# Integration of Database Programming and Query Languages for Distributed Object Bases

Markus Kirchberg

A dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Information Systems at Massey University

**Supervisor**:
Prof. Dr. Klaus-Dieter Schewe

**Co-Supervisor**:
Associate Prof. Dr. Ray Kemp

**Internal Examiner**:
Dr. Alexei Tretiakov

**Australasia Examiner**:
Prof. Dr. Xiaofang Zhou

**International Examiner**:
Prof. Dr. Bernhard Thalheim

**Date of Examination**:
July 13, 2007

# Abstract

Object-oriented programming is considered advantageous to other programming paradigms. It is believed to provide more flexibility, simplify development and maintenance, better model real-world situations, and is widely popular in large-scale software engineering. The idea behind object-oriented programming is that a program is composed of objects that act on each other. While programming languages mainly focus on processing, data storage and data sharing only play a minor role. Database management systems, on the contrary, have been built to support large bodies of application programs that share data. In todays database marketplace, relational and object-relational database systems are dominant. Object-oriented database systems were originally deemed to replace relational systems because of their better fit with object-oriented programming. However, high switching costs, the inclusion of object-oriented features in relational database systems, and the emergence of object-relational mappers have made relational systems successfully defend their dominance in the DB marketplace. Nevertheless, during the last few years, object-oriented database systems have established themselves as a complement to relational and object-relational systems. They have found their place as embeddable persistence solutions in devices, on clients, in packaged software, in real-time control systems etc.

In order to utilise the combined power of programming languages and SQL-like database query languages, research has focused on the embedded and integrated approaches. While embedded approaches are more popular, they suffer from the impedance mismatch, which refers to an inadequate or excessive ability of one system to accommodate input from another. This situation worsens when considering the numerous conceptual and technical difficulties that are often encountered when a relational database system is being used by a program that is written in an object-oriented language.

Research presented in this thesis addresses the integration of programming languages, database query languages, object-oriented programming and traditional database concepts. We investigate the stack-based approach to database programming and querying. The major objectives of our research are to:

- Develop a database architecture that is suitable to integrate database programming languages, object-oriented programming and more traditional database system features;
- Propose an intermediate-level database programming and querying language that combines the advantages of object-oriented programming and database querying languages. Before such a powerful language design can be achieved, a number of conceptual and technical challenges have to be investigated;
- Define a suitable run-time environment, which permits an efficient and effective evaluation of such an integrated language in a distributed database environment; and
- Provide proof of concept implementations.

# Acknowledgement

I would like to thank *Klaus-Dieter Schewe* for giving me the opportunity to pursue an academic career under his supervision. This thesis would not have been possible without his invaluable input, constant support and understanding.

My thanks also goes to *Ray Kemp* who kindly agreed to act as co-supervisor of this thesis.

Thanks to *Massey University* and the *Department of Information Systems* for their financial support.

Special thanks to *Sebastian Link*, *Sven Hartmann* and *Faizal Riaz-ud-Din* for their support, useful discussions and their friendship.

I am grateful to my wife *Gowri* for her love, encouragement and understanding. It is her support that gave me the strength to complete this work.

Finally, I would like to thank my parents *Sigrid* and *Dieter Kirchberg*, who have always been there for me and supported me in every way possible.

# Table of Contents

**1 Introduction** **9**

1.1 Database Systems and Programming Languages . . . . . . . . . . . . . . 10

1.1.1 The Impedance Mismatch . . . . . . . . . . . . . . . . . . . . . . . . 14

Objects and RDBMSs. . . . . . . . . . . . . . . . . . . . . . . . . 15

Objects and Object-Relational Databases. . . . . . . . . . . . . . 15

Objects and Pure Object Databases. . . . . . . . . . . . . . . . . . 15

1.1.2 On the Integration of Programming and Query Languages . . . 16

1.1.3 DBPL vs. Conventional Programming Languages . . . . . . . . 18

1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

1.3 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

1.4 Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**2 A Review of DBPLs and Related Concepts** **23**

2.1 The Stack-Based Approach . . . . . . . . . . . . . . . . . . . . . . . 23

2.2 Overview of DB Programming Environments . . . . . . . . . . . . . 25

2.2.1 The Database Programming Language DBPL . . . . . . . . . . . 25

2.2.2 The $O_2$ Object Database System . . . . . . . . . . . . . . . . . 26

2.2.3 The Object Base Management System TIGUKAT . . . . . . . . 26

2.2.4 The Parallel Database Programming Language FAD . . . . . . . 27

2.2.5 Additional Relevant Research Results on DBPLs . . . . . . . . 28

**3 An Integrated Approach to DB Programming and Querying** **31**

3.1 A Distributed Object-Oriented Database System . . . . . . . . . . . . 31

3.1.1 Architecture Overview . . . . . . . . . . . . . . . . . . . . . . . . 32

3.1.2 Properties of the OODM . . . . . . . . . . . . . . . . . . . . . . . 36

Notes on the Choice of the Data Model. . . . . . . . . . . . . . . 37

3.1.3 Fragmentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.1.4 Linguistic Reflection . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.1.5 A Simple University Application . . . . . . . . . . . . . . . . . . 39

The Global OODM Database Schema. . . . . . . . . . . . . . . . 39

Distributing the University Database. . . . . . . . . . . . . . . . 43

3.1.6 A Note on the Contribution of the Proposed ODBS . . . . . . . 50

3.2 Processing User Requests . . . . . . . . . . . . . . . . . . . . . . . . 50

**4 The Database Programming and Query Language iDBPQL 57**
4.1 Language Overview . . . 57
4.2 Basic Language Concepts . . . 58
4.2.1 Challenges . . . 59
4.2.2 Conventions . . . 60
4.2.3 Literals, Names and Other Rudimentary Language Elements . . 61
Literals. . . . 61
Identifiers. . . . 61
Comments. . . . 61
Names. . . . 61
4.2.4 Types and Values . . . 62
Primitive Types. . . . 62
The Record Type. . . . 63
Type Definitions. . . . 67
Type Parameters. . . . 68
Collection Types. . . . 69
`NULL`able Types. . . . 71
Value Initialisation and Default Values. . . . 72
Sub-typing and Type Conversion. . . . 73
Variables. . . . 74
Default Type Operations. . . . 74
Type Definitions for the University Application. . . . 75
4.2.5 Classes and Objects . . . 78
Structure of a Class. . . . 78
Variables and Reference-Types. . . . 80
Methods. . . . 81
(Multiple) Inheritance. . . . 82
Variables, Types, Objects, and Classes. . . . 88
The `FINAL` and the `ABSTRACT` Modifiers. . . . 88
The `UNION`-Type. . . . 89
Special Pre-Defined Classes. . . . 91
Class-Collections. . . . 93
Constraints. . . . 94
Database Schemata and Classes. . . . 95
Run-Time MetaData Catalogue Entries. . . . 96
Persistence. . . . 97
Class Definitions for the University Application. . . . 98
4.3 Evaluation Plans . . . 100
4.3.1 Challenges . . . 100
4.3.2 Components of Evaluation Plans . . . 101
4.3.3 Evaluation Blocks and Their Properties . . . 102
4.3.4 Statements . . . 104
Assignment Statements. . . . 104
Control Flow Statements. . . . 105
Type Operation Invocation, Method Calls and Object Creation. 107
4.3.5 Expressions . . . 108

Simple Expressions. . . . . 108
Parenthesised Expressions. . . . . 108
Assignment Expressions. . . . . 108
Type Operation Invocation, Method Calls and Object Creation. 108
Renaming Expressions. . . . . 108
Boolean Expressions. . . . . 109
Query Expressions. . . . . 110
Cast Expressions. . . . . 114
The `SUPER` Expression and Keywords `THIS` and `SUPER`. . . . . 114
4.4 Simultaneous Processing . . . . . 117
4.4.1 Implicit Inter-Transaction Concurrency . . . . . 120
4.4.2 Support for Explicit Concurrency . . . . . 120
4.4.3 Implications . . . . . 124
4.5 Examples . . . . . 125

**5 On the Implementation of iDBPQL 131**
5.1 MetaData Catalogues and Evaluation Plans . . . . . 131
5.1.1 Challenges . . . . . 132
5.1.2 MetaData Entries and Associated Information . . . . . 132
Representing Type Information. . . . . 134
Representing Class Information. . . . . 135
Representing Descriptors. . . . . 137
Representing Other Attributes. . . . . 137
Inheritance Relations. . . . . 138
5.1.3 The Representation of Objects and Values . . . . . 138
5.1.4 The Representation of Evaluation Plans . . . . . 141
5.1.5 Overview of Annotations . . . . . 144
5.2 DBS Component Interface Definitions . . . . . 145
5.2.1 A Persistent Object Store . . . . . 145
An Object Model for POS. . . . . 146
Access Methods. . . . . 150
The Architecture of POS. . . . . 150
The Service Interface. . . . . 151
POS as a Platform for iDBPQL. . . . . 159
5.2.2 A Multi-Level Transaction Management System . . . . . 162
The Service Interface. . . . . 165
5.2.3 The Remote Communication Module . . . . . 166
The Database Agent Communication Language (DBACL). . . . 166
5.3 The Execution of Evaluation Plans . . . . . 167
5.3.1 Challenges . . . . . 168
5.3.2 The Run-Time Environment . . . . . 169
The REE Stack Area. . . . . 171
The Environment Stack (ES). . . . . 172
The Result Stack (RS). . . . . 174
Operations on Stacks and Queues. . . . . 175
Initialising Result Queues. . . . . 178

Binding Behaviour Names to Evaluation Plans. . . . . . . . . . . 179
5.3.3 The `SYSTEMcall`, `POScall` and `TMScall` Primitives . . . . . . . . 179
The `SYSTEMcall` Primitive. . . . . . . . . . . . . . . . . . . . . . . . 179
The `POScall` Primitive. . . . . . . . . . . . . . . . . . . . . . . . . . 180
The `TMScall` Primitive. . . . . . . . . . . . . . . . . . . . . . . . . . 180
5.3.4 Overview of the Evaluation Process . . . . . . . . . . . . . . . . . . . 180
Machine Instructions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 180
Unnesting Objects and Values. . . . . . . . . . . . . . . . . . . . . . 181
The Main Evaluation Routine. . . . . . . . . . . . . . . . . . . . . . . 183
5.3.5 Maintaining the Run-Time Environment . . . . . . . . . . . . . . . 186
5.3.6 Evaluating Individual Statements and Expressions . . . . . . . . . 187
Simple Expressions: Literals and Names. . . . . . . . . . . . . . . 187
Object Identifiers and Stacks. . . . . . . . . . . . . . . . . . . . . . . 187
Expressions with Unary and Binary Operators. . . . . . . . . . . 188
Boolean Expressions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
Renaming Expressions. . . . . . . . . . . . . . . . . . . . . . . . . . . 194
Accessing Data Objects. . . . . . . . . . . . . . . . . . . . . . . . . . 195
Accessing Persistent Data: Beyond Direct Access. . . . . . . . . . 199
More Query Expressions. . . . . . . . . . . . . . . . . . . . . . . . . . 201
Controlling the Flow of Serial Data Processing. . . . . . . . . . . 209
Invocation of Behaviours. . . . . . . . . . . . . . . . . . . . . . . . . 213
Object Creation and Assignments. . . . . . . . . . . . . . . . . . . 214
Cast Expressions, `SUPER` and `THIS`. . . . . . . . . . . . . . . . . . 215
5.3.7 Evaluating Statements and Blocks of Statements . . . . . . . . . . 216
Blocks of Statements. . . . . . . . . . . . . . . . . . . . . . . . . . . . 216
Simple Transactions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 217
Atomic Statement Blocks. . . . . . . . . . . . . . . . . . . . . . . . . 221
5.3.8 Processing Evaluation Plans . . . . . . . . . . . . . . . . . . . . . . 222
5.3.9 Simultaneous Evaluation of Statements and Expressions . . . . 222
5.3.10 Distributed Processing of Evaluation Plans . . . . . . . . . . . . . 223
Migrating Objects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 223
Processing Evaluation Plans on a Remote Node. . . . . . . . . . . 224
Distributed Transactions. . . . . . . . . . . . . . . . . . . . . . . . . 224
5.4 Optimisation of the Evaluation Process . . . . . . . . . . . . . . . . . . 226

**6 Proof of Concept** **229**
6.1 Simultaneous Stack-Based Abstract Machines . . . . . . . . . . . . . . 229
6.2 The iDBPQL Prototype System . . . . . . . . . . . . . . . . . . . . . . 231

**7 Summary** **239**
7.1 Main Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 239
7.2 Future Plans and Open Problems . . . . . . . . . . . . . . . . . . . . . 241

**Bibliography** **245**

**A The Syntax of iDBPQL** **255**
A.1 The Lexical Syntax of iDBPQL . . . . . 255
A.2 The Syntax of MetaData Catalogue Entries . . . . . 256
A.2.1 Syntax of DBS MetaData Units . . . . . 256
A.2.2 Syntax of Run-Time MetaData Units . . . . . 256
A.2.3 Common Syntax of Type-System-Related Definitions . . . . . 256
A.2.4 Common Syntax of iDBPQL Types . . . . . 257
A.3 The Remaining Syntax of iDBPQL . . . . . 258
A.3.1 Syntax of Evaluation Units . . . . . 258
A.3.2 Syntax of Evaluation Blocks . . . . . 258
A.3.3 Syntax of Statements . . . . . 258
A.3.4 Syntax of Expressions . . . . . 259
A.3.5 Identifiers, Labels, Values and More . . . . . 259

**B The *Parentage* Database Example** **261**

# List of Figures

2.1 Relationship Between Data Models and SBQL's Abstract Storage Model ([131, Figure 1]). . . . . 23

3.1 Architecture of the Distributed Object-Oriented Database System. . . . 33
3.2 HERM Diagram of the University Database. . . . . 40
3.3 Relationship between User Requests, Data Models and iDBPQL. . . . . 51

4.1 The Type System of iDBPQL. . . . . 71
4.2 Union Types, Sub-typing and Inheritance ([53, Figure 1]). . . . . 91

5.1 Sample Evaluation Graph for the `verifyEnrolment` Method. . . . . 143
5.2 Overview of Embedded References Between Instances of Classes of the `University`$_{CC}$ Schema Fragment as Considered in Example 5.3. . . . . 149
5.3 Architecture of the Persistent Object Store. . . . . 151
5.4 Architecture of the Transaction Management System. . . . . 163
5.5 Local Heap with Embedded POS and RCM Shared Memory Areas. . . 170
5.6 Logical View of the Environment and Result Stacks. . . . . 173
5.7 Logical View of the Effects of `evalMultiES ( )` Evaluation Procedures on ES. . . . . 184
5.8 Overview of the Evaluation Process of the `verifyEnrolment` Method from Figure 5.1. . . . . 186

