in an object-oriented database schema, thus facilitating their dynamic querying. Both interface definitions for virtual classes and the implementation mappings are extracted from the ODL schema specifications and represented in the metamodel. Interface definitions are represented in the `InterfaceDef` metaclass, while their attributes, relationships and operations are represented in the associated `AttributeDef`, `RelationshipDef`, and `OperationDef` metaclasses. An OQL query defining the interface implementation is stored in the `QueryDef` metaclass which is also associated to the `InterfaceDef` metaclass. Thus, both the structure and extent definition of virtual classes can be queried from within the schema repository. However, direct mappings from virtual classes in the integrated schema to external classes in local databases are not represented in this metamodel. Furthermore, each attribute and relationship of a virtual class is mapped to a query defined upon its extent specification, and not directly to properties of the underlining external classes.

Global queries. The IRO-DB system uses ODMG OQL as a common query language to facilitate both local and global queries [SFF95]. Since OQL does not support updatability, queries in the IRO-DB are a retrieval only. However, query results can be updated from within the C++ programming language interface to which OQL query calls are embedded. The result of each query is represented as a set of surrogate objects in the integrated schema that maintain mappings to base objects physically storing data. Thus, an update on the surrogate object's property is propagated to an update of an equivalent property of its base object.

Each global query is evaluated in the home OODBMS that stores both federated schema and metamodel. Global query processing can be broken down into four general steps.

1. The `OQL Parser` processor performs syntax and semantic analysis of the global query to check whether the query references the correct class and property names. This analysis is preformed by examining the metamodel definitions of virtual classes upon which the query is based.

2. The `Global Query Processor (GQP)` decomposes a global query into multiple subqueries, where each subquery is targeted at one local database. Metadata information is used to resolve all derived virtual classes specified in the query definition to their actual OQL implementations. Thus, each derived class is replaced with an OQL query, and this process is repeated recursively until all subqueries refer to imported classes only.

3. At Local Layer databases, subqueries are executed in proprietary databases, and object identifiers for retrieved objects are generated. This set of identifiers is then returned through the communication interface to the Interoperable Layer.

4. The `Global Query Evaluator (GQE)` processor receives results from all subqueries, and generates imported surrogate objects corresponding to object identifiers contained in these results. Then the `GQE` recomposes these surrogate objects by starting the evaluation of the global query part. This effectively integrates data from multiple sources and produces a final result in a form of derived surrogate objects. These objects are sent back to the client application which can modify their state. Updates to surrogate objects are propagated through the object references back to corresponding local database objects.

When imported and derived surrogate objects are created in the home DBMS, they can be instantiated in one of three states. These instantiation modes are significant as they directly influence the optimisation of global query processing.

- **Minimal state.**
  It corresponds to the minimal instantiation, where no surrogate objects are created. Thus, the global query is completely evaluated at local sites, and no global objects are created.

- **Total State.**
  This state corresponds to the total instantiation of a virtual class in the home DBMS. This fully instantiates all surrogate objects belonging to the virtual class extent. The consequence is that all local data is moved to the Interoperable Layer before a query is evaluated. Therefore, a global query is fully resolved within the federated schema and no remote access is required. The downside of this approach is that large amounts of objects must be transfered and cached at the global database node. Also, all query processing is centralised within the integrated schema.

- **Partial State.**
  It corresponds to a partial instantiation when only part of the virtual class extent is instantiated in the federated schema. Query processing with a partial state requires a mixed evaluation. Some information is already cached in the home database, but not all. Thus, remote accesses are also required. The principle is that missing data is

retrieved from local sites by issuing subqueries, and then integrated with previously cached data to evaluate the global part of the query. In this approach processing and data transfer is balanced between local and global layers, but data integration can be complex as it requires numerous calculations.

Transactions. Updatability at the federated layer of the IRO-DB architecture is supported with a global transaction control system. This system is based upon the open nested transaction model [TW97], where each global transaction consists of multiple layers of nested subtractions. Global seralisability is ensured by the optimistic ticket method [GRS94] that does not violate the autonomy of local databases. This method forces conflicts between subtransactions on local database to determine their serialisation order. Thus, if subtransactions are serialised in the same relative order at all local databases, then the seralisability of the global transaction is also preserved. A two-phase commit is required for the processing of commits and aborts of global subtransactions. Therefore, each local database must natively support a visible prepare-to-commit state or it can be simulated by the `Local Transaction Manager (LTM)`. The `LTM` guarantees that a subtransaction cannot be unilaterally aborted by the local database if it is fully executed and does not issue further data operations.

The `Global Transaction Manager (GTM)` processor implements transaction control at the global level. Therefore, the execution of a global transaction by the `GTM` can be represented as a sequence of four steps.

1. **Take tickets.**
   Tickets are taken by the GTM from each participating local database the global transaction has accessed.

2. **Prepare the local transaction to commit.**
   The GTM acts as commit coordinator in a two-phase commit protocol.

3. **Validate the global serialisation order.**
   The GTM checks the order of tickets each subtransaction obtained, to determine if all subtransactions follow the same relative execution order.

4. **Enforce the decision.**
   If the validation of a global transaction was successful, the GTM commits all subtransactions, otherwise, an abort command is issued. This completes the global transaction.

Limitations. The canonical data model chosen for the IRO-DB system is the ODMG model. Although, it represents a standard for object-oriented databases, this model is not fully supported by commercial O-O databases. Therefore, the IRO-DB project was forced to develop complex adaptors to provide at least a minimum ODMG and OQL facility on top of existing object-oriented databases used in the system. Furthermore, the ODMG model does not provide sufficient support for large objects required for multimedia representation, nor can the server-side behaviour be defined to support performance intensive multimedia operations. The available literature does not discuss mappings between the ODMG canonical model and proprietary object-oriented and object-relational databases at the Local Layer. Basic ODMG wrappers are provided for only two databases: the relational database Ingres and the object-oriented database O_2. A relational wrapper is limited only to transformation of tuples into objects, while more advanced features such as referential integrity and behaviour are not supported.

Although the OQL query language is a natural choice as a global query language operating on top of an ODMG database, it has several deficiencies. Firstly, OQL does not support updates from within the query language. Secondly, all query language operators are hard-coded into the language itself, and cannot be modified to support multimedia data types. Finally, query translation from OQL to the proprietary languages of the Local Layer databases is non trivial, and published IRO-DB research does not discuss this issue. The evaluation of global queries in IRO-DB can be optimised by caching local data as surrogate objects in the federated schema. However, keeping this object cache synchronised is difficult when operating in the multi-user environment. The metamodel for IRO-DB follows the ODMG metamodel specification, and extends it with metaclasses for representing virtual classes in the federated schema. However, mappings from virtual to base classes are not defined in this metamodel.

Summary. The IRO-DB project delivers a system that integrates relational and object-oriented databases into database federations using ODMG standards. Therefore, the IDL schema definition language is extended to define federated schemas, while OQL is used as a global query language. The main advantage of the IRO-DB system is its ability to provide updatability at the global level. This is achieved by creating surrogate objects in the federated schema for all objects imported from local databases. Thus, each update on a surrogate object is propagated directly to its base objects in local databases. Although this feature allows for the definition of an updatable global query language, updates are not supported in the existing OQL standard. The IRO-DB metamodel stores definitions of virtual classes at the global layer, thus providing for dynamic querying. Global updatability based on surrogate objects is reused in the EGTV project, where it forms a platform for the definition of a new query language that supports global updates. We also reuse and improve the IRO-DB concept of transaction control and provide multimedia extensions to the query language.

2.7 Hera Project

The primary focus of the Hera project [VH01, VBH03] is to provide an integration interface for heterogeneous semi-structured data sources. This is achieved by constructing a global schema that integrates multiple semi-structured repositories into a queryable multidatabase system. Furthermore, the Hera project is designed specifically to enable capturing of web-based data sources and to provide efficient querying at the global level.

Architecture. The architecture of Hera system consists of four layers [VH02]: Source, Reconciliation, Mediating and Application layer.

- **Source layer.**
  The Source Layer contains external data sources such as relational or object-oriented databases, HTML pages, XML repositories, or RDF data sources. This layer provides the content to be integrated, but all data sources are expected to have capability of exporting their data in XML serialisation. However, since this is not supported by most legacy data sources, some external XML wrapping processes may be required. Due to the heterogeneity, none particular structure of the exported XML data is imposed, thus fully preserving the autonomy of the local data source.

- **Reconciliation layer.**
  This layer defines a canonical schema unique within the entire Hera system in a form of RDF/RDFS schemas. The Resource Description Framework (RDF) [Wor99] is a general purpose data language issued as a W3C standard that describes data in a terms of resources, their properties and property values. The RDF Schema (RDFS) [Wor99] is an extension to RDF that provides a support for creating vocabularies at the metadata (schema) level. The Reconciliation layer consists of multiple XML2RDF brokers that provide mapping between XML data repositories at the Source layer and the canonical schema in the RDF representation. Each XML2RDF broker is tailored specifically to its data source by encoding mapping rules in a simple mapping language LMX [VH01]. Mapping rules are automatically generated for structured data sources (i.e. relational and object-oriented databases) or when a full XML Schema definition of a data source is provided. However, for most of the semi-structured data sources, mappings must be manually defined.

- **Mediation layer.**
  The Mediation layer is responsible for integrating canonical representations of local data into a global schema. The global schema is constructed as a union of all canonical schemas defined at the Reconciliation layer. The other role of this layer is to facilitate evaluation of global queries in the Hera system. Each global query at this layer is firstly decomposed into multiple subqueries which are then distributed

among brokers. In this process it is attempted to push processing of the query as much as possible on the source data repositories, while taking into account the source query capabilities. In a second stage of global query processing, partial results are collected from the brokers and used as a source upon witch global part of the query is evaluated. This produces the final result which is then returned to the issuing client.

- **Application layer.**
  This layer acts as a client to the Mediation layer as it generates global queries and receives their results. It also facilitates the graphical front-end interface for the end-users of the system. This front-end displays global RDF schemas in a navigable tree representation and guide users to visually build queries by selecting RDF nodes and their properties. This generates global queries in the RQL language [KAC+02] which are then passed to the Mediation layer for the execution. Retrieved results are converted back to the graphical representation and displayed to the end-user.

Data model. Canonical data model chosen for the Hera project is RDF/RDFS [Wor99, Wor00] standard. This is a semi-structured model, therefore it is better suited for capturing web sources that traditional structured-data models. This model provides uniform interface to all data sources integrated in the Hera system. The RDF and RDFS languages are fully standardised by the W3C committee and can be represented either in a XML-like tagged encoding or in its DOM-like memory representation. This is illustrated in *Example* 2.8 where a RDFS metadata for the `Character` class is given. The RDF encoding for objects conforming to this class is provided in *Example 2.9*. Since, the RDF schema standard does not support data types, `String` and `Relationship` types used in *Example* 2.8 represent Hera's extensions of the RDFS. All extensions are defined in a proprietary `sys` namespace. RDF schemas can be further extended with higher level ontology languages to enhance the expressive power of the canonical model. However none of these is implemented in the Hera project.

Query language. The Hera project implements RDF/RDFS as its canonical model to facilitate capturing of web-based data sources. Therefore, its query language must be capable of retrieving and browsing semi-structured data in these repositories. The RQL query language [KAC+02] is chosen for a global query language as it provides querying of both metadata RDF schemas and their data instances. Furthermore, it defines predicates for position based queries on unstructured and semi-structured RDF data sources. The language, similarly to SQL, consists of `select-from-where` clauses. The `select` clause specifies resources (variables) to be retrieved; the `from` part defines one or more path expressions to which the variables are bound; while the `where` clause contains filtering conditions that are applied to variables bound in the from statement. This is illustrated in *Example* 2.10 where a simple RQL query is defined over the sample RDF schema. This

```
<?xml version='1.0' encoding='ISO-8859-1'?>
  <!DOCTYPE rdf:RDF [
    <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <!ENTITY rdfs 'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#'>
  ]>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">

<!--- Class Character -->
  <rdfs:Class rdf:about="Person">
</rdfs:Class>

<!-- Properties of the Character class -->
<rdf:Property rdf:about="name">
  <rdfs:domain rdf:resource="Character"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="description">
  <rdfs:domain rdf:resource="Character"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="appearsIn">
  <rdfs:domain rdf:resource="Character"/>
  <rdfs:range rdf:resource="Cartoon"/>
</rdf:Property>
</rdf:RDF>
```

Example 2.8: RDFS schema definition.

query selects names of all cartoons where Goofy character appears. The central part of this query is the `from` clause where two path expressions are used to bound `CARTOON_NAME` variable retrieved in the `select` clause and `CHARACTER_NAME` variable used in the filtering condition of the `where` clause. Variable bindings must be provided for each stage of path navigation when traversing an RDF schema. Although the RQL is flexible enough to support complex RDF schema queries, its syntax is not intuitive, and therefore a simple graphical front-end has been provided for the Hera project. This front-end allows visual (point-and-click) composition of queries which are then automatically converted to the RQL representation. However its functionality is limited only to very simple queries, and RQL language cannot be fully encapsulated.

Limitations. Although the RDF/RDFS defines an efficient representation for semi-structured data it does not provide all modelling primitives required to support structured data models. Namely, the notion of cardinality and inverse relationships is missing and there is also a lack of system types. The later is compensated in the Hera project by

```
<Character rdf:about=Goofy_Object>
  <name>
    <sys:String>
      <sys:data>Goofy</sys:data>
    <sys:String>
  </name>
  <description>
    <sys:String>
      <sys:data>Cartoon character</sys:data>
    <sys:String>
  </description>
  <appearsIn>
    <sys:Relationship>
      <sys:data>Goofy and Friends</sys:data>
    <sys:Relationshp>
  </appearsIn>
</Character>
```

Example 2.9: RDF schema instance example.

```
select CARTOON_NAME
from {CHARACTER:Character}appearsIn{CARTOON:Cartoon}.name{CARTOON_NAME},
     {CHARACTER}.name{CHARACTER_NAME}
where CHARACTER_NAME="Goofy"
```

Example 2.10: RQL query example.

defining a custom extensions to RDF/RDFS standards. However these extensions are proprietary and limit the interoperability of the system with other RDF/RDFS data sources. Furthermore, the query language RQL has complex and nonintuitive syntax and does not provide updatability and schema restructuring capabilities. Global schema is just an union of all canonical schemas and views cannot be specified.

Summary. The main advantage of Hera system is its ability to integrate semi-structured data sources with traditional databases in a multidatabase system and to provide an interface for global querying. This enables definition of queries that can combine results from web-based sources and structured databases to generate a single result set. However, the query language is read-only and updates are not supported. Furthermore, since queries are evaluated against in-memory representations of RDF objects and no multimedia specific operators are provided, this language is not suitable for querying large multimedia repositories. Therefore, our query language for the EGTV project takes a different approach by utilising structured object-oriented schemas to provide efficient object management and support for user-defined operators of the query language. Behaviour of these operators can be customised and optimised for any specific domain, including multimedia.

2.8 Conclusions

In this chapter some of the major research projects on object query languages were examined, together with some projects which specified metadata architectures, multimedia querying and global transaction control. During the study of these projects, some key characteristics emerged, which together with the analysis of federated and multimedia systems from chapter one, provide the functional requirements for a suitable global query language.

These requirements are summarised below: 1, 2 and 3 emerged as broad requirements from the study of existing object-oriented query languages; 4 and 5 are prerequisites that must be defined to support global queries in the federated environment; and 6 and 7 are outputs from the study of federated database systems.

1. **The SQL-like syntax of the query language.**
   The new query language for an EGTV database federation must provide a simple and easy to use client interface. Therefore, our query language should employ SQL-like syntax familiar to the majority of database users. All research projects examined in this chapter excluding LOQIS [SBMS94] follow this approach. SQL-like syntax is also used by the ODMG OQL query language, a standard for object-oriented databases.

2. **Updatability for queries and views.**
   Our query language must support updatability at both global and component layers of a federated architecture. This feature is essential, as clients of the federation are required to insert new and update existing multimedia and textual data in O-O and O-R databases. Updatability is provided in LOQIS [SBMS94] through direct referencing between virtual and base objects, and in IRO-DB [GGF$^+$96] which uses proxy objects in the global schema to relay updates to local databases. However, LOQIS does not facilitate the definition of a federated schema, nor does it support queries at the global level. IRO-DB uses OQL as its global query language, thus updates can only be facilitated through external behaviour and not within the query language itself. The remaining projects assessed in this chapter do not provide updates in the global schema.

3. **Multimedia extensions for query language.**
   The EGTV query language must provide extensions for manipulating different multimedia types stored in a database. However, contrary to the News-on-Demand (MOQL language [LOSO97]), our multimedia extensions should be flexible, and not hard-coded in the language itself. Therefore, operators of the query language should be defined within data types as *behaviour methods*. Thus, operators can be easily customised for each data type to optimise execution of complex multimedia operations.

Furthermore, the proposed query language is not bound to a single (multimedia) domain, and can be easily extended with different data types and operators.

4. **Federated metadata.**
Metadata information is crucial for any generic querying as it provides a runtime description of database schema and relationships between its elements. Therefore, specification of a common metamodel for the EGTV federation is an important prerequisite for a global query language. Of the projects involved in this study, only LOQIS [HRS02] fully discusses a metamodel and its ability to represent database schemas. However, this metamodel does not support data distribution, nor can it operate in a federated environment. There is very little metadata research in other assessed projects: News-on-Demand [OSEM+96] represents its schema metadata in a form of DTD files, IRO-DB [BFN94] extends the ODMG schema repository with virtual class metadata, while Hera [VBH03] uses RDFS standard to define semi-structured schemas. However, none of these projects fully supports dynamic schema repository interrogation, federated metadata or multimedia data types. Thus, a new metamodel must be defined for the EGTV project. This metamodel should be capable of providing a common representation for both component and global schemas in a federated architecture. Furthermore, it should be easily queried, and be capable of representing multimedia data types. Since all data in our federation is physically stored in O-O and O-R local layer databases, mappings must be provided between platform independent EGTV metamodel and local layer databases.

5. **Schema definition language.**
Schema definition for the EGTV federation should provide the ability of specifying multimedia database schemas in a platform neutral format. This is required both at the component layer where local O-R and O-O database schemas are captured, and at the federated layer where virtual classes must be defined to integrate multiple component database nodes. Related research projects specify their database schemas either in a form of SQL DDL (MOOD/MIND [Alt94]) or extended ODMG IDL schema definition language (Garlic [CHS+95] and IRO-DB [BFN94]). The News-on-Demand project [WLE+97] takes a different approach as it defines multimedia schemas using the SGML markup language. This benefits in providing simple algorithms for parsing and constructing the database schema. A similar approach is taken in the Hera project [VBH03] where schemas are specified in the XML encoding of the RDFS language. The EGTV schema definition language should be based on ODMG IDL, as it is well suited for definition of object schemas. However, the syntax of our query language should be defined in XML and supported with the XML Schema, as it provides the highest level of interoperability and simple parsing algorithms. Furthermore, mapping between this schema definition language and the EGTV metamodel must be specified to support dynamic generation of queryable metadata.

6. **Global query processing.**
   The query system should have clear semantics of query evaluation, both for local and global queries. Of the query systems involved in this study, LOQIS, MOOD/MIND and IRO-DB provide the more detailed descriptions of the generation of query results. LOQIS uses a stack based approach to evaluate queries and generate fully updatable results, although it does not support global queries. MOOD/MIND, IRO-DB and Hera use query decomposition in the federated schema to evaluate global queries, but their results are not fully updatable. Our aim is to provide full updatability within the EGTV query language at both component and global layers of the federated architecture.

7. **Global transaction management.**
   Updatability at the global layer requires an efficient transaction control system. Thus, EGTV query processing services should incorporate local and global transaction managers. This is similar to the approach taken in MOOD/MIND [DDK$^+$96] and IRO-DB [TW97] projects where global seralisability and consistency is ensured by implementing ticketing serialisation method and two-phase commit protocol.

The hypothesis presented in this thesis is that an efficient query language can be defined to support global queries and updates in the multimedia database federation. Therefore, a critical review of existing research projects was conducted in this chapter to identify requirements for such a language. The review was focused to the area of federated systems, global query languages and metadata architectures. The latter study was necessary once the requirement for a metamodel and schema definition language was identified in chapter one. This critical analysis enabled us to define methodology for our research and metrics for validating the feasibility of the hypothesis. It was concluded that a full specification of a new object-based metamodel is a vital prerequisite for providing generic query interface. Therefore, before the new EGTV query language is introduced in chapter five, it is necessary to present the EGTV metadata architecture (in chapter three) and new schema definition language (in chapter four). Transaction control system must also be defined to support updates at the global level. Thus, chapter six presents requirements for such a system and addresses its implementation. Prototype developed to support this research is discussed in chapter six, where thorough analysis of conducted experiments is also presented. The thesis concludes with the reexamination of hypothesis and detailed discussion of areas for future research, which is offered in chapter seven.

Chapter 3

The EGTV Metamodel

Current object-oriented and object-relational data models provide only basic support for handling multimedia and other complex data. This is limited to storing an object's state, while the server-side behaviour can only be specified in proprietary languages. Therefore, a new data model was defined within the separate research track of the EGTV project to provide a common interface to multimedia objects stored in object-oriented and object-relational databases. This model also supports definition of multimedia specific server-side behaviour using standard programming languages such as C++ or Java. However, this aspect of the EGTV research had no requirements for metamodel as it did not facilitate any schema integration or generic querying features. *Figure 3.1* presents all components of the EGTV project. Those represented in shaded boxes (`EGTV Model` and `Behaviour Processing`) are external to this thesis and are part of a separate research track within the EGTV. All other components of the EGTV projects (represented in transparent boxes) contain research covered in this thesis.

The ability to capture and store metadata of multiple database schemas is essential when constructing a database federation. This is emphasised in the EGTV project where the focus is on the integration of heterogeneous multimedia data sources. Thus, additional metadata information must be defined to capture multimedia specific data types and to

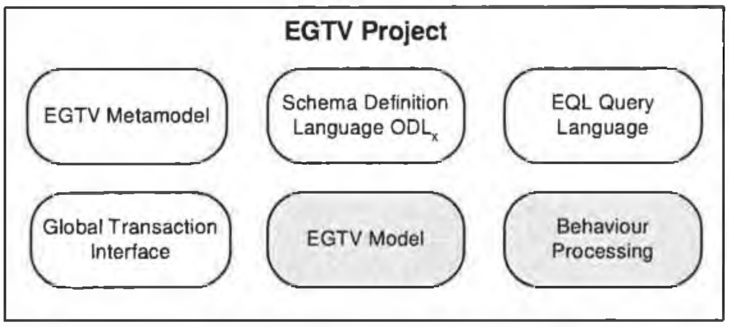


Figure 3.1: Components of the EGTV project.

describe heterogeneous and often incompatible data models. In this chapter we present a new metamodel for representing multimedia schemas stored in both object-oriented (O-O) and object-relational (O-R) databases. This metadata information is later used for the construction of a federated schema and during the evaluation of local and global queries. However, since EGTV metamodel represents multimedia schemas in platform independent format, it must be mapped to object-oriented and object-relational databases where actual data is stored. This is achieved by defining rules for transforming EGTV metamodel structures to equivalent O-O and O-R ones.

In §3.1 a general introduction to metadata and metamodeling is provided. Different metamodel representations for relational, object-relational and object-oriented databases are discussed. §3.2 provides a detailed description of the EGTV metamodel and its components using the UML notation. Special emphasis is placed on improvements and extensions to the ODMG metamodel. A meta-metamodel is also defined to represent different metamodel versions. This allows users to change their database model according to their needs. Metamodel mappings to object-oriented and object-relational metamodels are discussed in §3.3, as these define rules for metadata translation between the EGTV metamodel and object-oriented and object-relational standards. A simple language is introduced to formally define mapping rules. Finally, conclusions are presented in §3.4.

3.1 Introduction

A metamodel is a model that describes other models. It consists of data elements that define the structure of underlining models. Each metamodel can be seen as an instance of some other metamodel at the level above, and there is no restriction on number of levels. Usually, a four level approach [Ode95] is used when defining metadata levels in a software engineering and database applications.

1. **Data and Process Level:** This is the lowest level. Entities on this level are runtime objects, i.e. instances of classes and processes running on a concrete system.
2. **Model Level:** This level is an abstraction of the data and process level. Entities defined at the model level (i.e., classes, tables, relationships) describe the underlying physical system. In a database, this level is represented as the database schema.
3. **Metamodel Level:** The metamodel level describes the structure and capabilities of the model level. In databases, this is a description of how a class or table is defined and how it relates with other classes or tables.
4. **Meta-Metamodel Level:** This level defines the metamodel level. Different metamodel structures can be developed as instances of the meta-metamodel level. For example, a meta-metamodel can define the structure of the database metamodel.

Metamodel classes and their relationships are represented as objects in the meta-metamodel.

The main purpose of any metamodel is to provide a detailed specification of data models at lower levels in the metadata hierarchy. For example the metamodel for the Unified Modelling Language (UML) [BRJ99] describes the structure and elements of the UML modelling language. The UML metamodel corresponds to the third level of the metamodelling architecture and is itself defined recursively using UML [BRJ99].

The main function of a database metamodel is to describe the database schema. Schema metadata includes a catalogue of system data types and their properties, user defined entities such as tables or classes, relationships and database behaviour. Additional metadata may include physical data storage organisation, data distribution information, database users and security rules. The exact content and structure of a database metamodel depends of concrete implementation. A metamodel schema is usually stored within the database in a special segment called the *schema repository* [EN94].

The metamodel and schema repository of relational databases are relatively simple, due to the simplicity and highly formalised definition of the relational model itself. While the metamodel for relational databases is not standardised, the majority of implementations are very similar, although not always compatible. Relational database vendors usually represent a schema repository as a set of relational system tables that store metadata information on user defined tables, columns, data types, constraints and other relational model elements. The heterogeneity was partly overcome by ODBC (Open Database Connectivity) that standardises a data access interface for relational databases. The problem of heterogeneous and non-standard metamodels still remains.

The SQL:1999 specification [GP99] defines a schema repository standard for representing object-relational model in the form of an *Information Schema*. The Information Schema is a special database schema that defines a set of relational views and tables for representing both relational and object-relational metadata. However, this standard has not been widely accepted, and no object-relational database has implemented it yet. Object-relational databases extend relational databases by adding an object interface on top of a relational database engine. The majority of object-relational databases define a schema repository in the form of extensions to the existing relational metamodel[1]. Although it can represent object types and object tables, the object-relational schema repository itself is implemented as a set of relational tables. Thus, the metamodel can be regarded as semantically poorer than the model.

The main standard for object-oriented databases is defined by the Object Data Management Group (ODMG). The current version of the standard is 3.0 [CB99] and the majority of object-oriented databases conform (at some level) to this standard. The standard

[1]For this reason, we regard the Oracle 9i model as a standard as it has the most advanced object-relational model.

introduces an Object Definition Language (ODL) to define a database schema at an abstract level. This feature enables the portability of schema definitions between different implementations of ODMG compliant databases. Unlike ODL itself, the ODL Schema Repository is not fully specified in the standard. The ODL Schema Repository is a metadata repository for ODL definitions and allows runtime queries and updates. The Schema Repository specification consists of a set of ODL interface definitions, where each interface defines a single database construct. The metamodel presented in this specification is complex, with a large number of interfaces and association links between them. Interface definitions are cumbersome, and the same information is repeated at different places in the metamodel. Each interface defines a number of data access operations. These operations provide a means of ensuring a semantic integrity of metamodel by implementing rules for the creation, addition and removal of metadata from the metamodel. The drawback of this approach is in limitation of generic metadata access. Users are not able to perform generic queries and updates, but forced to use a predefined set of operations.

In [Jor98], they define a C++ implementation for the ODMG metamodel, where only some segments of the ODMG metamodel are implemented in a form of C++ classes, while others are restructured or totally omitted. Although this implementation corrects some redundancies and imprecise definitions, it preserves an overall structure of the metamodel as defined by the ODMG. Contrary to ODMG metamodel specification, this metamodel is supported by graphical diagrams and explanations in natural language making it more understandable to the reader. Still, this model inherits a lot of redundancies from the original ODMG metamodel: redundant relationships between metamodel elements, retrieval and manipulation operations that limits the generality of metamodel access, and redundancy of some metaclasses.

3.2 The EGTV Metamodel Specification

This section describes the object-oriented metamodel designed for multimedia databases, which has a special emphasis on database integration. In this context, the metamodel defines an object-oriented meta-schema for representing textual and multimedia metadata for databases at both the canonical and federated layers. An object-oriented model is chosen as it has already been shown that an object model is the most suitable for a canonical data model [SCGS91] in a federated database architecture.

The EGTV metamodel is based on the ODMG metamodel specification [CB99] and the C++ implementation defined in [Jor98]. Our metamodel eliminates ambiguities and redundancies present in both specifications by clearly defining the metamodel structure, and significantly reducing the overall complexity. Some modifications required for the representation of multimedia data types and behaviour were also introduced. The metamodel does not incorporate the metadata access interface (as the ODMG metamodel does) because this has been shown to limit generic query capabilities [HRS02]. All metaclasses in our metamodel are identified by the `sys_` prefix.

The specification defines the abstract, platform independent representation of the metamodel structure. The actual implementation of metaclasses is not discussed here as it depends on the type of database upon which the metamodel schema repository is implemented. The metamodel mappings developed for the object-relational and ODMG databases specify platform specific implementation details, and are presented later in this chapter. The full UML specification of the EGTV metamodel is provided in *Appendix A*.

3.2.1 Defining Name Scopes

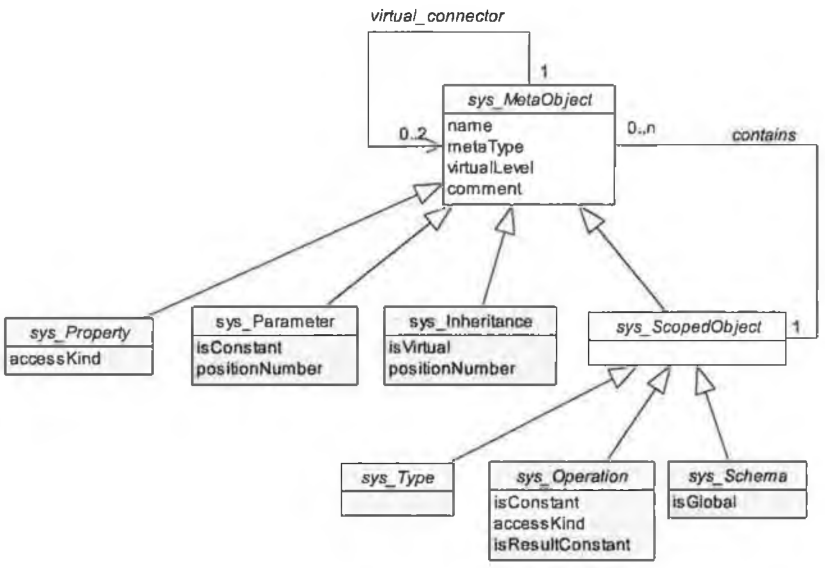


Figure 3.2: Metadata Definition of Name Scopes.

Each database entity has a name that must be unique within the scope to which it belongs. For example, attribute names are uniquely defined within the containing class, while class names are unique in the database schema. Name scopes and containment relationships in the metamodel are closely related because each metaclass provides a scope for all its contained elements. Our metamodel defines a single containment/scope relationship between metaclasses as illustrated in *Figure 3.2*.

The `sys_MetaObject` is an abstraction for all elements in the metamodel and defines common attributes such as `name`, metaclass type (`metaType`) and `comment`. These three attributes provide name, type and user defined comment properties for all elements stored in the metamodel. Metaclasses that are not capable of containing other metamodel elements, like `sys_Property`, `sys_Parameter` and, `sys_Inheritance` derive directly from the `sys_MetaObject`, while all container metaclasses derive from `sys_ScopedObject`. The `sys_ScopedObject` defines the containment relationship (`contains`) to sys_MetaObject, so that each instance of sys_ScopedObject can contain and define name scope for many sys_MetaObject instances. For example, an instance of sys_ScopedObject can contain attributes, relationships and operations which are uniquely identified within the scope of that class.

Since the sys_ScopedObject metaclass also derives from the sys_MetaObject metaclass, each container metaclass can recursively contain another container class. Metaclasses sys_Class, sys_Schema and sys_Operation derive from the sys_ScopedObject, as they provide naming scope for elements contained within them. The top level scope is the database schema itself (sys_Schema), and it contains the classes (sys_Class) defined by users.

3.2.2 Defining Types

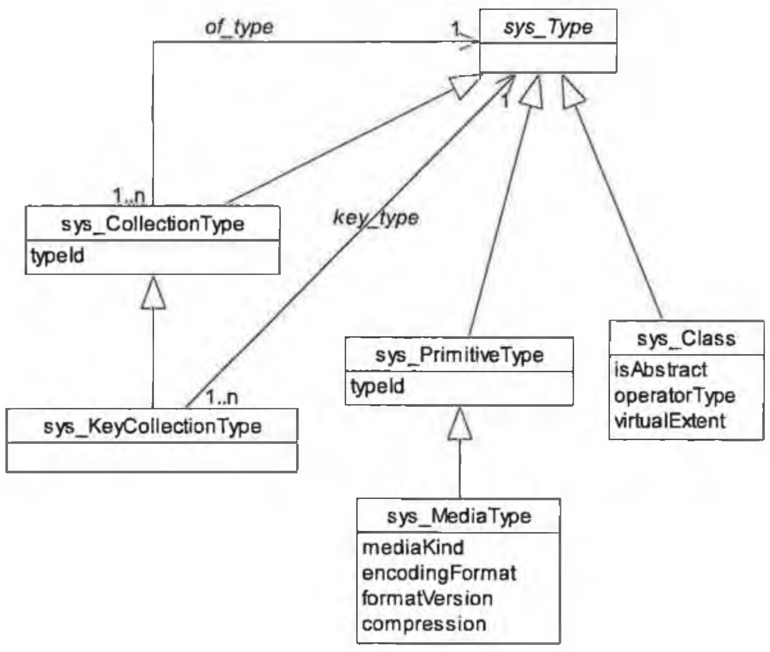


Figure 3.3: Metadata Definition of Data Types.

Type metaclasses derived from sys_ScopedObject provide a description for all built-in and user-defined types. All data types are represented in a metadata class hierarchy as illustrated in *Figure 3.3*. The sys_Type metaclass is an abstraction for all types in the database, and the more specific metaclasses which derive from it. Our metamodel is enhanced by permitting operations for the sys_Type metaclass, whereas the ODMG metamodel permits only user defined classes to have operations. Moving operation support to the level of the sys_Type base class enables the definition of operations not only for classes, but also for other data types (e.g. multimedia and collection types). The importance of this feature arises from the fact that the internal structure of complex data types is fully encapsulated and the only interface is provided through operations. For example, a `jpegImage` multimedia data type can have operations for query by pattern, resizing and rotating of an image it contains. These operations are registered in the metamodel and publicly available, while the implementation and storage details of the `jpegImage` data type are hidden from the user.

The built-in and user defined types are represented as a specialisation of the `sys_Type` metaclass. Built-in types are used as attributes of classes, or parameters of operations, and they cannot be instantiated to persistent self contained database objects. Built-in types can achieve persistence only as attributes of user defined classes. All built-in types can be classified as primitive or collection types.

- **Primitive types:** Represented in the `sys_PrimitiveType` metaclass. The full set of primitive types are `Integer`, `Float`, `String`, `Date`, `Blob` and `Boolean`. Additional primitive types can also be defined and added to the metamodel.
  Special data types for multimedia storage are incorporated into this metamodel and they are represented as instances of the `sys_MediaType` metaclass. The metamodel defines the `sys_MediaType` metaclass as a specialisation of the `sys_PrimitiveType`. The `mediaKind` property identifies multimedia type (`audio`, `video`, `text` or `image`), while `encodingFormat`, `formatVersion` and `compression` provide information on media encoding characteristics. The internal structure and storage details of multimedia data types are encapsulated from the user. An interface is provided through operations registered with the multimedia type. For example, operations provide an interface for searching, retrieval or updating of the corresponding multimedia objects.

- **Collection types:** Collections store multiple objects of one system type and are represented by the `sys_CollectionType` metaclass. Supported collection types include `Bag`, `Set`, and `List` as specified in the ODMG standard. Key collection types are defined in the `sys_KeyCollectionType` metaclass. They are collections of key-value pairs optimised for fast indexing and are represented with a `Map` type. Operations for element manipulation are defined for each collection type as operator and method behaviour.

The `sys_Class` metaclass represents user-defined types and corresponds to `d_Class` in the ODMG metamodel. Classes can contain attributes, relationships and operations for which they provide scope. All classes defined in the metamodel are classified in two categories: base classes and virtual classes. Base classes can be instantiated to persistent database objects that store data, while virtual classes are components of object views and they are constructed from the base classes. Virtual classes and object views are discussed in the §3.2.7. The `isAbstract` property is applicable only to base classes and is used to specify if the class is defined as an abstract one.

3.2.3 Defining Properties

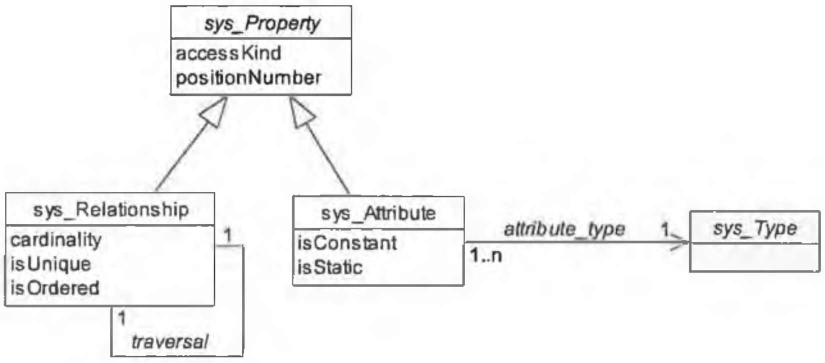


Figure 3.4: Metadata Definition of Properties.

The `sys_Property` metaclass is an abstraction for all property types that can be specialised as attributes or relationships and it is derived directly from the `sys_MetaObject`. The `positionNumber` property indicates the relative order of properties in the class definition, while the `accessKind` can be `private`, `protected` or `public`. Members of a class are specified in the `sys_Attribute` metaclass, where each attribute can be optionally defined as static or constant using `isStatic` and `isConstant` properties. Attributes can be of a system or a class type, where each attribute has only one type, while one type can be used by many attributes. This is represented with the `attribute_type` relationship between `sys_Attribute` and `sys_Type` metaclasses where the `sys_Type` is a superclass for all types in the metamodel.

The `sys_Relationship` metaclass defines bidirectional relationships between two classes where a cardinality of `one` or `many` is specified for each side of the relationship. The traversal property of the `sys_Relationship` returns the other side of the bidirectional relationship. Each relationship with cardinality greater than one can have ordered values (`isOrdered` property) or it can be defined as unique (`isUnique` property).

3.2.4 Defining Inheritance

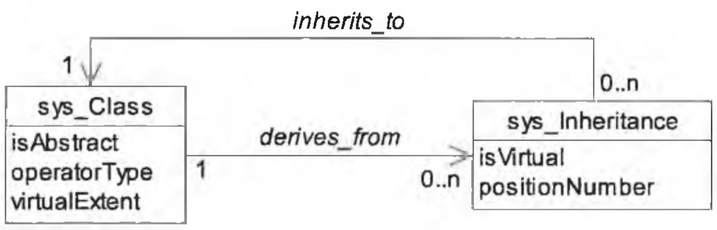


Figure 3.5: Metadata Definition of Inheritance.

Inheritance relationships are defined for classes only. Inheritance in the metamodel is represented by the `sys_Inheritance` metaclass which inherits from the `sys_MetaObject`. The instance of `sys_Inheritance` has an `inherits_to` and a `derives_from` relationship with two instances of `sys_Class`. Each class has a list of inherited classes and a list of derived classes. The `positionNumber` parameter indicates the order of base classes in multiple inheritance definitions.

3.2.5 Defining Operations

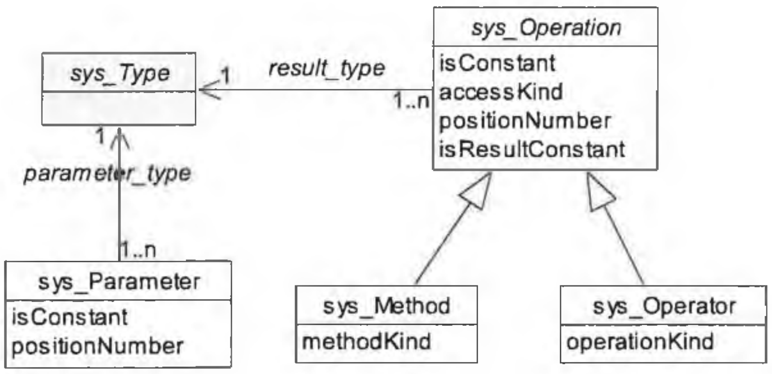


Figure 3.6: Metadata Definition of Operations.

Operations can be defined for both built-in types and classes. Operations specified for system types are part of the type definition and cannot be modified by the user, while operations on classes are user defined. The `sys_Operation` metaclass is an abstraction for all operations defined in the database and it is derived from the `sys_ScopedObject` metaclass. This metaclass corresponds to the `d_Operation` class in the ODMG metamodel with the difference that in our metamodel, `sys_Operation` is a generalisation for two types of operations: methods and operators defined by `sys_Method` and `sys_Operator` metaclasses respectively. Operators are not supported in the ODMG metamodel and thus new in this metamodel. An operator can be `unary` or `binary` as defined in `operatorKind` property, while `methodKind` property of the `sys_Method` indicates if method is prefixed as `static` or `virtual`.

Each operation can have a list of parameters and a return value. Parameters are specified by the `sys_Parameter` metaclass and can be of any system type. Classes cannot be passed by value as parameters or as return values of operations, and thus class references are used instead. The `positionNumber` attribute of the `sys_Parameter` metaclass indicates the relative position of the parameter within the parameter list. Operations can be `public`, `private` or `protected` as specified in the `accessKind` property. Each operation can be defined as constant (`isConstant` property), indicating that the internal state of the object cannot be changed. There are many issues concerning representation and invocation of database behaviour, but they are outside the scope of this research and are addressed elsewhere in the EGTV project [KR03].

3.2.6 Defining Schemas

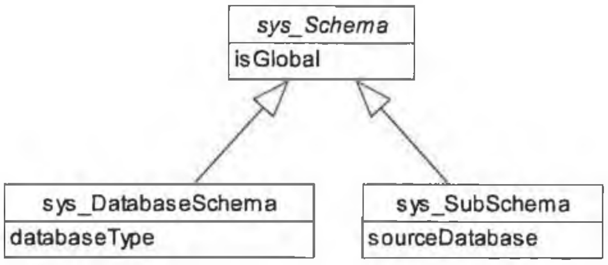


Figure 3.7: Metadata Definition of Schema.

A schema represents the top level container for classes and object views. The ODMG model implements a schema as an instance of the `d_Module` metaclass, while in our metamodel `sys_Schema`, derived from `sys_ScopedObject`, is an abstraction for database schema and view subschema metaclasses. An instance of `sys_DatabaseSchema` represents one database schema, defining a global scope for the database objects it contains. Objects that can be registered within a database schema are base and virtual classes.

3.2.7 Defining Views

Object views provide schema restructuring functionality for object-oriented database models. This feature is commonly used in federated database systems for the construction of different component and federated schemas. Object view support is not provided in the ODMG standard, but our metamodel defines extensions for representing view metadata. Views are commonly defined in a special view definition language and then transformed to the metamodel representation to facilitate runtime schema interrogation. Two types of object views can be represented in the EGTV metamodel: simple views and schema views.

Simple views. This is a concept where a view is represented as a single virtual class. The virtual class is defined as a stored query; its attributes are derived explicitly from the query definition; while the object extent is generated as a result of query execution. The query expression can be defined upon other base and virtual classes in the schema, thus providing the basic schema restructuring and integration features. Simple views are used in the EGTV architecture for construction of federated schemas as explained in chapter six. A virtual class is represented in the metamodel as an instance of the `sys_Class` metaclass, while the query expression is stored in the `virtualExtent` property of the `sys_Class`.

Schema views. A schema view consists of multiple interconnected virtual classes that define a *subschema*. Subschemas are represented in the `sys_SubSchema` metaclass of the EGTV metamodel, and one database schema can contain multiple subschemas. This representation is necessary as a view is always based on a subschema, and not on a single virtual class. Schema view support in this metamodel is designed to represent the view mechanism specified in [RKB01a]. Each object view, represented as an instance of `sys_SubSchema` metaclass, contains one or more virtual classes. A virtual class is constructed recursively from base classes or other virtual classes using restructuring operators specified in [RKB01a]. Operators define transformation rules applied to virtual class as it is constructed from the base or virtual classes defined at the level below. The `operatorType` property of the `sys_Class` specifies type of the operator applied to virtual class transformation.

Each virtual class (in both simple and schema views) can be based on one or two base or virtual classes at the level below. This class mapping is defined recursively until all virtual classes are resolved to base classes in the database schema. During this process, class definitions at one level map to corresponding classes at the level below, attributes map to attributes, relationships to relationships and inheritance to inheritance at the level below. The recursive structure of virtual classes is represented in the abstract metaclass `sys_MetaObject` from which all metamodel elements are derived. The `virtual_connector` relationship provides mapping to elements defined one virtual level below as illustrated in *Figure 3.2.* Each `sys_MetaObject` instance (class, attribute, relationship, inheritance, subschema) can have relationship to zero, one or two elements of the same type defined at lower level. The `virtualLevel` property indicates the virtual level at which the element is defined. The `virtualLevel` value of zero indicates base class, while classes and their components with the virtualLevel grater then zero are virtual.

3.2.8 Eliminated ODMG Metaclasses

The following metaclasses, defined in the ODMG metamodel specification are dropped from our metamodel.

`d_Constant_Type`: This class is dropped as constants can be defined only as attributes of user defined types or operation parameters. In both cases a constant value is denoted by `isConstant` parameter of `sys_Attribute` and `sys_Parameter` metaclasses and additional metaclass for constants representation is not required.

`d_Enumeration_Type`: The `d_Enumeration_Type` as defined in [Jor98] represent C-style enumerations. Object-relational databases does not have enumeration types, and this type is also not supported in the Java programming language. Enumeration types are not included because they are supported only in C and C++ and are not object data types.

`d_Exception`: Exceptions represent the internal segment of the behaviour implementation, and are not part of the operation signature. Our metamodel defines only metadata required for generic operation invocation such as operation names, parameters, return values and types.

`d_Structure_Type`: Contrary to user defined types, structures in object databases do not have object identification and cannot define operations. Structures are not included in the metamodel since nesting of the complex data types by value is not allowed. Also, passing a structure by value as a parameter or return value of operation is not supported in the EGTV model. The EGTV data model is discussed in chapter 5 where its features are explained.

3.2.9 A Meta-Metadata Level

Each metamodel describes the structure of the database schema at some level of abstraction. Our metamodel is specifically constructed to support multimedia metadata by recognising multimedia types as a special form of data type. The model in which the metamodel is specified and constructed is called the meta-metamodel. Metamodels for representation of specific database models (e.g. multimedia) can be easily defined in the meta-metamodel. Migration from the one metamodel structure to the another is accomplished by changing metamodel representation in the meta-metamodel. Our specification of meta-metamodel is illustrated in *Figure 3.8.* We recognise the meta-metamodel as beneficial to our system, because it allows us to specify new metamodels and to add new metaclasses to the existing ones. Furthermore, autonomous users are able to dynamically read metamodel structure and map metaclasses and properties to their locally defined data types.

The `m_Abstract` metaclass is an abstraction for any type of metamodel element. It defines `name` and `comment` attributes common for all entities in the meta-metamodel. The `m_Abstract` can be realised as element, attribute, association, generalisation or schema. The `m_Element` metaclass represents a general container element, and instances of the `m_Element` correspond to `sys_Class` metaclasses in the metamodel definition. Each `m_Element` can contain attributes and associations represented by the `m_Attribute` and `m_Association`. The `type` defines attribute type, while `isUnique` property specifies if attribute instances must be unique. Associations in the meta-metamodel can be unidirectional (`m_UniAssociation`) or bidirectional (`m_BiAssociation`). The `cardinality`, `isUnique` and `isOrdered` properties are defined for both association types, but only bidirectional associations have a `traversal` link to the inverse association element. Inheritance relationships between metadata elements are represented by the `m_Generalisation` metaclass, while the `m_Schema` is the root container for all elements

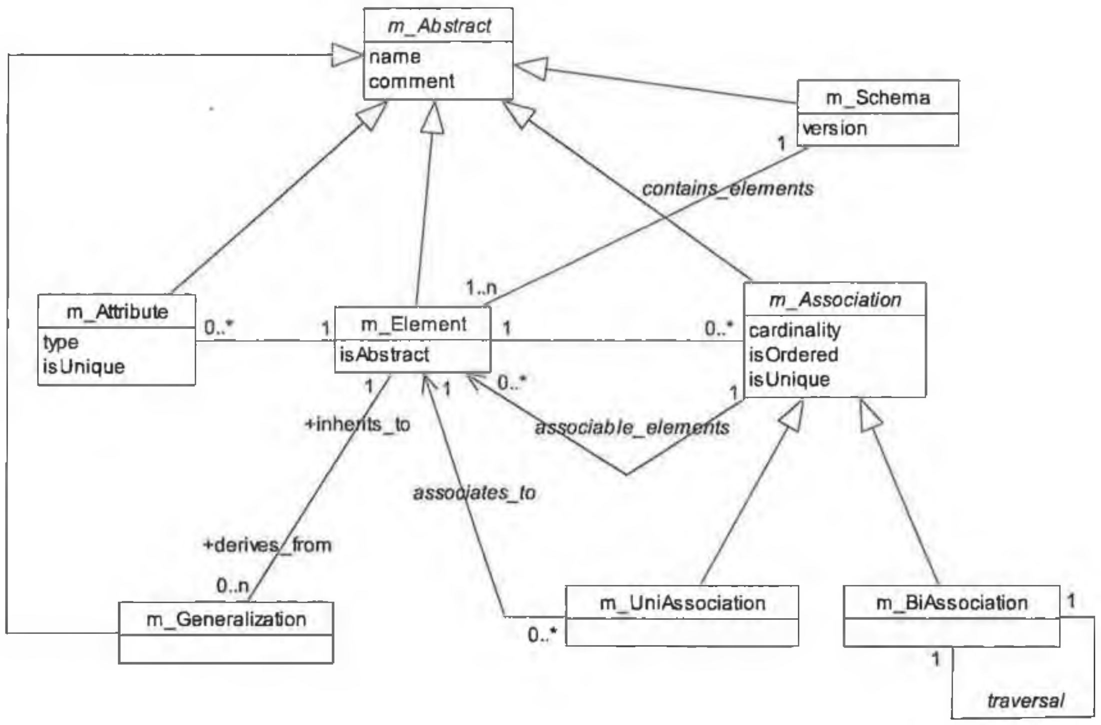


Figure 3.8: Meta-Metamodel Specification.

in the meta-metamodel. Each `m_Schema` instance corresponds to one metamodel schema defined in the meta-metamodel. The `associable_elements` relationship between `m_Association` and `m_Element` metaclasses defines association rules for metamodel instances of the meta-metamodel. The relationship specifies all subclasses to which an association defined between their superclasses, can be propagated. This feature enables definition of strict association rules that specify which subclasses are allowed to create an association link defined for their superclasses.

3.3 Metamodel Mappings

This section is aimed at providing a detailed specification of mappings from the EGTV metamodel [Beć02a] to object-oriented and object-relational schema repositories. This is required for representing EGTV multimedia schemas in different object-oriented and object-relational compliant databases.

Mappings are defined to transform schemas from the the EGTV metamodel to the object-relational and ODMG object-oriented metamodel representations [BR04a]. Mappings are generally straightforward except for some features of the EGTV metamodel not included in the ODMG and object-relational specifications such as multimedia types and object views. Therefore, the ODMG and object relational metamodels must be extended to support mappings to new EGTV features. For simplicity, we represent all ODMG and object-relational metamodel extensions either as a single ODMG class `d_Extension` or as an object-relational table `all_extensions`. The `d_Extension` class is a generic container that can map any EGTV metaclass property to the ODMG representation. It defines four attributes: `egtvClassName`, `propertyName`, `propertyValue` and `propertyType`. The `egtvClassName` attribute contains the name of the source EGTV metaclass, while the `propertyName` is the name of the property that is mapped to an extension. The `propertyValue` and `propertyType` attributes contain the actual value of mapped property and its data type (e.g. string or integer). An identical set of properties is defined for the `all_extensions` table. Without defining these extensions, the existing O-O and O-R metamodels would be incapable of supporting the full semantics of the EGTV metamodel. This is specially the case when representing metadata for multimedia types, operators and views. For a full description of the ODMG object-oriented standard, please refer to the [CB99], the object-relational SQL:1999 features are explained in the [GP99].

Mappings can be formally represented by the `map` function specified in *Definition 3.1*. This function transforms an EGTV metaclass to an equivalent O-O and O-R representation. Each EGTV metaclass is mapped to one or more ODMG classes, or to one or more O-R tables (O-R metamodels are implemented as tables and not object types). Metaclass properties are mapped to corresponding ODMG properties and O-R table columns. Mappings are defined in a simple mapping language and presented for each EGTV metaclass discussed in this section. The mapping language consists of a series of expressions, where each EGTV class, attribute, and relationship is assigned an equivalent O-O or O-R properties.

Definition 3.1 $\{\textit{ODMG Class}\}^{1..*} \leftarrow \textbf{map}(\textit{ EGTV Metaclass }) \mid$
$\{\textit{O-R table}\}^{1..*} \leftarrow \textbf{map}(\textit{ EGTV Metaclass })$

3.3.1 EGTV To ODMG Mapping

The EGTV metamodel is based on the ODMG metamodel specification, so both metamodels share a similar object-oriented platform. The mapping is relatively simple for metaclasses that have similar definition in both metamodels, but is more complex for multimedia types and object views. This section explains individual mappings, while major EGTV metaclasses and their counterparts in the ODMG metamodel are illustrated in *Table 3.2*. In all of these descriptions we provide a formal mapping language example to demonstrate.

Schema Mapping

The `sys_DatabaseSchema` metaclass defines database schema properties including a root naming scope for all schema elements. It is mapped to the `d_Module` metaclass

EGTV metaclass	ODMG metaclass
sys_DatabaseSchema	d_Module
sys_Class	d_Class
sys_Attribute	d_Attribute
sys_Inheritance	d_Inheritance
sys_Relationship	d_Relationship
sys_Method, sys_Operator	d_Operation
sys_Parameter	d_Parameter
sys_PrimitiveType	d_Primitive_Type
sys_CollectionType	d_Collection_Type
sys_MediaType	d_Class
sys_SubSchema	Extended ODMG

Table 3.2: The EGTV to ODMG Mapping.

which represents the equivalent class in the ODMG metamodel. For each database schema specified in the `sys_DatabaseSchema`, the mapping creates one `d_Module` instance in the ODMG metamodel. The schema mapping is illustrated in *Example 3.1.*

```
map sys_DatabaseSchema := d_Module, d_Extension
{
  attribute:
    isGlobal := d_Extension.isGlobal
    databaseType := d_Extension.databaseType
  relationship:
    containedIn := d_Module.definedIn
    containedObjects := d_Module.defines
}
```

Example 3.1: ODMG Schema Mapping.

The `map` function defines the mapping for the `sys_DatabaseSchema` to `d_Module` and `d_Extension` ODMG metaclasses. Attributes of the same name and type in both metamodels (`name` and `comment`) are omitted from this example as their mapping is implicit. The `isGlobal` and `databaseType` attributes are unique to the EGTV metamodel, so they are mapped to the `d_Module` extensions. The containment relationships between the database schema and its classes (`containedIn` and `containedObjects`) are mapped to the equivalent `d_Module` relationships `definedIn` and `defines`.

Class Mapping

Both metamodels represent database classes with a single metaclass. This mapping is illustrated in *Example* 3.2.

The `map` function defines the mapping between the EGTV `sys_Class` and ODMG `d_-Class` metaclasses. Properties of `sys_Class` for which equivalents cannot be found in

```
map sys_Class := d_Class, d_Extension
{
  attribute:
    isAbstract := d_Extension.isAbstract
  relationship:
    containedIn := d_Class.definedIn
    containedObjects := d_Class.defines
    inheritsTo := d_Class.inherits
    derivesFrom := d_Class.derives
}
```

Example 3.2: ODMG Class Mapping.

the d_Class are mapped to the ODMG extensions (d_Extension). Attributes having the same name and type in the both metamodels (name and comment) are implicitly mapped. The isAbstract attribute is new to EGTV metamodel, and maps to the same attribute in the d_Extension class. The containedObjects EGTV relationship represents attributes, operations and relationships contained within the instance of the sys_Cass, and maps to the definedIn relationship of the d_Class. The class containment in the database schema is represented with the containedIn relationship which maps to the defines relationship of the d_Class. The inheritance relationships inheritsTo and derivesFrom map to the d_Class inheritance relationship inherits and derives. The EGTV sys_Class defines additional attributes virtualLevel, operatorType, virtualExtent and relationship virtualConnector for representing virtual classes. These attributes are mapped only to the the ODMG extensions for object views as found in [RKB01a].

Attribute Mapping

Each attribute defined in the EGTV sys_Attribute class is mapped to one instance of ODMG d_Attribute metaclass. This is illustrated in *Example 3.3.*

```
map sys_Attribute := d_Attribute, d_Extension
{
  attribute:
    accessKind := d_Extension.accessKind
    isConstant := d_Attribute.is_read_only
    isStatic := d_Extension.isStatic
  relationship:
    containedIn := d_Attribute.definedIn
    attributeType := d_Attribute.type
}
```

Example 3.3: ODMG Attribute Mapping.

The map function maps the sys_Attribute EGTV metaclass to the d_Attribute and d_Extension classes. The name, comment, cardinality and traversal properties of sys_Attribute are not specified in *Example 3.3* as these mappings are implicit in this function. The isConstant attribute of the sys_Attribute EGTV metaclass maps to the is_read_only ODMG counterpart, while the attributeType EGTV relationship is mapped to the type relationship in the ODMG model. The accessKind and isStatic attributes do not have counterparts in the d_Attribute, and are mapped to the ODMG extension class.

Inheritance Mapping

Inheritance mapping is from the sys_Inheritance metaclass to the ODMG defined class d_Inheritance. All attributes of sys_Inheritance are mapped to the d_Extension metaclass since they are not defined in the ODMG metamodel. The inheritsTo and derivesFrom relationships between sys_Inheritance and sys_Class are mapped to the same relationships (inherits and derives) between d_Inheritance and d_Class. This is illustrated in *Example 3.4*.

```
map sys_Inheritance := d_Inheritance, d_Extension
{
  attribute:
    name := d_Extension.name
    comment := d_Extension.comment
    isVirtual := d_Extension.isVirtual
    positionNumber := d_Extension.positionNumber
  relationship:
    inheritsTo := d_Inheritance.inherits
    derivesFrom := d_Inheritance.derives
}
```

Example 3.4: ODMG Inheritance Mapping.

Relationship Mapping

The EGTV sys_Relationship metaclass maps to the ODMG metaclass d_Relationship. Attributes in the sys_Relationship which do not have an equivalents in d_Relationship, are mapped to the ODMG extensions. The relationship mapping is illustrated in *Example 3.5.*

The map function maps the sys_Relationship EGTV metaclass to the d_Relationship and d_Extension classes. The name, comment, cardinality and traversal properties of sys_Relationship are not specified in *Example* 3.5 as these mappings are implicit in this function. The isUnique and isOrdered attributes do not have counterparts in the d_Relationship, and are mapped to the ODMG extension class.

```
map sys_Relationship := d_Relationship, d_Extension
{
  attribute:
    cardinality:= d_Relationship.cardinality
    accessKind := d_Relationship.accessKind
    isUnique := d_Extension.isUnique
    isOrdered := d_Extension.isOrdered
  relationship:
    containedIn := d_Relationship.definedIn
    traversal := d_Relationship.traversal
}
```

Example 3.5: ODMG Relationship Mapping.

Operation Signature Mapping

Operations in the EGTV metamodel can be defined as methods or operators, but the ODMG specification supports only methods and not operators. This means that EGTV metaclass `sys_Method` maps directly to the ODMG `d_Operation`, while operator mapping must be further defined. Operators defined in the EGTV `sys_Operator` metaclass can be mapped to the ODMG `d_Operation` as methods with special system defined names. The rule is that operator name starts with the keyword `operator` followed by the operator type symbol. For example operator + is named `operator+` and mapped in the `d_Operation` metaclass. This and similar operator names are system reserved and cannot be used for regular methods. Methods and operator parameters represented in the EGTV metaclass `sys_Parameter` are mapped to the equivalent ODMG `d_Parameter` metaclass. The behaviour mapping is not specified further here as it forms part of separate research [KR03] into the specification of the EGTV model itself.

Data Type Mapping

Primitive types and collection types are represented in EGTV in the same way as in the ODMG metamodel. The `sys_PrimitiveType` is mapped to the `d_Primitive_Type` and `sys_CollectionType` is mapped to the `d_Collection_Type`. This mapping is defined in *Table 3.4*. However, since the ODMG standard does not define a data type for blobs, it was extended with the `BObject` (binary large object) class to which EGTV `Blob` type is then is mapped. In other words, the metamodel is extended with a user-defined class to encapsulate raw multimedia data. Multimedia types were introduced in the EGTV metamodel and do not have equivalent representations in the ODMG model. Furthermore, multimedia types can define behaviour operations, which is not supported for types in ODMG metamodel. The solution is to map the EGTV `sys_MediaType` to `d_Class` in the ODMG specification. This means that EGTV multimedia data types are represented in the ODMG metamodel as system defined classes. For example the `jpeg`

EGTV Type	ODMG Type	O-R Type
Integer	long	integer
Float	float	number
Double	double	number
String	string	varchar
Date	timestamp	date
Boolean	boolean	boolean
Blob	BObject	blob
Set	set	nested table
Bag	Bag	nested table
List	List	array
Map	Dictionary	Map

Table 3.4: EGTV Type Mappings.

multimedia type can be represented as `jpgImage` class in the `d_Class` metaclass. The names of the multimedia classes are system reserved and users cannot define classes of the same name. Since classes in the `d_Class` can have operations, the multimedia type operations are mapped to operations of the ODMG classes, however due to limitations of the ODMG, only interface mapping is provided, and not actual behaviour.

Object View Mapping

The ODMG metamodel does not provide support for object views, so they can be mapped only to the extended ODMG metamodel specified in [RKB01b]. This metamodel extends the ODMG specification with the set of metaclasses for view and virtual class representation. These classes are identified by the `v_` prefix and they reassemble the same structure as the original (`d_`) ODMG metaclasses. Simple EGTV views are mapped directly from the `sys_Class` to the `v_Class` in the extended ODMG model. The schema view definitions represented in the `sys_SubSchema` EGTV metaclass are mapped to the `v_SubSchema` metaclass in the extended ODMG. The virtual class definitions are mapped to the `v_Class`, while their attributes, relationships and inheritance are mapped to the `v_Attribute`, `v_Relationship` and `v_Inheritance` respectively. Multilevel structure of the virtual class definitions represented in the EGTV metamodel with the `virtual_connector` recursive relationship is mapped to the `base` and `virtual` connector relationships in the extended ODMG. The end result is that EGTV views can be mapped to the object-oriented view system specified in [RKB01a].

3.3.2 EGTV To Object-Relational Mapping

Object-relational databases do not use the SQL:1999 metamodel at present, so in this discussion we will regard the Oracle 9i metamodel as a standard as it has the most advanced features. The object-relational schema repository is represented as a set of relational tables which store both relational and object-relational metadata [WR03]. The EGTV metaclasses are mapped to the Oracle 9i schema repository tables as illustrated in *Table 3.6.* The attributes of the EGTV metaclasses are mapped to the table columns, while relationships are mapped to referential integrity constraints between schema repository tables.

EGTV Metaclass	O-R Table
sys_DatabaseSchema	all_users
sys_Class	all_types, all_types
sys_Attribute	all_type_attrs, all_tab_columns
sys_Inheritance	all_object_types
sys_Relationship	all_type_attrs
sys_Method, sys_Operator	all_type_methods
sys_Parameter	all_type_attrs
sys_PrimitiveType	all_types
sys_CollectionType	all_types
sys_MediaType	all_object_types
sys_SubSchema	Extended O-R

Table 3.6: EGTV to Object-Relational Mapping.

Schema Mapping

In the object-relational model, each database user owns one database schema which contains object types, tables and other model elements. The object relational schema repository represents database schemas in the `all_users` system table to which the EGTV `sys_DatabaseSchema` metaclass is mapped. The mapping represents each schema defined in the `sys_DatabaseSchema` as one tuple of the `all_users` table. Each schema defined in the `all_users` table provide root naming scope for all types and tables it contains. This is illustrated in *Example* 3.6.

The map function maps the `sys_DatabaseSchema` EGTV metaclass to the O-R table `all_users`. The name attribute is mapped to the username column of the `all_users` table. The other attributes of the `sys_DatabaseSchema` do not have counterparts in the `all_users` table, and are mapped to the O-R metamodel extensions (`all_-extensions`). The relationship `containedObjects` between `sys_DatabaseSchema` and `sys_Class` metaclasses is mapped to the referential integrity constraint between `all_users` and `all_types` tables.

```
map sys_DatabaseSchema := all_users, all_extensions
{
  attribute:
    name := all_users.username
    comment := all_extensions.comment
    isGlobal := all_extensions.isGlobal
    databaseType := all_extensions.databaseType
  relationship:
    containedObjects := all_users.username ref_to
                        all_types.owner
}
```

Example 3.6: O-R Schema Mapping.

Class Mapping

Classes in the object-relational model are represented as user defined types (UDT) and instantiated in the form of object tables [Ora02b]. For this reason, the `sys_Class` from the EGTV metamodel is mapped to two O-R schema repository tables: `all_types` and `all_object_tables`. The `all_types` table contains all user defined types in the database schema, while the `all_object_tables` represent object tables that instantiate these UDTs. The class mapping is illustrated in *Example 3.7.*

```
map sys_Class := all_types, all_object_tables,
                            all_extensions
{
  attribute:
    name  := all_types.type_name,
             all_object_tables.table_name
    comment := all_extensions.comment
    isAbstract := all_types.instantiable
  relationship:
    containedIn := all_users.username ref_to
                   all_types.owner
    containedObjects := all_types.type_name ref_to
                           all_type_attrs.type_name,
                        all_types.type_name ref_to
                           all_type_methods.type_name
    derivesFrom := all_types.supertype_name
    inheritsTo := all_extensions.inheritsTo
 }
```

Example 3.7: O-R Class Mapping.

The object-relational mapping requires a map function which transforms the `sys_Class` metaclass to `all_object_tables` and `all_types` O-R tables. The attribute name is mapped to name columns of both `all_types` and `all_object_tables` tables, while the comment attribute (not existing in the O-R schema repository) is mapped to the all_extensions table. The isAbstract attribute maps to the instantiable column of the all_types table and containedIn relationship between class and its containing schema is mapped to referential integrity constraint between all_types and all_users tables. Attributes and relationships defined within the class (containedObjects relationship) are mapped to referential integrity constraint between all_types and all_type_attrs tables, while operations are mapped to the constraint between all_types and all_type_methods tables. Inheritance in the sys_Class is represented with derivesFrom and inheritsTo relationships. The first relationship is mapped to the supertype_name column of the all_types table, while the second maps to the O-R schema repository extensions.

Attribute Mapping

Class attributes defined in the EGTV metaclass sys_Attribute are mapped to two object-relational tables. The all_type_attrs table defines UDT attributes, while all_tab_columns stores their object table instantiations. The containment relationship between attributes and user defined types is mapped to the referential integrity constraint between all_type_attrs and all_types schema repository tables. The sys_attribute properties isConstant, isStatic and accessKind does not have object-relational counterparts, so they are mapped to the O-R schema repository extensions. The attribute mapping is illustrated in *Example* 3.8.

```
map sys_Attribute := all_type_attrs, all_tab_columns,
                                     all_extensions
{
  attribute:
    name  := all_type_attrs.type_name,
             all_tab_columns.column_name
    comment := all_extensions.comment
    accessKind := all_extensions.accessKind
    isConstant := all_extensions.isConstant
    isStatic := all_extensions.isStatic
  relationship:
    containedIn := all_type_attrs.type_name ref_to
                       all_object_types.type_name,
    attributeType := all_type_attrs.attr_type_name
 }
```

Example 3.8: O-R Attribute Mapping.

Inheritance Mapping

Inheritance in the object-relational metamodel is represented as a relationship between a subclass and a superclass instances of the `all_types` schema repository table. The `sys_Inheritance` EGTV metaclass maps to the inheritance property `supertype_name` of this table. The inheritance mapping is illustrated in *Example* 3.9.

```
map sys_Inheritance := all_types,all_extensions
{
  attribute:
    name   := all_extensions.name
    comment := all_extensions.comment
    isVirtual := all_extensions.isVirtual
    positionNumber := all_extensions.positionNumber
  relationship:
    inheritsTo := all_extensions.inheritsTo
    derivesFrom := all_types.supertype_name
 }
```

Example 3.9: O-R Inheritance Mapping.

The `name`, `comment`, `isVirtual` and `positionNumber` attributes of the `sys_Inheritance` cannot be mapped directly to the O-R schema repository tables. Instead, the mapping to the O-R metamodel extensions is provided. The `derivesFrom` relationship of the `sys_Inheritance` maps to the `supertype_name` column of the `all_types` table, while the `inheritsTo` relationship is mapped to the `all_extensions` table.

Relationship Mapping

Relationships are represented as reference (REF) attributes and nested tables (collections) of REF attribute types [Ora02b]. The REF attribute type represents one side of the relationship, while the nested table of REF types is used to describe many side of the relationship. The object-relational mapping translates relationships from the EGTV `sys_Relationship` metaclass to the `all_type_attr` table. Since the class attributes are also mapped to the same table, the property `attr_type` of this table is used to distinguish relationships (REF attributes) from the non-reference attributes. This is illustrated in *Example 3.10.*

Operation Signature Mapping

Behaviour methods defined for user defined types are represented in the object-relational schema repository in the table `all_type_methods` to which EGTV `sys_Method` class is mapped. Type operators are not supported in object-relational schema repository, so they are mapped as methods with special system defined names which cannot be used for

```
map sys_Relationship := all_type_attrs, all_tab_columns,
                            all_extensions
{
  attribute:
    name   := all_type_attrs.type_name,
              all_tab_columns.column_name
    comment := all_extensions.comment
    accessKind := all_extensions.accessKind
    isUnique := d_Extension.isUnique
    isOrdered := d_Extension.isOrdered
    'REF' := all_type_attrs.attr_type_mod

  relationship:
    containedIn := all_type_attrs.type_name ref_to
                           all_object_types.type_name,
    traversal := all_extensions.traversal
 }
```

Example 3.10: O-R Relationship Mapping.

other methods. For example the operator = is represented in the `all_type_methods` table as a method named `operator=`. Operation parameters represented in the `sys_-Parameter` EGTV metaclass are mapped to `all_type_attrs` O-R metatable.

Data Type Mapping

Primitive and collection data types are mapped from the `sys_PrimitiveType` and `sys_CollectionType` EGTV metaclasses to the `all_types` table in the object-relational schema repository. The mappings are defined in *Table 3.4*. Multimedia types and the `Map` type are not defined in the O-R schema repository, so they are represented as a special user defined types in the `all_object_types` table. For example, the `mpeg` multimedia data type defined in the `sys_MultimediaType` EGTV metaclass is mapped to the `mpegVideo` user defined type in the `all_object_types` table. Operations defined for multimedia types are mapped to UDT operations. Regular UDTs cannot have the same name as these UDTs for multimedia representation, since these names are reserved for multimedia types representation.

Object View Mapping

The object-relational metamodel supports only a simplified form of views (as virtual tables). For this reason the EGTV view subschemas must also be mapped to the extended object-relational metamodel. These metamodel extensions should include a set of tables which supplement the existing object-relational schema repository with view metadata.

EGTV Metaclass	Virtual Table	Base Table
sys_SubSchema	vie_sub_schema	-
sys_Class	vie_types	all_types
sys_Attribute	vie_type_attr	all_type_attr
sys_Relationship	vie_type_attr	all_type_attr
sys_Inheritance	vie_types	all_types
sys_Method	vie_type_methods	all_type_methods
sys_Operator	vie_type_methods	all_type_methods

Table 3.8: EGTV to Extended Object-Relational Mapping.

Extension tables describe view subschemas, virtual object types, virtual attributes, relationship and inheritance elements. The extension tables listed in *Table 3.8* should preserve the same structure as the original object-relational schema repository tables, and are distinguished by the `vie_` prefix.

The EGTV metamodel mapping is defined so that the `sys_SubSchema` metaclass is mapped to the `vie_sub_schema` extended schema repository table. The virtual classes from the `sys_Class` are mapped to the to the `vie_types` table, while their operations are represented in the `vie_type_methods table`. Virtual attributes and relationships are both mapped to the `vie_type_attr` table as a regular attributes or REF attributes.

3.4 Conclusions

In this chapter we described our approach to designing and implementing a metamodel for multimedia databases. The metamodel defines a set of metaclasses for metadata storage in our multimedia database federation. It is based on the ODMG metamodel, but provides a more simplified design, includes multimedia data types and extended support for object views. We have developed and implemented a standard metamodel interface to object-based multimedia systems, while the contribution is in the fact that no federated database research project has published a single metamodel capable of representing both, base schemas and view subschemas. In [RKB01a] they provide a single interface to both base and view schemas, but view metadata is represented in a special metamodel segment, separate from base schemas. The detailed mappings were defined for each EGTV metamodel element and represented in the form of simple mapping language. This provides us with the ability to create EGTV multimedia databases and map their schemas to existing object-oriented and object-relational schema repositories. By publishing the metamodel, it is possible to create an architecture which is both open and extensible. This has proved to be beneficial when creating federations of multimedia databases, which is the focus of our research. Metamodel has a significant role in construction of the EGTV multimedia federation. Firstly, it provides a platform independent representation for multimedia schemas defined in local O-O and O-R databases. Secondly, it facilitates the construction of federated schemas in a form of views that integrate and restructure multiple local schemas. Thirdly, metadata can be defined to support multimedia data types and operations they provide. Finally, the EGTV metamodel provides infrastructure for dynamic (ad-hoc) interrogation of both base and virtual schemas. This feature is crucial for supporting query execution, as syntactic and semantic metadata validation of query expressions must be performed in runtime.