6.1 Overview of a Stack-Based Abstract Machine with Distribution Capabilities. . . . . 230
6.2 Overview of a Local Stack-Based Abstract Machine. . . . . 231
6.3 Usage Diagram for the iDBPQL Prototype. . . . . 234
6.4 Evaluation of the *HelloWorld* Example. . . . . 235
6.5 iDBPQL and Internal Representations of the *Parentage* Database. . . . 236
6.6 Executing Requests on the *Parentage* Database. . . . . 237

# List of Tables

4.1 The Primitive Types of iDBPQL. . . . . . . . . . . . . . . . . . . . . . 62
4.2 Primitive Types and Their Supported Operations. . . . . . . . . . . . . 64

5.1 Modifier Flags and Their Interpretation. . . . . . . . . . . . . . . . . . . . 135
5.2 Internal Object Structure of Instances of Classes Presented in Example 4.12. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
5.3 (Binary) Path-Operators that are Supported by POS. . . . . . . . . . . 152
5.4 Type Information Mapping to Machine Instructions. . . . . . . . . . . . 181
5.5 Overview of Machine Instructions for Binary Expressions without Side Effects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
5.6 Overview of Machine Instructions for Assignment Expressions with Side Effects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
5.7 Overview of Additional Boolean Expressions and their Machine Instructions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191

# Chapter 1

# Introduction

Programming paradigms and, thus, programming languages (PLs) advocating single or multiple paradigms have been at the centre of research for many decades. C++ [128] is one of the prime examples of the object-oriented (OO) programming paradigm. It is often mentioned as the object-oriented programming language (OOPL) that resulted in programmers and organisations adopting this class of PLs. While C++ is referred to as an OOPL, it is designed to support elements of the procedural programming paradigm, the object-based programming paradigm, the object-oriented programming paradigm, and the generic programming paradigm. It is then up to designers and programmers to decide how to build a C++ program using those paradigm elements. In contrast, OOPLs such as Java [46] only support one paradigm, i.e. the object-oriented programming paradigm. It is claimed by many people that object-oriented programming (OOP) is advantageous to other programming paradigms in various ways. OOP is believed to provide more flexibility, simplify development and maintenance, better model real-world situations, and is widely popular in large-scale software engineering. The idea behind OOP is that a program is composed of a collection of objects that act on each other. Each object is capable of receiving messages (i.e. methods), processing data and sending messages to other objects.

Programming languages traditionally focus on processing. Apparently, data storage and data sharing only play a minor role. Long-term data is 'exported' to a file system or database system (DBS). DBSs, on the contrary, have been built to support large bodies of application programs that share data. The emergence of OOPLs have brought those two concepts closer together, i.e. data is at the centre of attention. Nevertheless, in OOPLs, the messages that objects accept are most important while DBSs (in particular object-relational and object-oriented DBSs) largely evolve around object persistence and data sharing. This, of course, results in a number of common (i.e. object-related) concepts being dealt with differently. Such concepts include the degree of encapsulation (OOPLs adhere to it rigidly while DBSs demand a more relaxed approach to support ad-hoc querying), treatment of transient and persistent data, incorporation of types and classes, inheritance, concurrency, `NULL` values etc.

In today's database (DB) marketplace, relational DBSs (RDBSs) and object-relational DBSs (ORDBSs) are dominant. Object-oriented DBSs (ODBSs) were originally deemed to replace RDBSs because of their better fit with OOP. However, high switching costs, the inclusion of object-oriented features in RDBSs, and the emergence of object-relational mappers (ORMs) have made RDBSs successfully defend their dominance in the DB marketplace. ODBSs are now established as a complement, not a replacement for (object-)relational DBSs. They have found their place as embeddable persistence solutions in devices, on clients, in packaged software, in real-time control systems etc. Especially the open source community has created a new wave of enthusiasm that is fuelling the rapid growth of ODBS installations. Christof Wittig (db4objects, Inc.) summarises the aim of developing ODBSs in [29]: '*Our mission is to give OO developers a choice, when it comes to persistence: Relational databases do a great job in server-centric environments, where the database is "above" the application and O/R mappers take care of the mapping. However, in resource-constrained environments [...], persistence strategies [...] have to look different. There's no value in mapping objects to tables – only cost.*'.

Traditionally, database management systems (DBMSs) provide support for query languages (QLs), most commonly SQL[1]-like languages. It has been common practice for decades to link both domains (i.e. PLs and DBSs) by embedding QLs into PLs. Popular examples include programming interfaces such as ODBC[2] and JDBC[3], which are both based on SQL. However, this embedded approach suffers from problems collectively known as *impedance mismatch* (refer to Section 1.1.1). Alternative integrated approaches circumvent these problems. However, the majority of them either represent a PL with added query constructs (e.g. DBPL [10, 11]) or a QL with added programming abstractions (e.g. SQL-99 [36], Oracle PL/SQL [40], Microsoft SQL Server Transact-SQL [47]). Only a few research projects (e.g. LOQIS [129] with its stack-based language SBQL [131] and TIGUKAT [100]) have attempted a seamless integration of programming and querying languages. In this thesis, we follow this line of research, in particular we adopt ideas of the stack-based approach (SBA) as proposed in [129, 130, 131]. The SBA approach has been used successfully to integrate querying constructs and programming abstractions into a uniform database programming language. Unfortunately, there are a number of issues, which are yet to be addressed by the SBA approach. Not only does it neglect transactions, concurrent and distributed processing, but it also rejects any type checking mechanism that originates from type theory. Among other things, we intend to rectify these shortcomings. We will propose an integrated, object-oriented DBPL that combines the advantages of OOP and DBS programming.

Before considering the contribution of the research work presented in this thesis in explicit detail, we will elaborate on our motivation and challenges that have to be faced.

## 1.1 Object-Oriented Database Systems and Programming Languages

Object-oriented database systems (ODBSs) first appeared in the 1980s, as both prototype systems and commercial products. Prototype systems developed in the 1980s

[1] SQL (i.e. Structured Query Language) is the most popular database language used to create, retrieve and manipulate data from relational database management systems (RDBMSs).

[2] ODBC (i.e. Open Database Connectivity) provides a standard software application programming interface (API) method for using RDBMSs.

[3] JDBC (i.e. Java Database Connectivity) is an API (similar to ODBC) for the Java programming language. It provides methods for querying and updating data in RDBMSs.

and early 1990s include Avance [18], ENCORE/ObServer [121], IRIS [42], ORION [62], and Starburst [84]. Commercial products released in that period include GemStone [24], $O_2$ [14], ObjectStore [76], ONTOS [99], ORION-2 [63], and VERSANT [139]. Initially, ODBSs have been regarded as the solution to the discovery of the limitations of RDBMSs and the need to manage a large volume of objects with object semantics. However, most of those systems disappeared again within a few years. Among others, Kim [60] investigated the reasons for their disappearance and also outlined research directions to be addressed before being able to build a sophisticated and successful ODBS. Four major concerns that have contributed to the disappearance of most of those early developments are:

1. '*There is no standard object semantics.*' [60, page 6]. Though various (complete) object-oriented data models have been introduced, the problem, however, was and still is that there is no consensus on a single model.
2. '*Object-oriented database systems today are 'pot-boilers'.*' [60, page 6]. Meaning that the push for object-oriented concepts and the push for solutions to a long list of deficiencies in conventional DBSs have been translated into object-oriented database systems. However, most of those issues are orthogonal to object-oriented concepts.
3. '*The vendors of early systems offered systems which seriously lack in database features.*' [60, page 8]. Common database features, such as transaction management and query optimisation, have not been incorporated.
4. '*There is apparently no formal foundation for object-oriented database systems.*' [60, page 8]. A solid theoretical framework only exists in a fragmentary way.

Driven by the demand to research and develop better ODBSs and the failure of those early developments, a large number of discussion papers emerged. Main discussion objectives included the definition of characteristics of ODBSs and a recommendation of the future path of ODBS research. The three most important discussion papers are the three manifestos. The first manifesto [8] aimed at providing a basis for further discussions. To do so, main features and characteristics of ODBSs have been proposed. These characteristics are separated in three groups, namely mandatory features, optional features, and open choices. Authors of the first manifesto strongly support the opinion that ODBSs are intended to replace relational technologies. In contrast, authors of the second manifesto [134] strongly oppose this view. Instead, they revive the relational model, but still agree with the need for support of object-oriented concepts. They take the stand that accomplishments gained through the relational model should not be discarded that easily. Instead an extension of relational technologies is what they desire the database community to focus on. The third manifesto [34] strongly agrees to carry forward accomplishments gained via the relational model. However, they highlight that the relational model is more powerful than its current implementations. Thus, some rethinking and re-development of existing approaches and implementations is demanded. The third manifesto can be seen as a push for object-relational database systems.

Considering today's ORDBSs, none of them comes close to the desired characteristics as outlined in the third manifesto. RDBMS vendors have realised the shortcomings of their products with respect to certain advanced applications. This has led to an extension of their systems with features adding support for some object-oriented concepts. However, current systems do not use a consistent data model and different object-oriented features are supported with little standardisation.

In April 1989, the Object Management Group (OMG), an international not-for-profit corporation, was founded. This group has been formed to establish industry guidelines and detailed object management specifications to provide a common framework for application development. Being frustrated with the lack of progress towards ODBS standards, ODBS vendors have implemented another consortium, the Object Data(base) Management Group (ODMG), in 1991[4]. The ODMG group has 'completed' its work on object data management standards in 2001 and was disbanded. In 2000, the final release of the ODMG standard, ODMG 3.0 [28], was published. The ODMG standard mainly consists of an object model, the object definition language ODL, the object query language OQL and various object language bindings. Even though the ODMG standard was developed by the major vendors of ODBSs, it has not been accepted as a standard for ODBSs. Reasons for this include the fact that ODMG's object model does not have the mathematical rigour associated with the relational data model and its relational algebra. Similarly, Java Data Objects (JDO) [55], another attempt to standardise data access, has not seen adoption by the mainstream. Instead, recent research (e.g. [32]) suggests to use the PL itself to query objects and thus make a separate OO query standard redundant.

While ORDBSs dominate the present database marketplace, a new wave of enthusiasm is fuelling the rapid growth of second-generation, native ODBSs. However, ODBSs are no longer seen as a replacement of existing RDBSs and ORDBSs. Instead, they are considered as a complement to such (object-)relational systems. In accordance with [98], applications in which ODBSs tend to provide a better solution than current RDBSs include:

- *Embedded DBMS Applications.* Such applications demand super-fast response times which translates into the requirement of a self-contained, non-intrusive, and easy-to-deploy persistence solution for objects. Storing objects 'just as they are in memory' is always the leanest and least intrusive way to implement a persistence solution. Using a relational or object-relational DBMS requires the overhead of object-relational mapping, resulting in an increased demand on resources.
- *Complex Object Relationships.* In such applications, classes define multiple cross-references among themselves. Considering (object-)relational DBMSs, relationships among objects are often modelled using foreign keys. Thus, retrieving an object together with objects it references and also those objects they reference etc. can result in complicated and difficult-to-maintain code. On the contrary, ODBMSs implement reachability persistence. That means that any object referenced by a persistent object is also persistent. Thus, storing or retrieving objects can be achieved with a single call. The ODBMS engine handles the details of maintaining the references when objects are stored, and satisfying them when objects are retrieved.
- *'Deep' Object Structures.* Not all data is easily organised into the tabular form associated with RDBMSs. Some data is best organised in tree or graph structures

[4] Note: Neither OMG nor ODMG are recognised standards groups. Their aim is – or was, in the case of ODMG – to develop de facto standards that will eventually be acceptable to recognised standards groups such as ISO/ANSI.

that can be tricky to support in an RDBMS. ODBMSs, however, require no conversion of the original structure into a DB model and the structure's integrity is preserved by the DB engine itself.
- *Changing Object Structures.* Most applications evolve as they age. Hence, the data structures they support must evolve as well. An ODBMS will typically weather data structure changes more easily than a RDBMS.
- *Developments Including Agile Techniques.* ODBMSs fit more flawlessly into agile development environments than RDBMSs.
- *OO Programs.*
- *Objects Include Collections.* A collection within an object often represents a one-to-many relationship. Such relationships, modelled by an RDBMS, require an intermediate table that serves as the link between the parent object and the objects in the collection. Meanwhile, ODBMSs treat a collection as just another object.
- *Data is Accessed by Navigation Rather Than Query.* Considering RDBMSs, navigation through a tree translates into a sequence of `SELECT` statements. In an ODBMS, navigation through the tree is expressed naturally using the constructs of the native language.

The ODBMS.ORG Panel of Experts [98] reason that this new interest in ODBS technologies results from the facts that:

1. '*Object databases [...] have long been recognised as a solution to one of the biggest dilemmas in modern object-oriented programming [...] the object-relational [...] impedance mismatch.*' and
2. '*Now that OOP languages like Java and .NET are finally becoming mainstream, this problem* [i.e. the object-relational impedance mismatch] *rests at the heart of information technology.*'.

Current ODBSs include Cachè, db4o, FastObjects, Gemstone/S, GOODS, JADE, Jasmine, JYD Object Database, Matisse, ObjectDB, Objectivity/DB, ObjectStore, ozone, TITANIUM, Versant Developer Suite, and VOSS. These systems address many of the shortcomings of early ODBS developments (e.g. support of core object oriented concepts and traditional database features) but they are still not based on sound theoretical foundations.

This brings us back to the three manifestos mentioned above. These papers, as well as a large body of corresponding discussion papers, identify desirable characteristics a system should have in order to be referred to as an ODBS. These characteristics can be summarised as follows:

- *Mandatory object-oriented concepts:* Complex objects, object identity, encapsulation, types and classes, class and/or type hierarchies, inheritance, and polymorphism (overriding and late binding);
- *Traditional DB features:* Persistence, secondary storage management, transaction support, concurrency control, recovery, ad-hoc query facility, and schema evolution; and
- *Optional features:* Multiple inheritance, type checking or type inferencing, distribution, replication, generic update operations, and version management.

Considering these characteristics, they all relate (directly or indirectly) to the language(s) an ODBS makes available to their (high- and / or low-level) users. Bringing our attention back to the early 1980s, the emergence of ODBSs has been

> '*[...] due to the simultaneous coming of age of object-oriented programming and the push for post-relational database technology. The discovery of the limitations of the relational database systems and the need to manage a large volume of objects with object semantics found in object-oriented programming languages led to the introduction of commercial object-oriented database systems in the mid- to late-1980s.*' [60, page 2].