The metamodel and its database mappings are required prerequisites for specification of a schema definition language and a query language. Schema definition language provides a platform independent specification of EGTV multimedia schemas in the top-down approach. In chapter 4 we discuss the ODL_x language for EGTV schema specification. Our query language is also based on the EGTV metamodel, as it requires schema repository metadata for dynamic (run-time) evaluation of queries. This language is capable of querying schemas at both canonical and federated layers of our architecture and is discussed in chapter 5.

Chapter 4

Schema Definition

When storing data in heterogeneous databases, it is useful to have a single schema definition language. Such a language enables the definition of database schemas in a platform independent format, that can be subsequently mapped to any database representation. This is especially emphasised in federated database systems, where participating databases can be both heterogeneous and distributed. As XML represents a standard for encoding and distributing data across various platforms and the Internet, a language based upon XML has been developed in this research. The ODL$_x$ (Object Definition Language XML) language specifies a set of XML-based structures for defining object-oriented and object-relational database schemas in the database independent format. The XML Schema is used to facilitate formal definition of ODL$_x$ grammar and to provide blueprint for syntactic validation of user-defined ODL$_x$ schema files. The language is fully integrated with the EGTV metamodel through which ODL$_x$ schemas can be mapped to O-O and O-R databases.

The chapter is structured as follows: in §4.1 different approaches to database schema creation are discussed and our solution is motivated; in §4.2 an overview of the ODL$_{\mathrm{x}}$ language is given and explained through a series of examples; in §4.3 mappings of ODL$_{\mathrm{x}}$ language elements to the EGTV metamodel are elaborated; and finally in §4.4 some conclusions are given. The complete XML Schema definition of the ODL$_x$ language is given in *Appendix B*, while the full reference of all ODL$_{\mathrm{x}}$ language elements and their mappings is provided in *Appendix C.*

4.1 Introduction

A database schema is created from definitions commonly specified in the form of a Data Definition Language (DDL). A Data Definition Language provides a set of constructs that define database schema elements (types, tables, classes, etc.) and models the relationships between them. A schema is usually stored in a text file, and interpreted by the *schema* *definer* process which creates the database schema and populates metadata in some schema repository.

Relational databases commonly use SQL DDL [MS92] as a schema definition language. This language specifies a set of SQL `CREATE` statements for defining tables, integrity constraints and other elements of the relational database model. The language is standardised and generally portable across all relational database platforms. However, it cannot define schemas which include objects and concepts such as inheritance and relationships.

SQL:1999 specifies a Data Definition Language [GP99] for defining O-R schemas as an extension to the existing relational SQL. The standard extends SQL `CREATE` statement with new constructs for defining O-R schema elements. These extensions include object types and tables, inheritance and association relationships, and collection types. In addition to these, behaviour methods can be defined as part of an object-relational type. Although this specification is complete, it has not been widely adopted, and majority of O-R databases are still not fully compliant with the standard, most specifically when defining collection types, table inheritance, and user-defined methods. An O-R view is defined as a stored SQL query, the result of which can be interpreted as an extent of a User-Defined Type (UDT). However, updatability of these views is limited, and the semantics of virtual object identifiability remains unclear.

The ODMG standard [CB99] specifies the Object Definition Language (ODL) as a data definition language for object schema specification. This language supports the definition of classes and other entities that can be created in an ODMG schema. However, it does not provide constructs for defining views (virtual classes), and global schemas. ODL is based on the CORBA IDL language [OH98] (Interface Definition Language), so it inherits some of its limitations. Specifically, method overloading and static methods are not defined in the ODL syntax and thus, their semantics are unknown.

Since ODMG ODL and SQL:1999 DDL are two different and incompatible standards for database schema specification, our contribution is to specify and deploy a more general object schema definition language. In addition, we chose to use the emerging standard for data exchange, XML. This provides an added benefit in that our schema specifications are fully portable across different platforms. Furthermore, we use the XML Schema standard to provide a formal definition of the ODL$_x$ language grammar as this has several advantages over commonly used BNF notation. Firstly, in addition to unambiguous syntax definition of ODL$_x$, the XML Schema can describe relationships between different ODL$_x$ language elements and specify constraints on their values. Secondly, XML Schema has built-in support for data types and can enforce strong typing of ODL$_x$ database schemas. Finally, the XML Schema is a fully standardised and platform independent technology, hence each user-defined ODL$_x$ schema file can be validated for correctness using standard XML parsers. The ODL$_x$ language provides a schema definition syntax for both ODMG and O-R databases used in our architecture. The language is completely XML-based and supports the definition of virtual classes (simple views), multimedia data types, and global schemas.

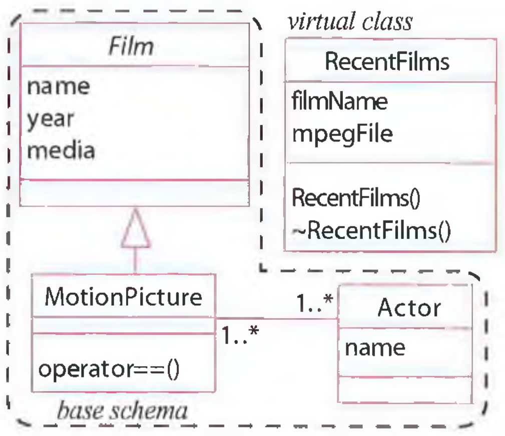


Figure 4.1: Film Archive Schema.

4.2 The Structure of ODL$_x$

ODL$_x$ is the definition language for specifying schemas in the EGTV model [KBR03]. This model is object-oriented, and capable of representing all features of both ODMG and O-R database models. The EGTV model introduces classes consisting of attributes, relationships and operations. Relationships in the EGTV model are bi-directional. Each operation can be either an operator or a method, differing in the invocation style. This is explained fully in [Kam04]. In this model, simple views can be defined as stored queries and represented as virtual classes. The role of a virtual class is to restructure and integrate database schemas, and is used in the construction of global schemas.

The ODL$_x$ is fully XML-based and structured as a hierarchy of *elements*. The top-level element is a database schema which contains multiple classes and virtual classes. The validity of each ODL$_x$ database schema is verified against its XML Schema [Wor01] specification. This provides an unambiguous definition of the ODL$_x$ syntax and defines consistency and referential integrity rules for ODL$_x$ database schemas.

Usage of the ODL$_x$ language is explained using a Film Archive subschema illustrated in *Figure 4.1.* This is an extract of a larger Multimedia Archive Schema defined in the EGTV project to store video recordings. The subschema consists of three classes: `Film`, `MotionPicture` and `Actor` which are part of an existing base schema. The `Film` is an abstract class from which a `MotionPicture` genre is derived to represent a special subcategory of films. The `MotionPicture` defines an `1-n` relationship to the `Actor` class which represents actors appearing in the motion picture. This relationship is bidirectional. The equivalent ODL$_x$ definition is provided in *example 4.1.* Using the base schema, a single virtual class `RecentFilms` is defined as illustrated in *example 4.2.* This virtual class selects only those motion pictures released after the year 2000. This example use the CDATA XML tag to encapsulate ODL$_x$ content non-compliant with XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="FilmArchive" databaseType="OR">
  <class name="Film" abstract="true">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="year">
      <primitiveType name="date"/>
    </attribute>
    <attribute name="media">
      <mediaType name="mpeg"/>
    </attribute>
  </class>
  <class name="MotionPicture">
    <inheritance name="MotionPictureGen" inheritsFrom="Film"/>
    <relationship name="MotionPictureRef" traversal="ActorRef"
                  cardinality="many"/>
    <operator name="==" constant="true" operatorKind="binary">
      <returnVal constant="true">
        <primitiveType name="bool"/>
      </returnVal>
      <parameter name="inClass" constant="true">
        <classType name="MotionPicture"/>
      </parameter>
    </operator>
  </class>
  <class name="Actor">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <relationship name="ActorRef" traversal="MotionPictureRef"
                  cardinality="many"/>
  </class>
</dbSchema>
```

Example 4.1: An ODL$_x$ definition of the Film Archive schema.

4.2.1 Class Definition

The ODL$_x$ `class` element represents a class in the database schema. The Film Archive schema in *example 4.1* defines three classes: `Film`, `MotionPicture`, and `Actor`, where the `Film` class is defined as an *abstract* class. Class attributes are defined as `attribute` subelements. Attribute definitions have a name and type properties, with attribute types defined as subelements of the `attribute` element. The `Film` class contains three attributes: `name`, `year` and `media` which are defined as `string` and `date` primitive types, and `mpeg` multimedia type respectively.

In *example 4.1* the class `MotionPicture` derives from the class `Film` by defining the `in-`

heritance element MotionPictureGen inside the specification of MotionPicture class. The inheritsFrom attribute of the inheritance element specifies the name of the superclass. In our example this is the Film class.

The bidirectional relationships of the EGTV model are represented as two relationship elements where each element defines one side of the relationship. *Example 4.1* specifies a many-to-many relationship between classes MotionPicture and Actor by defining relationship elements in both of these classes. The traversal attribute specifies the other side of the bidirectional relationship, and its value must correspond to the name attribute of the inverse relationship defined in the second class. This is enforced in the XML Schema using *key* and *keyref* constraints. The cardinality attribute defines the cardinality of the relationship and takes a value of one or many.

The behaviour of the class is declared using operator or method elements. In either case, the operation name must be specified together with mutability (attribute constant), list of parameters (subelements parameter) and the type of the return value (subelement returnVal). For each parameter, the type and name must be provided. The syntax of the definition is similar to the one used for class attributes. In our example, the operator== (testing for equality) is declared for class MotionPicture, and it compares two MotionPicture objects. Definition of behaviour (programming language code), its storage and processing can be found in [Kam04].

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="FilmArchive" databaseType="OR">
  <virtualClass name="RecentFilms">
    <extent><![CDATA[
        select name as filmName, media as mpegFile
        from MotionPicture
        where year > 2000;]]>
    </extent>
    <method name="RecentFilms" accessKind="public">
      <parameter name="filmName" constant=true>
        <primitiveType name="string"/>
      </parameter>
      <parameter name="recordingDate" constant="true">
        <primitiveType name="date"/>
      </parameter>
    </method>
    <method name="~RecentFilms" accessKind="public">
    </method>
  </virtualClass>
</dbSchema>
```

Example 4.2: An ODL$_x$ definition of the RecentFilms virtual class.

4.2.2 Virtual Class Definition

The `virtualClass` element facilitates the definition of a virtual class (simple view) as a stored EQL (EGTV Query Language) query [KBR03]. A detailed description of the EQL language is provided in next chapter. Each view definition is evaluated to one virtual EGTV class and represented in the EGTV Schema Repository. The attributes of this virtual class are derived implicitly from the EQL query definition, but its methods and operators are specified using the same syntax as for classes. *Example 4.2* defines a virtual class `RecentFilms` in the `FilmRepository` database schema. The `extent`[1] subelement defines a query from which the virtual class extent is generated. In our example, the extent is defined as a query that selects "all films made after the year 2000". The attribute list is specified in the `select` clause of the query and attribute types are read from the *Schema Repository*.

While updatability of virtual objects is provided by the EGTV model, operations play a crucial role in removing ambiguities when inserting new objects. For example, if the view contains a join, it is unclear how new objects are created, or existing ones deleted. Furthermore, a new object of a virtual class cannot be created if the virtual class hides some attributes of the underlining base class. The only way to unambiguously define semantics of create and delete operations on virtual classes is to specify constructor and destructor methods as a part of the view definition. Updates can also be redefined by overloading the assignment operator. Behaviour representation and processing is out of scope of this research, and a full discussion is provided in [Kam04]. In our example, the constructor of the virtual class `RecentFilms` is declared. This method receives two parameters: `filmName` and `recordingDate`, which are used to create new `MotionPicture` objects and set the values of its attributes. The `mpegFile` attribute is left as `null`, to be updated later. In a similar fashion, the destructor can be used to maintain integrity when objects are deleted. The destructor deletes the source object the virtual object was based upon, which effectively deletes the object from the database. Other operations can also be specified in a virtual class.

4.2.3 Import Class Definition

A virtual class imported into the global schema from a canonical layer database is specified using the `importClass` subelement of the `virtualClass` element. This ODL$_x$ element is required to unambiguously define source class as each imported class acts as a proxy for the original class in the canonical schema. Thus, the `importClass` element is used for defining global schemas in the EGTV database federation. Other virtual classes can be constructed upon classes imported from multiple local database nodes, thus facilitating global schema integration and restructuring. *Example* 4.3 illustrates the definition of

[1]The CDATA is an XML keyword which denotes text not to be interpreted by XML parsers.

the LocalRecentFilms virtual classes imported into the GlobalFilms global schema. Attributes of the importClass ODL$_x$ element provide information required to unambiguously locate original virtual class within the database federation. These include class name and identification of the schema and database where original class is located. Classes successfully imported into a global schema can be instantiated, queried and updated by global clients.

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="GlobalFilms" databaseType="OO">
  <virtualClass name="LocalRecentFilms"
    <importClass database="VideoRepository"
                 schema="FilmArchive" name="RecentFilms/>
  </virtualClass>
</dbSchema>
```

Example 4.3: An ODL$_x$ definition of the RecentFilms import class.

4.3 The ODL$_x$ to Metamodel Mappings

In this section we discuss mappings of the ODL$_x$ encoded database schemas to the EGTV metamodel representation. Metamodel mappings are required as all EGTV database schemas must be represented in the schema repository to allow for dynamic querying and global schema integration. Therefore, a formal mapping language was developed to transform ODL$_x$ elements and their attributes to metaclasses and properties in the EGTV metamodel. However, at this point we defer a discussion of formal mapping language specification. Instead, we introduce an ODL$_x$ to metamodel mapping process through a set of examples based upon the simple Film Archive Schema illustrated in *figure 4.1*. The full reference of all ODL$_x$ language elements and their metamodel mappings is provided in *Appendix C*.

4.3.1 Base Schema Mapping

Mapping of base ODL$_x$ schemas is a generally straightforward process where each ODL$_x$ element (i.e. dbSchema, class, attribute) is mapped to a single metaclass, while element attributes are mapped to metaclass attributes. This is illustrated in *figure 4.2* where mappings of the Film Archive schema (defined in *Example* 4.1) are represented as metaobjects in the UML collaboration diagram. The dbSchema ODL$_x$ element is mapped to the FilmArchive instance of the sys_DatabaseSchema metaclass, while each class element (Film, MotionPicture, Actor) defined in the Film Archive ODL$_x$schema is mapped to an instance of the sys_Class metaclass. The contained_in relationship defined between sys_Class and sys_DatabaseSchema metaclasses is instantiated to a set of object links that represent containment of class metaobjects within a schema

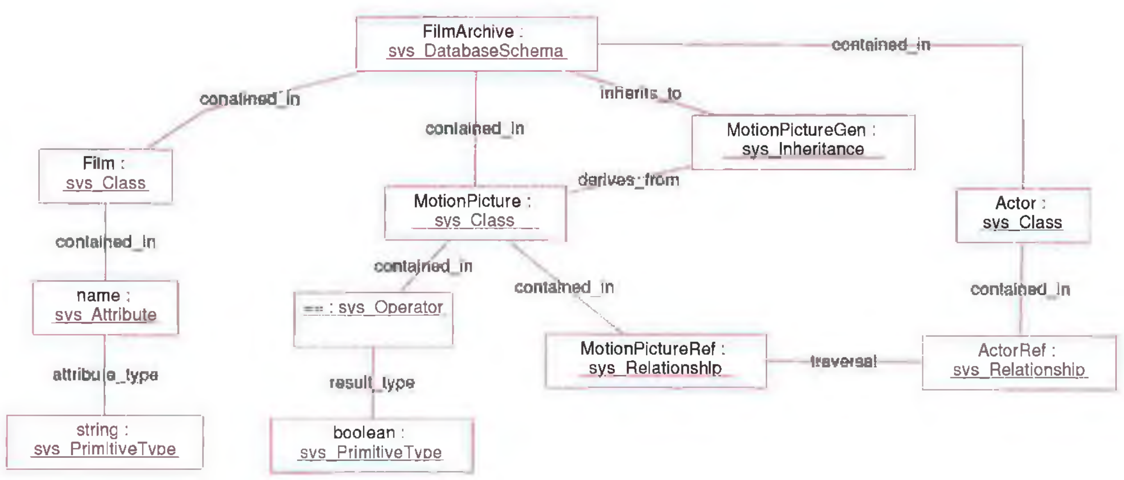

FilmArchive : sys_DatabaseSchema
contained_in
inherits_to
MotionPictureGen : sys_Inheritance
derives_from
Film : sys_Class
contained_in
MotionPicture : sys_Class
Actor : sys_Class
name : sys_Attribute
attribute_type
string : sys_PrimitiveType
== : sys_Operator
result_type
boolean : sys_PrimitiveType
MotionPictureRef : sys_Relationship
traversal
ActorRef : sys_Relationship

Figure 4.2: Metamodel Representation of Film Archive Schema.

instance. This relationship was implicitly defined in ODL$_x$ by nesting `class` elements within a `dbSchema` element.

Class attributes (`attribute` ODL$_x$ element) are represented as `sys_Attribute` instances, and linked to the containing `sys_Class` object (`contained_in` relationship). *Figure* 4.2 illustrates this by defining mapping for the `name` attribute of the `Film` class. Attribute type defied as a `primitiveType` element in the Film Archive ODL$_x$ schema is mapped to the `attribute_type` relationship in the EGTV metamodel. This relationship links an instance of the `sys_Attribute` metaclass with the instance of the `sys_Type` metaclass (supertype of the `sys_PrimitiveType`) to define attribute's type.

Relationships are mapped to the `sys_Relationship` metaclass, where each side of the bidirectional relationship (`relationship` ODL$_x$ element) corresponds to an instance of the `sys_Relationship` metaclass. For example, the `MotionPictureRef` relationship in *Example* 4.1 is mapped to `sys_Relationship` instance of the same name in *Figure* 4.2. This metaobject defines a `traversal` relationship to the `ActorRef` metaobject representing the other side of the bidirectional relationship.

The inheritance element `MotionPictureGen` is mapped to an object of the `sys_Inheritance` metaclass, while methods and operators are mapped to `sys_Method` and `sys_Operator` metaclass instances. The method's or operator's return value specified as `returnVal` ODL$_x$ element is mapped to the the `result_type` relationship defined in the `sys_Operation`, the superclass for both `sys_Operator` and `sys_Method` metaclasses. Thus, a return value of the `operator` == in *Example* 4.1 is mapped to an instance of the `result_type` relationship. This relationship points to a `boolean` instance of the `sys_PrimitiveType` metaclass that defines operator's return value.

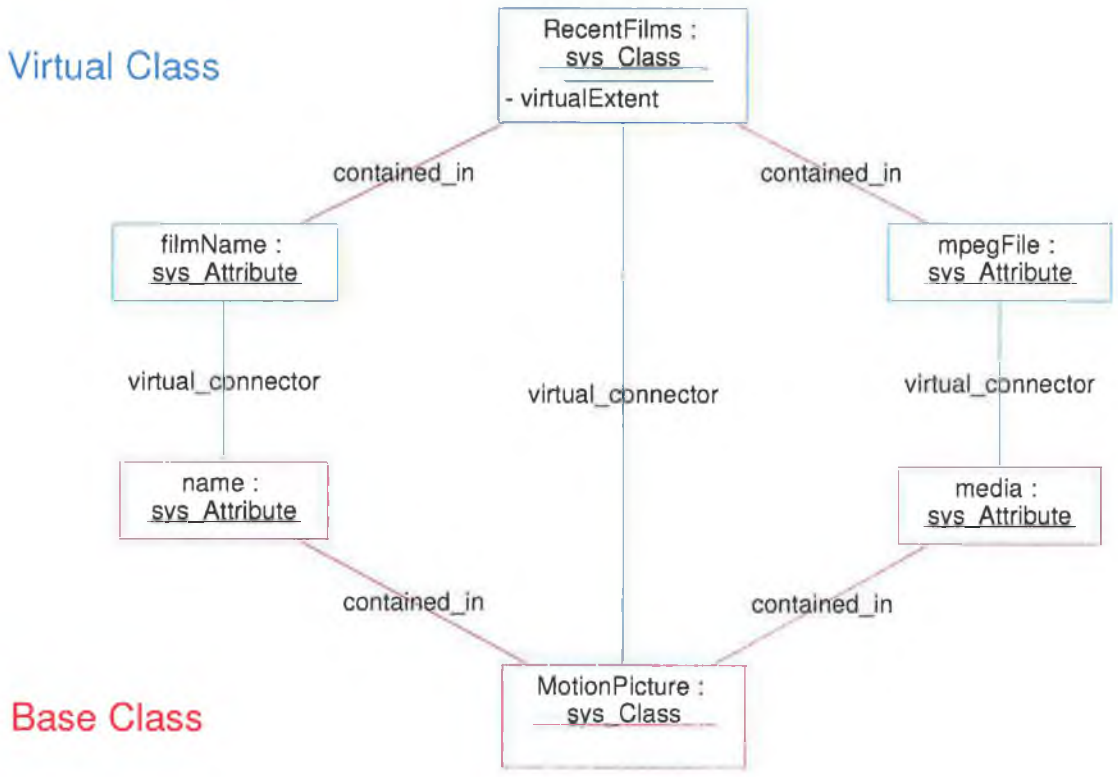


Figure 4.3: Metamodel Representation of `RecentFilms` Virtual Class.

4.3.2 Virtual Class Mapping

Virtual classes in the ODL$_x$ language are defined as stored queries and facilitate definition of simple views. Each virtual class definition (`virtualClass` ODL$_x$ element) is mapped to an instance of the `sys_Class` metaclass in the EGTV metamodel. Although this is the same metaclass to which base classes are mapped, its `virtualLevel` and `virtualEx-tent` properties are mapped differently for virtual classes. The `virtualLevel` property defines a non-zero value (zero level identifies base classes), while the `virtualExtent` contains an EQL query string which defines how virtual class extent is generated. The later property is directly mapped to the `extent` element in the ODL$_x$ specification. Each virtual class extent (EQL query) is evaluated to a set of virtual attributes that are mapped to `sys_Attribute` metaclass in the EGTV metamodel. This is illustrated in *figure 4.3* where mapping for virtual class `RecentFilms` (*Example 4.2*) and its attributes is defined in the UML collaboration diagram.

Each virtual class instance is linked with the `virtual_connector` relationship to one or more base metaclass instances from which it was constructed. Similarly, virtual class properties are linked to corresponding properties of base classes. This enables direct navigation from virtual to base classes and is used to facilitate direct updatability of virtual class instances. In *figure 4.3*, the `RecentFilms` virtual class is linked to the `MotionPicture` base class, while virtual attributes `filmName` and `mpegFile` are directly linked to `name` and `media` attributes of the base class.

Methods and operators of the virtual class cannot be reused from base classes, and must by explicitly defined within the virtual class specification. Therefore their metamodel mapping is identical to base classes.

4.3.3 Import Class Mapping

Virtual classes imported from canonical databases to the global schema are represented in the ODL$_x$ language with `importClass` subelement of the `virtualClass` ODL$_x$ element as illustrated in *Example* 4.3. Metamodel mapping is straightforward where each `virtualClass` element is mapped to one instance of `sys_Class` metaclass. The `database`, `schema` and `name` attributes of the `importClass` subelement are mapped to a single `virtualExtent` attribute in the `sys_Class` metaclass. This mapping is represented in `@<database>::<schema>::<name>` format, where each attribute is mapped to a segment of the same name, while `@` and `::` characters are used as segment delimiters. Therefore, imported class `LocalRecentFilms` in *Example* 4.3 is mapped to one instance of the `sys_Class` metaclass in the global schema repository. Its `database`, `schema` and `name` attributes are mapped to the value `@VideoReposi-tory::FilmArchive::RecentFilms` in the `virtualExtent` attribute of the `sys_-Class` metaobject.

4.4 Conclusions

When creating new database federations or extending existing ones, it is useful to have a common specification language for schemas. This requirement is highlighted when databases in the federation are heterogeneous and mutually incompatible. In this chapter we introduced ODL$_x$, the schema specification language used in the EGTV project. This language has been designed to facilitate an object schema definition in an implementation independent format. The language also addresses some deficiencies of the existing schema definition languages. The approach we have chosen is based upon standard technologies, and is portable across different platforms. These include, but are not restricted to, defining virtual classes, multimedia types and class operators. The language is supported with an XML Schema definition which provides a full syntax definition and specifies rules for integrity constraints enforcement. The complete XML Schema specification of ODL$_x$ is provided in *Appendix B*.

The ODL$_x$ language can be easily mapped to the EGTV metamodel, thus providing persistent metadata storage and dynamic browsing for both local and federated schemas. The ODL$_x$ database schemas transformed to the EGTV metamodel representation can be then easily mapped to O-O and O-R databases using metamodel mapping rules defined in the previous chapter.

By defining a platform independent schema definition language and its metamodel mappings, we are now in position to create multimedia enabled database schemas and integrate them into a database federation. The next step will be a specification of a query language capable of interrogating and updating both local and global EGTV multimedia schemas. In the next chapter we define a such language and discuss its features.

Chapter 5

The EQL Query Language

When storing data in heterogeneous databases, one of the top-down design issues concerns the usage of multiple query languages. A common language enables querying of database schemas in a platform independent format. This is particularly useful in federated database systems when newly added databases may be both numerous and heterogeneous. As the existing query language standards are generally incompatible and translation between them is not trivial, a new query language has been developed. The EQL language facilitates querying of both local and global EGTV multimedia schemas in a database and platform independent manner. The EQL language also provides an orthogonal type system, extensible operator semantics, the ability to define simple views, and updatability at the global level. In the previous chapter the schema definition language $\mathrm{ODL_x}$ was introduced. This chapter describes the structure of the EQL language, discusses its features and describes how $\mathrm{ODL_x}$ schemas are dynamically queried.

The ability to provide formal and unambiguous query representation is an important requirement for any query processing service. Query processing algorithms are able to evaluate only queries whose syntax and semantics are unambiguously specified. Thus, queries defined in a high-level query language are commonly transformed to an algebraic representation. Query algebra is a collection of formally defined operators for manipulating data structures defined in database model. Operators are atomic, and one operator's output can be used as an input for other operators. Thus, algebraic operators can be cascaded to form complex data transformation expressions that can represent high-level query definitions. The second part of this chapter discusses an object algebra for the EQL. This algebra is orthogonal as it manipulates classes only, which are both an input and output of each algebraic operator. A class is defined as the pair of metadata and object extent. EQL algebraic operators are type independent and they can be applied to any EGTV type, thus performing a high level transformations specific to the query language itself.

The remainder of this chapter is structured as follows: in §5.1 different approaches to schema querying are discussed and our solution is motivated; in §5.2 a brief explanation of the EGTV reference-based model is given; §5.3 introduces some general language concepts, while EQL language operators are categorised and discussed in §5.4; an object algebra for the EQL language is discussed in §5.5; in §5.6 some conclusions are given.

5.1 Query Language Standards

The EGTV project is primarily aimed at providing efficient query and update capabilities for a large distributed repository of multimedia objects. The individual repositories take the form of databases, independently designed and supplied by different vendors, thus heterogeneous in terms of data model and schema design. This assumes a federated database approach, where a canonical query language is used to provide a common interface to the system as whole.

The ODMG standard [CB99] specifies the Object Query Language (OQL) for querying object database schemas. The language is based on the ODMG data model and its type system. The main problem of this type system is different semantics used for representing object types and literals. As the consequence, the semantics of OQL is complex and inputs and output to the query language are not orthogonal. This results in an unclear semantics of nested queries and makes the process of query evaluation more difficult. The OQL language does not provide constructs for defining views (virtual classes), global schemas, nor does it support updates of database objects.

The SQL:1999 [GP99] specifies a standard for querying O-R databases by extending an existing relational query language with object features. Since the backward compatibility to relational model is preserved, the syntax and semantic of added object extensions is very complex and non-intuitive. This is especially the case when querying nested collections and references. The SQL:1999 type system adds User-Defined Types (UDT), collections and a reference type to an existing set of atomic literals, thus making query inputs and results non-orthogonal. Views can be defined as stored SQL queries, the result of a which can be interpreted as an extent of a UDT.

Since ODMG OQL and SQL:1999 SQL are two different and incompatible standards for database schema querying, query translation is not trivial. Also, both O-R and ODMG models have insufficient support for object views and lack the ability of defining efficient operators for multimedia types. Thus, OQL and SQL:1999 are not suitable for querying heterogeneous multimedia repositories. These problems motivated us to define a new query language that can be applied to provide an efficient interface to both O-R an O-O databases in our multimedia federation. This language provides mapping to both O-R and O-O databases and defines an orthogonal type system. The main contribution of the EQL language is the ability to query heterogeneous databases operating in a federated environment. The language supports global queries and provides full orthogonality between query inputs and outputs in a form of classes. EQL also defines semantics for updates of database objects and supports the definition of simple views. Operators of the query language are not hard-coded in the language itself, but defined as behaviour of data types. This adds additional flexibility to the query language, and enables definition of custom operators for multimedia handling. The EQL language is supported with a formally specified algebra, and a defined semantics of the process of query evaluation which are discussed later in this chapter.

5.2 The EGTV Data Model

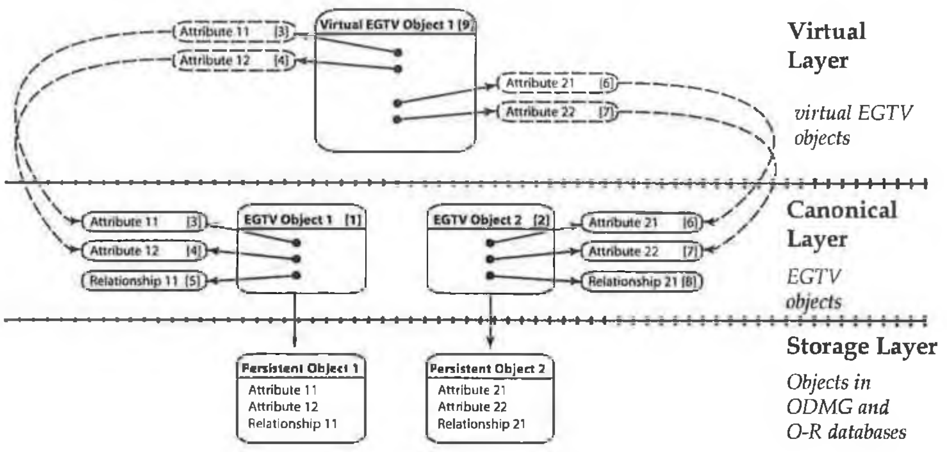


Figure 5.1: The EGTV Model.

The EQL query language was specifically designed for the EGTV data model [Kam04]. This model was developed as part of a separate research project within the EGTV project [KR03], but a brief overview is necessary to provide context for the remainder of this chapter. It provides a common interface to objects stored in ODMG or object-relational databases and is specifically designed to facilitate global updates. It uses *classes* as templates for the instantiation of *objects*. A class defines properties (attributes and relationships) that belong to the object, and set of *operations* that can be applied against the object. Classes can also define generalisation relationships with other classes in the schema, where multiple inheritance is possible. It is important to note that the EGTV model does not provide object storage itself, but acts as a *wrapper* for O-O and O-R models. Unlike many other object-oriented models, properties of an object are not contained in the object. Rather, they are independent objects, *referenced* from their parent object. Properties contain the actual values of the corresponding persistent object's properties. Using object-oriented modelling terminology, the EGTV model replaces all *containments* with *aggregations*, not affecting the expressiveness of the model. An advantage of this approach is that both objects and their properties can be directly referenced using the same type of reference. Of course, storage is required in the system in the form of object-oriented and object-relational databases at the Storage Layer.

Figure 5.1 illustrates a representation of a join operation, where two EGTV objects are form a *virtual* one. Persistent objects are first transformed to EGTV objects (at the Canonical Layer) and later joined (at the Virtual Layer). Object identities are shown in square brackets with their values generated by the system. Persistent objects are stored in an external database (at the Storage Layer), and can only be manipulated through an EGTV wrapper object. For each persistent object, a single EGTV object is materialised, forming a canonical representation of the persistent object. This EGTV object has a new object identity, which is bound to the identity of their original object for the lifetime of the EGTV object. The properties of all EGTV objects have object identities of their own, which are also immutable.

Virtual objects are materialised at the Virtual Layer as query results or views. An object identifier is generated for all virtual objects and thus, the materialisation of view or query objects uses an object generating semantics. The properties of virtual objects retain the identifiers of the EGTV properties upon which they are based. In *figure 5.1*, they are shown as dashed elements, with dashed arrows pointing to EGTV properties. This approach has benefits in updatable virtual objects, as updates to their properties are directly propagated to base and persistent objects.

5.2.1 Type System

Types in the EGTV model are categorised as built-in or user-defined. Built-in types are those provided by the model, and include atomic types (`integer`, `float`, `double`, `boolean`, `string`, `blob`, and `date`), collection types (`set`, `bag`, `list`), the bidirectional relationship type, and the reference type `ref`. A user-defined type is effectively a class that contains attributes and defines association and generalisation relationships to other classes in database schema.

The EGTV atomic types are typical to object models and require no further explanation. Additional atomic types can be added to capture the semantics of different multimedia formats such as `mpeg`, `jpeg` and `mp3`. Thus, we are able to extend the EGTV model with a native interface for multimedia storage and manipulation. The `ref` is a type used to reference objects (and their properties) and is internal to the EGTV model.

Collection types encapsulate multiple references; `list` is a *list of references* where order of elements is maintained; `set` is a *set of references* where order of elements is not relevant and duplicates are disallowed; and `bag` is an unordered *set of references* allowing duplicates. Collection types provide operations to traverse, extract and remove encapsulated references and to insert a new reference into a collection.

A relationship connects two related objects and is modelled as a linked pair of *relationship-sides*. In the EGTV model, relationships are *bidirectional*, and manipulation with one side of the relationship changes the other side of the same relationship. This preserves data consistency when objects are updated. The cardinality of each relationship-side is individually specified and can be *one* or *many*, where for cardinality many, it can be specified if the ordering is relevant (`ordered`) or not (`unordered`). An ordered relationship-side encapsulates a `list`, where an unordered one encapsulates a `set` of references.

An important feature of EGTV type system is the orthogonality between built-in and user-defined types. Contrary to ODMG, O-R, and some other database models, instances of EGTV built-in types are not literals, but objects with OIDs. Objects instantiated from built-in types can be directly referenced and are manipulated by the query language in the same manner as user-defined classes, thus providing full orthogonality and reducing the complexity of the type system. The other advantage of this approach is the ability to define type specific operators and methods (i.e. string comparison, date conversions). We commonly refer to both built-in types and user-defined classes as *types*.

5.3 Language Fundamentals

The EGTV Query Language (EQL) provides querying and updating for (distributed) EGTV multimedia database schemas. This language extends ODMG OQL [CB99] by providing new functionality and resolving existing ambiguities. However, contrary to some other query languages for reference-based models [SBMS94], the EQL language retains the familiar OQL-like syntax. Although the query syntax is relatively similar, the EGTV Query Language differs semantically from OQL in the way that queries are executed and results are generated. These differences are caused by the different representation of objects in the ODMG and EGTV models. We believe that this is superior as the EGTV model provides an orthogonal representation of types and full updatability of virtual objects.

Simple views can be defined as stored EQL queries where the view specification syntax is ODL_x based, as explained in chapter 4. For each view definition, one virtual class is created in the EGTV metamodel [RB02] representation and stored in a Schema Repository. Attributes of the virtual class are deduced from the EQL query result type, while methods and operators must be explicitly defined. Metadata representation for EGTV views is discussed in chapter 3, while view pragmatics and their role in global schema definition and querying are elaborated in chapter 7 after implementation details are discussed.

The main EQL language extensions are as follows:

- Clearly defined and fully orthogonal query input and output in the form of EGTV classes.
- Query language support for updates, creates and deletes.

- A new navigational join operator which simplifies path queries.
- The ability to define custom operators and redefine behaviour of existing ones.
- Support for multimedia data types.
- The ability to define simple views as stored queries.
- A fully updatable query result set.

The language is described through a series of examples based on the Multimedia Recording System database schema illustrated in *figure 5.2*. This schema stores information about *Físchlár* system users, their subscriptions and recording requests. The recordings system is used by administrators and users, each of which is represented using `Administrator` and `User` classes. The basic difference between administrators and users is that administrators can lock and supervise a user's account. The common properties of administrators and users (`name`, `login` and `password`) are placed in the abstract superclass `GenericUser`. The `User` class has a relationship to class `Rating`, which gives the maximum rating of the user. The class `Program` models individual programs: news, film etc. It aggregates optional `Trailers` and `Screenshots`, and has an association to `Rating`. The type of program is modelled using a relationship to `Category`. The `Category` facilitates recursive definition of categories, so that for example, Science Fiction can be defined as a sub-category of Movies. To place a recording request, a relationship between `GenericUser` and `Program` is established. Each individual broadcast of the program is represented using a `TVSchedule` object, which is associated with a `Channel` object. In each of the broadcasts, a program can have a different duration (for instance, because of commercial breaks). The user's subscriptions are modelled with the `Subscription` class, which has relationships to both `User` and `Channel`.

5.3.1 Query Input and Output

EQL provides full orthogonality between query input and output in the form of classes. This is different to OQL whereby each query can return either an object id, atomic value, structure or collection. Orthogonal input and output benefits EQL with easy subquerying, where subqueries can be freely composed as they are not constrained by incompatible result types. Input to an EQL query is a set of classes (either virtual or base ones), and the result of query evaluation is a single virtual class. This is illustrated in *Definition 5.2* where a query is formally represented as a generic function `EQL_Query` which takes a set of classes as input and generates a new virtual class as a result. The full semantics of base and virtual EGTV classes was explained in §5.2.

Definition 5.2 *Virtual_EGTV_Class* $\leftarrow$ *EQL_Query(* $\{EGTV\_Class\}^{1..*}$ *)*

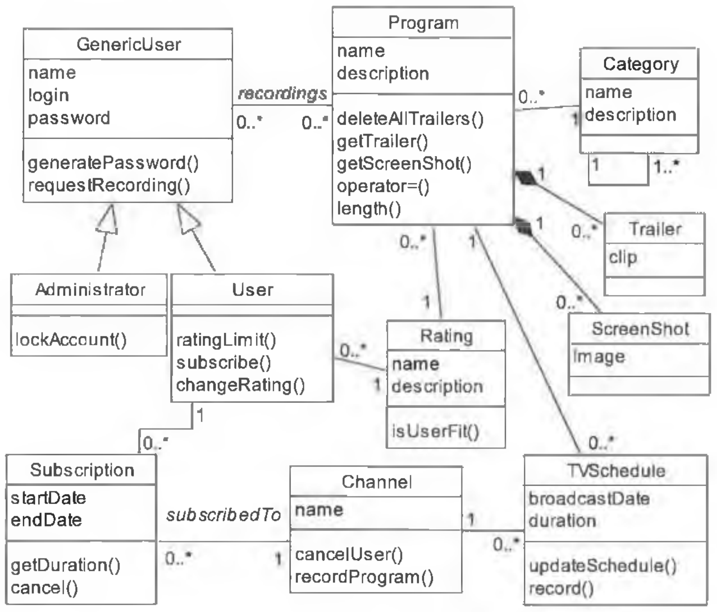


Figure 5.2: Database Schema for Multimedia Recording System.

The structure of an EGTV class (either base or virtual one) is formally specified in *Definition 5.3.*

Definition 5.3 *EGTV_Class ::= pair< metadata, object_extent >*

A class is defined as a pair of metadata information and an object extent. Metadata defines the structure of the class and the types of its properties, while an object extent contains all objects of the class. Metadata is represented in the EGTV metamodel format [RB02] and stored in the EGTV Schema Repository. Metadata representations and schema repository mappings were fully discussed in chapter 3. Class metadata has an important role in query evaluation where it is used to resolve class and property names specified in the query definition to objects in the class extent.

5.3.2 Query Structure

Each query consists of a *projection specification* (`select` clause), a *source specification* (`from` clause) and an optional *restriction specification* (`where` clause). A query can also contain two or more subqueries to which a set operation (union, intersection, or difference) is applied. This is formally specified in *Definition 5.4.*

Definition 5.4
```
EQL_Query ::= select <attribute_list>
               from  <source_predicate>
             ( where  <predicate> )*
             |
             ( Query1 <setop> Query2 )+
```

The *attribute_list* is a coma separated list of attributes representing the query's result. An attribute can be a selected class property, operation invocation, path navigation, assignment, or a complex expression involving any combination of these. The *source_predicate* defines the extent on which the query is processed. It can contain class names, join expressions, and subqueries. The *predicate* is any expression that evaluates to boolean `true` or `false` for each object in the initial extent generated in the `from` clause. This syntax is illustrated in *Example 5.1* where all recorded programs longer then 2 hours (120 minutes) are selected.

```
select name, description
from Program
where length() > 120;
```

Example 5.1: Query structure.

The structure of an EGTV query result extent is identical to the structure of *class extent*, which is a set of objects belonging to a single class. This identical structure facilitates easy *subquerying* (*nesting* queries), as subqueries can be freely nested in any part of the EQL query. Furthermore, client applications manipulate query extents and objects using the same interface (as both are of type `set`). This is distinct from the semantics of nested ODMG queries which are unclear, as the class extent is different in structure to the query extent. It also provides a simple view mechanism whereby views are represented as stored EQL queries. *Example 5.2* illustrates this feature by defining query that selects from the result of a its subquery. The subquery retrieves the `name` and the `length` only for those programs that are categorised as news.

```
select name, len
from ( select name, length() as len
       from Program
       where Program.categoryRef.Category = "News" ) as News;
```

Example 5.2: Nested query.

5.3.3 Deep and Shallow Extent

The EQL queries which return a set of objects can be evaluated against the deep or shallow extent of the classes specified in the source specification (`from` clause). The shallow extent includes only objects of the specified class, while the deep extent contains all objects instantiated from the specified class and all of its subclasses. This feature is defined in EQL with `deep` and `shallow` modifiers which precede the class name in the source specification. If a modifier is omitted, the default value is a deep extent. For example in the Multimedia Recording System, illustrated in *figure 5.2*, classes `User` and `Administrator` are derived from superclass `GenericUser`. *Example* 5.3 illustrates a shallow extent query where the set of objects consisting of `name` attributes of all `GenericUser` objects is returned, but this result set does not include any instances of the `User` and `Administrator` classes.

```
select User.name
from shallow GenericUser;
```

Example 5.3: Shallow modifier.

In *Example* 5.4, the deep extent query returns a set of `name` attribute objects of all objects instantiated from `User`, `Administrator` and `GenericUser` classes.

```
select User.name
from deep GenericUser;
```

Example 5.4: Deep modifier.

5.3.4 Operation Invocation

The EGTV model defines three categories of operations: methods, operators and class (static) methods. All three can be invoked in the EQL language, and the result they return is orthogonal to the query result which is an EGTV class. Operation behaviour is not defined in the EQL language but externally in the EGTV model. This makes our query language flexible, as behaviour is associated with data types, and not hard-wired to the language itself. The additional benefit is the ability to add new operations and redefine semantics of the existing ones independently of query language. However, new and redefined methods can be invoked from EQL in a same way as existing ones. As mentioned earlier, behaviour processing is outside the scope of this research and is covered elsewhere [KR01, Kam04].

Methods. Methods are always applied to individual objects, and they must be invoked by specifying method's name and an optional list of parameters. Methods can be invoked in any segment of the query (`select`, `from`, and `where`) using the same notation as for attributes and relationships. However, opened and closed brackets are a mandatory identification of a method. Method invocation syntax is illustrated in *Example 5.5* where `generatePassword()` method defined in `GenericUser` class is invoked in the `select` clause of the query to generate user's password. The `ratingLimit()` method is invoked in the `where` clause once for each object in the `User` extent, taking one `Rating` object as its input parameter.

Operators. Operators are a special category of methods for which an alternative, abbreviated invocation style is also available: an unary operator is specified in front of the

```
select u.login, u.generatePassword()
from User u
where u.ratingLimit( u.ratingRef.Rating ) < 5;
```

Example 5.5: Method invocation.

target object, and a binary one between the target object and the parameter. This is illustrated in *Example* 5.6 where the binary operator `operator+` is defined for the built-in type `string` with the effect of string concatenation. An extent of `User` objects is first selected in the `from` clause of the query, and then for each object in that extent an `operator+` is invoked on its `name` attribute. The `login` attribute is used as a parameter of the `operator+` method. All invocation rules defined for methods also apply to operators as they are simply a special category of methods.

```
select u.name + u.login
from User u
```

Example 5.6: Operator invocation.

Class methods. Class (static) methods differ from regular methods as they are not applied to single object instances, but to a class extent as a whole. Thus, class methods are commonly used for calculating aggregate values from a set of input objects. This is further explained in §5.4.1. Class methods must be specified in the `select` clause of the query as they require different invocation semantics to other operations. While non-static operations are individually invoked for each object in the `from` clause extent, class methods are invoked only once for an object extent as a whole. This is illustrated in *Example 5.7* where class method `min` defined for built-in type `date` is used in query that selects minimum duration of all `TVSchedule` objects. The query processing algorithm first evaluates the `from` clause of the query where the full `TVSchedule` object extent is materialised. This set of objects is then used as an input to class method `min` to determine the minimal duration value. The result is a virtual class containing the minimum duration object.

```
select min( s.duration )
from TVSchedule s
```

Example 5.7: Static method invocation.

5.3.5 Aliases

Aliases can be defined for expressions, class and attribute names. Once defined, alias names can be used anywhere in the projection, source and restriction specification where the original name or expression is expected. Aliases are defined with the keyword `as` which follows the original name. This keyword is not mandatory and can be omitted. *Example* 5.8 illustrates the usage of alias `p` for the class `Program`, and alias `CategoryName` for path expression in the `select` clause.

```
select p.ContainedPrograms.Category.name as CategoryName
from Program as p
where p.name = "Morning News";
```

Example 5.8: Name alias.

5.3.6 Undefined Values

An undefined value is a term that generally denotes any form of incomplete/missing/ uncertain/non-applicable information in the database schema. Database models generally represent this kind of data as a special *null* value.

Null values originated in the relational database model as a way of representing undefined states of relational table columns. The main reason for inclusion of undefined values was the inability of the relational model to represent complex inheritance and association relationships between tables. The consequence is that properties of conceptually different modelling entities are often merged into a single relational table, where they can have undefined states. For example, two tables with the inheritance relationships must be flattened to form a single table. When data corresponding to the parent table is inserted, the state of all child table columns is left undefined (null). Null values are fully incorporated in the SQL-92 standard [MS92], however this extension does not have any theoretical background and is rather is a technical trick which can be used for different purposes [SKLU96]. For example, null values in the `where` clause are always treated as a `false` value, while aggregate functions (`avg`, `sum`, `max`, `min`) completely ignore them when performing calculations. Since the object-relational model [Sto96] and its SQL:1999 standard [GP99] are extensions of the relational model, they fully support the relational syntax and semantics of null values.

Undefined values are generally not used in object-oriented models and programming languages as they violate strict type checking rules. However, the ODMG model defines an undefined value type *nil.* Every attribute of an ODMG object regardless of its type can be assigned a nil value, which means it can have an undefined state. Nil values are imported to ODMG from the OMG CORBA type model [OH98] on which it is based, but the OQL specification does not expand on this issue. The keyword `nil` in OQL is a special "object literal" denoting an absent object or non-initialised value [SKLU96]. An OQL expression returns nil if it cannot be evaluated correctly. A function that does not return a value also returns nil. Other than testing for equality, the ODMG standard does not specify the semantics of nil values in advanced query facilities such as aggregate functions, grouping, joins, and method invocations.

The EGTV data model is a pure object-oriented model and as such it rejects the notion of undefined values in the database schema. Objects in the EGTV model are strongly typed, thus the introduction of null values would violate the static type checking of class operations which are an essential feature of the model. However, null value functionality is substituted with the concept of default values [DD92, Ges91] in object constructors. Thus, constructor parameters not specified when object is created, are substituted with their default values defined in the body of constructor method.

The EQL language defines a special *nil* type for describing unassigned relationships. The EQL nil type, contrary to OQL, does not represent an undefined value, nor can it be assigned to an attribute of an object. Its role is limited solely to denoting if the relationship points to an object (or a set of objects), or if it is unassigned. The nil type is supported in the EQL language with the `nil` keyword. This can be used in the query language to decouple a relationship, or to test if the relationship points to an object.

5.4 Query Language Operators

All operators of the EQL query language can be classified into two categories: type operators and language operators. Type operators are individually defined for each built-in or user-defined type, while language operators are type independent and defined within the query language.

5.4.1 Type Operators

Type operators defined in EQL are applied to objects or collections of objects in the EGTV model. Unlike other query languages, EQL type operators are not hard-wired to the query language, but associated with types. Each EQL operator invokes the appropriate operator defined within the type. Thus, our query language is unique, as its operators can be individually specified for each database type. EQL type *operators* are classified as *comparison* (=, <, >, <=, =>, `!=`, `identical`), *arithmetic* (`*`, `/`, `+`, `-`), *logic* (`and`, `or`, `not`), *assignment* (`:=`), and *aggregate* (`max`, `min`, `avg`, `sum`, `count`).

Each EQL type operator is supported for built-in types, where the behaviour of built-in types is part of the model itself and cannot be changed. Operators in user-defined classes are defined by the class designer, where for some operations, the default behaviour is provided. This default behaviour can be redefined by modifying operator's definition. Operator redefinition is a feature of the EQL language that is not present in OQL. Each redefined operator is represented as a method in the user-defined class. For example, the EQL equality operator = is defined in the class `Program` as a method `operator=`. This is illustrated in *figure 5.2*. Thus, when two `Program` objects in an EQL query are compared for equality, the corresponding `operator=` method is invoked. The EGTV metamodel is used for mapping between class methods and EQL type operators. Operator redefinition enables us to define custom operators for each user-defined class independently of query language. This feature is beneficial for multimedia databases, where special behaviour must be defined for operations on the large multimedia objects. For example, the equality operator = can be redefined to support comparison of different video recordings based on their content, and not the file size.

With all of the operators described in this section, the result is always a virtual class where each object in the virtual class extent will have a newly generated identifier and comprise one or more properties. The complete classification of all EQL type operators is provided in *Appendix D*.

Comparison Operators. This category includes operators for testing equality of objects. All comparison operators are binary, they receive two objects as input and return a virtual `boolean` object (`true` or `false`) generated as a result value. The default behaviour of the equality operator is to compare all object properties for equality. By default, two objects are equal if all their properties are equal by value. This can be changed by redefining the `operator=` method of the class, and specifying the new behaviour. The `identical` operator is stricter than the equality one as it tests if two objects have the same object identifiers. The `identical` operator cannot be overloaded.

Arithmetic Operators. Arithmetic operators are binary, they receive two objects as input and generate a virtual object containing the result value. They are defined for some system types, but the custom behaviour can be specified for each user-defined class using operator redefinition.

Logic Operators. All logic operators are defined as binary except the logical negation which is an unary operation. All operators receive objects of `boolean` (`true` or `false`) type as input and return a virtual object of the same type.

Assignment Operator. The assignment operator := takes the form of lRef := rRef and is defined to assign the value of the right-hand side object to the value of the left-hand side object. The assignment operator plays an important role in update queries and it is further described in §5.4.2.

Aggregate Operators. Aggregate operators are applied to collections of objects, but their result is always a single virtual object. This result object can be based on an existing database object or a it can be a newly generated virtual object containing the aggregate value. The `max` and `min` aggregate operators return an existing object, so their result set is updatable. The other aggregate operators generate a new virtual object as their result, which is not updatable. Aggregate operators are defined for all system types, but each user-defined class must explicitly define aggregate operators which could be applied to objects of that class. This is achieved by specifying one class (static) method for each aggregate operator in the definition of the class. The query in *Example* 5.9 can only be executed if the `max` aggregate operator is defined for the `Program` class.

```
select max(User.recordings.Program)
from User
where User.name = John;
```

Example 5.9: Aggregate operator.

User-defined classes can have custom aggregate operators. These operators are also specified as class methods and can be applied only to objects of those classes for which they have been defined. For example the `length` operator can be defined for class `Program` to return the the total length in seconds of all recordings in the result set of `Program` objects as it is illustrated in the *Example* 5.10.

```
select length(User.recordings.Program)
from User
where User.name = "John";
```

Example 5.10: Custom aggregate operator.

5.4.2 Language Operators

Language operators are type independent, and can thus be applied to any EGTV object regardless of its type. These operators are not defined as behaviour of types, but at the more general level of the query language itself. Language operators are required as they provide high level operations of the query language not relevant for individual types, but important for expressiveness of EQL. This category of operators include set operators, path navigation, property and navigational joins, and update operators.

Set operators

Set operators defined in EQL are `union`, `unionall`, `intersection`, `difference`, `inset`, `distinct`. They allow set-like manipulation on collections of objects. Set operators are defined at the level of the EQL query language and not as the behaviour of built-in types and user-defined classes. This is because set operators are only ever applied to dynamically created virtual classes (resulting from two subqueries) which do not have any behaviour defined or included from input classes. This is illustrated in *Example 5.11* where name and login attributes are selected for those people that are both administrators and users. The intersection set operator is applied to virtual classes that result from two subqueries. Since these virtual classes are dynamically created, they contain state only, and do not define any behaviour to which the set operator can be mapped.

```
select name, login
from User
intersection
select name, login
from Administrator;
```

Example 5.11: Set operator.

All set operators except the operator inset generate a set of objects as a result of its evaluation. The inset operation takes as an input a set and a single object, returning a virtual boolean object with value true (encapsulated as a singleton object) if the object is contained within the set. All set operations are based on object value comparison, and not on the comparison of object references which are unique in the system. When duplicate elements are removed from the set, the language does not guarantee which of the duplicate objects are removed. This is because duplicate objects are equal by value and not by reference.

Path Navigation

Path navigation in the EQL language allow traversing through object graphs. The syntax, similar to path navigation in OQL, uses the operator "." (dot) for traversing between interrelated objects in the database schema. The expression o1.o2 where o1 and o2 are class names, evaluates to the set of o2 objects pointed from one instance of the object o1. The path navigation can be used in the projection, source and restriction specifications as long as all class names in the navigation path except the last one evaluate to the single objects. Different examples of the navigation are illustrated in *Examples* 5.12, 5.13, 5.14.

```
select Program.ratedIn.Rating.name
from Program
where name = "Morning News";
```

Example 5.12: Path navigation in projection specification.

```
select name
from User
where User.allowances.Rating.name = "All";
```

Example 5.13: Path navigation in condition predicate.

```
select p.name
from ( select Category
       from Category
       where name = "News" ).ContainedPrograms.Program p;
```

Example 5.14: Nested query path navigation.

The query in *Example* 5.12 identifies one `Program` object and uses path navigation in the projection specification to traverse to the `Rating` object and display its name. Each `Program` object has a relationship to exactly one `Rating` object. *Example* 5.13 illustrates the syntax for path navigation in the restriction specification, while *Example* 5.14 provides path navigation in the source specification. The nested query expression in *Example* 5.14 was used to retrieve one `Category` object to which the path navigation is then applied. This results in a set of `Program` objects from which the `name` attribute is selected.

Path navigation through `n-1` and `n-m` relationships is ambiguous as the left-hand side of the relationship is not a single object, but an object collection. Thus, there is no single way of traversing between objects as multiple paths exist. The only unambiguous way of traversing such a relationship is by using iterators to transform `n-1` and `n-m` relationships into multiple `1-1` relationships, which are easily navigable. OQL resolves this problem by using implicit iterator syntax in the `from` clause of the query as illustrated in *Example* 5.15. Here, for each `Program` object related `User` objects are retrieved and their names selected. However, this approach has several flaws. Implicit iterator definition cannot be formalised in the query algebra [Sub96], thus query optimisation is difficult. Furthermore, ambiguities can arise in the processing of nested queries where nested query aliases can be mistaken for implicit iterator syntax. These inconsistencies can severely limit the orthogonality of the query language, so we have chosen not to follow this OQL syntax. Instead we introduced a new *navigational join* operator which has a direct algebra mapping and resolves implicit naming ambiguities. We discuss navigational join operator later in this section.

```
select p.name as uName, Program.name as pName
from Program.recordings p, p.User;
```

Example 5.15: OQL implicit iterator syntax.

Joins

EQL defines two join operators: property join, and navigational join.

Property Join defines the standard inner join operator. This is illustrated in *Example 5.16* where classes `User` and `Administrator` are joined to create a virtual class containing only those persons that belong to both Administrators and Users.

```
select u.name, u.login, a.login
from User u join Administrator a on u.name=a.name;
```

Example 5.16: Property join.

A join is specified with the keyword `join` in the source specification of the EQL query, while the joining predicate is defined with the keyword `on`. It is important to note that equality in the join condition is tested using type operators. In *Example 5.16*, User and Administrator `name` properties are strings, so they are compared for equality using the `operator=` defined in the built-in type `string`.

Navigational Join creates a join-like result that spans two interconnected classes using path navigation. This operator is not defined in OQL, but is found in other object-oriented query languages [SBMS94]. The result set produced by the navigational join is a set of pairs containing the start and the end points of the navigation path. EQL introduces a new operator `connect` which denotes the navigational join operation. For example, consider the query where a list of programs marked for recording should be retrieved for each user. This query uses a navigational join to connect instances of `User` and `Program` classes (m–n relationship) as is illustrated in *Example* 5.17.

```
select User.name as uName, Program.name as pName
from Program connect User on User.recordings;
```

Example 5.17: Navigational join.

The result set for the query in *Example* 5.17 contains a set of objects consisting of user `name` and program `name` attributes. The keyword `on` specifies the name of the relationship that interconnects two joined classes. This keyword is optional and can be omitted if no more than one relationship is defined between joined classes. Multiple `connect` operators can be cascaded in the source specification to provide navigational joining of two or more classes.

Update Operators

OQL does not support create, update or delete operations on database objects. These operations can only be performed through user-defined methods. This provides a limited update functionality and as the operation invocation semantics is vaguely defined and the effect of such an operation invoked within a query remains unclear. The EQL language extends OQL with support for update operations. EQL defines only projection, source and optional restriction specification, with no special language elements provided for updates. This maintains a syntax that is both simple and consistent.

Create. New objects are created by invoking an object constructor in the projection specification. This is illustrated in *Example 5.18* where a new object of the class `User` is created by invoking a constructor method in the `select` clause.

```
select User( "Tom", "tomh", "tom54", nil, nil, ratingRef )
from User, Rating as ratingRef
where Rating.name = "all";
```

Example 5.18: Create.

Parameters of the constructor are provided in the order specified in the Schema Repository definition of the constructor method. The last three parameters in the `User` constructor are references to objects of `Program`, `Subscription`, and `Rating` classes. In *Example 5.18*, the `Program` and `Subscription` references are initialised to `nil`, while the `Rating` reference is set to the value selected from the query. The effect of this query is that an EGTV object is materialised, and the corresponding persistent object in the Storage Layer (see *figure 5.1*) is created. The query returns a virtual class containing a newly materialised object.

Delete. Objects are deleted from the schema by invoking their destructor method. Destructors are provided for all built-in types and default destructor behaviour is applied to each user-defined class which does not have an explicitly defined destructor method. Delete is represented in the EQL syntax by assigning `nil` to the object as illustrated in *Example* 5.19. This invokes a destructor on the object and releases the reference. This has the effect of deleting the physical object through the EGTV interface. The delete query returns a `nil` for each successfully deleted object as a result of its execution.

```
select User := nil
from   User
where  name = "Tom";
```

Example 5.19: Delete.

Update. Updates in EQL are specified using assignment operator '`:=`' to assign new values to the properties of the selected object or to the object itself. These modifications are then propagated to persistent objects in the database. The behaviour of the assignment operator is defined for each system type and default behaviour can be applied to any user-defined class. The default behaviour of the assignment operator for user-defined classes is to copy each attribute of the r-value object to the corresponding attribute of the l-value object. References defined in the r-value object are assigned to the corresponding reference properties of the l-value object and no new instances of the referenced objects are created. This is a shallow copy, although this default behaviour can be changed in each user-defined class by overloading the assignment operator and specifying the new behaviour. To maintain orthogonality, the result returned by the query is a virtual class containing updated objects. This result set can be used as an input to the other queries or to the parent subquery in the nested query. The database can also be updated by methods invoked in the EQL query, thus producing possible side effects discussed in [KR03].

The query in *Example* 5.20 illustrates an update of `login` and `password` attributes for one object of the class `User`. The new values are provided in the query itself. *Example* 5.21 illustrates an update of the same two attributes, but this time from the result set of a subquery, while in *Example* 5.22 the TV schedule with the maximum duration is first selected and then its duration is increased by 10. *Example* 5.23 demonstrates how `duration` properties, updated in the subquery of the nested query, are used as the input for an aggregate function `sum` in the parent query.

```
select login := "tomg", password := "tom34"
from User
were name = "Tom";
```

Example 5.20: Simple direct update.

```
(select login , password
 from User
 where name = "Tom") := select login, password
                        from User
                        where name = "John";
```

Example 5.21: Update from subquery result set.

```
select max(duration) := max(duration) + 10
from TVSchedule;
```

Example 5.22: Update of a method result set.