Object-oriented programming languages are mainly based on two concepts, encapsulation and extensibility. Encapsulation leads to the distinction between the state of an object (i.e. its associated data) and the behaviour of an object (i.e. the associated code that operates on the data). Extensibility corresponds to the ability to re-use and extend an existing system without modifying it. Extensibility is provided in two ways: Behavioural extension (i.e. add-on of new programs) and inheritance (i.e. specialisation of existing objects). Those object-oriented concepts have evolved over time. They first appeared in programming languages (e.g. Simula-67 [97] and Smalltalk [45]), then in artificial intelligence, and finally in databases. Focusing on DBSs, research into semantic data models has led to data modelling concepts similar to those embedded in object-oriented programming and knowledge representation languages. Examples include the ERM and the HERM [133]. Kim concludes that

> '*[...] object-oriented concepts are the common thread linking frame-based knowledge representation and reasoning systems, object-oriented programming (application development) environments, and object-oriented advanced human interface systems. Therefore, they may be the key to building one type of intelligent high-performance programming system of the foreseeable future.*' [60, page 2].

Before considering differences in the interpretation of object-oriented concepts in the DBS community and the OOPL community, we first explain the previously mentioned term impedance mismatch and also address common approaches to link programming languages and database languages in greater detail.

### 1.1.1 The Impedance Mismatch

The term *impedance mismatch* refers to an inadequate or excessive ability of one system to accommodate input from another. Considering DBMSs, the *object-relational impedance mismatch* is often named as one of the central problems of research. It refers to a set of conceptual and technical difficulties which are often encountered when a RDBMS is being used by a program written in an OOPL. Robert Greene (Versant Corp.) highlights in [29] that '*Objects in the language and Relations in the database have always been at odds, as articulated in the classic problem known as "impedance mismatch".*' He continues to identify the two fundamental ways in which this mismatch materialises, i.e. as developer burden and in slower performance and / or resource consumption. While standardisation of mapping has brought relief to the former, the latter still poses a significant burden. Thus, Robert Greene concludes that '*Object databases, which don't suffer from the "impedance mismatch" problem, have proven to be a better solution for certain kinds of applications and have a renewed opportunity to expand their scope of use.*'. Let us consider those impedance related issues in more detail (also refer to [48]).

**Objects and RDBMSs.** Managing objects in RDBMSs requires to:

- 'Map' the object structure (OOPL) to a table structure (SQL);
- Create the table using SQL;
- Write SQL code 'inside' OOPL code to create, read, update, and delete objects;
- Explicitly instantiate objects when reading them from the DB; and
- Explicitly control the translation between SQL data types and OOPL data types.

While these steps are necessary for simple objects, the list of conversion tasks grows much longer when dealing with complex objects, e.g. involving numerous object references. Since RDBMSs do not support references (apart from simple foreign key relationships), one has to model object references, e.g. by maintaining them in additional tables.

**Objects and Object-Relational Databases.** Object-relational DBMSs (ORDBMSs) can ease some of the conversion tasks required for RDBMSs. They manage the translation between objects and relational tables transparently. However, programmers still have to specify which objects are to be made persistent, and how their contents are mapped to the table. How this is done depends on the particular ORDBMS. Common tasks include:

- Derivation of those application classes that are to be stored in the DB from a base persistent class.
- Creation of a descriptor file, which includes mappings of classes to tables and instance variables to columns.
- Creation of a deployment descriptor file, which includes a DB driver class, DB aliases, DB authentication information etc.

**Objects and Pure Object Databases.** An ODBMS stores and retrieves objects. The structure of an object is commonly defined by its class. Relationships between objects – which would be handled by `JOIN` operations in RDBMSs – are modelled via references. This results in a number of distinguishing features of ODBMSs: On one hand, *the class structure is the schema*, i.e. associations and relationships are built into the class architecture. On the other hand, there is *the treatment of persistence*. ODBMSs provide two general persistence techniques, which are *explicit persistence* (e.g. db4objects [102]) and *transparent persistence* (e.g. JDO [55]). With explicit persistence, DB operations (e.g. storing, retrieving and deleting objects) appear in code. With transparent persistence, on the contrary, objects are moved to and from the DB invisibly.

### 1.1.2 On the Integration of Programming and Query Languages

The relationship between query languages and general-purpose programming languages has been studied for decades. As we have already mentioned, the popular classification distinguishes between the embedded approach and the integrated approach. While embedded approaches suffer from the impedance mismatch, integrated approaches circumvent these problems. The majority of current integrated approaches either represent a programming language with added QL constructs or a query language with added PL abstractions. The former approach provides full computational and pragmatic universality, and clean semantics whereas the latter, commercially more popular one, provides user friendliness, macroscopic programming, declaritiveness, and data independence.

There are a number of reviews of existing (integrated) database programming languages (DBPLs). For instance, [79] contains an elaborate analysis of requirements on the type system underlying object-oriented DBPLs together with a review of existing DBPLs. The review was written as part of the TIGUKAT project [100], which aimed at developing a novel ODBS. TIGUKAT researchers proposed a novel object model whose identifying characteristics include a purely behavioural semantics and a uniform approach to objects. Research results of interest to this thesis include a type system for object-oriented DBPLs [78] and a draft TIGUKAT user (query) language [81]. However, research has been terminated (in 1999, to the best of our knowledge) without addressing problems arising when including data creation and manipulation statements, and programming language constructs into a behavioural database language.

Another, from a practitioners point of view, more interesting approach is presented in [131]. The seamless integration of a query language with a programming language is investigated. Thus, a foundation of a QL-centralised programming language according to the traditional paradigms of the programming languages domain is built. Researchers follow an extended approach to stack-based machines as known from classical PLs such as Pascal.

This *Stack-Based Approach (SBA)* includes the SBQL language which is an untyped, query-centralised language. A number of extensions have been proposed more recently, introducing types [77], object roles [54] and views [74]. While the TIGUKAT project is based upon a powerful type system (resulting in implementation challenges and performance problems as acknowledged in [78, pages 207–208]), the SBA approach rejects any type checking mechanisms that originate from type theory. Instead, a type system consisting of a metabase, static stacks and type inference rules is proposed. In addition, SBA lacks of any efforts that aim towards efficient evaluation. Concepts such as concurrent processing, transactions and distribution are neglected by both approaches altogether. Considering SBA, on one hand, we can summarise its main contributions:

- It demonstrates the potential of combining the advantages of programming abstractions and database programming to yield a powerful and universal language design.
- An abstract machine model originating from the PL-domain has been extended to suit the evaluation of both, QL constructs and PL abstractions.
- A stack-based query-centralised language SBQL is proposed. Operational semantics are defined for atomic queries, compound queries, selection, projection, navigation,

path expressions, natural join, quantifiers, bounded variables, transitive closure, ordering, null values and variants, assignments, and `for each` statements.

On the other hand, there are a number of shortcomings that must be addressed before incorporating this SBA approach into a full-fledged database management system. These shortcomings include:

- *It tightly couples SBQL to the persistent object store.* A more modular approach enabling a stack-based language to be used with a variety of object stores is desired.
- *It does not aim towards efficiency.* Operational semantics of SBQL language constructs are only based on basic algorithms. It is desirable to have a number of different implementations for each language construct available from which the most suitable (e.g. efficient) one can be selected for execution on grounds such as the objects involved, the configuration of the affected ODBS node(s), and the current system load(s).
- *It does not support simultaneous processing.* Only a single stack-based machine, which evaluates all SBQL statements in a serial manner, is defined. It is desirable to have a collection of these machines that cooperatively execute requests that are passed down from higher ODBS layers.
- *It does not support the concept of transactions.* It is desirable to execute operations in a way that data consistency can be ensured. Thus, transaction support must be taken into consideration.
- *It does not support distribution.* It is vital for the success of most systems nowadays to operate in distributed computing environments, in particular for DBSs. For instance, data should be stored at (or close to) the location it is used most frequently. Thus, it is desirable to have a network of stack-based machines that cooperatively (over many locations if necessary) execute DB operations.
- *It is not suitable for large databases,* as the stacks are main memory based. This limits the size of objects and also restricts scalability of the ODBS. It is desirable to support objects of any size. Most importantly, database systems and applications have to be scalable to adapt 'easily' to changes in business structures, business demands, customer demands etc.

In addition to the above, we consider the following decision, adopted more recently in the SBA approach, as unfortunate:

- *Widely accepted results originating from type theory are dismissed.* In [52, page 11] it is stated that '*The notion that a type is a (possibly infinite) set of values and that a variable of that type is constrained to assume values of that type is a misconception and should be rejected*'. In our opinion, it is desirable to have a sound type system from which values and (complex) objects are built. A lack of (strong) typing is likely to result in low reliability and affects the productivity of programmers. The necessity of typing together with discussions of related issues are underlined by a large number of researchers, including [11, 25, 79].

Research results presented in this thesis will address these issues. In addition, a number of more conceptual (and resulting implementational) challenges have to be taken into consideration. These are discussed next.

### 1.1.3 Database Programming Languages vs. Conventional Programming Languages

In order to support the majority of desired features and characteristics a system should have to be referred to as an ODBS (as outlined earlier), a number of issues have to be considered that are dealt with differently when considering conventional OOPLs and languages for ODBSs. These issues include the degree of encapsulation, treatment of transient and persistent data, the interpretation of the notions of type, class and inheritance, support of `NULL` values etc. Among others, [8, 19, 31, 58, 61] contributed to the discussion of these issues affecting the design and implementation of object-oriented DBPLs. We will summarise the main issues next.

Database systems have mainly been built to support large bodies of application programs that share data. Thus, the main focus was on data. On the contrary, traditional programming languages have focused on processing. Data storage and sharing only played a minor role. As mentioned before, the latter has changed with the appearance of OOPLs. However, differences still remain. While OOPLs evolve around the messages an object accepts, DBMSs focus on persistence and data sharing (i.e. exposing both structure and behaviour of objects). Hence, encapsulation is interpreted differently. Without relaxing the encapsulation property, support for ad-hoc querying, as known from relational DBSs, cannot be carried over into an integrated object-oriented database programming language. Atkinson et al. [8] suggests that supporting encapsulation is essential but that it may be violated under certain conditions.

Another issue to consider is the interpretation of the term *class*. In fact, we also have to include the notions of *type* and *inheritance* in this discussion. Considering OOPLs, there is no common approach. While some languages support types, others support classes or even both with various degrees of closeness (or separation if you prefer). There is a large body of research papers that discuss these different approaches. We will not consider such OOPL issues in detail but rather refer the interested reader to [25, 26, 30, 91]. Instead, we focus on differences between the ODBS domain and the OOPL domain with respect to these notions. While types and / or classes serve as data structuring primitive in both domains, they also serve as means of access to all objects of a particular type or class in DBMSs. For instance, a user querying a DBMS expects the ability to access all objects of type or class `Person`. In particular, ad-hoc querying makes frequent use of this type of access. Such types or classes can be regarded as (preferably system-maintained) collections of objects. This is not the case in OOPLs. In fact, such a property is not desired in OOPLs. Not only does this make garbage collection almost impossible[5] it also has the potential to violate data abstraction. The latter is true since objects become accessible through the type or class even though they were meant to be hidden from the user (i.e. only accessible through an abstract layer). In addition, a type or class as collection approach does not make sense for all type or class definitions. Imagine a type or class definition used in numerous application domains. There might be no meaningful relationship between the respective objects. Thus, a collection of these objects is not desired.

[5] Garbage collection revolves around references. If there are no more references to an object, it can no longer be accessed. Thus, it is garbage-collected. Having a type or class as collection approach always provides a means of access to objects without using object references. Thus, we cannot collect garbage in the usual sense.

Support of inheritance is an essential feature in both communities. However, there is no agreement on which types of inheritance are to be supported. While modern OOPLs commonly support multiple interface inheritance (e.g. Java and C#), there are numerous discussions on whether multiple implementation inheritance has its advantages. Most OOPLs (e.g. Java and C#) only support single implementation inheritance. Considering ODBSs, in particular, schema design, structural properties of multiple supertypes or -classes should be inheritable. Thus, multiple inheritance becomes a necessary language feature. However, this introduces a number of (mainly) implementational challenges. Multiple implementation inheritance causes several semantic ambiguities. These ambiguities (caused by the same class, say $A$, being inherited over two different paths) fall into two classes: Replicated inheritance (i.e. two different copies of $A$'s members are inherited) and shared inheritance (i.e. $A$ is shared between at least two classes, one from each path). The former ambiguity can be resolved more easily (i.e. through renaming) than the latter.

Another issue concerns the life-time of objects. Persistence was always at the core of DBSs in contrast to the PL-domain. Programming languages tended to rely on file systems or DBSs to maintain long-term data. This, of course, resulted in a non-uniform treatment of transient and persistent data. It is commonly accepted that persistence should be orthogonal (i.e. each object, independent of its type or class, is allowed to become persistent without explicit translation) [10, 11, 145].

Further issues encompass the inclusion of `NULL` values and the different focus of concurrency support. The former has only recently found its way into OOPLs, i.e. into the second release of C#. The latter is commonly centred around transactions in DBMSs (i.e. competition for resources) and around multi-threading in PLs (i.e. cooperation).

## 1.2 Contributions

Research presented in this thesis is closely related to the development of a distributed object-oriented database system [72]. The core of the physical system, that is the evaluation engine, of this ODBS is of particular interest. We will propose the language iDBPQL that is used to program the evaluation engine and the corresponding run-time environment that makes up the evaluation engine and executes iDBPQL programs. In addition, we will briefly describe two prototype implementations accomplished during the process of our research as proof of concept.

The fact that presented research results are linked to a bigger research project leads to our first objective. We must achieve a sufficient degree of independence to ensure that results are applicable to ODBSs other than our own. The proposed distributed ODBS makes provisions for this undertaking by following a strictly modular approach. We have enforced this objective in our approach. Among other things, the proposed iDBPQL language is not tied to a particular conceptual data model nor is it tied to a particular persistent object store. However, it cannot be denied that certain design decisions made for the distributed ODBS have also found their way into the physical system composed to effectively and efficiently drive this database system.

The first major contribution of this thesis is the proposal of the integrated database programming and querying language iDBPQL. Driven by the desire to address conceptual and implementational challenges faced when integrating object-oriented programming principles and database environments, we have designed a language that includes the following properties:

- It distinguishes between values and objects;
- It supports types (with user-defined type operations) that structure values and classes (exposing both data and behaviour) grouping objects;
- It contains pre-defined collection types such as `BAG`, `SET`, `LIST` as well as the possibility to add user-defined (collection) types;
- It introduces the ability to add the `NULL` value to any other existing type;
- It supports structural sub-typing;
- It provides a means of name-based access to all objects of (system-maintained) class-collections;
- It supports (name-based) multiple inheritance;
- It allows for the definition of (static) domain and entity constraints;
- It adds a `UNION`-type that supports the unification of identical or similar objects;
- It supports database schemata as collections of class and type definitions with their corresponding objects and values;
- It allows any language entity to persist;
- It provides a mechanism to model blocks of atomic statements and (local and distributed) transactions;
- It enables the implementation of all types of behaviour through single concepts, evaluation plans;
- It contains the usual assignment and control flow statements together with statements supporting collections;
- It supports common query expressions such as selection, projection, navigation, ordering and various joins;
- It includes additional expressions such as two types of renaming expressions, quantifier expressions etc.; and
- It provides a means of specifying simultaneous processing explicitly.