```
select sum( DaySchedule.duration )
from ( select duration := duration + 10
       from TVSchedule
       where broadcastDate = "14/11/2002" ) as DaySchedule;
```

Example 5.23: Update within the subquery.

5.5 EQL Algebra

The query algebra is a theoretical language that can formally and unambiguously define queries on a database. The algebra itself is defined as a set of atomic operations that manipulate input collections of data to construct results. Both input operands and results are the same data structures, so the output from one operation can subsequently form the input to other operations. This property is called closure: data representations are closed under the algebra. Thus, queries in a high-level language are transformed into a sequence of unambiguous algebraic operations that can be easily executed by the query processor. For every syntactically correct query written in a query language, there is an equivalent algebraic expression as intermediate form, describing how to carry out the computation. This intermediate form is then input to a query processor. Furthermore, algebraic representation is suitable for query optimisation. A query, once translated into an algebra expression, can in many cases be transformed into an equivalent expression that can be evaluated much more efficiently.

Relational algebra [CBS96] is based upon the formal relational model that is highly standardised and used in all relational databases. The mathematical simplicity of the relational model allows a clean and formally consistent definition of algebraic operators and the data structures they manipulate. Inputs and outputs to all algebraic operators are relations (set of tuples), and all transformations in the algebra are based upon relations. However, this algebra cannot support complex data structures and lacks some advanced modelling concepts such as relationships, nested relations, and identifiability. These types of advanced modelling concepts are incompatible with the relational model, and thus cannot be represented in the relational algebra.

Object-relational databases extend the relational data model and SQL language with user-defined types, OIDs, and type methods. In addition to simple atomic types, attributes in an O-R relation may have complex structures such as references, nested tuples and collection values. However, the SQL:1999 standard [GP99] does not discuss the algebraic representation of these extensions. Thus, the object-relational algebra has not been formally standardised, although many independent implementations exist [LLO98]. O-R algebras extend the existing relational algebra with operators for nesting and unnesting non-normalised relations, and with the path navigation. However, these algebras generally ignore object identity and treat inputs and outputs to algebraic operations as complex (nested) relations [LLO98].

The main distinction between object-oriented algebras and their object-relational counterparts is in the type of data structures they manipulate. Inputs and outputs to object-oriented algebraic operators are uniquely identifiable objects and collections of these objects. This is different to relational and O-R algebras where algebraic operations manipulate literal data structures only. The AQUA object-oriented algebra [LMS$^+$93] is an example of such an algebra. It provides a large set of algebraic operators capable of supporting multiple object-oriented query languages. However, the query language mapping is complex and algebraic operators are not fully orthogonal as they must be defined for both objects and literals (immutable objects) that exist in some O-O query languages. The ODMG standard [CB99] defines the OQL query language, but does not discuss its algebraic representation. Recently, an independent algebra for OQL has been proposed [Zam02]. This algebra specifies a set of algebraic operators that operate on both records (literals) and objects. Although the algebra can support OQL queries by defining a minimal set of algebraic operators, it is not orthogonal as inputs and outputs to algebraic operators are both objects with identity and records with no identity.

The EQL algebra presented in this chapter defines a minimal set of operators required for algebraic representation of EQL queries. It represents the execution model for EQL language and is supported with formal EQL to algebraic mappings. Algebraic operators are type independent high level operations specific to the query language itself, and not to individual types. Furthermore, inputs and outputs of all EQL algebraic operators are fully orthogonal and represented in the form of EGTV classes. This ensures that our algebra remains compact and enables the simple translation to an equivalent query processing algorithms. This section discusses theoretical definitions of the EQL algebra, while its implementation and role in the query processing is explained in the next chapter.

5.5.1 Algebraic Operators

The EQL algebra is defined as a set of high level operators for manipulating EGTV classes. Algebraic operators have the same input and output as queries, so each EQL query can be easily transformed to an algebraic representation. This is illustrated in *Definition 5.5.* An EQL query takes as input a set of classes and a query expression (EQL query string), while the result is a new virtual class. EQL queries can be transformed to a sequence of algebraic operators, where the result of one operator is an input to other operators. Each operator takes as input one or two classes, and an expression to be applied to them. EQL algebraic operators can be binary or unary, while the expression is an optional argument. The operator's result is always a single virtual class, and is consistent with closure rules defined for relational algebraic operators [EN94] and orthogonality of the EQL language.

Definition 5.5

egtvClass $\leftarrow$ ***EQLQuery***($\{$*egtvClass*$\}^{1..*}$, *queryExpression*)
egtvClass $\leftarrow$ ***algebraicOperator***($\{$*egtvClass*$\}^{1..2}$, $\{$*expression*$\}^{0..1}$)
queryExpression ::= EQLQueryString
expression ::= booleanPredicate | *propertyList* | *pathExpression*
| *renameExpression*

All algebraic operators are classified into two categories: general and set operators. General operators are: *projection, filter, cartesian product, path, navigational join, property join,* and *rename.* Set operators define mathematical set operations on sets of EGTV objects and include *union, unionall, intersection, difference,* and *distinct.* All other EQL operators (comparison, logic, arithmetic, aggregate, and assignment) are not part of the algebra, since they are defined as behaviour of built-in types and user-defined classes.

Operators and methods defined in types can be accessed and invoked from within EQL algebra. EQL algebraic operators are not type dependant and can be applied to any EGTV type.

5.5.2 General Operators

This group of operators comprise the core of the EQL algebra and facilitate the high level operations of the query language. Operators can manipulate class metadata, objects or both.

The description of each operator is clearly separated into metadata and a data related set of actions. This separation is important for the evaluation of EQL views where the process of view metadata construction is independent of any subsequent view materialisations.

Filter (ϕ)

Filter is a unary operator that filters unwanted objects from the source class C_1 using the boolean predicate expression commonly specified in the `where` clause of the EQL query. This is formally represented in *Definition 5.6.* The boolean predicate expression is any sequence of methods and properties that return result of type `boolean`. A boolean predicate is usually constructed by applying EQL logic operators (`and`, `or`, `not`) to class properties and results of method invocations.

Definition 5.6 $C_2(p_{11}.. \ p_{1n}) \leftarrow \phi_{booleanPredicate} \ C_1(p_{11}.. \ p_{1n})$

Metadata processing. A new virtual class C_2 is generated to represent the structure of the result. This class contains an identical set of properties ($p_{11}..p_{1n}$) to the source class C_1. This is because the filter operator does not modify the metadata of the source class, but manipulates only the object extent.

Data processing. The source extent of objects belonging to class C_1 is filtered according to the `booleanPredicate` condition to generate a new object extent of class C_2. The `booleanPredicate` is applied to every object in the source class C_1, and only those objects of C_1 for which the `booleanPredicate` evaluates to `true` are included in the result class C_2.

Projection (π)

Projection is used to project properties into a new class: its input is a class and an expression in the form of a property list. The property list corresponds to the `select` clause of the EQL query and defines attributes of the result class. Each property in the property list can be a constant specified in the query definition, class attribute, method invocation, or a path navigation. This is illustrated in *Definition 5.7.* Methods and type operators can be cascaded to form complex expressions where the output of the one operation is used as the input to other operations.

Definition 5.7

$C_2(p_{21} .. \; p_{2m}) \leftarrow \pi_{propertyList} \; C_1(p_{11} .. \; p_{1n})$
$propertyList ::= \{ \; constant \mid attribute \mid operator \mid method \mid pathNavigation \; \}^{1..*}$

Metadata processing. The expression specified in the form of a property list is evaluated, and a virtual class C_2 representing the result type is constructed. Attributes of this virtual class match properties specified in the property list. Each property in the list is an expression which defines how a corresponding attribute of the result class C_2 is generated from the source class C_1. Thus, each expression in the the property list is resolved to an attribute of class C_2. For example, a method expression is evaluated and its result type is represented as an attribute of the class C_2.

Data processing. The result of this processing stage is as an object extent of the virtual class C_2. The transformation rules defined in the property list are applied to objects of the class C_1 to generate the corresponding objects of the class C_2. In this case, for each object in the class extent C_1, one object in the class extent C_2 is generated.

Cartesian Product (×)

This is a binary operator which generates a cartesian product of two source classes. The cartesian product operator is never used directly and is required only for construction of more complex join operators. It does not require any parameters (predicate expression) as its processing is straightforward. The formal definition is presented in *Definition 5.8.*

Definition 5.8 $C_3(p_{11} .. \; p_{1n}, p_{21} .. \; p_{2m}) \leftarrow C_1(p_{11} .. \; p_{1n}) \times C_2(p_{21} .. \; p_{2m})$

Metadata processing. The source classes C_1 and C_2 contain two object extents and corresponding metadata definitions. The metadata definition of the result class C_3 contains all attributes of both C_1 and C_2 classes.

Data processing. The object extent of class C_3 is created to contain all combinations of C_1 and C_2 object pairs. Each object in the object extent C_1 is paired with all objects from the C_2 extent to generate the cartesian product of two object sets.

Path ($\vdash$)

The path operator takes as an input one class and a path expression. For each object in the source class extent, a path expression is evaluated to an object or to a set of objects located one nesting level below. These objects comprise the extent of the result class and are directly referenced by the source class objects. For example, the path expression `C.p` applied to object of class `C` returns a set of `p` objects. This is illustrated in *Definition 5.9.*

Definition 5.9 $C_2(p_{21}..\ p_{2m}) \leftarrow \vdash_{pathExpression} C_1(p_{11}..\ p_{1n})$
$pathExpression := className$

Metadata processing. A path expression is applied to class C_1 to retrieve the metadata of the referenced class C_2 from the Schema Repository.

Data processing.

1. For each object in the class C_1 object extent, a set of referenced objects is retrieved. These objects belong to the virtual class C_2.
2. The result set is generated by combining all partial results retrieved in the previous processing step to construct the object extent of the class C_2.

Property join ($\bowtie$)

Property join is a binary operator which takes two classes as input and an expression defining the joining criteria. Result is a new virtual class containing all properties of both source classes. This is effectively an inner join operator applied to EGTV classes. The operator is presented formally in *Definition* 5.10.

Definition 5.10

$$C_3(p_{11}..\ p_{1n},\ p_{21}..\ p_{2m}) \leftarrow C_1(p_{11}..\ p_{1n}) \bowtie_{booleanPredicate} C_2(p_{21}..\ p_{2m}) \Leftrightarrow$$
$$\phi_{booleanPredicate}\ (C_1(p_{11}..\ p_{1n}) \times C_2(p_{21}..\ p_{2m}))$$

The operator can be formally expressed as the cartesian product of classes C_1 and C_2 to which is the filter operator then applied with the `booleanPredicate` condition.

Metadata processing. A new virtual class C_3 representing the join is constructed. This class contains all the properties of both source classes C_1 and C_2.

Data processing.

1. The object extent of the virtual class C_3, constructed during the metadata processing stage is created as the cartesian product of both source extents.
2. The filter operator is applied to the object extent C_3 to remove unwanted objects. A `booleanPredicate` evaluates each object in the extent.
3. The result is returned as a virtual class C_3. All objects in the C_3 object extent must satisfy the predicate condition.

Navigational join ($\rtimes$)

This algebraic operator maps directly to the `connect` operation in EQL. It is a binary operator that takes two classes as input and creates a join-like result based on the relationship defined between them. The joining condition is specified as the name of the relationship as is illustrated in *Definition 5.11.*

Definition 5.11

$$C_3(p_{11}.. \; p_{1n}, p_{21}.. \; p_{2m}) \leftarrow C_1(p_{11}.. \; p_{1n}) \rtimes_{pathExpression} C_2(p_{21}.. \; p_{2m}) \Leftrightarrow$$
$$\phi_{\exists pathExpression} \; (C_1(p_{11}.. \; p_{1n}) \times C_2(p_{21}.. \; p_{2m}))$$

The navigational join operator can be formally represented as the cartesian product of the classes C_1 and C_2, to which the filter operator is then applied to remove all objects not satisfying the `pathExpression` condition.

Metadata processing. The result virtual class C_3 is constructed to contain all properties from both joined classes (C_1 and C_2) except the relationship on which the navigational join is performed. Since all relationships in the EGTV model are bidirectional, this relationship is excluded from both source classes.

Data processing.

1. For each object in the extent C_1, related objects in the extent C_2 are retrieved. A new set of object pairs (first set object, second set object) is created as the partial result.
2. The final result is generated as a union of all partial results. All objects of the result extent are instantiated from the virtual class C_3.

Rename (R)

Rename is a unary operator that can change class name or names of its properties. Rename can be formally represented as the special case of the projection operator, where all properties are projected to the equivalent but, only with different names. This is illustrated in *Definition 5.12.*

Definition 5.12

$C_2(p_{11}.. \ p_{1n}) \leftarrow R_{renameExpression} \ C_1(p_{11}.. \ p_{1n}) \Leftrightarrow \pi_{propertyList} \ C_1(p_{11}.. \ p_{1n})$
$renameExpression := \{ \ (\ old\_name, \ new\_name \ ) \ \}^{1..*}$
$name := old\_name \mid new\_name$
$property \ list := \{ \ name \ \}^{1..*}$

Metadata processing. A new virtual class C_2 is generated to contain all properties of the source class C_1, and only the names specified in the expression are changed to new values.

Data processing. The full extent of the class C_1 is projected to the newly generated virtual class C_2.

5.5.3 Set Operators

Set operators in EGTV algebra include: *union*, *unionall*, *intersection*, *difference* and *distinct*. Set operators are the only type operators of the EQL query language explicitly defined in the algebra, and not as the behaviour of the built-in types and user-defined classes. This is because the query processor applies set operators only to dynamically created virtual classes (resulting from two subqueries) which do not have any behaviour defined or included from source classes. For this reason, set operators must be supported at the EQL algebra level as global operators. All other operators of the EQL language are always mapped to the existing behaviour in the built-in types, user-defined classes or views. The semantics of each operator is straightforward and described earlier in this chapter.

Unionall ($\uplus$)

Unionall is binary operator that takes two classes as input and merges their object extents to produce the virtual class result. Duplicate objects are not eliminated in this process. Both source classes must have the same number of properties, and property types must match. This is illustrated in *Definition 5.13.*

Definition 5.13 $C_3(p_{11}.. \ p_{1n}) \leftarrow C_1(p_{11}.. \ p_{1n}) \uplus C_2(p_{21}.. \ p_{2n})$
$typeOf(p_{1i}) = typeOf(p_{2i})$

Metadata processing. The new virtual class C_3 is constructed in the EGTV metamodel representation to contain the identical set of attributes as source classes C_1 and C_2.

Data processing. An object extent of the result class C_3 is created by merging object extents of C_1 and C_2 classes..

Intersection ($\cap$)

This is a binary operator that provides intersection of two object sets. The result is a new virtual class with only those objects that are equal in both source classes. This is illustrated in *Definition 5.14*. Since all objects in the EGTV model have distinct object identifiers, object equality is value based. If all attributes of two source objects are equal by value, then these objects are equal. Equality is tested using the equals (=) operator defined for each attribute type. All built-in types in the EGTV model have the equality operator. A default equality operator is generated for each user-defined class, but it can be redefined by a class designer.

Definition 5.14 $C_3(p_{11}..\ p_{1n}) \leftarrow C_1(p_{11}..\ p_{1n}) \cap C_2(p_{21}..\ p_{2n})$
$typeOf(p_{1i}) = typeOf(p_{2i}) \wedge p_{1i} = p_{2i}$

Metadata processing. Virtual class C_3 is constructed to contain result objects. Its metadata structure is identical to the structure of source classes C_1 and C_2.

Data processing. The object extent of the result class C_3 contains only those objects of C_1 extent for which equal objects exist in the object extent of a class C_2. The equality is tested using the equals (=) operators of class properties.

Difference ($\setminus$)

This operator implements set difference of two source classes. Both classes must have the same number of attributes, and attribute types must match. This operator is illustrated in *Definition 5.15*.

Definition 5.15 $C_3(p_{11}..\ p_{1n}) \leftarrow C_1(p_{11}..\ p_{1n}) \setminus C_2(p_{21}..\ p_{2n})$
$typeOf(p_{1i}) = typeOf(p_{2i}) \wedge p_{1i} \neq p_{2i}$

Metadata processing. This processing stage constructs a metadata representation of the result class C_3. It defines the identical set of attributes as source classes C_1 and C_2.

Data processing. The result is generated as an object extent of class C_3. It contains only those objects of class C_1 extent that are not present in the C_2 extent. This is evaluated by testing objects for equality. Two objects are equal only if all their properties are equal.

Distinct ($\oplus$)

The role of the distinct operator is to eliminate duplicate objects from the source class. This is an unary operator and its result is a new virtual class with no duplicate objects in its extent. This is illustrated in *Definition 5.16.*

Definition 5.16 $C_2(p_{11}..\ p_{1n}) \leftarrow \oplus\ C_1(p_{11}..\ p_{1n})$

Metadata processing. A virtual class C_2 with an identical structure to the source class C_1 is created in the EGTV schema repository.

Data processing. The object extent of result class C_2 is created to contain only unique objects, and discard all duplicates from the source class C_1. Uniqueness is defined based on the object equality, where two objects are equal if all their attributes are equal. This is a default equality which can be redefined for each user-defined class. When two or more objects are duplicated in the source class C_1, the distinct operator does not guarantee which objects are discarded and which are moved to the result class C_2. This is because object equality is determined by value and not by identity. Thus, duplicate objects are indistinguishable.

Union ($\cup$)

This operator performs an union operation on two object extents. It differs from the unionall operator as it eliminates duplicate objects from the result virtual class. Thus, this operator can be formally represented as a composition of unionall and distinct operators as illustrated in *Definition 5.17.*

Definition 5.17

$C_3(p_{11}..\ p_{1n}) \leftarrow C_1(p_{11}..\ p_{1n}) \cup C_2(p_{21}..\ p_{2n}) \Leftrightarrow \oplus\ (C_1(p_{11}..\ p_{1n}) \uplus C_2(p_{21}..\ p_{2n})\ )$
$typeOf(p_{1i}) = typeOf(p_{2i})$

Metadata processing. A result virtual class C_3 is created in the EGTV metamodel representation to contain an identical set of attributes as source classes C_1 and C_2.

Data processing. Object extents of two source classes C_1 and C_2 are merged, and then the distinct operator is applied to filter out duplicate objects. The result is generated as an object extent of the virtual class C_3.

5.5.4 Mapping Operators

The operator set defined in the EQL algebra can be used to formally represent any EQL query. Algebraic operators are directly mapped to algorithms in the query execution tree, where each algorithm defines one stage of the query processing. However, since the EGTV model is a canonical model and effectively a wrapper for persistent objects stored elsewhere, two additional operators have been defined. Their role is to provide on-demand materialisation of EGTV object extents and behaviour invocation. These operators are referred to as *mapping* operators, they are not part of the formal algebra and are defined only as algorithms in the query execution tree. These two operators are `extent`, and `eval`.

extent

`extent` is an unary operator whose role is to materialise EGTV objects and retrieve their metadata. This is then used as input to other operators in the query execution tree. The `extent` operator is always invoked at the start of query processing as it facilitates direct interaction with the EGTV model and the EGTV Schema Repository.

Metadata processing.

1. Class metadata is retrieved from the Schema Repository.
2. All superclasses of the retrieved class are then recursively flattened to create a new virtual class.
3. The result is returned as a virtual class (created in a previous step) containing properties from the base class and its superclasses.

Data processing. An object extent belonging to the virtual class constructed in the metadata phase is materialised from the database. This extent is then added to the class created in the metadata processing.

eval

The role of this operator is to provide an invocation interface for behaviour (methods and operators) defined in the built-in types and user-defined classes. Thus, this operator effectively acts as a wrapper for behaviour defined in EGTV model types.

5.6 Conclusions

The EGTV project involves the top-down design of a multimedia based federated database system. The top-down approach allowed us to choose target database systems and models, but the heterogeneity across the standard for O-O and O-R databases meant that neither query language met the requirements of a canonical query interface. Also, both ODMG OQL and SQL:1999 languages were unable to provide efficient representation and querying of complex multimedia contents. Thus, we developed a new query language using conventional syntax but with different semantics for specific query types.

In this chapter we demonstrated the EQL query language that facilitates querying of O-O and O-R schemas in a database and a platform independent manner. This built on our earlier work which specified a new metamodel and schema definition language to act as a wrapper for both database model types. In EQL, we preserved the familiar OQL syntax, but resolved some of the negative issues associated with OQL and made it orthogonal in terms of query input and output. EQL also provides a clear semantics for the updatability of the result set, facilitates primitives for object creation, update and deletion, and includes operation invocation support. EQL support for multimedia data types and ability to define custom operators are crucial when constructing and querying large multimedia database federations. Simple updatable views can be defined as stored EQL queries to provide schema restructuring functionality. This is later used for construction of federated schemas and updatable global queries. Global queries and views are fully explained in the next chapter.

The query algebra is a formal language that can unambiguously define database queries. The algebra itself is defined as a set of atomic operators that manipulate input sets of data to construct results. Each syntactically correct query can be represented as a composition of algebraic operators where the output of one operator forms the input for others. When inputs and outputs to algebraic operators are object structures, the algebra is considered as an object algebra.

An object algebra for the EQL language was formally defined in this chapter. The EQL algebra is fully orthogonal as its inputs and outputs are EGTV classes and thus, identical to inputs and outputs of the EQL query language. Each EQL query can be easily transformed to the algebraic representation. EQL algebraic operators are type independent and can be applied to to any built-in type and user defined class in the EGTV model. For each algebraic operator a formal definition has been provided, while its evaluation is discussed separately for data and metadata processing stages. This separation is important for the evaluation of stored queries (views) where the process of virtual class metadata construction is independent of any subsequent query materialisations. The main benefit of specifying a query language algebra is the ability to formalise query evaluation algorithms. Each EQL algebraic operator can be easily represented as an algorithm in the query execution tree. Therefore, we are now able to unambiguously represent EQL queries and define a methodology for their evaluation.

By defining a platform independent query language and an orthogonal object algebra, we are now in position to interrogate existing multimedia database schemas and populate them with new data. The next step will be the the specification of query processing semantics. In the next chapter we discuss an implementation of the EQL query processor service and introduce an architecture for defining and executing global queries.

Chapter 6

Implementation

The term interoperability is frequently used in research work into distributed and federated systems. Federated database systems are generally created from systems which were not designed to interact with outside processes, and the issue of accessing data is one of the first problems to face system engineers. This is further emphasised when federating large multimedia collections as they are commonly represented in proprietary data stores which are difficult to interface with and do not facilitate querying. Therefore, the EGTV project firstly defined canonical data and metamodels capable of capturing multimedia schemas, and then mapped them to standard object-oriented and object-relational databases. This provided a basis upon which our query language and its algebraic representation were constructed.

The role of a prototype is to explore the hypothesis and to validate research. Thus, it defines a test environment where experiments are conducted and performance results are assessed. These are then compared with the initial hypothesis of the thesis to prove the workability of research and identify areas for future improvements. The prototype system discussed in this chapter was based upon the new EGTV architecture which assumes that all databases in the system are available through a single common interface. In order to provide this level of interoperability, new services for query processing and schema definition are specified, while transaction control is provided to support write operations. However, the prototype developed to cover research presented in this thesis forms one segment of an overall EGTV project implementation. Other prototype systems developed within the EGTV are outside the scope of this research, but they provide some services used by our implementation. Specifically, implementation of the EGTV model and behaviour invocation is defined in [Kam04], and we use it as a basis upon which query and transaction processing is implemented.

The remainder of this chapter is structured as follows: §6.1 provides an overall deployment architecture for the EGTV federated system; In §6.2 definition of EGTV schemas is explained, while in §6.3 local and global query processing is discussed; §6.4 describes the

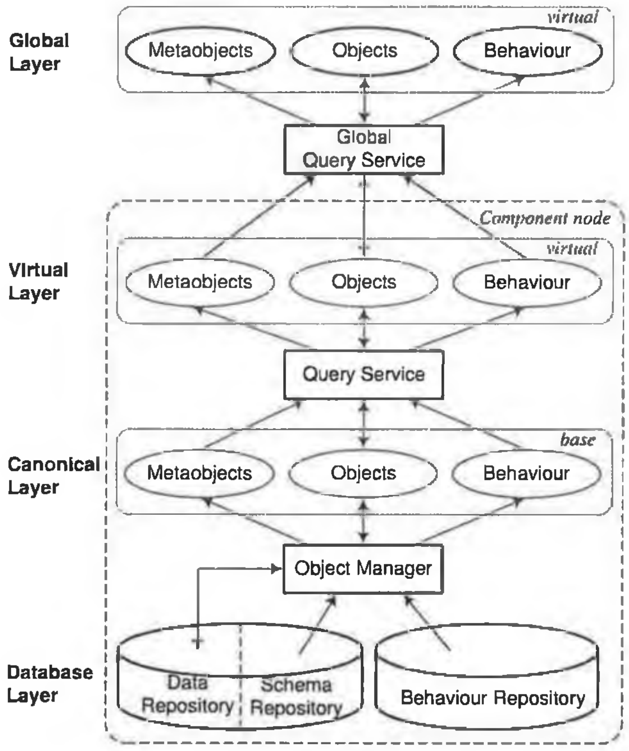


Figure 6.1: EGTV Deployment Architecture.

transaction system developed to support updatable queries; In §6.5 details of experiments are described, and in §6.6 some conclusions are drawn.

6.1 Deployment Architecture

The EGTV project facilitates the construction of a global schema for integrating different multimedia data sources into a database federation. It is based on the standard architecture for federated database systems [SL90] with some modifications required for multimedia data handling. The architecture can support large multimedia libraries integrated from many inexpensive general purpose databases, where the emphasis is on metadata and generic querying. Metadata has a significance as it is required for the construction of a global schema, while the query language facilitates generic schema interrogation at both local and global layers of the federation.

The architecture, illustrated in *Figure 6.1* consists of four layers, where each layer is defined in a form of database schema. Schemas are constructed and manipulated by processors that are located between layers. Our architecture differs from the generic five layer architecture [SL90], in several aspects. Firstly, multimedia data stores at the Database Layer are restricted to the ODMG object-oriented and object-relational databases. Secondly, server-side behaviour is an integral part of EGTV objects, and can be invoked from within the query language. Thirdly, the canonical schema is defined in the form of EGTV metamodel and EGTV model representations. It uses one processor (`Object Manager`) to interact with the data, metadata and behaviour repositories at the Database Layer. Objects instantiated from the canonical schema are represented in the platform independent EGTV model representation [KBR03]. Finally, the client interface is provided at both Global and Virtual Layers in a form of EGTV objects, and no External Layer is used.

- **Database Layer.**
  In the EGTV federation, all data is physically stored in either ODMG or object-relational databases at this layer. This does not constrain the application domain of this research as other projects (such as [RKB01a]) have shown that object wrappers can be used to represent data in legacy (non object) database systems and multimedia data stores to objects. Metadata is represented in the Schema Repository segment of the database, while behaviour is stored separately in a special Behaviour Repository.

- **Canonical Layer.**
  The Canonical Layer contains both data and metadata in a common EGTV representation. The canonical metaschema is represented in the EGTV metamodel format [RB02], while data objects and behaviour are provided in the EGTV model representation. This layer is the entry point for database schema definition and for local queries. From the user's perspective, the Database Layer is completely encapsulated and accessed through a single common interface.

- **Object Manager.**
  The `Object Manager` processor maintains objects at the Canonical Layer and interacts with the EGTV Schema Repository. It uses metadata information to transform persistent objects into EGTV base objects, and facilitates CRUD (create, retrieve, update and delete) functionality over EGTV objects. This processor is also responsible for propagating updates on the EGTV objects back to to the local database and maintaining transaction consistency. To implement behaviour requests, the `Object Manager` loads behaviour libraries from the Behaviour Repository and makes them available to EGTV objects.

- **Virtual Layer.**
  The Virtual Layer represents results of local queries and view materialisations. Thus, it contains virtual EGTV objects and their metadata definitions. Virtual objects are constructed from base EGTV objects at the Canonical Layer to which query processing transformations are applied. These transformations correspond to the EQL algebraic operators discussed in the previous chapter. Virtual EGTV objects are also used to represent that subset of the Canonical Layer schema to be shared within other databases. Thus, they effectively define an export schema which can be

further queried and updated from the Global Layer. Behaviour at the Virtual Layer can only be specified for views (stored queries). Virtual objects generated as results of ad-hoc queries cannot define new behaviour as their structure is not known in advance.

- **Query Service.**
  The Virtual Layer is maintained by the `Query Service` processor which similar to the `Object Manager`, provides a CRUD interface for virtual EGTV objects. This processor is responsible for query evaluation and construction of virtual EGTV objects. Virtual objects generated as query results are fully updatable and any update on their properties is directly propagated to the EGTV objects at the Canonical Layer on which they are based upon. Furthermore, behaviour invocations in virtual objects are processed through the behaviour interface at the Canonical Layer. The `Query Service` can form virtual objects based on other virtual objects thus making a recursive loop, in cases where the *source clause* of a query is another query.

- **Global Layer.**
  The Global Layer contains an integration of multiple Virtual Layer schemas. It stores global metadata in the EGTV metamodel representation and provides an access point for global queries. Virtual EGTV objects at this layer are either directly imported from different *Component Nodes*, or generated as results of global queries and views. Object interchange between the Global Layer and component nodes is facilitated through the CORBA based object exchange protocol. This interface creates a CORBA proxy for each imported virtual object to support updatability across different database nodes. Updates on global objects are propagated to their Virtual Layer counterparts, thus maintaining the overall data consistency. The CORBA bridge does not form part of this research and is covered in [Kam04].

- **Global Query Processor.**
  The `Global Query Service` facilitates the construction of global schemas and provides query and transaction interface for global clients. It defines the same set of services as the local `Query Service`, with the addition of global transaction control. The architecture and implementation of the EGTV transaction system is discussed later in this chapter.

6.2 Schema Definition

The EGTV schema definition process creates a Canonical Layer database and defines metadata in the *EGTV Schema Repository*. Thus, data and metadata are represented in a canonical format that can be queried and updated from any EGTV application. Depending on the existence of local data, each canonical database can be defined in either a bottom-up or top-down approach. The bottom-up schema definition transforms existing

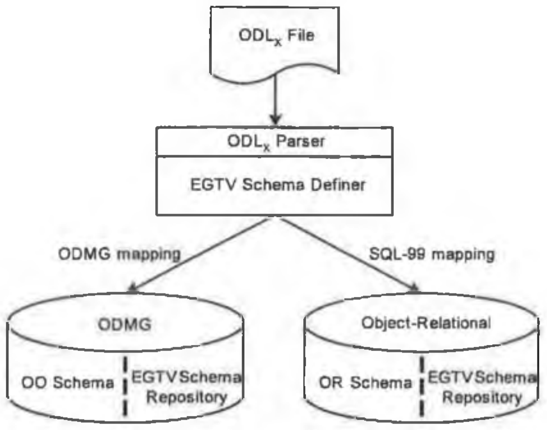


Figure 6.2: Schema Generation Process.

database schemas to the canonical EGTV representation, while the top-down approach creates new schemas at both Canonical and Local layers.

- **Bottom-up schema definition.**
  Database schemas are firstly specified in the native Data Definition Language (DDL) of object-oriented or object-relational database. These DDL definitions are then processed to create classes, properties, relationships and other metadata elements in the Schema Repository. Finally, metamodel mapping rules (discussed in chapter three) are applied to transform these schema definitions from the proprietary database metamodel to the EGTV metamodel representation.

- **Top-down schema definition.**
  The database schema metadata is firstly created in the EGTV metamodel representation using the platform independent data definition language ODL$_x$ discussed in chapter four. Metamodel mapping rules are then applied to transform metadata from the EGTV representation to the object-oriented or object-relational database metamodel, and to create database schemas.

At present, only a top-down schema definition process is implemented. This is because the existing multimedia data is not stored in queryable databases, but in a specialised file-based repository. Therefore, new multimedia database schemas cannot be reused, as they must be generated from "scratch" in a top-down approach.

The existing prototype for the top-down schema definition process is illustrated in *figure 6.2*.

A database schema is first specified in a platform neutral ODL$_x$ schema definition language, and then stored in a file. This file is processed by the `ODL`$_x$ `Parser` processor, which reads the ODL$_x$ specification of the schema and transforms it to in-memory object tree representation. Since the ODL$_x$ language is encoded in a standard XML format, it can be easily parsed using the standard DOM (Document Object Model) libraries for XML. We use the Apache XERCES implementation of DOM in C++, as it provides portability across different operating systems. The ODL$_x$ schema file is checked for syntax errors by validating its DOM representation against the XML Schema specification of ODL$_x$ language.

The syntactically correct database schema in an object tree representation is the input for the `EGTV Schema Definer` processor. Its role is to create database types and insert metadata to the Schema Repository. Semantic validation of the database schema against the metamodel definition (specified as an XML Schema document) is also performed by this processor. Since the ODMG and O-R databases differ in data model and schema definition interface, two methods of schema and metadata generation were developed. The first one is implemented for a Versant O-O database, while the second method is developed for the Oracle 9i O-R database.

For ODMG databases, the `Schema Definer` creates an object-oriented schema as a database import file, containing the Versant specific definition of database classes and their structures. This file is then loaded to the database, to create the Versant (ODMG) database schema consisting of persistent classes and relationships between them. For O-R databases, the `Schema Definer` generates a sequence of SQL `CREATE` statements. These statements conform to SQL:1999 syntax for object types and object tables. The generated SQL is saved to an SQL script file which is then executed against database to create an Oracle 9i (O-R) database schema.

After the schema is successfully created, the `Schema Definer` generates metadata from the ODL$_x$ specification and inserts it into the EGTV Schema Repository database segment. Translation from the ODL$_x$ schema definition to the EGTV metamodel (stored in the EGTV Schema Repository) is a straightforward process where each ODL$_x$ element (i.e. `dbSchema`, `class`, `attribute`) is mapped to one metamodel class as defined in ODL$_x$ to metamodel mappings discussed in chapter four.

View definitions are represented in the ODL$_\mathrm{x}$, and thus processed by the `Schema Definer` processor together with base schema definitions. The `Schema Definer` parses the view definition and transforms it to the EGTV metamodel representation which is then stored in the EGTV Schema Repository. This information is later used by the `Query Processor` to materialise view extents.

In terms of platform, Versant (ODMG) and Oracle 9i (O-R) databases were used on both Windows XP and Linux. The schema definition prototype was fully developed using standard C++ and STL (standard Template Libraries). The ODL$_x$ definitions are processed using the Apache XERCES DOM parser, while database access is facilitated using OCCI (Oracle C++ Call Interface) object library for Oracle 9i and ODMG C++ mappings for Versant database.

6.3 Query Processing

EQL queries are processed by `Query Service` processors illustrated in *figure 6.1.* These processors are categorised as local and global ones, where local query processors evaluate queries at the canonical layer, while global query processing manipulate object extents from multiple distributed sources and includes global transaction control.

6.3.1 Local Query Processing

The processing of a local EQL query can be divided into three stages: query parsing, execution tree construction, and result materialisation. In the parsing stage, the syntactic and semantic validation of the EQL query is performed. The second stage translates a query to its algebraic representation. This effectively generates an execution tree and invokes metadata processing to determine the metadata for the result class. The final stage generates the result by applying the operations defined in the execution tree to class extents.

In terms of platform, the local query processor is implemented as a C++ service that receives EQL queries in a string format as input and generates a virtual EGTV class as a result. The result class consists of an object extent and metadata definition for objects in that extent. EGTV objects and metadata are created and manipulated using EGTV model and metamodel API which fully encapsulates Local Layer databases. Therefore, the query processor is platform independent, as it manipulates only canonical EGTV objects. An ANTLR parser [ANT02] is used for parsing of query definitions and generation of the syntactic tree. The EQL grammar definition in ANTLR BNF syntax is provided in *Appendix G.* Query evaluation is implemented entirely in standard C++, where STL containers are used for in-memory representation of object collections and query execution tree manipulation.

Query Parsing

- input: EQL query string
- output: EQL syntactic tree

In the first stage of processing, an EQL query string is parsed to the syntactic tree representation. The nodes of the tree are class, property and operation names grouped in three branches: `select`, `from`, and `where`. *Example* 6.1 illustrates a simple EQL query which is transformed to the syntactic tree represented in *Figure 6.3.* During the construction of the tree, the query is syntactically validated. Syntax validation checks if the syntax of the EQL query string is correct and in accordance with the EQL language specification. The result of the parsing stage is an EQL syntactic tree. The benefit is that the query string is tokenised and transformed to the tree representation suitable for further processing.

```
select User.name as userName,
       Program.name as programName
from   User connect Program on User.recordings
where  User.ratingLimit() > 5;
```

Example 6.1: Query processing example.

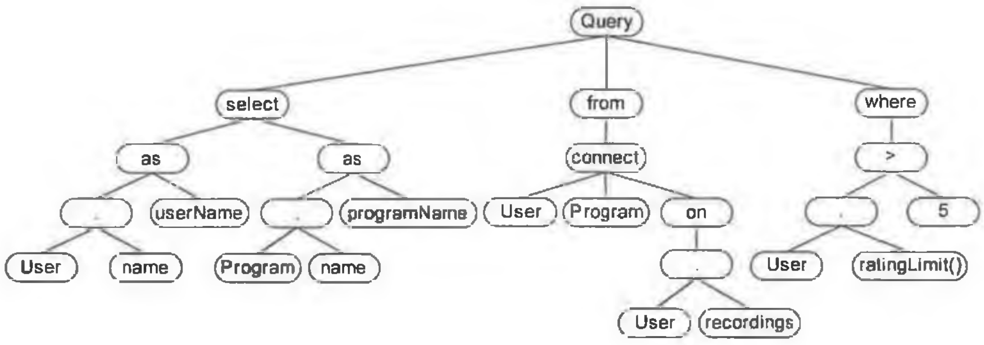


Figure 6.3: Syntactic tree example.

Execution Tree Construction

- input: EQL syntactic tree
- output: Algebraic execution tree

In the second stage of processing, the query in the syntactic tree is transformed to its algebraic representation. The transformation process starts by examining the `from` branch of the syntactic tree and resolving class names to the virtual classes in the Schema Repository. These classes represent the starting point for query evaluation. EQL algebraic operators are successively applied until the virtual class representing the query result set is constructed. In this process the whole syntactic tree is traversed and all EQL syntactic elements (select, from, where, path navigation, join, and behaviour invocations) are transformed to algebraic operators and represented as the query execution tree illustrated in *Figure 6.4*. This effectively performs semantic validation of the query to check if types used in the query exist in the Schema Repository and if operations are compatible with these types. The execution tree can be optimised for speed and resource utilisation, but this would form part of future research for this project. The benefits of this stage are the following: semantic analysis of the query; creation of the virtual class representing the query result type; and query execution tree construction.

Result Materialisation

- input: Algebraic execution tree

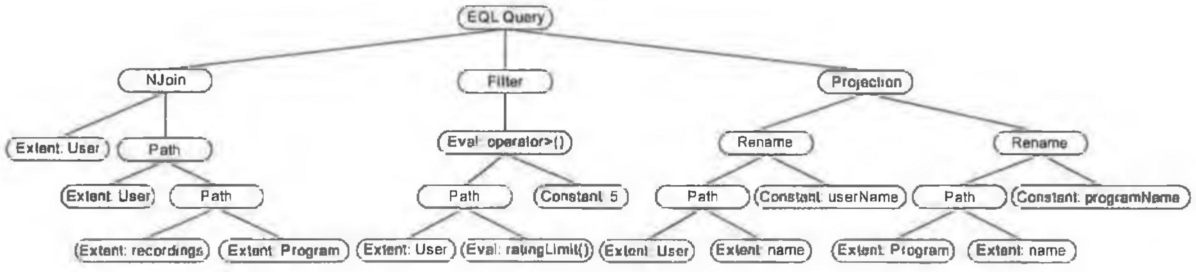


Figure 6.4: Execution tree example.

- output: Virtual class representing the query result

In the materialisation phase, the execution tree is processed and a query result generated. The processing starts with the materialisation of all class extents required as inputs for algebraic operators. Then, operations defined in the execution tree are sequentially applied (post-order traversal method) to create the final result. During this process, temporary objects can be created as intermediary results. This is because each algebraic operator returns a result as a virtual class which subsequently becomes an input for other operators. However, when the final result is generated, all intermediary results (temporary objects) are closed. The final result is a single virtual class. The metadata component of the virtual class was defined in the previous (EQL algebra transformation) processing step, while this step constructs its object extent. Properties of result objects directly reference corresponding properties of the source EGTV objects providing updatability.

6.3.2 Global Query Processing

Global queries are specified at the Global Layer of our federated architecture using the same EQL syntax as local queries and processed by the `Global Query Service` processor. They are evaluated against a global schema constructed using views imported from different component databases. In this section the construction of the global schema is first explained and then the generation of the global query extent is discussed.

Global Schema Construction

The construction of the global schema begins at each component node where a set of views is defined at the Virtual Layer. These views represent export schemas [SL90] which can be accessed and queried from the Global Layer. They are defined as EQL queries, and their result is represented as a virtual class. This is illustrated in *Figure 6.5*, where `FilmProgram` (*Example* 6.2) and `TVRecording` (*Example* 6.3) views are defined in two different databases of the *Físchlár* system. These are based on `Program` and `Film` classes as represented in processing step (1) of *Figure 6.5*. The UML definition of two

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="MultimediaRecordingSystem" databaseType="OO">
  <virtualClass name="FilmProgram">
    <extent><![CDATA[
        select name as filmName, description
        from Program p
        where p.categoryRef.Category.name = "Film";]]>
    </extent>
    <method name="FilmProgram" accessKind="public">
      <parameter name="filmName" constant=true>
        <primitiveType name="string"/>
      </parameter>
      <parameter name="description" constant="true">
        <primitiveType name="string"/>
      </parameter>
      <parameter name="category" constant="true">
        <primitiveType name="string"/>
      </parameter>
    </method>
    <method name="~FilmProgram" accessKind="public">
    </method>
  </virtualClass>
</dbSchema>
```

Example 6.2: An ODL$_x$ definition of the FilmProgram virtual class.

canonical database schemas is provided in *Appendix E*. The global schema is then defined by importing view metadata (2) to the Global Layer. Other views can be constructed upon the imported ones, thus providing the restructuring of the global schema. A global view RecentFilm (*Example* 6.4) is defined in the global schema as an EQL query that joins FilmProgram and TVRecording imported classes (3).

Extent Generation

The process of global query evaluation is the same as for local queries, and it follows the three processing stages defined in §6.3.1. The only difference is in the materialisation of extents for imported virtual classes (views). The extent of each imported class is first materialised at its local node by evaluating its EQL query definition, and is then imported to the global schema. This is represented as step (4) in *Figure 6.5.* Properties of objects in imported extents are only references to corresponding objects materialised at local nodes, so performance and memory overhead is minimal. Properties are dereferenced and physical data is passed to the Global Layer only when required for query processing. The interconnection between different nodes and the exchange of objects is facilitated through the CORBA service. Any update to objects in imported extents are propagated through

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="VideoArchive" databaseType="OO">
  <virtualClass name="TVRecording">
    <extent><![CDATA[
        select name, recordingDate
        from Film
        where Film.langRef.Language.name = "French";]]>
    </extent>
    <method name="TVRecording" accessKind="public">
      <parameter name="name" constant=true>
        <primitiveType name="string"/>
      </parameter>
      <parameter name="recordingDate" constant="true">
        <primitiveType name="date"/>
      </parameter>
      <parameter name="language" constant=true>
        <primitiveType name="string"/>
      </parameter>
    </method>
    <method name="~TVRecording" accessKind="public">
    </method>
  </virtualClass>
</dbSchema>
```

Example 6.3: An ODL$_x$ definition of the TVRecording virtual class.

references to original objects at component database node. This is a feature of EGTV model and was explained in chapter five. Global transaction consistency is guaranteed by the two-phase commit protocol.

For this research only a limited prototype of the Global Query Service processor has been implemented. It facilitates query evaluation and transaction control system, but all objects must originate from a single database. This is due to the fact that CORBA object exchange segment of the EGTV architecture is part of external research [Kam04] and not completed yet, thus data interconnection between multiple databases cannot be facilitated. However, it is expected that the full implementation of this system will be available in a near future. Therefore, this protocol will be used as a basis upon which our data distribution features are built.

6.4 EGTV Transaction Processing

Efficient transaction control is the important requirement for every multiuser database system. This issue is even more emphasised in federated database systems where transaction consistency must be preserved across multiple heterogeneous databases. The EGTV feder-

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema name="GlobalSchema" databaseType="OO">
  <virtualClass name="RecentFilm">
    <extent><![CDATA[
        select t.name, f.description
        from FilmProgram f join TVRecording t on f.filmName = t.name
        where recordingDate > "01/03/2004";]]>
    </extent>
    <method name="RecentFilm" accessKind="public">
      <parameter name="name" constant=true>
        <primitiveType name="string"/>
      </parameter>
      <parameter name="description" constant="true">
        <primitiveType name="string"/>
      </parameter>
      <parameter name="language" constant="true">
        <primitiveType name="string"/>
      </parameter>
      <parameter name="recordingDate" constant="true">
        <primitiveType name="date"/>
      </parameter>
    </method>
    <method name="~RecentFilm" accessKind="public">
    </method>
  </virtualClass>
</dbSchema>
```

Example 6.4: An ODL$_x$ definition of the `RecentFilm` virtual class.

ated architecture integrates transaction control mechanisms with the new reference-based EGTV model and the EQL query language. Here we introduce the EGTV transaction model and explain the global and local transaction processing. Our transaction model facilitates transaction scheduling and extends the standard locking mechanism by introducing write-copy lock.

6.4.1 Transaction Model

A transaction encapsulates a set of operations executed on database objects, and is always processed atomically, meaning that all operations in transaction are successfully executed or none are. Operations perform retrievals and updates on database objects and can take the form of EQL queries or behaviour invocation. Each transaction is started with the `begin_transaction` and ended with the `commit` or `rollback` command.

The EGTV transaction architecture is illustrated in *figure 6.6*. The `Global Query Service` receives global transactions, partitions them into a set of subtransactions and

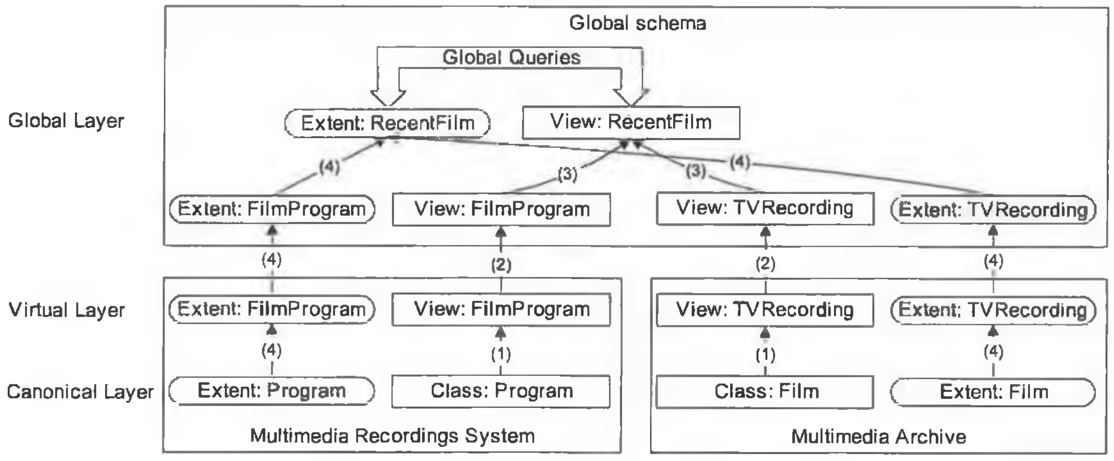


Figure 6.5: Global query processing.

then submits them to multiple `Local Query Services` for execution. It is responsible for ensuring the atomicity and serialisability of global transactions. The `Local Query Service` executes local transactions and global subtransactions at each participating database node. Its responsibility is to guarantee transaction consistency by applying locks on persistent objects in the database. The database must be ODMG or object-relational compliant, and must have a transaction interface available to users. Although some databases provide a *nested transaction* interface, this is generally not supported in available ODMG [CB99] and SQL-99 [GP99] compliant databases. Therefore, our implementation supports only flat transactions, although nested transaction interface will be investigated in other EGTV research projects.

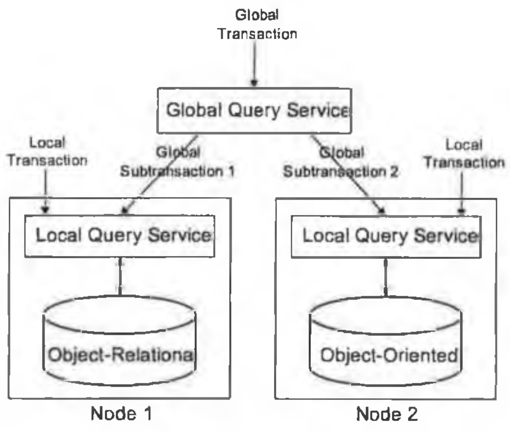


Figure 6.6: EGTV Transaction Architecture.

6.4.2 The EGTV Object Pool

Object materialisation denotes the process of creating EGTV objects as a canonical representation of persistent objects in an O-O or O-R database. EGTV objects are materialised in the *EGTV Object Pool* and are not shareable with other concurrent sessions. This is required for guaranteeing atomicity of concurrent transactions. The EGTV Object Pool contains all EGTV and virtual EGTV objects materialised in one client session. One Object Pool is created per each client session and all objects in the pool belong to the same transaction. Persistent objects from the Storage Layer can be simultaneously materialised in multiple object pools, but EGTV objects in one pool are inaccessible to other object pools. If an object is materialised as updatable in one pool, it is read-only in other pools to preserve transaction atomicity. The query processor is a client of the EGTV Object Pool and can read and update any object in the pool. When the client (query processor) attempts to access a non-materialised single object or full object extent in the EGTV Object Pool, the automatic materialisation procedure is triggered. This includes materialisation of one EGTV object for each persistent object in the database and applying transaction control (locking) mechanisms. The query processor can also explicitly close objects materialised in the pool. This process includes writing of all modified objects back to the Storage Layer database and deallocating EGTV objects from the pool. When client session is closed, all objects in the pool are automatically closed.