The second major contribution addresses (operational) semantics and the implementation of the iDBPQL language. We propose its internal representation, a corresponding run-time environment and interfaces of related ODBS components such as the persistent object store, the transaction management system and remote communication facilities. The run-time environment, together with iDBPQL metadata catalogues and ODBS system interfaces, is then used to specify the evaluation of iDBPQL expressions, statements, blocks, evaluation plans, and entire user requests. This specification is done in terms of operational semantics. While outlining operational semantics of the iDBPQL language, implementational challenges that stem from the integration of OOPLs and DBPLs are addressed.

Our final contribution corresponds to proof of concept implementations. The feasibility and practicality of the presented approach is demonstrated in terms of two prototype implementations. An initial prototype that extends the SBA approach with capabilities to process many transactions concurrently and function in a (virtual) parallel computing environment is introduced. The second, more sophisticated prototype implements the proposed run-time environment and processes iDBPQL evaluation plans in a single-node or distributed database environment.

## 1.3 Assumptions

Throughout the 1980s and early 1990s, object bases have been advocated as the superior database technology that will supersede relational database technologies. While various object base implementations emerged during that time, these systems did not 'live up to their expectations'. Among others, this failure can be explained by developers grounding their implementations on ideas that originated from relational technologies. Apparently, the latter is not a suitable technology for the processing of large sets of highly structured complex objects. In this thesis, we break with that school of thought in order to avoid the repetition of the mistakes made in the 1980s and 1990s. Instead, we have been looking for alternative (including previously neglected) ways to approach the integration of object-oriented programming and database technologies. In particular, research advances in the areas of programming languages, database programming languages and compiler construction have been beneficial to this research.

The proposal of a new and novel approach for the development of a complete distributed object-oriented database system is a research problem large enough for a dozen or more PhD theses. In order to achieve our particular research objectives, we have made a number of assumptions. These include:

- User requests arrive in a form that is suitable for processing by the underlying object base engine. That is, concepts such as user interaction, code compilation, fragmentation and allocation, code optimisation, code rewriting, execution plan generation etc. are beyond the scope of this thesis. Instead, we adopt a black box approach and assume that user requests are 'magically' transformed into a suitable form for request evaluation.
- Restricting our considerations to the level of request evaluation implies that typical high-level programming language concepts such as packages or modules, interfaces, type and class definitions, classes (as code structuring primitives) etc. are left aside. Those concepts, however, have to be considered when defining suitable high-level language interfaces.
  In order to assist with the readability and understanding of the language part of this thesis, various syntactically rich language constructs have been introduced in the proposed language. While it is not common to have a syntactically rich, intermediate-level language, the final version of the proposed integrated language will be less rich. But this does not mean that our efforts are wasted. Instead, most of those richer concepts will find their way into a corresponding high-level language once the envisioned ODBS environment has been finalised.
- The run-time environment that processes evaluation plans replies on the support by other database components such as a persistent object store, a multi-level transaction management system, caching mechanisms, and remote communication mechanisms. While we describe the general functionality and service interfaces of corresponding DBS components, it has to be acknowledged that there are still a number of open research problems, which need to be addressed before those components reach the necessary degree of maturity.

Assumptions are discussed in greater detail in the respective chapters. An overview of open research problems can be found at the end of this thesis.

## 1.4 Outline

The remainder of this thesis is organised in six chapters. Chapter 2 provides an overview of selected (database) programming and evaluation environments, and related research contributions that had a significant impact on the ODBS and DBPL research communities. Subsequently, in Chapter 3, we present our approach to develop a distributed ODBS that is based on a sound theoretical framework. The main focus is on how an integrated object-oriented DBPL fits into the envisioned ODBS. To assist this objective, an overview of how user requests are processed is presented. Chapter 4 proposes the intermediate-level, integrated database programming and querying language iDBPQL. The main focus is on the language's design and characteristics. In a nutshell, iDBPQL consists of metadata catalogues (holding transient and persistent entity definitions), evaluation plans that represent behaviour implementations (such as the user request, a type operation implementation or a method implementation), and the iDBPQL library (containing pre-defined language entities and their implementations). The internal representation of metadata catalogues and evaluation plans is presented in Chapter 5. In addition, service interfaces of ODBS components are outlined. These components are used subsequently when the operational syntax of iDBPQL's language constructs is presented. In Chapter 6, we discuss two prototype implementations that have been created to verify the feasibility of the chosen approach and that serve as proof of concept. Finally, Chapter 7 summarises this thesis and also provides an outlook for future research.

# Chapter 2

# A Review of Database Programming Languages and Related Concepts

As previously mentioned, the Stack Based Approach has been used successfully to seamlessly integrate querying constructs and programming abstractions into a uniform database programming language. First, we introduce this approach in greater detail. Subsequently, we will address other integrated languages and object-oriented (database) environments that have made significant contributions to the respective research areas.

## 2.1 The Stack-Based Approach

Subieta et al. [131] investigates the 'seamless' integration of a query language with a programming language. Thus, a foundation of a QL-centralised programming language according to the traditional paradigms of the programming language domain is built. An extended approach to two-stack abstract machines – known from classical programming languages such as Pascal – is presented. This proposal includes definitions of an abstract storage model, an abstract machine model, and semantics of query and programming language operators that are defined through operations on these stacks. Figure 2.1 shows the relationship between data models and the abstract storage model.

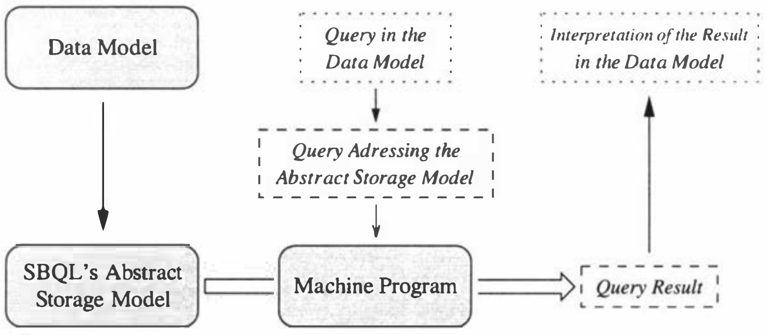

Data Model
Query in the Data Model
Interpretation of the Result in the Data Model
Query Adressing the Abstract Storage Model
SBQL's Abstract Storage Model
Machine Program
Query Result

**Fig. 2.1.** Relationship Between Data Models and SBQL's Abstract Storage Model ([131, Figure 1]).

In the abstract storage model, objects are defined as triples $< id, name, value >$, where *id* is an internal object identifier, *name* represents its external identifier, and *value* is either an atomic value, an identifier of another object or a set of objects. A database instance is viewed as a set of objects that satisfy the referential integrity constraint.

The abstract machine model introduces two stacks. These are the environment stack *ES* and the query result stack *QRES*. The environment stack determines scoping and binding[1]. *ES* never stores objects directly, instead it only holds (collections of) references to objects. The query result stack is a storage for intermediate results, used either for the evaluation of query operators or for the evaluation of arithmetic-style expressions. Results are represented as tables, which are bags of atomic values and internal object identifiers.

Semantics of operations are part of the proposed language SBQL (Stack-Based Query Language). It is an untyped, query-centralised programming language in the 4GL style. Evaluation of SBQL operations is performed by a single machine. The state of this machine is defined by the current database instance, *ES* and *QRES*. Semantics of SBQL operations are defined using top-down recursion in accordance with the respective query tree. In particular, semantics are defined for the following operations: Atomic queries, compound queries (using both algebraic and non-algebraic operators), selection, projection, navigation, path expressions, natural join, quantifiers, bounded variables, transitive closure, ordering, null values, variants, assignments, and `for each` statements. In addition, remarks are provided on how to deal with procedures, classes, class inheritance, methods, and encapsulation. As an example, let us consider operational SBQL semantics for the navigational join [131, pages 33 and 34] in greater detail.

EXAMPLE 2.1. SBQL supports only a single join operator, which is the navigational join. Its operational semantics are defined as follows:

```
procedure eval ( query: string );                          // the main eval procedure
begin
  ...
  if query is recognised as $q_1 \bowtie q_2$ then
  begin                                   // commence evaluation of a navigational join
    var RESULT: Table;                         // define result variable as a table
    RESULT := $\emptyset$;                                // initialise result variable
    eval ( $q_1$ );                                       // evaluate $q_1$ expression
    for each $r \in$ top ( QRES ) do          // for each row of the result of $q_1$ do
    begin
      push ( ES, nested ( r ) );                           // open a new scope on ES
      eval ( $q_2$ );                                     // evaluate $q_2$ expression
      RESULT := RESULT $\sqcup$ ( $\underline{r} \otimes$ top ( QRES ) );   // join and add to result
      pop ( QRES );                                    // cancel the result of $q_2$
      pop ( ES );                             // restore the previous state of ES
    end;
```

[1] Binding refers to the association between two things, such as the association of values with identifiers. Binding is closely related to scoping. In fact, scopes decide when binding occurs. In most modern PLs, the scope of a binding is determined at compile-time (i.e. static scoping). Alternatively, binding may be dependent on the flow of execution (i.e. dynamic scoping).

```
      pop ( QRES );                            // cancel the result of q1
      push ( QRES, RESULT );                        // compose result
    end;
    else ...
  end (*eval*);
```

$\underline{r}$ denotes a single-row table from the row $r$. $\otimes$ denotes the horizontal composition of bags. □

The SBA approach has been implemented in the LOQIS system [129].

More recently, this SBA approach has been refined (refer [130]). In contrast to the earlier approach, it now supports a hierarchy of object store models ranging from a simple version (called M0), which covers (nested-)relational and XML-oriented DBSs to higher-level object store models covering classes and static inheritance (in the model known as M1), object roles and dynamic inheritance (in the model known as M2), and encapsulation (in the model known as M3). In addition, a number of extensions have been proposed introducing types [52, 77], object roles [54] and views [74]. However, the shortcomings outlined in Section 1.1.2 still apply.

## 2.2 Overview of Existing Database Programming Environments

While the SBA approach is of particular interest to our research, there are numerous research findings that have influenced the iDBPQL proposal. In this section, we intend to provide a brief overview of research results that are important to the DBPL research community and, in particular, to our research.

In general, there are two approaches to develop ODBSs. On one hand, a type system from an existing OOPL can be adopted and then extended to include concepts vital for DBSs. On the other hand, research may start by proposing a new, purpose-built object model. For instance, the former approach has been adopted by the DBPL project (which is based on Modula-2), ObjectStore (which is based on C++) and Gemstone (which is based upon Smalltalk). The latter approach has been more popular among research prototypes such as $O_2$, SBA, TIGUKAT, and iDBPQL.

Persistence is a property not commonly supported by main stream programming languages. Thus, a large body of research has been devoted to address this shortcoming.

Next, we will introduce selected DB programming environments and important research contributions in more detail.

### 2.2.1 The Database Programming Language DBPL

The Database programming language *DBPL* [115] integrates programming abstractions and querying constructs by extending the Modula-2 [144] system programming language. Main extensions include the addition of bulk data management (through a bulk type constructor for 'keyed sets' or relations), access expressions (for accessing relational variables), persistent modules (by extending Modula-2's modularisation mechanism with a special `DATABASE MODULE` construct), and transactions (as special procedures).

While DBPL is not an object-oriented language, it still had a major impact on the development of object-oriented database languages. It demonstrated the feasibility and practicality of a database programming language that has been designed to advocate simplicity and orthogonality. In addition, DBPL supported an implementation (procedure) type, but without incorporating a notion of methods.

### 2.2.2 The $O_2$ Object Database System

The $O_2$ object database system [14] has been developed in the late 1980's and 1990's. Starting from various (research) prototype systems, $O_2$ became one of the most successful commercial object base systems by the end of the 1990's. The main objective underlying the development of $O_2$ was to build an environment for data intensive applications that integrates the functionality of a DBMS, a programming language, and a programming environment.

$O_2$ distinguishes between values and objects. Each value has a type, which describes its structure. Pre-defined primitives are associated with types that may be invoked upon corresponding values. Besides atomic values, sets, lists and records are supported. Objects belong to classes, have an identity and encapsulate values and user-defined methods. A class has a name, a type and a set of methods. Classes and types are partially ordered according to an inheritance relation. Multiple inheritance is supported. Persistence is user-controlled. An $O_2$ schema consists of classes, types, named objects and named values. A query language that uses the distinction between objects and values and the existence of primitives to manipulate structured values has been designed.

The $O_2$ object base contains the $O_2$ object manager [138] that is concerned with complex object access and manipulation, transaction management, persistence, disk management, and distribution in the adopted server / workstation environment. The object manager is divided into four layers:

- A layer that supports the manipulation of objects and values as well as transaction control;
- A memory management layer;
- A communication layer executing object transfers, execution migration, and application downloading; and
- A storage layer, on the server, provides persistence, disk management and transaction support.

Applications correspond to system processes. For every application there is one process on the workstation and one (mirror) process on the server. The lock table, the buffer and the object manager's memory are shared among all processes.

### 2.2.3 The Object Base Management System TIGUKAT

The *TIGUKAT* project [100] has been another attempt at developing a novel ODBMS that integrates object-oriented concepts with database systems. Characteristics of this approach include a purely behavioural object model (i.e. user interactions only happen through behaviour invocations), a uniform object model (i.e. everything is a first-class object; there is no separation between objects and values), and the vision that this uniformity property is extended to other system entities such as queries, transactions etc. Research started by investigating requirements of OOPLs, DBPLs and ODBSs. The selection of requirements was mainly theory-driven. As a result, numerous desired properties that are not commonly found in practical systems even within the respective domain have been added. For instance, support of multi-methods and parametric polymorphism together with inclusion polymorphism is uncommon even in today's OOPLs due to their complexity and associated implementation challenges. Leontiev et al. [79] outlines all identified requirements in more detail and also proposes ten tests that may be used to verify whether or not an object-oriented type system satisfies some or all of the desired properties. Leontiev [78, page 206] already acknowledges the complexity and difficulty of meeting the outlined set of requirements: '*[...] the desired combination of features remains elusive in that no type system, theoretical or practical, exists in a programming language or proposed, has all the features necessary for the consistent and uniform treatment of object-oriented database programming.*'. In the same article [78], a corresponding type system is proposed that passes all ten tests. While theoretical correctness was proven, practical verification or prototyping remained unaccomplished. Among other things, the complexity of verification algorithms is assumed to be exponential.

The TIGUKAT research project was terminated (in 1999, to the best of our knowledge) without addressing problems that arise when data creation and manipulation statements, PL constructs, transactions and distribution are included into the proposed behavioural DBS language. Nevertheless, this research project not only provides an extensive review of existing database and system programming languages, but it also influenced our research. In particular, the shape of parts of TIGUKAT's type system hierarchy and the adopted approach to class-based object access has affected the respective concept in the proposed language iDBPQL.

### 2.2.4 The Parallel Database Programming Language FAD

The *Franco-Armenian Data Model (FAD)* has been designed for highly parallel database systems. Besides query operations, the optimisation of programming language constructs, in terms of utilising parallelism, is considered to enhance system performance. The language FAD is presented in [13]. FAD provides a rich set of built-in structural data types and operations. It operates on sets and tuples, which can be nested within each other to an unlimited degree. FAD operators mainly utilise pipelining and set-oriented parallelism. For instance, a `filter` operator is proposed targeting sets. In addition, a `pump` operator, which supports parallelism by a divide and conquer strategy, targets aggregate functions. Both operators have influenced how iDBPQL supports simultaneous processing.

FAD was later extended [51] with communication primitives, in particular with asynchronous message passing mechanisms. The resulting language is called *PFAD*, which is restricted to a shared-nothing, distributed model of execution.

### 2.2.5 Additional Relevant Research Results on Database Programming Languages

The *Object Database and Environment (ODE)* [2] is based on the C++ [128] object paradigm. C++ classes are used for both database and general purpose manipulation. ODE provides its own database programming language O++ [1], which allows the definition, querying and manipulation of ODE databases. Persistence is modelled on the 'heap'. In particular, memory is partitioned into volatile and persistent. Volatile objects are allocated in main memory (on the heap). Persistent objects are allocated in the persistent store. An ODE database is a collection of persistent objects. Each object is identified by a unique object identifier, which is realised as (persistent) pointer to a persistent object.

*PS-Algol* [9] is an experimental language, which demonstrates that persistence can be achieved regardless of type. It is considered to be the first programming language that supports this orthogonal persistence property.

The support of *orthogonal persistence* is strongly advocated, especially for DBPLs. Among others, [10, 11] underline this importance. Three principles of persistence are outlined '*that should govern language design:*

– *Persistence should be a property of arbitrary values and not limited to certain types.*
– *All values should have the same rights to persistence.*
– *While a value persists, so should its description (type).*' [11, page 109]

In addition, [11] also advocates the support of *type completeness* (i.e. all data types should be of equal status) and adequate *expressive power* (i.e. computational power, database manipulation and database access).

This need for orthogonal persistence is reinforced in [10]. Technologies required to support orthogonal persistence are presented. These include a stable and reliable persistent object store, implementing persistence by reachability and type-safe linguistic reflection as defined in [126].

*Napier88* [93] is a persistent programming system that supports orthogonal persistence and type completeness. In contrast to its predecessor, PS-algol, the Napier88 system consists of its own, special-purpose built Napier88 language and a persistent environment. As such, it uses objects within the persistent store.

In Napier88, the philosophy that types are sets of values from the value space is adopted. This is in accordance with research results obtained in type theory (e.g. refer to [27]).

Concurrency is provided by threads and semaphores for co-operative and competitive concurrency, and designer transactions.

The Napier88 system is designed as a layered architecture consisting of a compiler, the Persistent Abstract Machine (PAM) and persistent storage architecture. Multiple incarnations of persistent stores and activations of the PAM are supported. However, only one PAM incarnation may work on one persistent store at any one time.

*Fibonacci* [5] is an object-oriented database programming language, which is based on the Galileo language [4]. Its main contribution is the proposal of three orthogonal mechanisms: Objects with roles, classes and associations.

Objects are encapsulated entities that can only be accessed by method invocation. Objects are grouped into classes. Internally, objects (e.g. persons) are organised as acyclic graphs of roles (e.g. students and academic staff members). An object can only be accessed through one of its roles. Also, the associated behaviour depends on the role used to access the object. This enables a person to be both a student and an academic staff member (without supporting multiple inheritance on the level of objects / classes). Associations relate objects that exist independently. In fact, they represent modifiable $n$-ary symmetric relations between classes.

Considering database functionality, Fibonacci supports common concepts such as persistence, transactions, queries, and integrity constraints. In addition, a modularisation mechanism is provided enabling the structuring of complex databases in interrelated units, and for the definition of external schemata.

The *Java* [46] application programming language has become the most popular object-oriented language to date. There are numerous languages derived from Java, such as *Orthogonally Persistent Java (PJava)* [12]. PJava extends Java by supporting the three principles of persistence as outlined earlier. Persistence is achieved through promotion. As the first object of a class is promoted, so is its class and all classes that are used to define it. Thus, the promotion algorithm maintains reachability (sub-)graphs.

In addition, research results from our own research group are incorporated in Chapter 3. Findings from other researchers that relate to a particular iDBPQL concept (e.g. support of `NULL`able types, union types and simultaneous processing), related ODBS components (e.g. the persistent object store and the multi-level transaction management system) and further implementation-specific issues are stated when the corresponding concepts are to be discussed.

# Chapter 3

# An Integrated Approach to Database Programming and Query Languages for Distributed Object Bases

The work presented in this thesis forms part of a bigger research project [72], which aims at developing a distributed ODBMS that is based on a sound theoretical framework. In this chapter, we will briefly cover the scope of this research project. After summarising challenges that have to be faced, we present an overview of the proposed ODBS architecture and, then, introduce selected concepts that are most relevant to this thesis in greater detail. In this process, we will see how an integrated, object-oriented DBPL fits into the envisioned ODBS.

Another aim of this chapter is to provide an overview of how user requests are processed. Main focus will be on the mapping of such requests to the ODBS component performing the evaluation [69].

## 3.1 A Distributed Object-Oriented Database System

The architecture of a distributed ODBS that is based on a sound theoretical framework is presented in [72]. While a large number of ODBSs have been proposed and implemented, a significant amount of fundamental problems have not been addressed fully. The latter, together with support for distribution, are the main focus of this research.

As mentioned in Section 1.1, the lack of standard object semantics was and still is one of the main disadvantages of ODBSs compared to commercially successful (object-)relational DBSs. Research in the last two decades has investigated complex values (i.e. data constructed by various type constructors) and references between data – which in fact lead to infinite, yet finitely representable structures. The *Object-Oriented Data Model (OODM)* [114] allows these different aspects to be combined. Starting from an arbitrary underlying type system, a schema is defined as a set of classes, each of which combines complex values and references. Thus, the theory of that data model can be tailored according to the underlying type system. This has been utilised in [109] in order to define a generic query algebra.

In order to satisfy the identified needs, it is a natural idea to develop a distributed database system based on this OODM. The first problem that has to be addressed is the distribution of the data. The fragmentation and allocation of OODM schemata have recently been addressed [86, 87, 110]. The result of fragmentation and allocation will be still an OODM schema, but each class will be allocated to exactly one node – or in the case of replication several nodes – of a network. As a consequence, each global object, which corresponds to the original schema, is represented by one or more local objects that correspond to the fragmented schemata. The term 'local' simply means that these objects together with all their sub-objects are physically located in the same ODBS node.

However, the structure of these local objects is still complex, whereas efficient storage and retrieval will require the provision of just records stored on pages. This implies a further decomposition of objects as we move closer to the physical storage. As a result, we obtain multiple levels of objects. The existence of multiple levels of objects allows the exploration of the concept of multi-level transactions [6, 16, 111, 141, 142]. Multi-level transaction schedulers take advantage of the fact that many low-level conflicts become irrelevant, if higher-level operation semantics are taken into account.

The multiple object levels are also reflected in the operational system, which utilises the ideas of stack-based abstract machines [131] to implement database functionality. However, these machines have to be extended in a way that they:

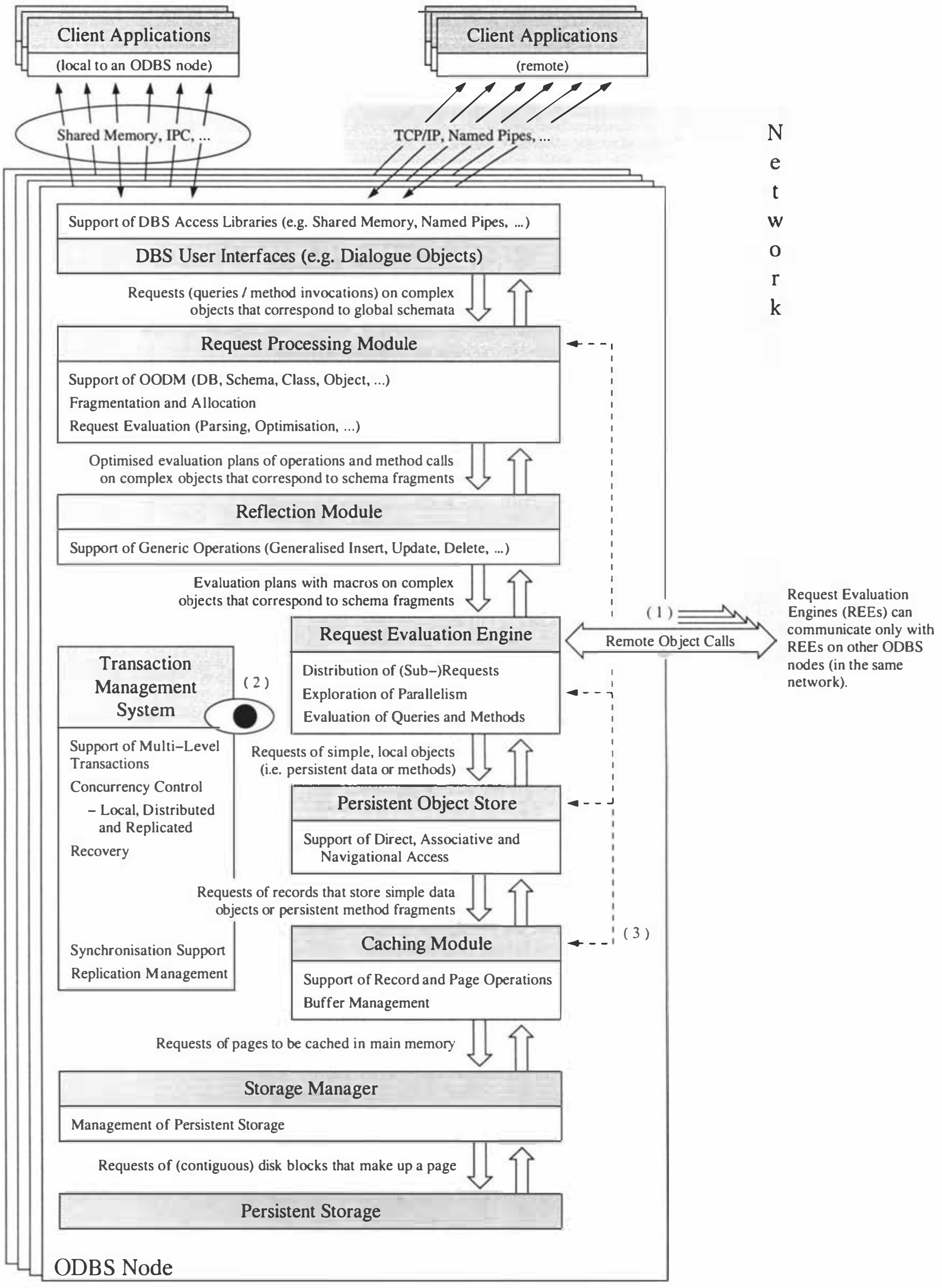
- Communicate with each other including communication via remote object calls;
- Run simultaneously;
- Are coupled with the transaction management system, the persistent object store and the caching module; and
- Reflect the operations on higher levels of the DBS.

A number of additional problems arise. These include the transformation of high-level queries and operations to the level of stack-based machines. For this, we utilise linguistic reflection [125, 126]. We provide a macro language, in which the high-level constructs in transactions such as generic update operations and the high-level algebra constructs for querying can be formulated. In [124] it has been shown how linguistic reflection can be used to expand such macros for the case of query algebra constructs. In [113] linguistic reflection has been applied to expand macros for generic update operations.

Next, we will introduce the architecture of a corresponding distributed ODBS (DODBS) in more detail. This is then followed by a discussion of some high-level concepts that are important to this thesis.

### 3.1.1 Architecture Overview

In this section, we provide a more detailed overview of the architecture of the distributed object-oriented database system as introduced in [72]. Figure 3.1 illustrates this architecture. Similar to most modern DBSs, the proposed system has a layered internal architecture. Each layer has an interface that provides services invoked by the


Client Applications
(local to an ODBS node)
Client Applications
(remote)
Shared Memory, IPC, ...
TCP/IP, Named Pipes, ...
Network
Support of DBS Access Libraries (e.g. Shared Memory, Named Pipes, ...)
DBS User Interfaces (e.g. Dialogue Objects)
Requests (queries / method invocations) on complex objects that correspond to global schemata
Request Processing Module
Support of OODM (DB, Schema, Class, Object, ...)
Fragmentation and Allocation
Request Evaluation (Parsing, Optimisation, ...)
Optimised evaluation plans of operations and method calls on complex objects that correspond to schema fragments
Reflection Module
Support of Generic Operations (Generalised Insert, Update, Delete, ...)
Evaluation plans with macros on complex objects that correspond to schema fragments
Request Evaluation Engine
Distribution of (Sub-)Requests
Exploration of Parallelism
Evaluation of Queries and Methods
(1)
Remote Object Calls
Request Evaluation Engines (REEs) can communicate only with REEs on other ODBS nodes (in the same network).
Transaction Management System
(2)
Support of Multi-Level Transactions
Concurrency Control
– Local, Distributed and Replicated
Recovery
Synchronisation Support
Replication Management
Requests of simple, local objects (i.e. persistent data or methods)
Persistent Object Store
Support of Direct, Associative and Navigational Access
Requests of records that store simple data objects or persistent method fragments
Caching Module
(3)
Support of Record and Page Operations
Buffer Management
Requests of pages to be cached in main memory
Storage Manager
Management of Persistent Storage
Requests of (contiguous) disk blocks that make up a page
Persistent Storage
ODBS Node

(1) Distribution of parts of evaluation plans that are scheduled to be executed on remote ODBS nodes
(2) Overlooks (i.e. synchronisation and serialisation of) database operations on local, distributed and replicated resources
(3) Requests to access / store data items (e.g. indices) that are cached (on pages) by the Cache Manager on behalf of a higher level module

**Fig. 3.1.** Architecture of the Distributed Object-Oriented Database System.

layer above it to implement their higher-level services. The DODBS consists of a collection of ODBS nodes (or ODBS instances) that process client application requests cooperatively. In Figure 3.1, modules (or components or layers if you prefer) are outlined together with the core functionality they provide. Also, the linkage between these modules is shown. Comments are added outlining how requests are passed downwards. Next, we introduce these modules in more detail starting with the persistent storage.