6.4.3 Requirements

Both, the `Local Query Services` and the `Global Query Service` must provide a set of services required for atomic processing of global and local transactions.

Local Query Service Requirements

1. Strict two-phase locking protocol [BH87] supported in each object-oriented and object-relational database at the Storage Layer.
2. Each database must provide transaction interface which supports `begin_transaction`, `commit` and `rollback (abort)` operations.
3. Database supports visible `prepare-to-commit` state, or simulated `prepare-to-commit` state [Geo91] is provided by the `Local Query Service`.
4. Database notifies the `Local Query Service` of any unilateral transaction abort.
5. Database transaction interface must support per-object locking granularity.

Global Query Service Requirements

1. There is at most one subtransaction per component database node for each global transaction.
2. Two-phase commit protocol is supported [OV99].
3. Support for *Cascadless Ticketing* [GRS94] global serialisability protocol.

6.4.4 Lock Types

The `Local Query Service` is responsible for persistent object locking. It invokes the transaction interface provided by the database directly to set and release locks on persistent objects. Locking granularity is per object, which means that locks can be applied to a single object or to an object set of any cardinality. Two types of object locks can be set by the `Local Query Service`: *read* and *write-copy* locks.

Read lock. This lock maps directly to the shared (read) lock as defined in the two-phase locking protocol. Multiple non-conflicting read locks can be applied to the same persistent object, but a read lock conflicts with a write lock. All read locks are applied to persistent objects before they are materialised at the Canonical Layer. This ensures the materialisation of consistent set of EGTV objects.

Write-copy lock. The write-copy lock is the enhancement of the standard write lock in two-phase locking protocol. It exclusively locks a persistent object in the database and then creates a copy of its original state (before locking). The `Local Query Service` applies write-copy lock by materialising two EGTV objects for each persistent object locked in the database. The first EGTV object is updatable, it can be modified by the transaction, and all updates are propagated to the persistent object. The second object cannot be modified, and it is used as a read-only data source for other transactions. Thus, transactions that request read-only access to exclusively locked objects are able to proceed without waiting for locks to be released. The overall transaction concurrency is increased since transactions that otherwise would have to be serialised are simultaneously executed without violation of data integrity. An example illustrating this feature is given in §6.4.6.

6.4.5 Transaction Types

The EGTV model defines two types of transactions: *read-only* and *read-write*.

Read-only transactions do not acquire any locks on the database objects, except for a short interval of time while EGTV objects are materialised at the Canonical Layer. Persistent objects in the local database are temporally locked in the read (shared) mode to ensure the consistency of the retrieved object extent. All locks are released immediately after the EGTV objects are materialised. The read-only transaction effectively creates a database snapshot which is not updatable. The materialised EGTV objects are not bound to objects in the database.

Read-write transactions are required for query updates, or when the client application requests the retrieval of updatable query result. A read-write transaction acquires write-copy locks on all persistent objects retrieved by the query. These locks can be released only when transaction commits or aborts. The EGTV objects materialised at the Canonical Layer are bound to the corresponding persistent objects in the database for the whole duration of the read-write transaction.

The formal syntax for EQL transactions is presented in *Definition 6.18.* A sequence of EQL queries is enclosed within the `begin_transaction` and `end_transaction` block. Read-only transactions must not contain any query which modifies database state. *Example 6.5* illustrates a typical read-only transaction defined as a sequence of EQL queries.

Definition 6.18 *transaction := <begin_transaction>*
*<EQL Query>**
<end_transaction>

<begin_transaction> := ***transaction read-only*** | ***read-write***
<end_transaction> := ***commit*** | ***rollback***

```
transaction read-only;
  select name, description from Program;
  select name, login from User;
commit;
```

Example 6.5: Read-only transaction example.

6.4.6 Local Transactions

The `Local Query Service` materialises EGTV objects and binds them to the persistent objects in the database. It is responsible for maintaining data consistency of locally executed transactions by applying locks to persistent objects in the database. The locking mechanism is not applicable to transient objects. The EGTV transaction model guarantees statement-level read consistency, which means that all the data that the query sees come from the single point of time.

EGTV objects can be materialised from persistent database objects using either *extent* or *navigational* materialisation.

Extent materialisation materialises a full class extent into a set of EGTV objects at the Canonical Layer. The extent to be materialised is specified in the source specification (`from` clause) of the EQL query. Depending of the type of transaction, the appropriate locking mechanisms are applied to the materialised extent.

Navigational materialisation is applied only when navigating between objects using references. An attempt to navigate from materialised EGTV object to a non-materialised EGTV object, or a set of objects (`1-many` relationships) triggers the navigational materialisation. When EGTV objects are materialised, the appropriate locking mechanisms are applied to their corresponding persistent objects in the database. Persistent objects are accessed directly through object references, and no queries are used during the retrieval.

Local Transaction Processing

Each transaction executed by the Local Query Service is subject to strict locking rules. The execution of any transaction can be generalised to following set of steps that guarantees data consistency and transaction atomicity.

1. The transaction is started at the local database by invoking the `begin_transaction` command of the database transaction interface.
2. Each query in the transaction is processed by the `Local Query Service`. The processing returns result as an extent of EGTV objects and includes the following steps.
   (a) The full extents of all classes specified in the `from` clause of the query are locked in the database and EGTV objects are materialised at the Canonical Layer. Read locks are used in read-only transactions, and write-copy locks in read-write transactions. Read locks materialise one non-updatable EGTV object for each persistent object locked in the database. Write-copy locks materialise one updatable, and one read-only EGTV object per a persistent object. The read-only object preserves the original state of the persistent object and it can be read by other transactions. The updatable object propagates all updates to the persistent object in the database and its state cannot be accessed from other transactions.
   (b) A restriction condition is applied to materialised EGTV object extent. EGTV objects not satisfying the condition are closed and locks on their persistent object counterparts are released. This is not a violation of the locking protocol, since the released objects have not been modified since the transaction started. However, when an EGTV object is modified, locks on its persistent object counterpart must not be released until the transaction is committed or rolled back. This is in accordance with the separation of growing and shrinking phases in the strict two-phase locking protocol [BH87].

   (c) Updates specified in the query are applied against EGTV object extents filtered in the previous step. The result set generated as a set of EGTV objects is returned to the issuing client where it can be further updated. Any updates on the result set objects are propagated to the persistent objects in the database.

3. The transaction is completed by issuing `commit` or `rollback` commands. Commit flushes all cached updates from EGTV objects to persistent database objects and commits transaction in the database. This effectively releases all locks held by the transaction, while EGTV objects are invalidated and closed. Rollback closes all EGTV object without flushing the changes and releases locks on the persistent objects in the database. The database is restored to the state before the start of transaction. This is guaranteed by the transaction interface of the local database.

Transaction Scheduling

Scheduling improves the overall transaction concurrency by allowing simultaneous execution of multiple transactions on the same class extent. The only condition is that object sets locked by different transactions must be disjoint. Let `Ti`, `Tj` and `Tk` be independent transactions operating on the `Ec` class extent. Transaction `Ti` starts its execution before `Tj`, which precedes `Tk`. `Ti` and `Tj` operate on disjoint subsets `Ei` and `Ej` of `Ec` class extent, while extent `Ek` of transaction `Tk` overlaps with the `Ei` extent. The remaining (unused) subset of `Ec` extent is denoted as sub-extent `Er`. Read-only extents are denoted with `r-o` subscript (i.e. $Ei_{r\text{-}o}$).

```
Ti < Tj < Tk
Ei ∩ Ej = ∅
Ei ∩ Ek ≠ ∅
Ec = Ei ∪ Ej ∪ Ek ∪ Er
```

The execution of `Ti`, `Tj` and `Tk` transactions is represented as a sequence of time intervals (`t0..t5`).

```
t0: Ti started; Ec = Ei ∪ Er
t1: Ti processing, Tj started; Ec = Eir-o ∪ Ej ∪ Er
t2: Ti processing, Tj processing, Tk blocked; Ec = Eir-o ∪ Ejr-o ∪ Er
t3: T1 completed, Tj processing, Tk started; Ec = Ejr-o ∪ Ek ∪ Er
t4: T1 completed, Tj completed, Tk processing; Ec = Ekr-o ∪ Er
t5: T1 completed, Tj completed, Tk completed; Ec
```

At time `t1`, transaction `Tj` was able to start because it had the full access to the `Ec` extent (including the read-only sub-extent $Ei_{r\text{-}o}$) and it attempted to lock only unused objects (`Ej` sub-extent) from the Ec class extent. The `Tk` transaction was blocked in `t2` although it had read-only access to the full `Ec` extent, but it required locking of objects already locked by transaction Ti (Ei and Ek object sets were not disjoint). Tk was serialised after the Ti and it was allowed to resume execution in t3, when Ti completed and released locks on objects it held.

Local Transaction Example

This example demonstrates the usage of write-copy locks to facilitate concurrent execution of two read-write transactions operating on a disjoint set of objects from the same extent. The persistent class X containing one attribute a of type int is defined in the local database. The extent of the class X contains the following values (objects): X{1, 2, 3, 4, 5, 6, 7}. Two read-write transactions T1 and T2 concurrently update the values of the extent X. The transaction T1 decrements all objects which are less than 4, while the transaction T2 increments all X objects greater than 4. This is illustrated in *Table 6.1.* Transactions are processed by the Local Query Service which utilises the write-copy lock mechanism.

Transaction T1	**Transaction T2**
transaction read-write;	transaction read-write;
select a := a - 1	select a := a + 1
from X	from X
where a < 4;	where a > 4;
commit;	commit;

Table 6.1: Concurrent Transaction Schedule.

Suppose the transaction T1 was submitted first, but before it completed, the transaction T2 had been started. The local transaction schedule is defined as: { update(T1), update (T2), commit(T1), commit(T2) }. This execution can be represented as the following sequence of steps.

1. T1: The full extent X is materialised at the Canonical Layer, and all persistent objects of class X are exclusively locked in the database.
2. T1: The read-only copies of the EGTV objects {1, 2, 3, 4, 5 ,6 ,7} are constructed at the Canonical Layer. These *copy* objects retain the original object state, before any changes are applied by transaction T1.
3. T1: The restriction condition is applied. Those EGTV objects with a value a not less than 4 are closed and locks on the corresponding persistent objects are removed. The objects {1, 2, 3} remain locked.
4. T1: EGTV objects are updated, and their values are decremented by 1. The result set of the transaction T1 is generated as {0, 1, 2}.
5. T2: The materialisation of the full extent X is requested by the transaction T2. Since one segment of this extent is still locked by transaction T1, only available objects

(unlocked) will be materialised for transaction `T2` and locked in the database. The write-copy locks are applied.

6. `T2:` The transaction `T2` applies its restriction condition. It is evaluated against the objects materialised and locked by transaction `T2` and read-only copy objects created by transaction `T1` in *step 3*. The evaluation concludes that only objects `{5, 6, 7}` are required for transaction `T2`, while the others can be released. The transaction `T2` is allowed to continue because objects locked by the transaction `T1` are not required for execution of `T2`.

7. `T1:` Transaction `T1` commits and releases locks held on permanent objects in the database. EGTV objects are invalidated and closed. The new state of database is `{0, 1, 2, 4, 5, 6, 7}`.

8. `T2:` transaction `T2` commits, releases locks and closes EGTV objects. The database has the following state: `{0, 1, 2, 4, 6, 7, 8}`.

If the restriction condition of transaction T2 had been different and had evaluated to objects locked by transaction `T1`, the transaction `T2` would have been blocked and would have to wait until the `T1` commits and releases the locks on objects it was holding. Any deadlock situation involving two or more transactions is detected by the locking system of the local database and conflicting transactions are aborted.

Implications of the EGTV Locking System to Transaction Concurrency

Read-write transactions always lock the full class extent in the database and then apply restriction conditions. This approach can limit the overall transaction concurrency, so we introduced the write-copy lock. This lock allows multiple non-conflicting read-write transactions to simultaneously operate on the same extent. However, conflicting transactions attempting to lock the same objects are still serialised.

Read-only transactions also apply locks to full class extents. However, this does not introduce significant concurrency degradation since all persistent objects are locked in the shared (read) mode, and read-only locks are released immediately after all EGTV objects are materialised. Read-only transactions do not have to wait for objects locked by the read-write transactions, since they are able to access and use their read-only object copies.

6.4.7 Global Transactions

The `Global Query Service` executes global transactions and provides global atomicity and serialisability. Global atomicity ensures that all or no subtransactions are successfully committed. The serialisability of global transactions is preserved only if their subtransactions are committed in the same relative order at all participating nodes. The global atomicity protocol used in the EGTV transaction model is the two-phase commit [OV99], while the global serialisability is enforced by the *Cascadless Ticket Method* [GRS94].

Atomicity. The two-phase commit is a two-stage global commitment protocol. After subtransactions are submitted to participating nodes, the Global Query Service sends a `prepare` command to all Local Query Services. The subtransaction is in the prepared state only if it is fully executed and is guaranteed to commit. Each Local Query Service, executes subtransaction and sends a `prepared` or `not-prepared` response to the Global Query Service. If all participating nodes respond with `prepared` message, the global command is `commit`, otherwise all nodes are instructed to `abort` their subtransaction and roll back any database modifications. The global transaction is committed when all participating nodes `commit` or `abort` their subtransactions.

Serialisability. Local databases maintain only their local transaction schedules and are not aware of the global schedule. This is the reason why global serialisability must be controlled externally by the `Global Query Service`. The serialisability of global transactions in the Cascadless Ticket Method is preserved by forcing direct conflicts between their subtransactions on the each participating database node. All subtransactions compete to acquire a ticket object and the order in which they take the ticket represents the serialisation order of subtransactions at the participating node. If relative serialisation order of global transactions at all participating nodes is the same, the serialisability of global transactions is preserved and global transactions are allowed to commit, otherwise conflicting transactions are aborted and re-executed.

The execution of a global transaction is distributed between the `Global Query Service` and participating database nodes. A global transaction is processed by the `Global Query Service`, while the `Local Query Services` at participating nodes execute its subtransactions. Different algorithms are applied for processing of read-only and read-write subtransactions. This is due to differences in locking mechanisms between these two subtransaction types.

Global Transaction Processing

1. A global transaction is broken into a set of subtransactions, where each subtransaction is aimed for a different database node.
2. The transaction timer is started. It defines the time interval in which each subtransaction must complete its execution. This feature is important for detecting global deadlocks caused by different serialisation schedules at different participating nodes. If the timeout expires, the global transaction is aborted and restarted.

3. Subtransactions are submitted for execution to participating nodes.

4. The `Global Query Service` waits until the response to `prepare` command is received from all `Local Query Services` or until the timer timeouts.

5. If all subtransactions responded with the `prepared` message, the global command is `commit`. If any of the subtransactions is not ready or the timeout has been reached while waiting for response, all `Local Query Services` are instructed to `abort` its subtransactions and global transaction is restarted.

Read-Only Subtransaction Processing

1. Acquire read (shared) locks on all extents accessed by subtransaction.

2. Materialise EGTV objects at the Canonical Layer.

3. Filter the EGTV object extent and close all EGTV objects not satisfying the filtering condition.

4. Release all read locks.

5. Take a ticket (attempt to acquire a write lock on the ticket object). This step is required to guarantee correct serialisation order of global transactions.

6. If the ticket was successfully acquired, the subtransactions is prepared for global commit. The `prepared` message is sent to the `Global Query Service`.

7. The `commit` or `abort` command is received from the `Global Query Service`. This initiates the commit or rollback in the database and the release of the ticket object. All EGTV objects are closed.

Read-Write Subtransaction Processing

1. Acquire a write-copy lock on all extents accessed by the subtransaction.

2. Materialise EGTV objects at the Canonical Layer.

3. Process subtransaction queries and selectively release objects not required for processing. All objects released before the commit must not be modified by the subtransaction.

4. Perform updates on database objects as specified in subtransaction definition.

5. Take a ticket (attempt to acquire a write lock on the ticket object).

6. If successful, signal the `prepared` state to the `Global Query Service` and wait for a global command.

7. If the `commit` command is received, the subtransaction is committed and all changes are made persistent. The `abort` command triggers the rollback of all modifications and the database is returned to the state prior to the start of subtransaction. In both cases, the ticket object is released and EGTV objects closed.

6.5 Experiments

In this section, a discussion on the performance of schema definition and query processing is presented. All experiments were conducted within a laboratory environment where a limited amount of multimedia data was extracted from the *Físchlár* system [LSO+00] and inserted to the Versant O-O database. While this does not reflect all working environments, it does demonstrate that integration using the EGTV metamodel is possible, updates through query results are feasible, and when outside influences are controlled, the materialisation of most EGTV objects is achieved at an acceptable performance level. The results of experiments are categorised into the times taken to define the database schema from the ODL_x specification file, and the times taken to evaluate query results once data resides in a Local Layer database. Due to the unavailability of the CORBA based object exchange protocol (developed within a separate project), we were not able to perform tests involving multiple databases in the system. Thus, all results presented in this section reflect only a single database acting as both component and federated node of the architecture.

Three separate database users designed schemas according to their different goals and design patterns: multimedia recording system, multimedia editing system and multimedia archive database. The UML diagrams for these three schemas are provided in *Appendix E*. The multimedia content was mainly drawn from the *Físchlár* system, although once stored locally, it was possible for users to manipulate the data set according to their own needs. In terms of platform, Versant 6.0 (ODMG) and Oracle 9i (O-R) databases were used on both Windows XP and Linux. The existing prototype allows users to define database schemas on both target platforms, although the query interface is currently developed only for Versant databases running on Windows platforms.

6.5.1 Schema Definition Tests

Schema definition testing was based upon the database schema for the Multimedia Recording System that contains 11 classes, 16 attributes, 11 bidirectional associations and 2 generalisation relationships. It also defines 1 behaviour operator and 16 behaviour methods. This schema was fully defined in ODL_x and then processed by the `SchemaDef` program to create both Versant and Oracle 9i database schemas. The complete ODL_x definition of the schema is provided in *Appendix F*. The Versant implementation of the `SchemaDef` was developed for the Windows platform, while the Oracle 9i implementation is for Linux.

	ODL$_x$ parsing	**Metadata definition**	**Schema definition**	**Total time**
Versant (Windows)	0.3 sec	0.5 sec	7.1 sec	7.9 sec
Oracle (Linux)	0.2 sec	0.7 sec	1.2 sec	2.1 sec

Table 6.2: Schema Definition Performance

Schema processing consists of three stages: ODL$_x$ parsing, Schema Repository population and database schema creation. Performance was measured for each processing stage and results are provided in *Table 6.2*. The poor schema definition results for Versant are the direct consequence of the complex procedure for creating new schemas in this database. The schema definition file generated by the `SchemaDef` program is firstly compiled by the Versant `schcmp` utility to produce a database import file, which is then loaded into Versant to create the database schema (`sch2db` utility). The schema definition process for Oracle is much simpler and only requires execution of a series of SQL DDL statements through an open database connection. This is reflected in the measured values which are much lower for Oracle.

In a terms of platform, the `SchemaDef` program was tested on a PC with a Pentium 4 1.7 GHz processor and a 512 MB of RAM. The operating system for the O-R schema definition was RedHat Linux 9.0, while Versant schemas were defined on the Windows XP Professional platform. Windows version was compiled with Microsoft Visual C++ 7.0 compiler, while on Linux we used the GNU g++ 3.2 compiler.

The `ObjectPopulator` application was specially constructed to facilitate loading of the initial set of multimedia data into the database schema. It also extracts content metadata information (duration, frame rate, resolution, etc.) from the MPEG and JPEG files and maps it to the corresponding properties of the EGTV multimedia data types. The performance depended on the size of the memory buffer used for caching data reads from a file. On average, with the buffer size of 100MB, the loading process took 3.6 minutes for each 100MB of the multimedia file. Textual data was produced using a random value generator and then inserted into the database schema together with multimedia objects. This process resulted in the instantiation of persistent objects from the classes defined previously by the `SchemaDef` program.

From these experiments it was discovered that the only negative performance factor was due to multimedia data load.

6.5.2 Query Processing Tests

To test the performance of the Query Service, queries can be divided into five broad categories: basic queries, join queries, navigational queries, exported queries, and update queries. In this section, query types are briefly described, together with performance

Query type	Number of objects	Average execution time
Basic queries	1000	4 sec
Join queries	1000	10 sec
Navigational queries	1000	6 sec
Exported queries	1000	5 sec
Update queries	100	4 sec

Table 6.3: Query Processing Performance.

details. Behaviour based EQL operators used in these queries are limited to built-in types `string` and `int`. This is because content specific behaviour for multimedia types has not been developed yet. Thus, our test queries perform data comparisons and updates only on textual attributes of EGTV objects. Query sets used for these tests are provided in *Appendix H*, while their execution times are listed in *Table 6.3*.

1. **Basic Queries.** This category of queries is defined locally (at the component schema) and does not involve join operations. Their purpose was to test selection and projection of EGTV objects materialised from Versant database. All EGTV object extents were materialised in less than 1 second, as they contain only object references and not the actual data. However, projection and selection operations on these extents required dereferencing of some attributes, thus consuming more time. With 1000 objects, selection and projection on textual attributes was evaluated in under 4 seconds. These results were consistent with our initial performance estimations. Fast materialisation of EGTV objects from the local database was largely due to usage of native database API (Oracle OCCI and ODMG C++ bindings) for object retrieval and updating. This eliminated the need for translation of EQL queries to their SQL/OQL counterparts which would consume mush more time and resources. No object caching and query rewrite optimisations were used for materialised EGTV objects. Thus, all selection and projection operations on materialised object extents evaluated within the initial performance expectations for unoptimised query system.

2. **Join Queries.** This category of queries is defined locally, and employ integration operators to join classes. Since neither database contained natural joining properties, string values were used to perform the joins. For example, one query used `login` attribute to join the `User` and `Administrator` classes of the Multimedia Recording schema. With 1000 objects in the join operation, all queries were evaluated in under 10 seconds. Slower execution time was expected as no optimisation algorithms were used in the implementation of join operator. Thus, all objects in the `User` extent had to be iteratively compared to all objects of the `Administrator` extent. Implementation of more advanced join algorithms (merge join, has join, etc.) and EGTV object caching would significantly improve performance of the join operation.

3. **Navigational Queries.** This group of queries include path expressions and navi-

gational join operations. These operations include object navigation through references, and materialisation of individual EGTV objects at the Canonical Layer. The obtained results indicate no significant performance degradation as all queries were evaluated under 6 seconds. Thus, execution times for navigational join queries were better that those obtained for property join queries. This is because navigational join is based upon object reference navigation and does not involve any value matching. The only performance overhead is related to the materialisation of navigated objects at the canonical layer. No sorting and joining algorithms are required, thus materialised objects can be immediately retrieved or passed to other algebraic operators for further processing. Navigational query experiments show that in object-based data models querying through path navigation is less performance intensive then property joining.

4. **Exported Queries.** This category of queries were defined locally and exported to the global schema which is stored on the same database node. Thus, an addition layer of indirection was introduced. These tests were designed to monitor any degradation in performance as a result of the query definition located in a different database schema from base classes. No noticeable difference in the processing times were recorded. Due to the unavailability of CORBA object exchange protocol experiments involving more than one database node could not be performed. However, performance degradation can be estimated by analysing the efficiency and costs of the EGTV inter-node communication. The EGTV communication protocol consists of three layers: TCP/IP network transport, CORBA services and EGTV object exchange which is built upon the first two layers. The network interconnect used for the prototype is the fast 100MBs switched network, thus its performance degradation is reasonably low, up to 8% compared to a single instance database. CORBA layer adds additional 5% overhead due to name referencing and object marshaling and unmarshaling operations. The biggest performance degradation of a range of 10% is expected for the EGTV object exchange protocol which creates and maintains virtual objects in the global schema as proxies of equivalent objects in the canonical layer databases. Therefore, it is estimated that the total performance degradation of a networked EGTV system compared to the existing single node setup would be 23%. However, this can be reduced by optimising the prototype implementation. Firstly, TCP/IP performance degradation can be reduced by implementing faster 1GBs networks or aggregating multiple existing 100MBs communication channels. Secondly, implementation of global caching for EGTV virtual objects and optimisation the existing algorithms for maintaining proxy objects could significantly improve the overall response time of the interconnection system.

5. **Update Queries.** This group of queries facilitate update, create and delete operations on EGTV objects. The results obtained reflect total performance of the EGTV system, as the execution time is influenced by the performance of both Ver-

sant database and the EGTV model that maintains object references. Each update was propagated through EGTV objects to Versant database, where it was applied to corresponding persistent objects. Performance tests updated three textual attributes of 100 EGTV objects in 4 seconds. The execution time includes time required to materialise EGTV objects in the canonical schema, and time required to propagate updates from EGTV objects back to persistent objects in the local database. Transaction control is mandatory for updatable queries and can also increase the overall processing time. Thus, update operations incur additional overhead in writing objects back to the database and maintaining transactional consistency. Update experiments were consistent with the initial estimations as the write segment of an update EQL operation takes significantly more time then the initial object materialisation and evaluation of the query extent. This is largely due to the initial implementation of the EGTV model, where all updates are immediately propagated to local databases and transaction consistency is maintained between multiple concurrent sessions. This can be improved by caching updates in the EGTV model and propagating them to local database only when write access is requested by other concurrent transactions. Furthermore, the implementation of the transaction system can be further optimised to result in better response time when applying locking mechanisms to EGTV objects.

6.6 Conclusions

In this chapter an overview of the EGTV deployment architecture was presented, and this was followed by a description the prototype system. This included the full specification of schema definition process and the analysis of query processing algorithms for both local and global queries. Furthermore, a transaction control system was also developed to support update operations. By providing generic update operations, one of the object-orientation design principles, data encapsulation, is compromised. However, we and other researchers ([Sub96, RKB01b]) regard the encapsulation of data in a database as an unnecessary requirement, as views and security procedures can be used to emulate data hiding. Therefore, simple views (stored EQL queries) were provided in the EGTV prototype to facilitate schema restructuring and integration operations.

A description of the prototype environment and experiments were also provided. This chapter has demonstrated that the research ideas developed in this thesis are implementable, and limited experiments demonstrate that even a crude query evaluation strategy (in terms of optimisation) can provide acceptable results. However, query optimisation and a strategy for caching updates on virtual objects in global schema provide interesting areas for further research. Suggestions as to how this research could progress are offered in the final chapter.

Chapter 7

Conclusions

The aim of this research was to demonstrate that an object-oriented canonical query language can be developed to facilitate efficient queries and updates on distributed multimedia objects. Unlike other research projects, one objective of our research was to provide dynamic extensibility where the behaviour of query language operators can be redefined independently of the language itself. Thus, the EQL language can be easily extended to support other application domains apart from multimedia. A second objective was to enable this language to operate in the federated environment. Therefore, a metamodel and a schema definition language were defined to facilitate construction of a global schema, while a transaction control system was implemented to support updatability at the global layer. Furthermore, the EQL language also provided full orthogonality between query inputs and results, while its algebraic form was defined to facilitate formal and unambiguous query representation. The limited prototype system provided us with a platform both to test the ability to construct various multimedia schemas and to measure the performance or queries. In this chapter a review of the thesis is presented in §7.1; options for further research are discussed in §7.2; while in §7.3 final conclusions are offered.

7.1 Thesis Summary

In chapter one, an introduction to federated database systems was presented and existing standards for multimedia archiving and retrieving were discussed. It was concluded that existing multimedia repositories do not provide the required level of interoperability and query features. Therefore, a distributed database storage of large multimedia data was proposed. The Sheth and Larson architecture adopted by many researchers was described; requirements for canonical data model were discussed; and a hypothesis for this research emerged. Federated database systems form an important research topic because they provide a solution to many industrial problems. Previous canonical models were either relational models which were unable to manage the semantics of participating systems, or object models which failed to conform to any standard. Presently, two standards exist for object-based database models: the SQL:1999 object-relational standard and the ODMG 3.0 object-oriented database standard. However, neither of these models satisfy all requirements for a canonical data model, nor do they provide efficient multimedia manipulation and global schema construction features. Thus, the choice was to define a new federated model that includes canonical data and metadata representation for multimedia objects stored in standard O-O and O-R databases. The aim was to provide generic query capabilities for a set of inexpensive object-based databases organised into the federated system, and demonstrate its usability through a working prototype.

In chapter two, several research projects covering object query languages, metamodels, federated database architectures and multimedia databases were analysed and discussed. An examination and comparison of some of the existing global query languages was conducted to uncover how query definitions and their metadata representations were specified in each project, and to assess their execution capabilities. The output from this analysis provided the requirements for the design of a query language for this research. It was also concluded that a metadata repository and language for global schema specification are required prerequisites for defining any query language in the EGTV architecture. Further requirements were provided by the examination of federated database systems and multimedia repositories in previous chapter. Both chapters are used to prepare the introduction to the metamodel, schema definition language and finally the global query language in chapters three, four and five.

Chapter three presented a new metamodel developed to capture and store metadata from multiple heterogeneous database schemas. It was based upon the ODMG metamodel, but improved with a more simplified design, the ability to represent multimedia data types and with an extended support for object views. A metamodel is an important prerequisite for the definition of the global query language as it stores metadata required for generic (run-time) querying. The meta-metamodel was also specified to represent different metamodel versions. Metamodel mapping rules were defined for both object-oriented and object-relational databases and represented in the form of simple mapping language. This provided us with the ability to create EGTV multimedia databases and map their schemas to existing object-oriented and object-relational schema repositories.

A schema definition language for an EGTV federation was presented in chapter four. This language was designed to facilitate an object schema definition in an implementation independent format. Both Canonical Layer base schemas and global virtual schemas can be specified in the ODL$_x$ language, that also supports multimedia data types. The language was based upon the XML as it represented a standard for encoding and distributing data across various platforms and the Internet. Furthermore, it was supported with an XML Schema definition which provided a full syntax definition and specified rules for integrity constraints enforcement. The ODL$_x$ language was mapped to the EGTV metamodel, for which a simple mapping language was developed. Base schemas were then easily mapped to O-O and O-R databases using metamodel mapping rules defined in previous chapter, while global schemas were constructed as a union of virtual class definitions.

In chapter five we presented the EQL query language that facilitates querying of O-O and O-R schemas in a database and platform independent manner. The EQL preserved the familiar OQL syntax, but resolved some of the negative issues associated with OQL and made it orthogonal in terms of query input and output. Furthermore, it also provided a clear semantics for the updatability of query results, facilitated primitives for object creation, update and deletion, and included operation invocation support. EQL support for multimedia data types and ability to define custom operators were crucial when constructing and querying multimedia database federations. The EQL language was also supported with a query algebra, a formal language that can unambiguously define queries on a database. The EQL algebra was defined as a set of atomic operators that manipulate input sets of data to construct results. All algebraic operators are fully orthogonal as their inputs and outputs are EGTV classes which is identical to inputs and outputs of the EQL query language.

Chapter six discussed the implementation of the query system. Firstly, the overall deployment architecture of the EGTV project was presented and this was then followed by a description of the prototype system. This included the full specification of schema definition process and the analysis of query processing algorithms for both local and global queries. A transaction control system developed to support global updates was also discussed in this chapter. Chapter six then provided details of a prototype implementation which was constructed using multimedia data extracted from the *Físchlár* system. Three separate multimedia schemas were constructed and queried to test the performance of different query types, and to provide data for future research into areas such as query optimisation and EGTV object caching.

The initial hypothesis of this research was that an object query language can be defined to provide efficient queries and updates for large multimedia database federations. Therefore, a research methodology set to verify this hypothesis identified metamodel and schema definition language as important prerequisites for construction of such a query language. Thus, a new multimedia metamodel was firstly specified and then implemented to enable run-time access to schema metadata and to provide an infrastructure for generic querying. Furthermore, the ODL$_x$ schema definition language enabled specification of global multimedia schemas in a platform and database independent format. These two provided a foundation upon which the EQL query language was defined. It delivered simple SQL-like query interface and an orthogonal type system that facilitates easy subquerying. These features are combined with the ability to customise query language operators (at the data type level) to deliver powerful tool for querying and updating federated multimedia repositories. However, the existing implementation of the EQL does not facilitate any query optimisation strategy which limits the overall performance of the system. Furthermore, global multimedia schemas are created using simple (one virtual class) views that can limit schema restructuring capabilities of the federation. Although the primary goal of this research has been achieved and the initial hypothesis verified, some open issues related mainly to performance of the prototype and optimisation strategies need to be further addressed. These and other issues are identified as areas for future improvements and are thoroughly discussed in the remainder of this chapter.

7.2 Areas for Further Research

Object-based database systems are rapidly becoming a standard storage mechanisms in the database world. However, this is primarily due to recent expansion in object-relational database models and the increasing trend of their standardisation. While pure object-oriented ODMG databases struggle to make an impact, the irony is that the relational database community is moving closer to the usage and storage of objects. The object-relational model has already been standardised in the form of SQL:1999 specification, and major relational database vendors are starting to implement it in their database servers. This trend will lead to gradual evolution of relational database systems, until the object interface becomes standard and pure relational tables become obsolete.

A new trend in the database world is the arrival of semi-structured and XML databases. These systems store and query semi-structured data in its native representation and do not attempt to restructure or convert it to another format (i.e. relational). Although database representation and query interface for semi-structured data are not yet standardised, this field offers a major potential for future expansion. Current development in this area is mainly focused on independent academic and non-commercial projects (with an exception of Tamino), while mainstream database vendors are still lagging behind. However, the same trend as with object-based databases can be observed here. Although, standalone independent databases (such as eXist and Xindice) are first to deliver workable prototypes, it may well be the big relational database vendors who will eventually put this technology to mainstream users, and initial signs are already evident. Most mainstream relational databases are now able to provide some form of interfacing between relational and semi-structured data representations, while more advanced integration mechanisms are expected soon.

Support for semi-structured data. The expansion of Internet, and web-based technologies dramatically increased the amount of data stored in HTML, XML and other semi-structured representations. This trend is also evident in the multimedia domain where hypertext is commonly used to integrate different multimedia contents into a single browsable document, and where XML encoding of MPEG-7 [Mar02] is a standard way of representing content specific video metadata. Some aspects of this trend were already identified in the Hera project [VH02] assessed in chapter tree. However, the existing implementation of the EGTV federated system can only manipulate structured database schemas where data is encapsulated as objects. Therefore, EGTV system could possibly be enhanced with support for storing and querying semi-structured multimedia data. However, this extension would have implications to all segments of the EGTV architecture. Extensions at the database level should include development of an integration layer to facilitate incorporation of native XML databases as new data sources for the EGTV federation. A more complex problem would be canonical representation for semi-structured data and its integration with the existing structured database schemas. One option [LAW99] is to wrap semi-structured data into generic objects and then provide connection points between structured and semi-structured segments of the integrated schema. Metadata integration for semi-structured data is another area of research where issues such as EGTV metamodel integration with the XML Schema standard, support for semi-structured data types (e.g. MPEG-7), and construction of global schemas must be investigated. Finally, extensions to the EQL language should be defined to facilitate querying and updating of semi-structured multimedia data. This should include navigation through hyperlinks and transaction control for updates on semi-structured data.

Query optimisation. A topic which has not been covered in this research is that of the optimisation when processing EQL queries. There are several areas where this research could be focused, ranging from optimisation of query execution plans to more efficient caching algorithms for materialisation of EGTV objects.

- **Query rewriting.**
  This involves transforming query statements into a form which gives better performance. Traditionally, this involves placing selections before joins, but for some systems, it may be possible to provide query predicates to reduce the amount of locking at the local level, and minimise data retrieval at the canonical level.

- **Access to index structures in local databases.**
  Unfortunately, some local database systems may not provide index information, and some databases may drop or change an index during the lifetime of the database system. A dynamic approach might involve the analysis of EQL statements of current applications, and extending the Object Manager processor to detect the presence of indexes in local databases and make this information available to the query processor.

- **Improved algorithms for join operations.**
  Techniques such as hash join and merge join algorithms can be used to improve query performance when joining large collections of data.

- **Behaviour analysis.**
  EQL language defines most of its operators as behaviour of data types. This approach benefits in flexibility and adaptability of the query language, but presents a challenge for query optimisation. Behaviour definitions are encapsulated within their

data types and therefore cannot be analysed by the query optimiser. Thus, a query optimiser is unable to evaluate the performance impact of these behaviour operations and generate alternative execution plans. One solution is to generate behaviour specific metadata. This metadata is created at compile time and contains information on classes that are accessed from within the behaviour code and on the type of that access (i.e. read, update, object materialisation, property de-referencing, etc.). The other approach could involve performance analysis of behaviour invocations. Consecutive behaviour executions can build up reliable statistical data that can be then used for future query optimisations.