The *Database (DB)* can be seen as a collection of physical objects (i.e. disk blocks) stored on a collection of persistent storage devices. These objects can be accessed through the *Storage Manager*, the lowest layer of the distributed DBMS software. It deals with disk space management and supports efficient access to physical objects. Thus, a set of routines is provided to enable higher layers to allocate, deallocate, read, and write pages (i.e. a fixed number of disk blocks).

The *Caching Module* (also referred to as the *Buffer Manager*) maintains those pages in main memory. It partitions the available main memory into segments that hold collections of pages of the same fixed size (e.g. 8 KB, 16 KB, 24 KB, or 32 KB) and the same type (e.g. data pages or index pages). The collection of these segments are commonly referred to as the buffer pool. In addition, the caching module employs page replacement policies that aim to predict which pages are accessed next. Those pages can then be loaded into / kept longer in main memory. Thus, response times of page requests decrease if predictions are correct.

Besides page management in main memory, the caching module also supports the concept of records. Records are arranged on pages. Records that are commonly accessed together are mapped to the same page (whenever possible). Since multiple records are stored on a single page, record operations have to be synchronised. This is done by using short-term locks (i.e. latches).

The caching module has two well-defined interfaces: The page interface and the record interface. The page interface is available to all higher layers. Among others, it is used to store operational, organisational and auxiliary data (e.g. the transaction log, navigational access and associative index structures, lower portions of stacks of abstract machines etc.) persistently. The record interface is used for all (database) data requests. It is available only to the module on the next higher layer, the *Persistent Object Store (POS)*.

POS provides another level of abstraction by supporting storage objects. Storage objects are constructed from records and have a unique (storage) object identifier. POS maintains direct physical references between storage objects, and offers object-related, associative and navigational access to these objects. Access refers to the linkage between storage objects in order to reconstruct objects of a more complex structure. Object-related access refers to direct object access using object identifiers. Associative access means the well-known access via key values. Navigational access is related to the propagation along physical references. Section 5.2.1 discusses related concepts and also describes a corresponding prototype implementation in more detail.

The *Request Evaluation Engine (REE)* resides on top of POS. Actually, it is the module that executes client application requests as passed down from higher layers. This module is the focal point of this thesis. The REE employs a large number of agents that – cooperatively whenever possible – perform the work that applications request. Agents are realised as threads. The collection of agents that cooperatively evaluate an application request, on a particular DODBS node, belong to the same process. Within a process, agents are classified according to their role (i.e. master or slave). The layer of REEs is the lowest layer in the distributed DBMS that is aware of distribution.

Agents are designed to evaluate requests (i.e. requests formulated in iDBPQL) on locally or remotely stored objects. They utilise distribution and multi-threading, support the concept of (distributed) transactions and further optimise the processing of requests.

The existence of multiple object levels enables us to take advantage of a more sophisticated *Transaction Management System (TMS)*. It is based on the multi-level transaction model [16, 141, 142]. The transaction management system controls the execution of operations performed by REE agents and by POS. Hence, it ensures local and global serialisability. In general, a TMS consists of two components, the *Transaction Manager* and the *Recovery Manager*. The transaction manager takes advantage of 1) benefits resulting from the detection of pseudo-conflicts (i.e. conflicts that do not stem from a higher-level conflict), and 2) the fact that serialisability of a multi-level schedule can be achieved by serialising concurrent operations (also called sub-transactions) level-by-level. Hence, different level-specific concurrency control protocols can be employed. The recovery manager guarantees atomicity, durability and data consistency. This is achieved by maintaining local logs reflecting all updates to objects on all levels, by supporting complete and partial undo-operations of (sub-)transactions, redo-operations of (sub-)transactions, crash recovery etc. The ARIES/ML recovery mechanism [66, 111] is used to provide this functionality. It is an extension of the well-known ARIES recovery algorithm [92] to multi-level systems. Section 5.2.2 discusses related concepts and also describes a corresponding prototype implementation in more detail.

On the logical level, data is described in terms of a data model. The proposed system is based on the generic object-oriented data model (OODM) [114]. It considers objects as abstractions of real world objects. The OODM distinguishes between values and objects. Every object consists of a unique, immutable identifier, a set of (type, value)-pairs, a set of (reference, object)-pairs and a set of methods. The OODM is based on any arbitrary underlying type system. Types are used to structure values. Classes serve as structuring primitives for objects having the same structure and behaviour. A schema is given by a collection of classes. The operations provided by the underlying type system plus a single join operator allow to define a corresponding generic query algebra. Section 3.1.2 introduces the OODM in more detail.

In order to support distribution, certain fragmentation techniques are employed. These are splitting, horizontal fragmentation and vertical fragmentation. For this purpose, classes are considered. Each class is assigned to exactly one DODBS node – or in case of replication, to several DODBS nodes – in a network. Hence, fragmentation decomposes the global objects that correspond to the original schema into several local objects that correspond to the fragmented schema. Having fragmentation and a class / node relationship, we still have to allocate the fragments – including fragmented methods – to the corresponding DODBS node(s). Section 3.1.3 discusses fragmentation techniques in more detail.

Objects resulting from the fragmentation process do not correspond directly to objects as processed by REE agents. Furthermore, high-level queries, transactions, object methods etc. need to be translated into a language that can be interpreted by these agents. This conceptual gap between the logical OODM-level and the REEs is bridged by the *Reflection Module*. It supports linguistic reflection. A macro language is provided, in which the high-level constructs in transactions (e.g. generic update operations and the high-level algebra constructs for querying) are formulated. Section 3.1.4 discusses reflection in more detail.

The *Request Processing Module (RPM)* supports the OODM, fragmentation and allocation as discussed above. In addition, a request optimiser, which produces an *evaluation plan* for executing the user request, is employed. This evaluation plan (think of a blueprint for implementing the user request) is then passed to the reflection module, which replaces high-level constructs by macros. REEs will then use this evaluation plan with macros to evaluate the user request.

The internal representation of objects, data distribution, transactions etc. are hidden from the user. This is realised through user interfaces. In general, transactions started by a user will involve different operations on various classes. For instance, the work in [112] provides a mechanism to integrate the interface with the database. This is done by defining *dialogue classes*. Corresponding *dialogue objects* are defined by extended views. These dialogue objects can be created anywhere in the network. Initiating an operation associated with such a dialogue object would result in the execution of a top-level transaction. In this thesis, we will not directly focus on such high-level interfaces. Instead, in Section 3.2, we will outline our assumptions on how user requests arrive at a particular ODBS node and how they will then be processed.

### 3.1.2 Properties of the OODM

In the OODM [114] a *database schema* is a finite set of classes. Based on the fundamental distinction between general abstractions called values and application-dependent abstractions called objects, the OODM distinguishes classes from types [15]. *Types* can basically be seen as denoting sets of values[1]. Thus, we provide an underlying type system, e.g. $t = b \mid x \mid (a_1 : t_1, \ldots, a_n : t_n) \mid \{t\} \mid [t] \mid \langle t \rangle$ (using abstract syntax). Here, $b$ represents any collection of base types including one type $ID$ to be used for object identifiers, and at least one further type. $x$ represents type variables. $(\cdot)$, $\{\cdot\}$, $[\cdot]$ and $\langle\cdot\rangle$ provide constructors for tuple, set, list and multi-set (i.e. bag) types, respectively.

Given any type system, the structural part of a *class* $C$ is defined by a *structure expression* $exp_C$, which results from a type without occurrence of $ID$ by replacing all type variables $x_i$ by references $r_i : C_i$ with reference names $r_i$ and class names $C_i$, and by the set of *super-classes*. Therefore, the class names appearing in references and super-classes must be defined in a schema.

The behavioural part of a class is defined by operations (i.e. methods that may be invoked on any object of this class or its sub-classes) that are associated with the class. Such operations are defined using the usual control constructs of imperative languages.

In order to define *databases* over a given schema, we also need the *representation type* $T_C$ for a class $C$, which is simply obtained from $exp_C$ by replacing the references by the type $ID$. Then, in a database $\mathcal{D}$ each class $C$ of the schema is represented by a value $\mathcal{D}(C)$ of type $\{(\text{id} : ID, \text{val} : T_C)\}$ (i.e. by a finite set of identifier-value pairs).

[1] This is not completely correct according to the existence of type polymorphism as discussed in [27, 91]. However, for our purposes here this view suffices.

Clearly, we have to require some constraints to be satisfied: Uniqueness of identifiers, inclusion integrity with respect to super-classes, and referential integrity with respect to references.

However, there is another important requirement on databases called *value-representability*, a necessary and sufficient condition for the existence of generic update operations and the unique identifiability of objects. To explain this property, assume that for each $(i, v) \in \mathcal{D}(C)$, we expand the value $v$ into a rational tree [33], i.e. whenever an identifier $i'$ occurs within $v$ and this identifier corresponds to the reference $r : C'$, then we replace $i'$ in $v$ by the unique value $v'$ such that $(i', v') \in \mathcal{D}(C')$. This results in an infinite, yet finitely representable tree. The $\psi$-terms in [3] provide an example for such a finite representation. Value representability requires these rational tree values to be unique within each class. For formal details, we refer the reader to [114].

**Notes on the Choice of the Data Model.** The advantage of the OODM over other object models is the orthogonality of the type system, i.e. types can be arbitrarily nested. In particular, object references can appear deeply inside these nested structures, whereas in many other approaches, such as the ODMG model [28], the object model only supports attributes, which are either linked to types or to classes. In this way only types with an outermost record-constructor will be supported, and references can only appear directly inside this outermost constructor. This type orthogonality leads to a simple and handy, but very powerful structure that can be used equivalently to object identifiers, the structure of *rational trees* [33]. In particular, rational trees can be used to characterise value-representability as a desirable property of schemata, which becomes necessary for generic updates [114], and they can be used to define generic query algebras that can be parameterised by the type system [109]. This grounding in a solid mathematical theory enables one to identify theoretical challenges. Their solution may tell us which restrictions, for the sake of practical feasibility, should be requested – instead of starting with such restrictions in the first place.

### 3.1.3 Fragmentation

In order to support distribution of an object base we have to fragment the underlying OODM schema. The works in [110] and in [86] generalise horizontal and vertical fragmentation from the relational data model to the OODM. In addition, a third kind of fragmentation called split fragmentation is introduced.

*Split fragmentation* is based on the simple split operation, which replaces a class by two new classes, one referencing the other. Suppose the schema contains a class $C$ and the structure expression $exp$ occurs within the structure expression $exp_C$. Then we simply add a new class $C'$ with $exp_{C'} = exp$ to the schema and replace $exp$ in $exp_C$ by a new reference $r' : C'$.

The generalisation of *horizontal fragmentation* is also straightforward. According to the definition of databases for an OODM schema, each class $C$ will be associated with a set of pairs. Hence, we may partition this set into $\mathcal{D}(C) = \bigcup_{i=1}^{n} \sigma_{\varphi_i}(\mathcal{D}(C))$ with disjoint sets $\sigma_{\varphi_i}(\mathcal{D}(C))$. The fragments $\sigma_{\varphi_i}(\mathcal{D}(C))$ are obtained by operations of the query algebra.

We then replace $C$ in the schema by $n$ new classes $C_i$, all with $exp_{C_i} = exp_C$. However, as there may be classes $D$ referencing $C$, i.e. $r : C$ occurs within $exp_D$, we have to replace this reference as well. This is only possible, if the type system provides a (disjoint) union type constructor so that we can replace $r : C$ in $exp_D$ by $(a_1 : r_1 : C_1, \dots, a_n : r_n : C_n)$ with new pairwise distinct reference names $r_1, \dots, r_n$.

For *vertical fragmentation* of a class $C$ we assume that the outermost constructor in the structure expression $exp_C$ was the record type constructor, say $exp_C = (a_1 : exp_1, \dots, a_n : exp_n)$. As for the relational data model, we would like to replace $C$ by new classes $C_1, \dots, C_k$ with $exp_{C_i} = (a_1^i : exp_{i_1}, \dots, a_{n_i}^i : exp_{i_{n_i}})$ such that $\{a_1, \dots, a_n\} = \bigcup_{i=1}^{k} \{a_1^i, \dots, a_{n_i}^i\}$ holds, and for any database $\mathcal{D}$ we can reconstruct $\mathcal{D}(C)$ by joining the projections $\pi_{X_i}(\mathcal{D}(C))$.

There are two more problems we have to be aware of. The first concerns the handling of references to the class $C$, say $r : C$ in some $exp_D$. The second problem concerns the preservation of value representability. In relation to the first problem, the easiest solution is to replace $r : C$ in each $exp_D$ by a new structure expression with an outermost record constructor and new references $r_1, \dots, r_k$, i.e. replace $r : C$ by $(b_1 : r_1 : C_1, \dots, b_k : r_k : C_k)$.

The second problem is a bit more tricky. We must require that at least one of the new classes $C_i$ is value-representable. As the new classes will use all the same object identifiers, i.e. we always have $(i, v_j) \in \mathcal{D}(C_j)$, we could replace $r : C$ simply by $r : C_{i_0}$ choosing one of the new classes that is value-representable. The selected class $C_{i_0}$ must become a super-class for all the other new classes.

As to the operations associated with a class $C$, the fragmentation of the class will require to 'fragment' the operations as well. In the case of vertical fragmentation, each assignment will lead to several assignments with method calls among them. If the fragments are allocated to different nodes of the network, these method calls will become remote object calls.

### 3.1.4 Linguistic Reflection

In general, database requests correspond to user-issued transactions and queries. These transactions and queries will involve the complex operations of the query algebra, e.g. the generalised join and generic update operations (i.e. insert, update and delete). Each of these operations requires the analysis of the schema and the computation of the required types. It is known that both joins and generic updates are not parametric operations [113, 124]. In order to realise such complex operations, we apply a technique called *linguistic reflection.* We will consider these operations as macros for operations. Then the purpose of linguistic reflection is to expand these macros and to replace them by ordinary operations.