- **Object caching.**
  EGTV objects are closed and released (from canonical and global schemas) when the user commits a transaction or has finished working with query results. It should be possible for some local database systems to permit the materialised objects to remain in Canonical Layer schemas until an update takes place at the local database. It may also be possible to update the cached data using a system of triggers. It is likely that this is possible for some local systems, but may be impractical for others.

Real-time wrappers for multimedia repositories. The existing implementation requires that all multimedia data must be physically relocated to a set of O-O and O-R databases. However, there could be situations where this data relocation would be unachievable, or where multimedia files would continue to be updated in their proprietary repositories. Therefore, the EGTV system should provide real-time interfaces for such repositories. This would expand the overall applicability of the EGTV system and provide read-only querying of multimedia data stored in their native repositories. Development of the object wrapper for Oracle Video Server should be the priority, but should later be followed by implementations for other multimedia repositories integrated to the EGTV system. However, the performance and usability of these wrappers must be tested.

Advanced object view system. The view system for EGTV is based upon simple views that evaluate to a single virtual class only. The main limitation of this approach is the inability to define relationships between virtual classes in the global schema. One solution would be the expansion of the EGTV view system to support more complex schema views as provided in [RKB01a]. In this scenario, each schema view would evaluate to a set of interrelated virtual classes. Although schema views can already be defined in the EGTV metamodel, they are not supported in other components of the EGTV architecture. Therefore an extension of the ODL_x schema definition language is required to facilitate syntactic and semantic definition of these views. An alternative solution would investigate the possibility of adding new relationships to existing virtual classes in the global schema, created as results of simple (class) views.

Transaction processing. The current implementation of the EGTV transaction system requires that all local databases support strict two-phase locking and the two-phase commit protocol (either provided natively or simulated). This can prove to be a limiting factor when selecting database systems for multimedia storage, as not all databases fulfil these strict transaction requirements. The problem is especially evident when O-O and O-R databases at the Storage Layer act only as object wrappers for multimedia data externally stored in proprietary repositories with limited transaction support. One solution would be to relax the existing transaction requirements by implementing support for compensated and resubmitted transactions. This approach eliminates the requirement for two-phase commit protocol and strict two-phase locking making the EGTV transaction system applicable to larger number of existing database systems. The area of compensating transactions has already been extensively researched in projects such as AQUA [NZ96], InterBase [MBE96] and others. Therefore, their experiences should be assessed and adapted to support multimedia transactions in the EGTV federated system. The other area of possible research is the investigation of specific transaction mechanisms that can be applied to semi-structured data and XML documents stored in the EGTV federated system. This research would also require relaxation of existing transaction rules, as semi-structured data does not conform to existing database transaction standards [YEH00].

External layer. An external layer is not defined in the current version of the EGTV federated architecture. Therefore all federated clients are required to use a native C++ object API as their only interface to the system. However, there is an increased demand for representing query results in the XML format as this would enable a more diverse set of clients to access EGTV federation. Thus, an external layer should be constructed to transform existing EGTV schemas to a XML representation. This would then allow construction of a query interface that could use standard XQuery and XPath languages to interrogate EGTV global schemas. Since these two languages do not facilitate updates and transactions, extensions must also be investigated. Furthermore, a SOAP-based object exchange protocol should be considered as a replacement to the existing CORBA implementation. This protocol would be used for communication between XML enabled clients and the newly constructed external layer of the EGTV federated architecture.

Extending the EQL language with grouping predicates. The current specification of the EQL query language does not provide support for the predicates `group by` and `having`. This is due to the inability of the EGTV reference-based model to guarantee updatability of results for grouping queries. We now briefly explain this issue. The evaluation of a grouping query starts with the partitioning of a source object extent into multiple groups (according to the grouping condition), and then one virtual object is generated as an aggregate result for each of these groups. However, properties of these result objects cannot be unambiguously mapped to corresponding properties of base objects from which they were constructed as multiple mapping options exist. Thus, it is not possible to define update semantics as it is unclear how updates on the result virtual objects can be propagated to its base objects. This is the main reason that prevented inclusion of grouping predicates to the current specification of EQL language. However, as significance and usability of these predicates is high, the resolution of the problem should be a priority in future versions of the EQL language. A simple solution would be to disallow any updates on results of grouping queries, but this is perceived as too restrictive. Therefore, a new approach should be developed where a special `on_update` rule would be constructed to unambiguously define update semantics of stored queries. This rule would be triggered each time a property of an object generated as result of grouping query is updated. The `on_update` rule should override the default update behaviour and it could be implemented as a special operator of each database type. This could be further extended to provide custom updatability semantic for any virtual object generated as a result of a stored query. Finally, the EQL algebra should also be updated to support grouping operators.

A graphical tool for schema definition and integration. A graphical tool to assist with definition of external schemas and with their subsequent integration into a global schema would be beneficial. This tool should be capable of reading existing ODL$_x$ schema files, transforming them into a graphical representation where they could be easily modified and writing those modifications back to ODL$_x$ format. Furthermore, the graphical tool should be also capable of generating new ODL$_x$ schemas and assisting administrators in defining export and global schemas. The later functionality should include features for detecting semantic discrepancies during schema integration and it should provide some assistance in resolving these issues.

7.3 Final Thoughts

In this research, great emphasis has been placed on issues such as database representation for multimedia collections, the publication of a semantically rich metamodel, the definition of global schemas that integrate multiple multimedia repositories, and finally specification of a new object query language.

Some form of object-based storage mechanisms represent the future for database technology, and with the arrival of the World Wide Web (WWW) as an information sharing medium, the storage of new data types cannot be served by older, traditional systems such as relational databases. However, the arrival of novel technologies that store and query semi-structured data in their native representations introduces new challenges for system integrators. This is especially evident in the multimedia area where the integration of multimedia and hypermedia systems has already begun. The challenge to integrate these more complex forms of data is likely to arise sooner rather than later.

Bibliography

[Alt94] Altinentel, M., *Design and implementation of a Dynamic Function Linker and an Algebra for an object-oriented database system*, Master's thesis, Middle East Technical University, 1994.

[ANT02] ANTLR, *ANTLR Reference Manual*, ANTLR, 2002.

[BE96] Bukhres, O. and Elmagarmid, A., *Object Oriented Multidatabase Systems: A Solution for Advanced Applications*, Prentice Hall, 1996.

[Bećc02a] Bećarević, D., A Metadata Model for a Multimeda Database Federation, in *Technical Report ISG-02-01*, pp. 1–13, 2002, URL `http://www.computing.dcu.ie/~isg`.

[Bećc02b] Bećarević, D., The Federated Database System For Multimedia, in *PhD Transfer Report*, pp. 1–23, 2002, URL `http://www.computing.dcu.ie/~isg`.

[BFN94] Busse, R., Fankhauser, P. and Neuhold, E., Federated Schemata in ODMG, in *Proceedings of the Second International East/West Database Workshop*, pp. 356–379, Springer, 1994.

[BH87] Bernstein, P. and Hadzilacos, V., *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[BHP92] Bright, M., Hurson, A. and Pakzad, S., A Taxonomy and Current Issues in Multidatabase Systems, in *IEEE Computer*, vol. 25(3), pp. 50–60, 1992.

[Boo94] Booch, G., *Object-Oriented Analysis And Design with Applications (2nd edition)*, Benjamin Cummings, 1994.

[BP97] Blaha, M. and Premerlani, W., *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1997.

[BR01] Bećarević, D. and Roantree, M., Distributed Transactions for ODMG Federated Databases, in Zielinski, K., Geihs, K. and Laurentowski, A., eds., *DAIS*, vol. 198 of *IFIP Conference Proceedings*, pp. 317–322, Kluwer, 2001.

[BR04a] Bećarević, D. and Roantree, M., A Metadata Approach to Multimedia Database Federations, in *Information and Technology Journal*, vol. 46(3), pp. 195–207, 2004.

[BR04b] Bećarević, D. and Roantree, M., The EGTV Query Language, in *Proceedings of the 21th British National Conferenc on Databases (BNCOD 21)*, pp. 45–56, Springer, 2004.

[BRJ99] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[CB99] Catell, R. and Barry, D., *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 1999.

[CBS96] Connolly, T., Begg, C. and Strachan, A., *Database Systems: A Practical Approach to Design, Implementation and Management*, Addison Wesley, 1996.

[CHS$^+$95] Carey, M. et al., Towards Heterogeneous Multimedia Information Systems: The Garlic Approach, in *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, pp. 124–131, IEEE-CS, 1995.

[DAO$^+$95] Dogac, A. et al., METU Object-Oriented DBMS Kernel, in *Proceedings of the 6th International Conference on Database and Expert Systems Applications*, pp. 14–27, Springer Verlag, 1995.

[DD92] Date, C. and Darwen, H., *Relational Database Writings, 1989-1991*, Addison-Wesley, 1992.

[DDK$^+$96] Dogac, A. et al., A Multidatabase System Implementation on CORBA, in *6th Int Workshop on Research Issues in Data Engineering: Nontraditional Database Systems*, pp. 2–11, 1996.

[DOAO94] Dogac, A. et al., METU Object-Oriented DBMS, in *On Object-Oriented Database Systems*, pp. 513–541, Springer Verlag, 1994.

[EN94] Elmasri, R. and Navathe, S., *Fundamentals of Database Systems*, Addison Wesley, 1994.

[EP97] Eriksson, H.-E. and Penker, M., *UML Toolkit*, Wiley, 1997.

[ER98] Eaglestone, B. and Ridley, M., *Object Databases: An Introduction*, McGraw-Hill, 1998.

[Geo91] Georgakopoulos, D., Multidatabase Recoverability and Recovery, in *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, 1991.

[Ges91] Gessert, G. H., Handling Missing Data by Using Stored Truth Values, in *SIGMOD Record*, vol. 20(3), pp. 30–42, 1991.

[GGF$^+$96] Gardarin, G. et al., *IRO-DB : A Distributed System Federating Object and Relational Databases*, pp. 684–709, 1996.

[GP99] Gulutzan, P. and Pelzer, T., *SQL-99 Complete, Really*, R&D Books, 1999.

[GRS94] Georgakopoulos, D., Rusinkiewicz, M. and Sheth, A. P., Using Tickets to Enforce the Serializability of Multidatabase Transactions, in *Knowledge and Data Engineering*, vol. 6(1), pp. 166–180, 1994.

[HB96] Hurson, A. R. and Bright, M. W., Object-Oriented Multidatabase Systems, in *[BE96]*, pp. 1–36, 1996.

[HMN$^+$99] Haas, L. et al., Transforming Heterogeneous Data with Database Middleware: Beyond Integration, in *IEEE Data Engineering Bulletin*, vol. 22(1), pp. 31–36, 1999.

[HRS02] Habela, P., Roantree, M. and Subieta, K., Flattening the Metamodel for Object Databases, in *Proceedings of the Sixth East-European Conference on Advances in Databases and Information Systems (ADBIS 2002)*, pp. 263–276, Springer, 2002.

[HV99] Henning, M. and Vinoski, S., *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

[IBM01] IBM Corporation, *Informix DataBlade Version 8.11*, 2001.

[ISO86] ISO-8879, *Information Processing - Text and Office Information Systems - Standard Generealized Markup Language*, Internal Standards Organization, 1986.

[ISO92] ISO-10744, *Hypermedia/Time-based Structuring Language: HyTime*, Internal Standards Organization, 1992.

[ISO02a] ISO/IEC-13249-2:2002, *Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, 2nd edition*, Internal Organization for Standardization, 2002.

[ISO02b] ISO/IEC-13249-3:2002, *Information technology - Database languages - SQL Multimedia and Application Packages - Part 3: Spatial, 2nd edition*, Internal Organization for Standardization, 2002.

[ISO02c] ISO/IEC-13249-5:2002, *Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Still Image, 2nd edition*, Internal Organization for Standardization, 2002.

[Jor98] Jordan, D., *C++ Object Databases: Programming with the ODMG Standard*, Addison Wesley, 1998.

[KAC+02] Karvounarakis, G. et al., RQL: a declarative query language for RDF, in *Proceedings of the 11th International World Wide Web Conference, WWW2002*, pp. 592–603, ACM Press, 2002.

[Kam04] Kambur, D., *Behaviour in Object Views*, Ph.D. thesis, School of Computing, Dublin City University, 2004.

[KBR03] Kambur, D., Bećarević, D. and Roantree, M., An Object Model Interface for Supporting Method Storage, in *In proceedings of the 7th East European Conference on Advances in Databases and Information Systems ADBIS*, 2003.

[KK95] Kim, W. and Kelley, W., *On View Support in Object-Oriented Databases Systems*, pp. 108–129, 1995.

[KLS03] Kozankiewicz, H., Leszczylowski, J. and Subieta, K., Updatable XML Views, in *Proceedings of the Seventh East-European Conference on Advances in Databases and Information Systems (ADBIS 2003)*, pp. 381–399, Springer, 2003.

[KR01] Kambur, D. and Roantree, M., Using stored behaviour in object-oriented databases, in *Proceedings of the 4th Workshop EFIS 2001*, pp. 61–69, IOS Press, 2001.

[KR03] Kambur, D. and Roantree, M., Storage of complex business rules in object databases, in *5th International Conference on Enterprise Information Systems (ICEIS 2003)*, 2003.

[LAW99] Lahiri, T., Abiteboul, S. and Widom, J., Ozone: Integrating Structured and Semistructured Data, in *Proceedings of the 7th International Workshop on Database Programming Languages (DBPL'99)*, pp. 297–323, Springer, 1999.

[Lee98] Lee, J., Parallel Video Servers: A Tutorial, in *IEEE Multimedia*, vol. 5(2), pp. 20–28, 1998.

[LLO98] Li, H., Liu, C. and Orlowska, M., A Query System for Object-Relational Databases, in *Proceedings of the Australia Database Conference (ADC98)*, pp. 39–50, Springer, 1998.

[LMS+93] Leung, T. et al., The AQUA Data Model and Algebra, in *Workshop on Database Programming Languages*, pp. 157–175, 1993.

[LOSO97] Li, J. et al., Moql: A Multimedia Query Language, in *TR-97-01*, pp. 1–10, 1997, URL `http://www.cs.ualberta.ca/~duane/pdf/1997wmmis.pdf`.

[LSO$^+$00] Lee, H. et al., The Físchlár Digital Video Recording, Analysis, and Browsing System, in *Proceedings of the RIAO 2000 - Content-based Multimedia Information Access*, pp. 1390–1399, 2000.

[Mar02] Martinez, J., MPEG-7 Overview (version 8), in *Online document*, 2002, URL `http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm`.

[MBE96] Mullen, J., Bukhres, O. and Elmagarmid, A., InterBase: A Multidatabase System, in *[BE96]*, pp. 652–683, 1996.

[Mel03] Melton, J., *Advanced SQL:1999 Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann, 2003.

[Mot87] Motro, A., Superviews: Virtual Integration of Multiple Databases, in *IEEE Transactions on Software Engineering*, vol. 13(7), pp. 785–798, 1987.

[MS92] Melton, J. and Simon, A., *Understanding the New SQL: A Complete Guide*, Morgan Kaufmann, 1992.

[NKOD96] Nural, S. et al., Query decomposition and Processing in Multidatabase Systems, in *Proceedings of Object Oriented Database Symposium of the 3rd European Joint Conference on Engineering Systems Design and Analysis, France*, pp. 41–52, 1996.

[NZ96] Nodine, M. and Zdonik, S., The Impact of Transaction Management on Object-Oriented Multidatabase Views, in *[BE96]*, pp. 57–104, 1996.

[Ode95] Odell, J., Meta-Modeling, in *OOPSLA '95 Workshop on Metamodeling in OO*, 1995.

[OH98] Orfali, R. and Harkey, D., *Client/Server Programming with Java and CORBA*, Wiley, 1998.

[OMM$^+$01] O'Connor, N. et al., Físchlár: an on-line system for indexing and browsing broadcast television content, in *Proceedings of the 26th International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, vol. 3, pp. 1633–1636, IEEE, 2001.

[Ora02a] Oracle Coorporation, *Oracle9i InterMedia Reference Release 9.2*, 2002.

[Ora02b] Oracle Coorporation, *Oracle9i SQL Reference Release 9.2*, 2002.

[OSEM$^+$96] Ozsu, M. et al., Database Management Support for a News-on-Demand Application, in *SPIE Procceedings of the First International Symposium on Technologies and Systems for Voice, Video, and Data Communications - Multimedia: Full-Service Impact on Business, Education, and the Home, Vol. 2617*, pp. 24–32, 1996.

[OV91] Ozsu, T. and Valduriez, P., Distributed Database Systems: Where Are We Now?, in *IEEE Computer*, vol. 12(3), pp. 68–78, 1991.

[OV99] Ozsu, T. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice Hall, 1999.

[PBE95] Pitoura, E., Bukhres, O. and Elmagarmid, A., Object orientation in multidatabase systems, in *ACM Computing Surveys*, vol. 27(2), pp. 141–195, 1995.

[RB02] Roantree, M. and Bećarević, D., Metadata Usage in Multimedia Federations, in *First International Meta informatics Symposium (MIS 2002)*, pp. 132–147, 2002.

[RKB01a] Roantree, M., Kennedy, J. and Barclay, P., Integrating View Schemata Using an Extended ODL, in *Proceedings of the 9th International IFCIS Conference on Cooperative Information Systems (CoopIS)*, pp. 150–162, Springer, 2001.

[RKB01b] Roantree, M., Kennedy, J. and Barclay, P., Using a Metadata Software Layer in Information Systems Integration, in *Proceedings of the 13th International Conference on Advanced Information Systems Integration (CAiSE 2001)*, pp. 299–314, Springer, 2001.

[RS02] Roantree, M. and Smeaton, A., Research in Information Management at Dublin City University, in *Sigmod Record*, vol. 31(4), 2002.

[SBMS94] Subieta, K. et al., A Stack-Based Approach to Query Languages, in *Proceedings of the Second International East/West Workshop*, pp. 159–180, Springer Verlag, 1994.

[SCGS91] Saltor, F., Castellanos, M. and Garcia-Solaco, M., Suitability of Data Models as Canonical Models for Federated Databases, in *SIGMOD Record*, vol. 20(4), pp. 44–48, 1991.

[SFF95] Smahi, V., Fessy, J. and Finance, B., Query Processing in IRO-DB, in *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, DOOD'95*, pp. 299–318, Springer, 1995.

[Sie96] Siegel, J., *Corba Fundamentals and Programming*, Wiley, 1996.

[SKL95] Subieta, K., Kambayashi, Y. and Leszczyłowski, J., Procedures in Object-Oriented Query Languages, in *Proceedings of the 21st International Conference on Very Large Data Bases*, pp. 182–193, Morgan Kaufmann, 1995.

[SKLU96] Subieta, K. et al., Null Values in Object Bases: Pulling Out the Head from the Sand, in *Technical Report*, pp. 1–19, 1996, URL `http://www.ipipan.waw.pl/~subieta/EngPapers/index.html`.

[SL90] Sheth, A. and Larson, J., Federated Database Systems for Managing Distributed, heterogeneous and Autonomous Databases, in , vol. 22(3), pp. 183–226, 1990.

[Sto96] Stonebraker, M., *Object relational DBMSs the next great wave*, Morgan Kaufmann, 1996.

[Sub96] Subieta, K., Remarks on the ODMG Standard, in *Technical Report*, pp. 1–18, 1996, URL `http://www.ipipan.waw.pl/subieta`.

[TW97] Tesch, T. and Wasch, J., Global Nested Transaction Management for ODMG-Compliant Multi-Database Systems, in *Proceedings of the Sixth International Conference on Information and Knowledge Management (CIKM97)*, pp. 67–74, ACM Press, 1997.

[VBH03] Vdovjak, R., Barna, P. and Houben, G.-J., Designing a Federated Multimedia Information System on the Semantic Web, in *Proceedings of the Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003*, pp. 357–373, Springer, 2003.

[VH01] Vdovjak, R. and Houben, G.-J., RDF-Based Architecture for Semantic Integration of Heterogeneous Information Sources, in *Proceedings of the International Workshop on Information Integration on the Web 2001*, pp. 51–57, Springer, 2001.

[VH02] Vdovjak, R. and Houben, G.-J., Providing the Semantic Layer for WIS Design, in *Proceedings of the Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002*, pp. 584–599, Springer, 2002.

[WLE+97] Wong, J. et al., Enabling Technology for Distributed Multimedia Applications, in *IBM Systems Journal*, vol. 36(4), pp. 489–507, 1997.

[Wor99] World Wide Web Consortium, *Resource Description Framework (RDF) Model and Syntax Specification*, 1999.

[Wor00] World Wide Web Consortium, *Resource Description Framework (RDF) Schema Specification 1.0*, 2000.

[Wor01] World Wide Web Consortium, *XML Schema Parts 0-2 [Primer, Structures, Datatypes]*, 2001.

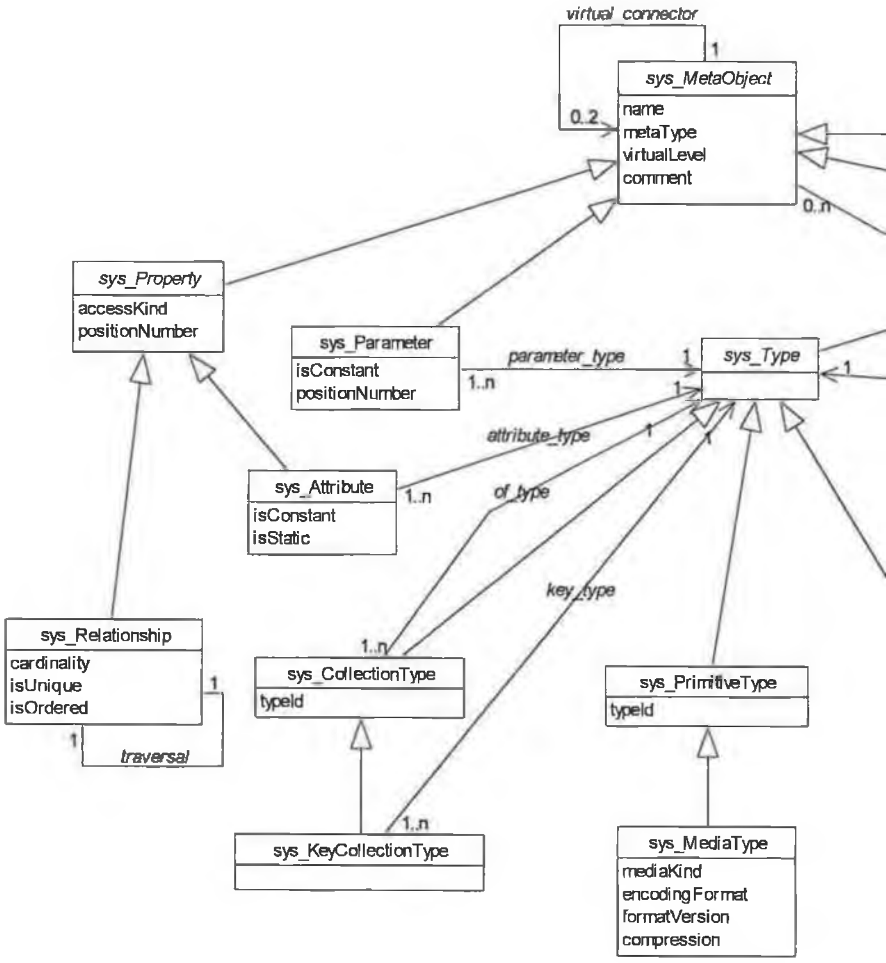
[WR03] Wang, L. and Roantree, M., Designing Roles For Object-Relational Databases, in *Proceedings of the 5th Workshop EFIS 2003*, pp. 106–116, IOS Press, 2003.

[YEH00] Younas, M., Eaglestone, B. and Holton, R., A Review of Multidatabase Transactions on the Web: From the ACID to the SACReD, in *Proceedings of the 17th British National Conferenc on Databases (BNCOD 17)*, pp. 140–152, Springer, 2000.

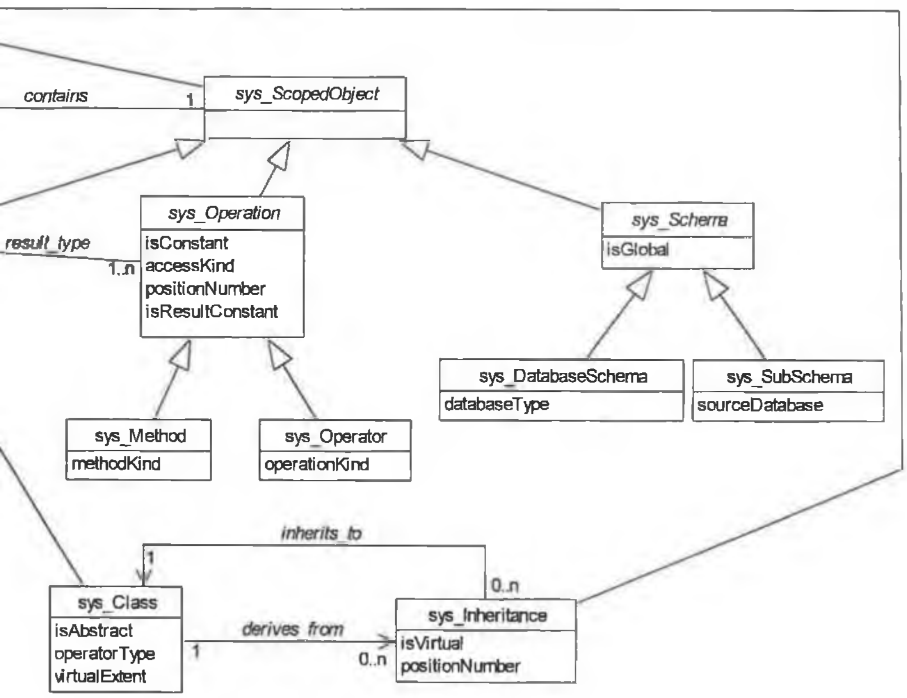
[Zam02] Zamulin, A., An Object Algebra for the ODMG Standard, in *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems (ADBIS 2002)*, pp. 291–304, Springer, 2002.

Appendix A

EGTV Metamodel UML Diagram



virtual connector
1
sys_MetaObject
name
metaType
virtualLevel
comment
0..2
0..n
sys_Property
accessKind
positionNumber
sys_Parameter
isConstant
positionNumber
parameter_type
1..n
1
sys_Type
1
1
attribute_type
1
sys_Attribute
isConstant
isStatic
1..n
of_type
1
key_type
sys_Relationship
cardinality
isUnique
isOrdered
1
1
traversal
1..n
sys_CollectionType
typeId
sys_PrimitiveType
typeId
1..n
sys_KeyCollectionType
sys_MediaType
mediaKind
encodingFormat
formatVersion
compression


contains
1
sys_ScopedObject
result_type
1..n
sys_Operation
isConstant
accessKind
positionNumber
isResultConstant
sys_Schema
isGlobal
sys_DatabaseSchema
databaseType
sys_SubSchema
sourceDatabase
sys_Method
methodKind
sys_Operator
operationKind
inherits_to
1
0..n
sys_Class
isAbstract
operatorType
virtualExtent
derives_from
1
0..n
sys_Inheritance
isVirtual
positionNumber

Appendix B

An XML Schema Definition of the ODL$_x$ Language

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" at-
tributeFormDefault="unqualified">
  <xs:annotation>
    <xs:documentation xml:lang="en">Definition of Primitive, Media, Class ans Collec-
tion types</xs:documentation>
  </xs:annotation>
  <xs:complexType name="generalType">
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:group name="attributeTypeGroup">
    <xs:choice>
      <xs:element name="primitiveType" type="generalType"/>
      <xs:element name="mediaType" type="generalType"/>
      <xs:element name="collectionType" type="collectionType"/>-->
    </xs:choice>
  </xs:group>
  <xs:group name="parameterTypeGroup">
    <xs:choice>
      <xs:group ref="attributeTypeGroup"/>
      <xs:element name="classType" type="generalType"/>
    </xs:choice>
  </xs:group>
  <xs:complexType name="collectionType">
    <xs:group ref="parameterTypeGroup"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="size" type="xs:integer" use="optional" default="0"/>
  </xs:complexType>
  <xs:annotation>
    <xs:documentation xml:lang="en">Operation result types, parameters, operators and meth-
ods</xs:documentation>
  </xs:annotation>
  <xs:complexType name="returnValType">
    <xs:group ref="parameterTypeGroup"/>
    <xs:attribute name="constant" type="xs:boolean" use="optional" default="false"/>
  </xs:complexType>
  <xs:complexType name="parameterType">
    <xs:group ref="parameterTypeGroup"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="constant" type="xs:boolean" use="optional" default="false"/>
  </xs:complexType>
  <xs:group name="operationTypeGroup">
    <xs:sequence>
      <xs:element name="returnVal" type="returnValType" minOccurs="0"/>
      <xs:element name="parameter" type="parameterType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="code" minOccurs="0"/>
    </xs:sequence>
  </xs:group>
  <xs:attributeGroup name="operationTypeAttrGroup">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="constant" type="xs:boolean" use="optional" default="false"/>
    <xs:attribute name="accessKind" type="xs:string" use="optional" default="public"/>
    <xs:attribute name="library" type="xs:string" use="optional"/>
  </xs:attributeGroup>
  <xs:complexType name="operatorType">
    <xs:group ref="operationTypeGroup"/>
    <xs:attributeGroup ref="operationTypeAttrGroup"/>
    <xs:attribute name="operatorKind" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="methodType">
    <xs:group ref="operationTypeGroup"/>
    <xs:attributeGroup ref="operationTypeAttrGroup"/>
    <xs:attribute name="methodKind" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:annotation>
    <xs:documentation xml:lang="en">Attribute, relationship and inheri-
tance</xs:documentation>
  </xs:annotation>
  <xs:complexType name="attributeType">
    <xs:group ref="attributeTypeGroup"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="constant" type="xs:boolean" use="optional" default="false"/>
    <xs:attribute name="static" type="xs:boolean" use="optional" default="false"/>
    <xs:attribute name="accessKind" type="xs:string" use="optional" default="public"/>
```

```
</xs:complexType>
<xs:complexType name="relationshipType">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="comment" type="xs:string" use="optional"/>
  <xs:attribute name="traversal" type="xs:string" use="required"/>
  <xs:attribute name="cardinality" type="xs:string" use="required"/>
  <xs:attribute name="unique" type="xs:boolean" use="optional" default="false"/>
  <xs:attribute name="ordered" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>
<xs:complexType name="inheritanceType">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="comment" type="xs:string" use="optional"/>
  <xs:attribute name="inheritsFrom" type="xs:string" use="required"/>
  <xs:attribute name="virtual" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>
<xs:annotation>
  <xs:documentation xml:lang="en">Class definition</xs:documentation>
</xs:annotation>
<xs:complexType name="importClassType">
  <xs:attribute name="database" type="xs:string" use="required"/>
  <xs:attribute name="schema" type="xs:string" use="required"/>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="comment" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="classType">
  <xs:choice maxOccurs="unbounded">
```

```
      <xs:element name="inheritance" type="inheritanceType" minOccurs="0" maxOc-
curs="unbounded"/>
      <xs:element name="attribute" type="attributeType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="relationship" type="relationshipType". minOccurs="0" maxOc-
curs="unbounded"/>
      <xs:element name="method" type="methodType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="operator" type="operatorType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="abstract" type="xs:boolean" use="optional" default="false"/>
  </xs:complexType>
  <xs:complexType name="extentType">
    <xs:simpleContent>
      <xs:extension base="xs:anySimpleType"/>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="virtualClassType">
    <xs:choice>
      <xs:sequence>
        <xs:element name="extent" type="extentType"/>
        <xs:element name="method" type="methodType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="operator" type="operatorType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:element name="importClass" type="importClassType"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:complexType name="libraryType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="include" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="systemLibrary" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:annotation>
    <xs:documentation xml:lang="en">Schema definition</xs:documentation>
  </xs:annotation>
  <xs:complexType name="schemaType">
    <xs:sequence>
      <xs:element name="class" type="classType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="virtualClass" type="virtualClassType" minOccurs="0" maxOc-
curs="unbounded"/>
```

```
      <xs:element name="library" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="libraryType">
              <xs:attribute name="language" type="xs:string" use="required"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="importSchema" type="importType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="comment" type="xs:string" use="optional"/>
    <xs:attribute name="federated" type="xs:boolean" use="optional" default="false"/>
    <xs:attribute name="databaseType" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:annotation>
    <xs:documentation xml:lang="en">Schema root element and constraint defini-
tions</xs:documentation>
  </xs:annotation>
  <xs:complexType name="importType">
    <xs:attribute name="startWithClass" type="xs:string" use="required"/>
    <xs:attribute name="relationships" type="xs:boolean" use="required"/>
    <xs:attribute name="inheritance" type="xs:boolean" use="required"/>
  </xs:complexType>
  <xs:element name="dbSchema" type="schemaType">
```

```
    <xs:key name="ClassID">
      <xs:selector xpath="class"/>
      <xs:field xpath="@name"/>
    </xs:key>
    <xs:key name="RelationshipID">
      <xs:selector xpath="class/relationship"/>
      <xs:field xpath="@name"/>
    </xs:key>
    <xs:keyref name="RelationshipRef" refer="RelationshipID">
      <xs:selector xpath="class/relationship"/>
      <xs:field xpath="@traversal"/>
    </xs:keyref>
    <xs:keyref name="ClassGeneralizationRef" refer="ClassID">
      <xs:selector xpath="class/generalization"/>
      <xs:field xpath="@inheritsFrom"/>
    </xs:keyref>
    <xs:key name="LibraryID">
      <xs:selector xpath="library"/>
      <xs:field xpath="@name"/>
    </xs:key>
  </xs:element>
</xs:schema>
```

Appendix C

The ODL$_x$ Language Specification and Metamodel Mappings

In this appendix we provide a full specification of all ODL$_x$ language elements and discuss their mappings to the EGTV metamodel. Metamodel mappings are required as all EGTV schemas must be represented in the schema repository to allow for dynamic querying and global schema integration. Mappings define rules for translation of ODL$_x$ elements and their attributes to metaclasses in the EGTV metamodel. Translation is generally a straightforward process where each ODL$_x$ element (i.e. `dbSchema`, `class`, `attribute`) is mapped to a single metaclass, while element attributes are mapped to metaclass attributes. Mappings can be formally represented by the xmap function specified in *Definition 3.19*. This function takes as input one ODL$_x$ element to which a mapping rule is applied to create a new meta-object in the EGTV Schema Repository. During this process, attributes of the newly created meta-objects are initialised from the values specified in the ODL$_x$ element. Mapping rules are defined in a simple mapping language and presented for each ODL$_x$ element discussed in this appendix. ODL$_x$ elements are on the left hand side, while EGTV metaclasses are on right hand side of mapping language definitions.

Definition 3.19 *`EGTV Metaclass := `***`xmap`***`( ODL`$_x$` Element )`*

C.1 Schema Element

- **Parent element:** none
- **Example:**
```
<dbSchema name="FilmArchive" global="false"
          databaseType="OR">
```

The `dbSchema` is the root element of the ODL$_{\text{x}}$ database schema specification and each `dbSchema` element contains multiple class and virtual class definitions. Attributes `name` and `global` define database schema name and a schema type, while the `databaseType` denotes the type of the underlying database. Schema type can be either global in terms of federated database architecture, or component, thus bound to a single database. Database type can be object-oriented (`OO`) or object-relational (`OR`). Boolean attribute `global` is optional, and if not specified, a value of `false` is assumed.

Metamodel Mapping. The dbSchema element is mapped to the `sys_DatabaseSchema` metaclass and its `sys_Schema`, and `sys_MetaObject` superclasses. This mapping is illustrated in *Example C.1*.