The basic idea of linguistic reflection is to use *reflection types* such as $SCHEMA_{rep}$, $CLASS_{rep}$, $TYPE_{rep}$, $METHOD_{rep}$, $COMMAND_{rep}$ etc. for the representation of abstract syntax expressions depicting schemata, classes, types, methods, commands (method bodies) etc., respectively. For each of these types, there exists a function *raise*, which associates a true schema, class, type etc., respectively, with the corresponding syntactic expression.

In particular, the macros for the complex query and update operations will first turn the classes, types etc., for which they are to be defined, into values of the corresponding reflection types. This is the effect of applying the operation *drop*. Then the computation will be performed on these values of the representation types. Finally, the result will be raised, which generates actual (low-level) operations. For technical details of reflection, we refer the reader to [113, 124].

### 3.1.5 A Simple University Application

Let us consider an example of an OODM database schema and corresponding fragmentation specifications. We will continuously revisit and refine this example throughout the thesis. Among other things, this will demonstrate how the conceptual specifications are represented and used at the layer of REEs.

EXAMPLE 3.1. Let us consider a simple university application similar to the one introduced in [133, pages 71–75]. The following information should be captured in our sample database schema:

- A collection of people working or studying at the university. A person has an identification number uniquely identifying this person. People have a name (consisting of titles, first name and last name) and an address (with street, city and postal code).
- A collection of students who are characterised by their student identification numbers. A student is also a person. Students have a major and may have a minor specialisation. They are supervised by at most one academic staff member.
- A collection of academic staff members with their specialisation. Academic staff members are people. Each staff member is associated with one department (consisting of a unique department name, location and a set of phone numbers).
- A collection of courses offered by the university and characterised by a unique course number, and a course name. A course can have different prerequisites.
- A course has a collection of lectures associated per semester (with year and semester number). Furthermore, each lecture has a room (with campus, building and room number) and a lecturer.
- A collection of projects characterised by a unique project identifier, a title, the beginning, and the end.
- Collections of student enrolments and student records. Students can enrol in a certain course during a semester. They obtain a final grade upon completion of the course.

Figure 3.2 shows the HERM diagram [133] modelling the university database. □

**The Global OODM Database Schema.** Now that we have described the university application informally, we will formulate a corresponding schema using the OODM data model from Section 3.1.2.

EXAMPLE 3.2. Let us continue with Example 3.1. A corresponding OODM database schema (without behaviour specifications) can be formulated as follows:

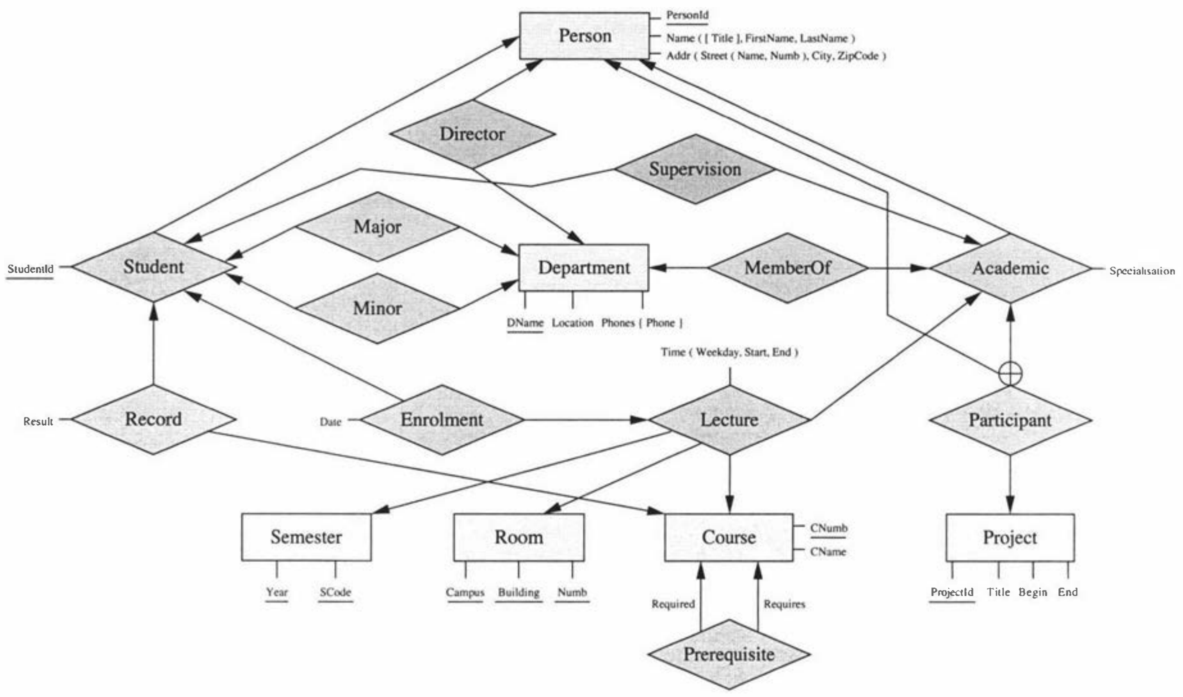


**Fig. 3.2.** HERM Diagram of the University Database.

```
SCHEMA University
  TYPE NameT    = ( titles : [ STRING ], firstName : STRING, lastName : STRING )
                  END NameT
  TYPE StreetT  = ( name : STRING, numb : STRING ) END StreetT
  TYPE AddressT = ( street : StreetT, city : STRING, zipCode : NATURAL )
                  END AddressT
  TYPE PersonT  = ( personId : NATURAL NOT NULL, name : NameT NOT NULL,
                  addr : AddressT ) END PersonT

  CLASS PersonC
    STRUCTURE PersonT
    CONSTRAINT UNIQUE ( personId )
  END PersonC

  TYPE CourseT = ( cNumb : STRING NOT NULL, cName : STRING NOT NULL ) END CourseT

  CLASS CourseC
    STRUCTURE
      CourseT,
      prerequisites    : { CourseC } REVERSE isPrerequisiteOf,
      isPrerequisiteOf : { CourseC } REVERSE prerequisites
    CONSTRAINT UNIQUE ( cNumb )
  END CourseC

  TYPE CampusT : ENUM ( "City Centre", "Lake Side", "The Oval" )
  TYPE RoomT   = ( campus : CampusT NOT NULL, building : STRING NOT NULL,
                 numb : STRING NOT NULL ) END RoomT
```

```
CLASS RoomC
  STRUCTURE RoomT
  CONSTRAINT UNIQUE ( campus, building, numb )
END RoomC

TYPE YearT         : INTEGER
TYPE SemesterCodeT : ENUM ( "first", "second", "double" )
TYPE SemesterT     = year : YearT NOT NULL, sCode : SemesterCodeT NOT NULL )
                     END SemesterT

CLASS SemesterC
  STRUCTURE SemesterT
  CONSTRAINT UNIQUE ( year, sCode )
END SemesterC

TYPE PhoneT      = ( phone : STRING ) END PhoneT
TYPE DepartmentT = ( dName : STRING NOT NULL, location : CampusT NOT NULL,
                   phones : { PhoneT } ) END DepartmentT

CLASS DepartmentC
  STRUCTURE
    DepartmentT,
    director      : PersonC,
    majorStudents : { StudentC }  REVERSE major,
    minorStudents : { StudentC }  REVERSE minor,
    staff         : { AcademicC } REVERSE staffMemberOf
  CONSTRAINT UNIQUE ( dName )
END DepartmentC

TYPE StudentT = ( studentId : NATURAL NOT NULL ) END StudentT

CLASS StudentC IsA PersonC
  STRUCTURE
    StudentT,
    major      : DepartmentC NOT NULL,
    minor      : DepartmentC,
    supervisor : AcademicC REVERSE supervises
  CONSTRAINT UNIQUE ( studentId )
END StudentC

TYPE AcademicT = ( specialisation : STRING ) END AcademicT

CLASS AcademicC IsA PersonC
  STRUCTURE
    AcademicT,
    staffMemberOf   : DepartmentC  REVERSE staff NOT NULL,
    lectures        : { LectureC } REVERSE lecturer,
    supervises      : { StudentC } REVERSE supervisor
  CONSTRAINT UNIQUE ( personId, staffMemberOf )
END AcademicC

TYPE MonthT   : NATURAL
```

```
  TYPE DayT     : NATURAL
  TYPE DateT    = ( year : YearT, month : MonthT, day : DayT ) END DateT
  TYPE ProjectT = ( projectId : NATURAL NOT NULL, title : STRING NOT NULL,
                    begin : DateT NOT NULL, end : DateT ) END ProjectT

  CLASS ProjectC
    STRUCTURE
      ProjectT,
      participants : { UNION ( AcademicC, PersonC ) };
    CONSTRAINT UNIQUE ( projectId )
  END ProjectC

  TYPE WeekDayT : ENUM ( "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" )
  TYPE TimeT        = ( hour : NATURAL, minute : NATURAL ) END TimeT
  TYPE LectureTimeT = ( weekDay : WeekdayT NOT NULL, start : TimeT NOT NULL,
                        end : TimeT ) END LectureTimeT
  TYPE LectureT     = ( time : LectureTimeT ) END LectureT

  CLASS LectureC
    STRUCTURE
      LectureT,
      course   : CourseC   NOT NULL,
      lecturer : AcademicC REVERSE lectures,
      semester : SemesterC NOT NULL,
      room     : RoomC
    CONSTRAINT UNIQUE ( course, semester )
  END LectureC

  TYPE EnrolmentT  = ( date : DateT ) END EnrolmentT

  CLASS EnrolmentC
    STRUCTURE
      lecture : LectureC NOT NULL,
      student : StudentC NOT NULL,
      EnrolmentT
    CONSTRAINT UNIQUE ( lecture, student )
  END EnrolmentC

  TYPE PassGradesT : ENUM ( "A+", "A", "A-", "B+", "B", "B-", "C+", "C" )
  TYPE FailGradesT : ENUM ( "D", "E" )
  TYPE MiscGradesT : ENUM ( "DNC", "Withdrawn" )
  TYPE GradesT : ENUM ( PassGradesT, FailGradesT, MiscGradesT )
  TYPE RecordT = ( result : GradesT ) END RecordT

  CLASS RecordC
    STRUCTURE
      course  : CourseC  NOT NULL,
      student : StudentC NOT NULL,
      RecordT
    CONSTRAINT UNIQUE ( course, student )
  END RecordC
END University
```

□

**Distributing the University Database.** Once the OODM schema is finalised, the database designer and / or database administrator will decide about the fragmentation and allocation of the global OODM schema. This may be done as described in the following example.

EXAMPLE 3.3. Let us continue with Example 3.2. Assume, we have three DBS nodes $N_{CC}$, $N_{LS}$ and $N_{TO}$ – one at each campus (refer to type `CampusT`; the index $_{CC}$ refers to "City Centre", $_{LS}$ refers to "Lake Side" and $_{TO}$ refers to "The Oval"). Information about rooms (i.e. class `RoomC`), lectures (i.e. class `LectureC`) and enrolments (i.e. class `EnrolmentC`) relevant to a particular campus should be stored at that campus only. General information about courses (i.e. class `CourseC`) and semesters (i.e. class `SemesterC`) is to be held at the main campus `"City Centre"`. Thus, the global OODM schema is amended as described below:

– Fragment class `RoomC` horizontally on attribute `campus` with $\varphi_{1_i}$ specified as follows:

$$\begin{aligned}\varphi_{1_{CC}} &\equiv \texttt{campus = "City Centre"};\\ \varphi_{1_{LS}} &\equiv \texttt{campus = "Lake Side"}; \text{ and}\\ \varphi_{1_{TO}} &\equiv \texttt{campus = "The Oval"}.\end{aligned}$$

– Fragment class `LectureC` horizontally on attribute `room` with $\varphi_{2_i}$ specified as follows:

$$\begin{aligned}\varphi_{2_{CC}} &\equiv \texttt{room.campus = "City Centre"};\\ \varphi_{2_{LS}} &\equiv \texttt{room.campus = "Lake Side"}; \text{ and}\\ \varphi_{2_{TO}} &\equiv \texttt{room.campus = "The Oval"}.\end{aligned}$$

– Fragment class `EnrolmentC` horizontally on attribute `lecture` with $\varphi_{3_i}$ specified as follows:

$$\begin{aligned}\varphi_{3_{CC}} &\equiv \texttt{lecture.room.campus = "City Centre"};\\ \varphi_{3_{LS}} &\equiv \texttt{lecture.room.campus = "Lake Side"}; \text{ and}\\ \varphi_{3_{TO}} &\equiv \texttt{lecture.room.campus = "The Oval"}.\end{aligned}$$

– Move classes `SemesterC` and `CourseC` to DBS node $N_{CC}$.