```
xmap dbSchema := sys_DatabaseSchema, sys_Schema, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  federated := sys_Schema.isGlobal
  databaseType := sys_DatabaseSchema.databaseType
}
```

Example C.1: Schema Mapping.

C.2 Class Element

- **Parent element:** `dbSchema`
- **Example:** `<class name="Film" abstract="true">`

The `class` element represents a class defined in the database schema. Each class belongs to one schema and a class can contain multiple `attribute`, `relationship`, `inheritance`, `method` and `operator` elements. Attributes defined for the `class` element are `name`, `comment` and `abstract`. The `name` attribute represents a class name that must be unique within the database schema. The `comment` is an optional user defined string value that can be specified not only for class, but for any other ODL$_x$ element. The `abstract` attribute specifies if the class is defined as abstract. If not specified, it defaults to the boolean value `false`. Abstract classes provide only definitions used by their subclasses and cannot be instantiated to EGTV objects.

Metamodel mapping. The ODL$_x$ `class` element maps to the `sys_Class` metaclass and its superclasses. The mapping is straightforward where each `class` attribute is mapped to the corresponding attribute of the `sys_Class`. Class containment in the database schema is implicitly defined in the ODL$_x$ by nesting the `class` element within the `dbSchema` element. The mapping language represents this relationship by defining a virtual attribute `<parent>` which is then mapped to the `contained_in` relationship of the `sys_MetaObject` abstract class. This is illustrated in *Example C.2.*

```
xmap class := sys_Class, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  abstract := sys_Class.isAbstract
}
```

Example C.2: Class Mapping.

C.3 Attribute Element

- **Parent element:** `class`
- **Example:** `<attribute name="ReleaseDate" constant="true"`
  `accessKind="Public">`

Attributes are defined within a class, and each class can contain multiple attribute definitions. Attributes in ODL$_x$ are represented with an `attribute` element. Each `attribute` element must have a `name`, while other attributes are optional. The `constant` defines if attribute values are mutable, and defaults to `false`. The `accessKind` defines the visibility of an attribute within a class. It can be `public`, `protected` or `private`, and if unspecified, the default value is `public`. The `static` attribute is of type `boolean` and defines attribute scope. It is set to `true`, the `attribute` element has a class (static) scope, otherwise an attribute is instantiated per object basis. This attribute is optional and defaults to `false`.

Metamodel Mapping. All attributes of the `attribute` element are mapped to the corresponding properties of the `sys_Attribute` metaclass and its superclasses, as illustrated in the *Example* C.3. Attribute type is defined with as nested subelement `<type>` which is then mapped to relationship `attribute_type` between metaclasses `sys_Attribute` and `sys_Type`. Detailed discussion of data type mapping is provided in §C.10.

C.4 Inheritance Element

- **Parent element:** `class`
- **Example:** `<inheritance name="MotionPictureGen" inheritsFrom="Film"`
  `virtual="false"/>`

```
xmap attribute := sys_Attribute, sys_Property, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  constant := sys_Attribute.isConstant
  static := sys_Attribute.isStatic
  accessKind := sys_Property.accessKind
  <name> := sys_Attribute.attribute_type
}
```

Example C.3: Attribute Mapping.

The generalisation relationship between two classes is represented with the `inheritance` element specified in the definition of each derived class. The `inheritsFrom` attribute specifies the name of the superclass, while the attribute `virtual` is optional, and it specifies if inheritance is defined as virtual. The default value for this attribute is `false`.

Metamodel Mapping. The mapping rule for inheritance element is illustrated in *Example C.4*. The `positionNumber` attribute of the `sys_Inheritance` metaclass defines the order of inheritance definitions in the case of multiple inheritance. The value of this attribute is determined by the relative position number of the `inheritance` element in the `class` definition.

```
xmap inheritance := sys_Inheritance, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  inheritsFrom := sys_Inheritance.inherits_to
  virtual := sys_Inheritance.isVirtual
}
```

Example C.4: Inheritance Mapping.

C.5 Relationship Element

- **Parent element:** `class`
- **Example:** `<relationship name="MotionPictureRef" traversal="ActorRef"`
  `cardinality="many"/>`

Each class can define relationships with other classes in the same database schema. Only bidirectional relationships can be defined in the EGTV database schemas and ODL$_x$ represents them as a pair of two interconnected `relationship` elements where each element defines one side of the relationship. The name is a mandatory attribute of the relationship definition as it uniquely identifies the relationship within the class. The traversal attribute specifies the other side of the bidirectional relationship, and its value must correspond to the name attribute of the inverse relationship defined in the same database schema. The cardinality defines the cardinality of the relationship and can have values of one or many. The boolean attributes unique and ordered specify if relationships values are ordered or unique. These two attributes are applicable only to relationships with cardinality greater than one, and if left unspecified, they default to false.

Metamodel Mapping. The relationship mapping is illustrated in *Example* C.5. Containment of the relationship within the class implicitly defined in the ODL$_x$ is represented with the virtual attribute <parent>. This attribute is mapped to contained_in relationship of the sys_MetaObject superclass.

```
xmap relationship := sys_Relationship, sys_Property, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  traversal := sys_Relationship.traversal
  cardinality := sys_Relationship.cardinality
  unique := sys_Relationship.isUnique
  orderd := sys_Relatsys_MetaObjectionship.isOrdered
  accessKind := sys_Property.accessKind
}
```

Example C.5: Relationship Mapping.

C.6 Method Element

- **Parent elements:** class, virtualClass
- **Example:** <method name="getScreenShot" constant="true"
  accessKind="public">

The method element is used to represent a signature of a method defined within the EGTV class. Each method can contain a result value and a list of parameters defined as ODL$_x$ elements nested within the main method element. The name attribute is mandatory and specifies the name of the method. It must be unique within the containing class. The accessKind can be public, private or protected, and if not specified, its default value is public. By default method is not constant, but mutability can be defined by the boolean constant attribute. The methodKind is an optional attribute which defines if method is prefixed as `static` or `virtual`. If `methodKind` is unspecified, method is neither static nor virtual.

Metamodel Mapping. The `method` ODL$_x$ element is mapped to the `sys_Method` metaclass and its abstract superclass `sys_Operation` as illustrated in *Example C.6.*

```
xmap method := sys_Method, sys_Operation, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  constant := sys_Operation.isConstant
  accessKind := sys_Operation.accessKind
  methodKind := sys_Method.methodKind
}
```

Example C.6: Method Mapping.

C.7 Operator Element

- **Parent elements:** `class`, `virtualClass`
- **Example:** `<operator name="==" constant="true" accessKind="public" operatorKind="binary">`

Class operators represent an additional type of behaviour supported in the EGTV metamodel. Multiple operators can be defined for each class using `operator` element provided in the ODL$_x$. Similarly to methods, operators can have parameters and a return value. The `name` attribute is a mandatory and it specifies the operator name. The `accessKind` and `constant` attributes are optional and they have the same default values as corresponding attributes in the `method` element. The `operatorKind` is a mandatory and specifies if operator is defined as `unary` or `binary` one.

Metamodel Mapping. The mapping of `operator` element to the metaclass `sys_Operator` and its superclasses is illustrated in *Example* C.7.

C.8 Return Value Element

- **Parent elements:** `method`, `operator`
- **Example:** `<returnVal constant="true">`

```
xmap operator := sys_Operator, sys_Operation, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  constant := sys_Operation.isConstant
  accessKind := sys_Operation.accessKind
  operatorKind := sys_Operator.operatorKind
}
```

Example C.7: Operator Mapping.

The `returnVal` element specifies the result value type returned from a method or operator function call. This element has only one optional attribute `constant` which specifies if the return value is constant. The default value is `false`.

Metamodel Mapping. Since the EGTV metamodel does not define specific metaclass for operation result, the ODL$_x$ `returnVal` element is mapped to the `sys_Operation`, the superclass for the `sys_Method` and `sys_Operator` metaclasses. Result type (defined as nested `<type>` element) is mapped to the `result_type` relationship between metaclasses `sys_Operation` and `sys_Type`. This is illustrated in *Example* C.8.

```
xmap returnVal := sys_Operation
{
  constant := sys_Operation.isResultConstant
  <type> := sys_Operation.result_type
}
```

Example C.8: Return Value Mapping.

C.9 Parameter Element

- **Parent elements:** `method`, `operator`
- **Example:** `<parameter name="inClass" constant="true">`

The `parameter` element belongs to the operation, and it is always nested within the `method` or `operator` element. Attributes of the `parameter` elements are `name`, `comment` and `constant`. The `name` attribute defines parameter name that must be unique within the operation to which it belongs. The `constant` specifies if the parameter value is mutable. This attribute is optional, and if left unspecified, the default value is `false`.

Metamodel Mapping. Parameter mapping is illustrated in *Example C.9.* Parameter type is defined as a nested subelement `<type>` and mapped to the `parameter_type` relationship between metaclasses `sys_Parameter` and `sys_Type`.

```
xmap parameter := sys_Parameter, sys_Metaobject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
  constant := sys_Parameter.isConstant
  <type> := sys_Parameter.parameter_type
}
```

Example C.9: Parameter Mapping.

C.10 Type Element

- **Parent elements:** `attribute`, `parameter`, `returnVal`
- **Example:** `<primitiveType name="integer"/>`

Primitive, collection and multimedia built-in types defined in the metamodel are represented as ODL$_x$ elements `primitiveType`, `mediaType`, `collectionType`, and `keyCollectionType` respectively. Type elements in ODL$_x$ are used to denote data types for `attribute`, `parameter` and `result` elements. Each type element has only one mandatory attribute, the `name` that uniquely identifies data type already registered within the metamodel. Collection and key collection types can recursively contain other type elements, including the other collections. Key collection types must also define a type for the key element by nesting additional ODL$_x$ type element.

The `classType` is a special type element that represents a user-defined class in the EGTV database schema. Its `name` attribute denotes the name of a user-defined class already registered in the EGTV metamodel.

Metamodel Mapping. Data type mappings are illustrated in *Example C.10.* A mapping rule is provided for each category of built-in and user-defined types. Mapping is straightforward, where each type category maps to corresponding metaclass in EGTV metamodel type hierarchy.

C.11 Virtual Class Element

- **Parent element:** `dbSchema`

```
xmap primitiveType := sys_PrimitiveType
{
  name := sys_PrimitiveType.name
}

xmap mediaType := sys_MediaType
{
  name := sys_MediaType.name
}

xmap collectionTypeType := sys_CollectionType
{
  name := sys_CollectionType.name
}

xmap keyCollectioType := sys_KeyCollectionType
{
  name := sys_KeyCollectionType.name
}

xmap classType := sys_Class
{
  name := sys_Class.name
}
```

Example C.10: Type Mapping.

- **Example:** `<virtualClass name="RecentFilms">`

The `virtualClass` element represents a virtual class defined in the database schema. One database schema can contain multiple virtual classes, where each virtual class can be either a simple view definition, or an imported class. The view definition contains one mandatory `extent` element and multiple optional `method` and `operator` elements. The `extent` element is defined as an EQL query which is constructed upon other virtual and base classes in the schema. The import class (the `importClass` element) is used for construction of global schemas and its role is to specify virtual classes to be imported into the global schema from the canonical layer. Attributes defined for the `virtualClass` element are `name` and `comment`. The `name` attribute represents the name of the virtual class and must be unique within the database schema. The `comment` is an optional user defined string value.

Metamodel Mapping. Virtual class mapping is illustrated in *Example* C.11. All virtual classes map to `sys_Class` metaclass in the EGTV metamodel. Although this is the same metaclass to which base classes are mapped, the `virtualLevel` and `virtualExtent` properties are mapped differently for virtual classes. The `virtualLevel` property contains non-zero value (level zero identifies base class), while the `virtualExtent` defines the EQL query string. A detailed description of metamodel representation for virtual classes was presented in chapter 3.

```
xmap virtualClass := sys_Class, sys_MetaObject
{
  name := sys_MetaObject.name
  comment := sys_MetaObject.comment
  <parent> := sys_MetaObject.contained_in
}
```

Example C.11: Virtual Class Mapping.

C.12 Extent Element

- **Parent element:** `virtualClass`
- **Example:** `<extent> <![ CDATA [ EQLQuery ] ]> </extent>`

The `extent` element defines an EQL query from which the virtual class extent is generated. This is only a metadata definition represented as a string value. The EQL query is enclosed in the the CDATA XML type, because the query can contain some characters not compliant with XML encoding. The `extent` element defines only the `comment` attribute which is a user defined string.

Metamodel Mapping. The virtual class extent (defined as a query) is mapped to the `virtualExtent` property of the sys_Class. An EQL query string in the ODL$_x$ is not represented as an attribute of the `extent` element, but as its textual value `TEXT` subelement. The mapping language represents this subelement as a `<content>` virtual attribute and maps it to the `virtualExtent` property of the `sys_Class` metaclass. This is illustrated in *Example C.12.*

```
xmap extent := sys_Class, sys_MetaObject
{
  comment := sys_MetaObject.comment
  <content> := sys_Class.virtualExtent
}
```

Example C.12: Extent Definition Mapping.

C.13 Import Class Element

- **Parent element:** `virtualClass`

- **Example:** `<importClass database="VideoRepository" schema="FilmArchive" name="RecentFilms">`

The `importClass` ODL$_x$ element defines a virtual class imported into the global schema from the canonical layer. This element is required when creating global schemas in the federated database architecture, since imported class acts as a proxy for the original class defined in the canonical layer schema. Global schemas, their construction and querying are fully explained in chapter 6. Any data manipulation on the imported class in the global schema is propagated to its canonical schema counterpart. Database interconnectivity and data exchange is facilitated by the EGTV reference model which was discussed in chapter 5. Attributes of the `importClass` element define name and location of the class to be imported. This includes the `database` name, the name of the `schema` in that database, and the `name` of virtual class in the schema. All these attributes are mandatory, while the `comment` attribute is optional.

Metamodel Mapping. The `database`, `schema` and `name` attributes of the `import-Class` element are mapped to a single `virtualExtent` attribute of the `sys_Class` metaclass. The mapping representation of the `virtualExtent` attribute is specified in *Example* C.13. The mapping is straightforward, where each ODL$_x$ attribute is mapped to the `virtualExtent` segment of the same name. The `@` and `::` characters are used as segment delimiters. This mapping provides sufficient information for retrieving and materialisation of imported virtual class.

```
xmap importClass := sys_Class, sys_MetaObject
{
  comment := sys_MetaObject.comment
  database, schema, name := sys_Class.virtualExtent
                            [@<database>::<schema>::<name>]
}
```

Example C.13: ODL$_x$ Import Class Mapping.

Appendix D

EQL Operator Classification

EQL Symbol	Description
=	equality
<	less then
<=	less or equal then
>	greater then
>	greater or equal then
!=	non equality
identical	object identity

Table D.2: Comparison Operators

EQL Symbol	Description
*	multiplication
/	division
+	addition
-	subtraction

Table D.4: Arithmetic Operators

EQL Symbol	Description
and	logical and
or	logical or
not	logical negation

Table D.6: Logic Operators

EQL Name	Description
max	Maximum element in the collection
min	Minimum element in the collection
avg	Average of all elements in collection
sum	Summary of all elements in collections
count	Number of elements in collection

Table D.8: Aggregate Operators.

Name	Description
union	Union of two object sets with duplicates eliminated
unionall	Union of two object sets with duplicates preserved
intersection	Intersection of two object sets
difference	Difference of two object sets
inset	Boolean true/false if the element is contained in the set
distinct	Eliminate duplicate elements

Table D.10: Set Operators.

Appendix E

Sample Multimedia Database Schemas

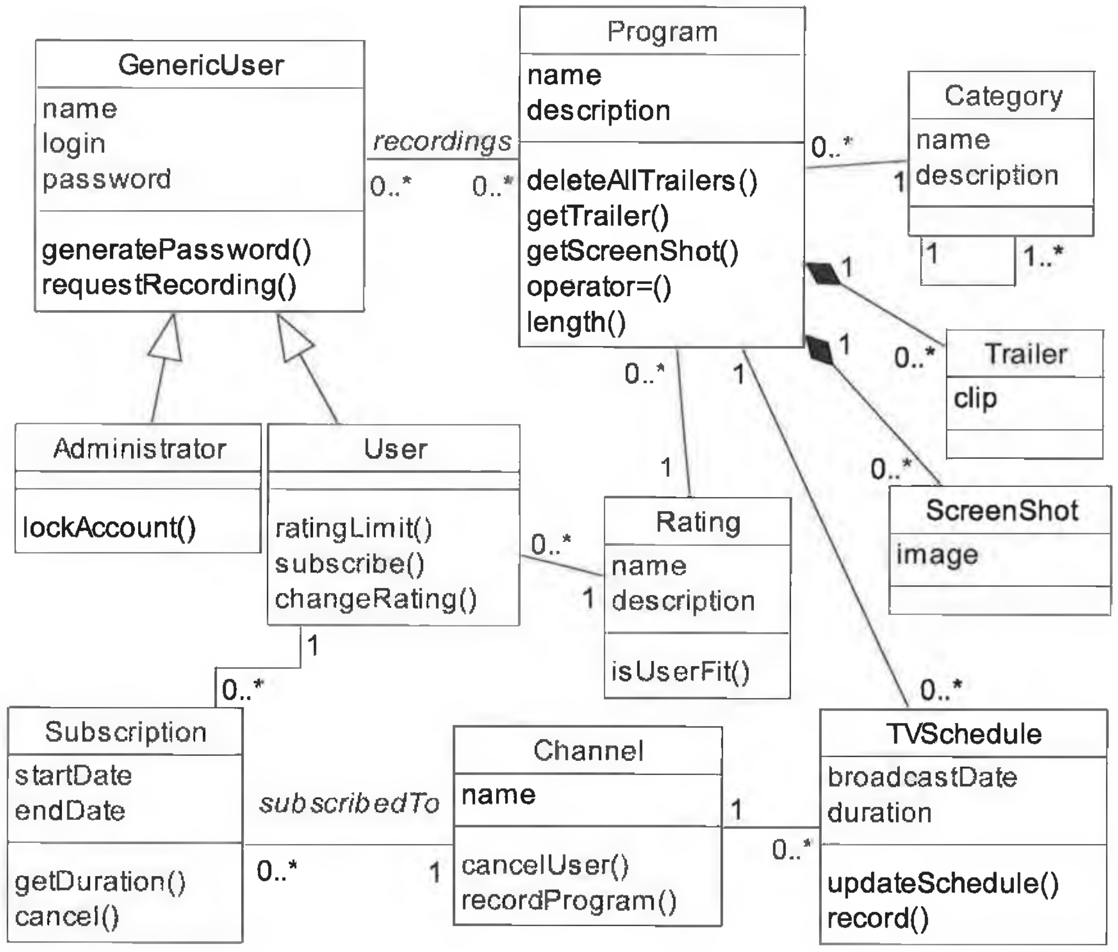


Figure E.1: Multimedia Recording System schema.

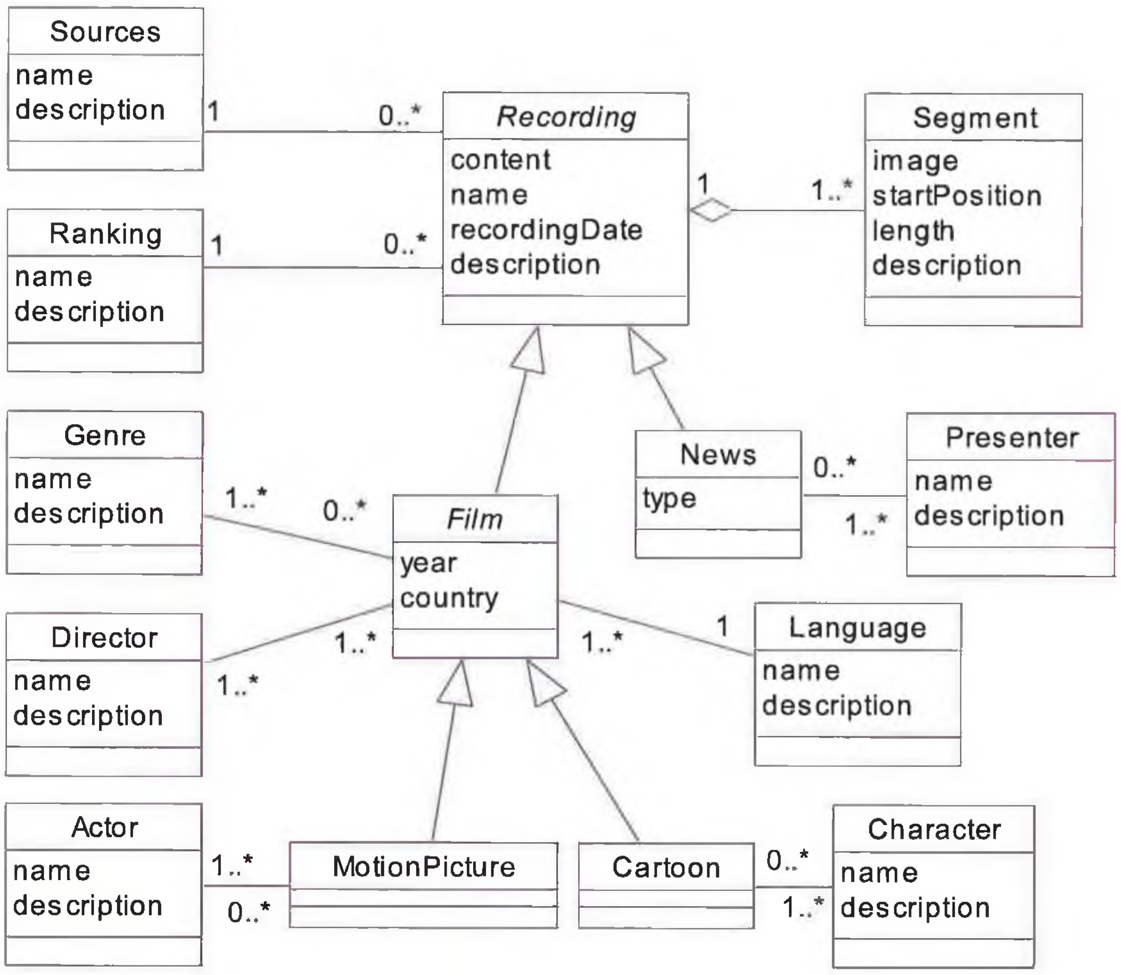


Figure E.2: Multimedia Archive System schema.

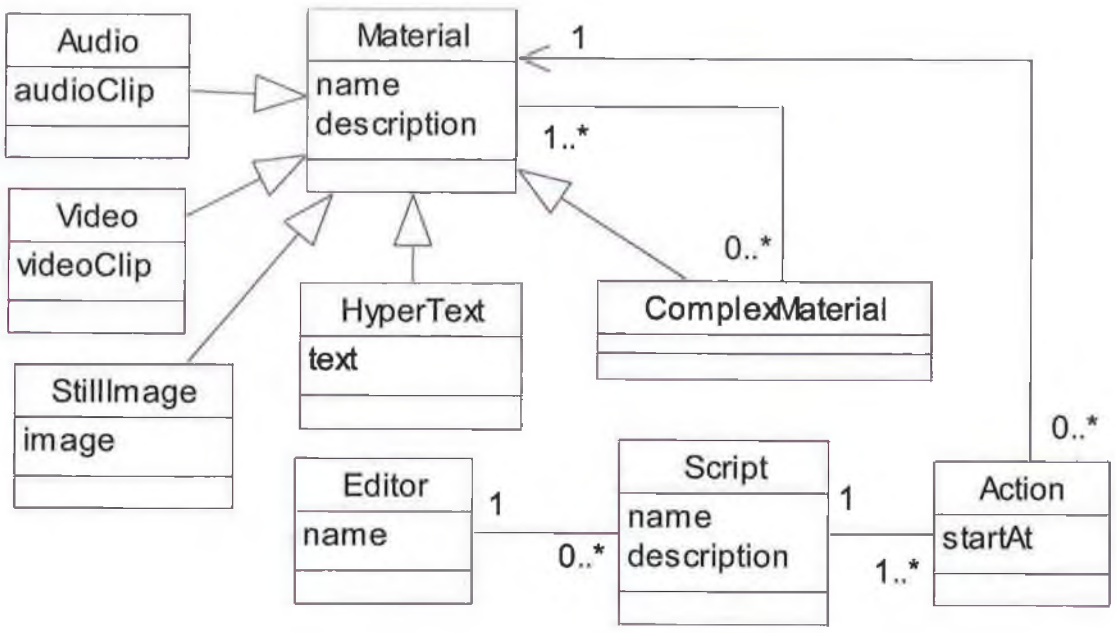


Figure E.3: Multimedia Editing System schema.

Appendix F

The ODL$_x$ Definition of a Test Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<dbSchema xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="metaSchema.xsd" name="MMRecordSys" database-
Type="OO">
  <!--######GenericUser class definition######-->
  <class name="GenericUser" abstract="true">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="login">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="password">
      <primitiveType name="string"/>
    </attribute>
    <method name="generatePassword" accessKind="public"/>
    <method name="requestRecording" accessKind="public">
      <returnVal>
        <primitiveType name="string"/>
      </returnVal>
    </method>
    <relationship name="GenericUserProgramRef" traversal="ProgramGenericUserRef" cardinal-
ity="many" unique="false"/>
  </class>
  <!--######Administrator class definition######-->
  <class name="Administrator">
```

```
    <inheritance name="AdministratorGenericUser" inheritsFrom="GenericUser" virtual="false"/>
    <method name="lockAccount" accessKind="public">
      <returnVal>
        <primitiveType name="bool"/>
      </returnVal>
      <parameter name="pUser" constant="true">
        <classType name="User"/>
      </parameter>
    </method>
  </class>
  <!--######User class definition######-->
  <class name="User">
    <inheritance name="UserGenericUser" inheritsFrom="GenericUser" virtual="false"/>
    <method name="ratingLimit" accessKind="public"/>
    <method name="subscribe" accessKind="public">
      <parameter name="pProgram">
        <classType name="Program"/>
      </parameter>
    </method>
    <method name="changeRating" accessKind="public">
      <parameter name="pRating">
        <classType name="Rating"/>
      </parameter>
    </method>
    <relationship name="UserRatingRef" traversal="RatingUserRef" cardinal-
ity="one" unique="false"/>
```

```
    <relationship name="UserSubscriptionRef" traversal="SubscriptionUserRef" cardinal-
ity="many" unique="false"/>
  </class>
  <!--######Program class definition######-->
  <class name="Program">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="description">
      <primitiveType name="string"/>
    </attribute>
    <method name="deleteAllTrailers" accessKind="public"/>
    <method name="getTrailer" accessKind="public">
      <returnVal>
        <classType name="Trailer"/>
      </returnVal>
      <parameter name="numTrailer">
        <primitiveType name="integer"/>
      </parameter>
    </method>
    <method name="getScreenShot" accessKind="public">
      <returnVal>
        <classType name="ScreenShot"/>
      </returnVal>
      <parameter name="numScreenShot">
        <primitiveType name="integer"/>
      </parameter>
    </method>
    <operator name="=" operatorKind="binary" accessKind="public">
      <returnVal>
        <primitiveType name="bool"/>
      </returnVal>
      <parameter name="pProgram">
        <classType name="Program"/>
      </parameter>
    </operator>
    <relationship name="ProgramGenericUserRef" traversal="GenericUserProgramRef" cardinal-
ity="many" unique="false"/>
    <relationship name="ProgramCategoryRef" traversal="CategoryProgramRef" cardinal-
ity="one" unique="false"/>
    <relationship name="ProgramRatingRef" traversal="RatingProgramRef" cardinal-
ity="one" unique="false"/>
    <relationship name="ProgramTVScheduleRef" traversal="TVScheduleProgramRef" cardinal-
ity="many" unique="false"/>
    <relationship name="ProgramTrailerRef" traversal="TrailerProgramRef" cardinal-
ity="many" unique="false"/>
    <relationship name="ProgramScreenShotRef" traversal="ScreenShotProgramRef" cardinal-
ity="many" unique="false"/>
  </class>
  <!--######Category class definition######-->
  <class name="Category">
    <attribute name="name">
```

```
      <primitiveType name="string"/>
    </attribute>
    <attribute name="description">
      <primitiveType name="string"/>
    </attribute>
    <relationship name="CategoryProgramRef" traversal="ProgramCategoryRef" cardinal-
ity="many" unique="false"/>
    <relationship name="Category1Ref" traversal="Category2Ref" cardinal-
ity="one" unique="false"/>
    <relationship name="Category2Ref" traversal="Category1Ref" cardinal-
ity="many" unique="false"/>
  </class>
  <class name="Trailer">
    <attribute name="clip">
      <mediaType name="jpeg"/>
    </attribute>
    <relationship name="TrailerProgramRef" traversal="ProgramTrailerRef" cardinal-
ity="one" unique="false"/>
  </class>
  <!--######ScreenShot class definition######-->
  <class name="ScreenShot">
    <attribute name="image">
      <mediaType name="mpeg"/>
    </attribute>
    <relationship name="ScreenShotProgramRef" traversal="ProgramScreenShotRef" cardinal-
ity="one" unique="false"/>
```

```
  </class>
  <!--######Rating class definition######-->
  <class name="Rating">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="description">
      <primitiveType name="string"/>
    </attribute>
    <method name="isUserFit" accessKind="public">
      <returnVal>
        <primitiveType name="bool"/>
      </returnVal>
      <parameter name="pProgram">
        <classType name="Program"/>
      </parameter>
      <parameter name="pUser">
        <classType name="User"/>
      </parameter>
    </method>
    <relationship name="RatingUserRef" traversal="UserRatingRef" cardinal-
ity="one" unique="false"/>
    <relationship name="RatingProgramRef" traversal="ProgramRatingRef" cardinal-
ity="many" unique="false"/>
  </class>
  <!--######Subscription class definition######-->
  <class name="Subscription">
    <attribute name="startDate">
      <primitiveType name="string"/>
    </attribute>
    <attribute name="endDate">
      <primitiveType name="date"/>
    </attribute>
    <method name="getDuration" accessKind="public">
      <returnVal>
        <primitiveType name="integer"/>
      </returnVal>
    </method>
    <method name="cancel" accessKind="public">
      <returnVal>
        <primitiveType name="bool"/>
      </returnVal>
    </method>
    <relationship name="SubscriptionUserRef" traversal="UserSubscriptionRef" cardinal-
ity="one" unique="false"/>
    <relationship name="SubscriptionChanelRef" traversal="ChanelSubscriptionRef" cardinal-
ity="one" unique="false"/>
  </class>
  <!--######Channel class definition######-->
  <class name="Channel">
    <attribute name="name">
      <primitiveType name="string"/>
    </attribute>
    <method name="cancelUser" accessKind="public"/>
    <method name="recordProgram" accessKind="public"/>
    <relationship name="ChanelTVScheduleRef" traversal="TVScheduleChanelRef" cardinal-
ity="many" unique="false"/>
    <relationship name="ChanelSubscriptionRef" traversal="SubscriptionChanelRef" cardinal-
ity="many" unique="false"/>
  </class>
  <!--######TVSchedule class definition######-->
  <class name="TVSchedule">
    <attribute name="broadcastDate">
      <primitiveType name="date"/>
    </attribute>
    <attribute name="duration">
      <primitiveType name="float"/>
    </attribute>
    <method name="updateSchedule" accessKind="public"/>
    <method name="record" accessKind="public"/>
    <relationship name="TVScheduleProgramRef" traversal="ProgramTVScheduleRef" cardinal-
ity="one" unique="false"/>
    <relationship name="TVScheduleChanelRef" traversal="ChanelTVScheduleRef" cardinal-
ity="one" unique="false"/>
  </class>
</dbSchema>
```

Appendix G

EQL Grammar

The grammar for the EQL language is implemented using ANTLR and is presented in ANTLRs own BNF. The semicolon symbol is used to signify the end of each grammar rule.

```
//////////////////////////////////////////////////
// EQL Grammar (ANTLR)
//////////////////////////////////////////////////
statement
    : ! q:queryExpression {## = #([EQLQUERY ,"EQL-
Query"], q);}
    ;
queryExpression
    : (querySpecification SEMICOLON! EOF!) => querySpec-
ification |
      subQueryExpression ( (UNION^ | UNIONALL^ | INTER-
SECTION^ | DIFFERENCE^) subQueryExpression)+ SEMI-
COLON! EOF!
    ;
subQueryExpression
    : ! (LPAREN!)? q:querySpecification (RPAREN!)? {## = #([EQL-
QUERY ,"EQLQuery"], q);}
    ;
querySpecification
    :
    selectClause
    fromClause
    (whereClause)?
    ;
```

```
selectClause
  : SELECT^ attributeList
  ;

fromClause
  : FROM^ sourceList
  ;

whereClause
  : WHERE^ logicalORExpr
  ;
attributeList
  : assignmentExpr (COMMA! assignmentExpr)*
  ;
sourceList
  : sourceExpr (COMMA! sourceExpr)*
  ;
parameterList
  : generalExpr (COMMA! generalExpr)*
  ;
primaryExpr
  : IDENTIFIER
  | constant
  | (LPAREN! generalExpr RPAREN! )
  ;
signExpr
  : (MINUS)? primaryExpr
  ;
operationExpr
  : (signExpr ~LPAREN!) => signExpr |
   ! se:signExpr ( (lp:LPAREN) (pl:parameterList)? (rp:RPAREN) )?
     { ##=#(se, pl ); }
  ;
pathExpr
  : operationExpr ((DOT^) pathExpr)?
  ;
mulExpr
  : pathExpr (( TIMES^ | DIVIDE^ | MOD^ ) mulExpr)?
  ;
generalExpr
```

```
    : mulExpr (( PLUS^ | MINUS^ ) generalExpr)?
    ;
  renameExpr
    : generalExpr (AS^ IDENTIFIER)?
    ;
  assignmentExpr
    : renameExpr (ASSIGN^ assignmentExpr)?
    ;
  subQueryGeneralExpr
    : renameExpr |
      ! LPAREN! q:querySpecification RPAREN!
        {## = #([EQLQUERY ,"EQLQuery"], q);}
    ;
  logicalNOTExpr
  : (NOT^)? subQueryGeneralExpr
  ;
  comparisonExpr
  : logicalNOTExpr ( (EQUAL^ | NOTEQUAL^ |
                      LESSTHANOREQUALTO^ |
                      LESSTHAN^ |
                      GREATERTHANOREQUALTO^ |
                      GREATERTHAN^) comparisonExpr)?
  ;
  logicalANDExpr
  : comparisonExpr ( (AND^) logicalANDExpr )?
  ;
  logicalORExpr
  : logicalANDExpr ( (OR^) logicalORExpr )?
  ;
  pjoinExpr
  : ! lhs:subQueryGeneralExpr PJOIN
      rhs:subQueryGeneralExpr ON LPAREN
      ex:logicalORExpr RPAREN
           { ##=#(PJOIN, lhs, rhs, #(ON, ex) ); }
  ;
  njoinExpr
  : ! lhs:subQueryGeneralExpr NJOIN
      rhs:subQueryGeneralExpr ON LPAREN
      ex:pathExpr RPAREN
           { ##=#(NJOIN, lhs, rhs, #(ON, ex) ); }
  ;
```

```
joinExpr
: (pjoinExpr) => pjoinExpr |
  (njoinExpr) => njoinExpr
;
sourceExpr
: (subQueryGeneralExpr) => subQueryGeneralExpr |
   joinExpr
;
constant
: (INTEGER | CHARACTER)
;
////////////////////////////////////////////////
// ANTLR DECLARATIONS
////////////////////////////////////////////////
class EQLLexer extends Lexer;
options {
        k = 2;
        exportVocab=EQLExpr;
        caseSensitiveLiterals = false;
        charVocabulary='\u0000'..'\uFFFE';
}
tokens
{
    SELECT = "select";
    FROM = "from" ;
    WHERE = "where" ;
    AS = "as";
    INTERSECTION = "intersection" ;
    UNION = "union" ;
    UNIONALL = "unionall" ;
    INSET = "inset" ;
    AND = "and" ;
    OR = "or" ;
    NOT = "not" ;
    PJOIN = "join";
    NJOIN = "connect";
    ON = "on";
}
Whitespace     : (' ' | '\t' | '\n' | '\r')
    { _ttype = Token.SKIP; }
    ;
```

```
SingleLineComment
    : "//"( ~('\r' | '\n') )*
    { _ttype = Token.SKIP; }
    ;
MultiLineComment
    : "/*" (~'*')* '*' ('*' | ( ~('*' | '/') (~'*')* '*') )* '/'
    { _ttype = Token.SKIP; }
;
IDENTIFIER :  ('a'..'z' | 'A'..'Z' | '_-
' ) ( ('a'..'z' | 'A'..'Z' |
               '_') | ('0'..'9' ))* ;
INTEGER : '0'..'9' ('0'..'9')* ;
CHARACTER : "'" '\0'..'\255' "'" ;
COMMA : ',' ;
DOT : '.' ;
SEMICOLON : ';' ;
LPAREN : '(' ;
RPAREN : ')' ;
LCURL : '{' ;
RCURL : '}' ;
PLUS : '+' ;
MINUS : '-' ;
TIMES : '*' ;
DIVIDE : '/' ;
MOD : '%' ;
EQUAL : '=' ;
ASSIGN : ":=" ;
NOTEQUAL : "!=" ;
LESSTHANOREQUALTO : "<=" ;
LESSTHAN : "<" ;
GREATERTHANOREQUALTO : ">=" ;
GREATERTHAN : ">" ;
```

Appendix H

Query Experiments

In this appendix, the query specifications used in the experiments discussed in chapter six are listed.

H.1 Basic Local Queries

Experiment 1 (basic select)

```
select name, recordingDate, description
from Recording;
```

Experiment 2 (method invocation)

```
select name, description
from Program
where length() > 120;
```

Experiment 3 (operator invocation)

```
select u.name + u.login
from User u;
```

H.2 Join Queries

Experiment 4 (property join)

```
select d.name, d.description, a.description
from Director d join Actor a on d.name = a.name;
```

Experiment 5 (property join with filter)

```
select u.name, u.login, a.login
from User u join Administrator a on u.name=a.name
where u.login != a.login;
```

H.3 Navigational Queries

Experiment 7 (path navigation)

```
select name, recordingDate
from Film
where Film.langRef.Language.name = "French";
```

Experiment 7 (path navigation with filter)

```
select Program.ratedIn.Rating.name
from Program
where name = "Morning News";
```

Experiment 8 (navigational join)

```
select User.name as uName, Program.name as pName
from Program connect User on User.recordings;
```

H.4 Exported Queries

Experiment 9 (querying exported virtual class)

```
select name, recordingDate
from TVRecording;
```

H.5 Update Queries

Experiment 10 (update of two attributes)

```
select login := "tomg", password := "tom34"
from User
were name = "Tom";
```

Experiment 11 (update of three attributes)

```
select year := "2003", country := "Ireland", descrip-
tion := "empty"
from Film
where recording date > "01/01/2003";
```