In addition, information about departments (i.e. class `DepartmentC`) and staff they employ (i.e. class `AcademicC`) is to be stored at the respective campus only. Thus, the global schema is further amended in the following ways:

– Fragment class `DepartmentC` horizontally on attribute `location` with $\varphi_{4_i}$ specified as follows:

$$\varphi_{4_{CC}} \equiv \texttt{location} = \texttt{"City Centre"};$$
$$\varphi_{4_{LS}} \equiv \texttt{location} = \texttt{"Lake Side"}; \text{ and}$$
$$\varphi_{4_{TO}} \equiv \texttt{location} = \texttt{"The Oval"}.$$

– Fragment class `AcademicC` horizontally on attribute `staffMemberOf` with $\varphi_{5_i}$ specified as follows:

$$\varphi_{5_{CC}} \equiv \texttt{staffMemberOf.location} = \texttt{"City Centre"};$$
$$\varphi_{5_{LS}} \equiv \texttt{staffMemberOf.location} = \texttt{"Lake Side"}; \text{ and}$$
$$\varphi_{5_{TO}} \equiv \texttt{staffMemberOf.location} = \texttt{"The Oval"}.$$

All remaining classes (i.e. `PersonC`, `StudentC`, `ProjectC`, and `RecordC`) are hosted at the main campus on DBS node $N_{CC}$.
As a result, the following schema fragments are present at each of the three ODBS nodes:

– Node $N_{CC}$ contains the following OODM fragment:

```
SCHEMA University_CC
  IMPORT SCHEMA University_LS, University_TO

  TYPE NameT    = ( titles : [ STRING ], firstName : STRING, lastName : STRING )
                  END NameT
  TYPE StreetT  = ( name : STRING, numb : STRING ) END StreetT
  TYPE AddressT = ( street : StreetT, city : STRING, zipCode : NATURAL )
                  END AddressT
  TYPE PersonT  = ( personId : NATURAL NOT NULL, name : NameT NOT NULL,
                  addr : AddressT ) END PersonT

  CLASS PersonC
    STRUCTURE PersonT
    CONSTRAINT UNIQUE ( personId )
  END PersonC

  TYPE CourseT = ( cNumb : STRING NOT NULL, cName : STRING NOT NULL )
                 END CourseT

  CLASS CourseC
    STRUCTURE
      CourseT,
      prerequisites    : { CourseC } REVERSE isPrerequisiteOf,
      isPrerequisiteOf : { CourseC } REVERSE prerequisites
    CONSTRAINT UNIQUE ( cNumb )
```

```
END CourseC

TYPE CampusT : ENUM (  "City Centre", "Lake Side", "The Oval" )
TYPE RoomT   = ( campus : CampusT NOT NULL, building : STRING NOT NULL,
                  numb : STRING NOT NULL ) END RoomT

CLASS RoomC_CC
  STRUCTURE RoomT
  CONSTRAINT UNIQUE ( campus, building, numb )
END RoomC_CC

TYPE YearT          : INTEGER
TYPE SemesterCodeT : ENUM ( "first" = 1, "second" = 2, "double" = 12 )
TYPE SemesterT     = year : YearT NOT NULL,
                      sCode : SemesterT NOT NULL ) END SemesterT

CLASS SemesterC
  STRUCTURE SemesterT
  CONSTRAINT UNIQUE ( year, sCode )
END SemesterC

TYPE PhoneT      = ( phone : STRING ) END PhoneT
TYPE DepartmentT = ( dName : STRING NOT NULL,
                     location : CampusT NOT NULL, phones : { PhoneT } )
                     END DepartmentT

CLASS DepartmentC_CC
  STRUCTURE
    DepartmentT,
    director      : PersonC,
    majorStudents : { StudentC }    REVERSE major,
    minorStudents : { StudentC }    REVERSE minor,
    staff         : { AcademicC_CC } REVERSE staffMemberOf
  CONSTRAINT UNIQUE ( dName )
END DepartmentC_CC

TYPE StudentT = ( studentId : NATURAL NOT NULL ) END StudentT

CLASS StudentC IsA PersonC
  STRUCTURE
    StudentT,
    major      : UNION ( DepartmentC_CC, DepartmentC_LS, DepartmentC_TO )
                 NOT NULL,
    minor      : UNION ( DepartmentC_CC, DepartmentC_LS, DepartmentC_TO ),
    supervisor : UNION ( AcademicC_CC, AcademicC_LS, AcademicC_TO )
                 REVERSE supervises
  CONSTRAINT UNIQUE ( studentId )
END StudentC

TYPE AcademicT = ( specialisation : STRING ) END AcademicT

CLASS AcademicC_CC IsA PersonC
  STRUCTURE
```

```
    AcademicT,
    staffMemberOf    : DepartmentC_CC     REVERSE staff NOT NULL,
    lectures         : UNION ( { LectureC_CC }, { LectureC_LS },
                       { LectureC_TO } ) REVERSE lecturer,
    supervises       : { StudentC }       REVERSE supervisor
  CONSTRAINT UNIQUE ( personId, staffMemberOf )
END AcademicC_CC

TYPE MonthT : NATURAL
TYPE DayT   : NATURAL
TYPE DateT  = ( year : YearT, month : MonthT, day : DayT )
                END DateT
TYPE ProjectT  = ( projectId : NATURAL NOT NULL, title : STRING NOT NULL,
                 begin : DateT NOT NULL, end : DateT ) END ProjectT

CLASS ProjectC
  STRUCTURE
    ProjectT,
    participant : { UNION ( { AcademicC_CC }, { AcademicC_LS },
                    { AcademicC_TO }, { PersonC } ) }
  CONSTRAINT UNIQUE ( projectId )
END ProjectC

TYPE WeekDayT      : ENUM ( "Monday", "Tuesday", "Wednesday", "Thursday",
                       "Friday" )
TYPE TimeT         = ( hour : NATURAL, minute : NATURAL ) END TimeT
TYPE LectureTimeT  = ( weekDay : WeekDayT NOT NULL,
                       start : TimeT NOT NULL, end : TimeT )
                       END LectureTimeT
TYPE LectureT      = ( time : LectureTimeT ) END LectureT

CLASS LectureC_CC
  STRUCTURE
    LectureT,
    course   : CourseC   NOT NULL,
    lecturer : UNION ( AcademicC_CC, AcademicC_LS, AcademicC_TO )
               REVERSE lectures,
    semester : SemesterC NOT NULL,
    room     : RoomC_CC
  CONSTRAINT UNIQUE ( course, semester )
END LectureC_CC

TYPE EnrolmentT = ( date : DateT ) END EnrolmentT

CLASS EnrolmentC_CC
  STRUCTURE
    lecture : LectureC_CC NOT NULL,
    student : StudentC   NOT NULL,
    EnrolmentT
  CONSTRAINT UNIQUE ( lecture, student )
END EnrolmentC_CC

TYPE PassGradesT : ENUM ( "A+", "A", "A-", "B+", "B", "B-", "C+", "C" )
```

```
  TYPE FailGradesT : ENUM ( "D", "E" )
  TYPE MiscGradesT : ENUM ( "DNC", "Withdrawn" )
  TYPE GradesT : ENUM ( PassGradesT, FailGradesT, MiscGradesT )
  TYPE RecordT = ( result : GradesT ) END RecordT

  CLASS RecordC
    STRUCTURE
      course  : CourseC  NOT NULL,
      student : StudentC NOT NULL,
      RecordT
    CONSTRAINT UNIQUE ( course, student )
  END RecordC
END University_CC
```

– Node $N_{LS}$ contains the following OODM fragment:

```
SCHEMA University_LS
  IMPORT SCHEMA University_CC, University_TO

  TYPE CampusT : ENUM ( "City Centre", "Lake Side", "The Oval" )
  TYPE RoomT   = ( campus : CampusT NOT NULL, building : STRING NOT NULL,
                   numb : STRING NOT NULL ) END RoomT

  CLASS RoomC_LS
    STRUCTURE RoomT
    CONSTRAINT UNIQUE ( campus, building, numb )
  END RoomC_LS

  TYPE PhoneT      = ( phone : STRING ) END PhoneT
  TYPE DepartmentT = ( dName : STRING NOT NULL,
                       location : CampusT NOT NULL, phones : { PhoneT } )
                       END DepartmentT

  CLASS DepartmentC_LS
    STRUCTURE
      DepartmentT,
      director      : PersonC,
      majorStudents : { StudentC }    REVERSE major,
      minorStudents : { StudentC }    REVERSE minor,
      staff         : { AcademicC_LS } REVERSE staffMemberOf
    CONSTRAINT UNIQUE ( dName )
  END DepartmentC_LS

  TYPE AcademicT = ( specialisation : STRING ) END AcademicT

  CLASS AcademicC_LS IsA PersonC
    STRUCTURE
      AcademicT,
      staffMemberOf   : DepartmentC_LS    REVERSE staff NOT NULL,
      lectures        : UNION ( { LectureC_CC }, { LectureC_LS },
                        { LectureC_TO } ) REVERSE lecturer,
      supervises      : { StudentC }      REVERSE supervisor
    CONSTRAINT UNIQUE ( personId, staffMemberOf )
  END AcademicC_LS
```

```
    TYPE WeekDayT      : ENUM ( "Monday", "Tuesday", "Wednesday", "Thursday",
                            "Friday" )
    TYPE TimeT         = ( hour : NATURAL, minute : NATURAL ) END TimeT
    TYPE LectureTimeT = ( weekDay : WeekDayT NOT NULL,
                            start : TimeT NOT NULL, end : TimeT )
                            END LectureTimeT
    TYPE LectureT      = ( time : LectureTimeT ) END LectureT

    CLASS LectureC_LS
      STRUCTURE
        LectureT,
        course   : CourseC   NOT NULL,
        lecturer : UNION ( AcademicC_CC, AcademicC_LS, AcademicC_TO )
                   REVERSE lectures,
        semester : SemesterC NOT NULL,
        room     : RoomC_LS
      CONSTRAINT UNIQUE ( course, semester )
    END LectureC_LS

    TYPE YearT        : INTEGER
    TYPE MonthT       : NATURAL
    TYPE DayT         : NATURAL
    TYPE DateT        = ( year : YearT, month : MonthT, day : DayT ) END DateT
    TYPE EnrolmentT = ( date : DateT ) END EnrolmentT

    CLASS EnrolmentC_LS
      STRUCTURE
        lecture : LectureC_LS NOT NULL,
        student : StudentC   NOT NULL,
        EnrolmentT
      CONSTRAINT UNIQUE ( lecture, student )
    END EnrolmentC_LS
  END University_LS
```

– Node $N_{TO}$ contains the following OODM fragment:

```
  SCHEMA University_TO
    IMPORT SCHEMA University_CC, University_LS

    TYPE CampusT : ENUM ( "City Centre", "Lake Side", "The Oval" )
    TYPE RoomT   = ( campus : CampusT NOT NULL, building : STRING NOT NULL,
                     numb : STRING NOT NULL ) END RoomT

    CLASS RoomC_TO
      STRUCTURE RoomT
      CONSTRAINT UNIQUE ( campus, building, numb )
    END RoomC_TO

    TYPE PhoneT       = ( phone : STRING ) END PhoneT
    TYPE DepartmentT = ( dName : STRING NOT NULL,
                          location : CampusT NOT NULL, phones : { PhoneT } )
                          END DepartmentT
```

```
CLASS DepartmentC_TO
  STRUCTURE
    DepartmentT,
    director      : PersonC,
    majorStudents : { StudentC }    REVERSE major,
    minorStudents : { StudentC }    REVERSE minor,
    staff         : { AcademicC_TO } REVERSE staffMemberOf
  CONSTRAINT UNIQUE ( dName )
END DepartmentC_TO

TYPE AcademicT = ( specialisation : STRING ) END AcademicT

CLASS AcademicC_TO IsA PersonC
  STRUCTURE
    AcademicT,
    staffMemberOf   : DepartmentC_TO    REVERSE staff NOT NULL,
    lectures        : UNION ( { LectureC_CC }, { LectureC_LS },
                      { LectureC_TO } ) REVERSE lecturer,
    supervises      : { StudentC }      REVERSE supervisor
  CONSTRAINT UNIQUE ( personId, staffMemberOf )
END AcademicC_TO

TYPE WeekDayT     : ENUM ( "Monday", "Tuesday", "Wednesday", "Thursday",
                      "Friday" )
TYPE TimeT        = ( hour : NATURAL, minute : NATURAL ) END TimeT
TYPE LectureTimeT = ( weekDay : WeekDayT NOT NULL,
                      start : TimeT NOT NULL, end : TimeT )
                      END LectureTimeT
TYPE LectureT     = ( time : LectureTimeT ) END LectureT

CLASS LectureC_TO
  STRUCTURE
    LectureT,
    course   : CourseC   NOT NULL,
    lecturer : UNION ( AcademicC_CC, AcademicC_LS, AcademicC_TO )
               REVERSE lectures,
    semester : SemesterC NOT NULL,
    room     : RoomC_TO
  CONSTRAINT UNIQUE ( course, semester )
END LectureC_TO

TYPE YearT       : INTEGER
TYPE MonthT      : NATURAL
TYPE DayT        : NATURAL
TYPE DateT       = ( year : YearT, month : MonthT, day : DayT ) END DateT
TYPE EnrolmentT = ( date : DateT ) END EnrolmentT

CLASS EnrolmentC_TO
  STRUCTURE
    lecture : LectureC_TO NOT NULL,
    student : StudentC   NOT NULL,
    EnrolmentT
  CONSTRAINT UNIQUE ( lecture, student )
```

```
    END EnrolmentC_TO
  END University_TO
```

Amendments with respect to the global schema have been highlighted. Identifiers of replicated types are underlined. Amendments resulting from fragmentation processes appear in italic text.

□

### 3.1.6 A Note on the Contribution of the Proposed ODBS

In terms of the potential contribution to the ODBS research community, the proposed architecture corresponds to a first milestone on our way to build a sound ODBS that is based on a solid theoretical framework. The proposed system meets the majority of desired features a system should have in order to be considered as an ODBS (refer to Section 1.1 for corresponding details). The only two concepts that are not yet taken into account are:

- Support for schema evolution. However, schema evolution is hardly provided by any DBS nowadays. We strongly think that this issue requires a separate research initiative;
- Support for version management. We chose not to consider this optional feature at this stage. Support for this feature can be added once a commercialisation process of our system gets on its way.

Having introduced the basic concepts underlying distributed ODBSs, we move on to consider the processing of user requests in such an environment.

## 3.2 Processing User Requests

In a truly distributed DBS (such as the one introduced in Section 3.1.1), a high-level DBS user is not aware of the distributed nature of the system. In fact, data independence, network transparency, replication transparency, and fragmentation transparency are key properties of this type of DBS. Thus, user requests are free of location- and communication-specific information. Only during the compilation, fragmentation, allocation, code rewriting, linking and optimisation processes such information is added (in the form of annotations).

Let us assume that high-level user requests arrive in the form of DBPQL[2] programs (or more precise modules). The request processing module employs a number of components that include a DBPQL compiler, code optimisers (performing compile-time code optimisation and also query optimisation), code rewriters (e.g. mapping operations on objects that correspond to the global OODM schema to operations on objects that relate to OODM schema fragments; such mappings are based on the fragmentation and

[2] DBPQL is a high-level database programming and querying language based on the intermediate-level iDBPQL language introduced in this thesis. DBPQL is a modular language that will support not only processing of transactions and queries but also provide support for generic requests, type, class and object creation and manipulation commands etc. We will not introduce DBPQL in detail but only refer to some of its aspects that relate to / stem from iDBPQL.

allocation catalogue), a reflection module (e.g. adding support for genericity by utilising linguistic reflection) etc. Details about these components and corresponding processes are beyond the scope of this thesis. Instead, we follow a black box approach. We assume that RPMs transform incoming user requests (i.e. DBPQL modules) into optimised evaluation plans, which are then translated into iDBPQL code, an intermediate-level version of DBPQL. Details about query optimisation and the generation of execution plans for ODBSs can be found in the literature, e.g. [127].

iDBPQL code is then evaluated by a collection (or better a network) of agents. Code evaluation is governed by REEs (there exists one REE instance per ODBS node). Agents of REEs are aware of the distributed nature of the DBS, they know about transactions, utilise simultaneous processing etc. Agent technologies[3] are utilised by all lower-level DBS components. Agents of different components speak different languages – usually referred to as *Agent Execution or Evaluation Language (AEL)*. However, they all use the same *(Agent) Communication Language DBACL* [67]. Separating AELs and DBACL is (kind of) straightforward due the fact that original, high-level user requests do not contain any location- or communication-specific information. While REE agents process evaluation plans formulated in iDBPQL they also interpret annotations that have been added by RPM's optimiser. In addition, REE agents also enhance processing further by utilising concurrent and distributed processing capabilities. We will discuss such details in Chapter 5.

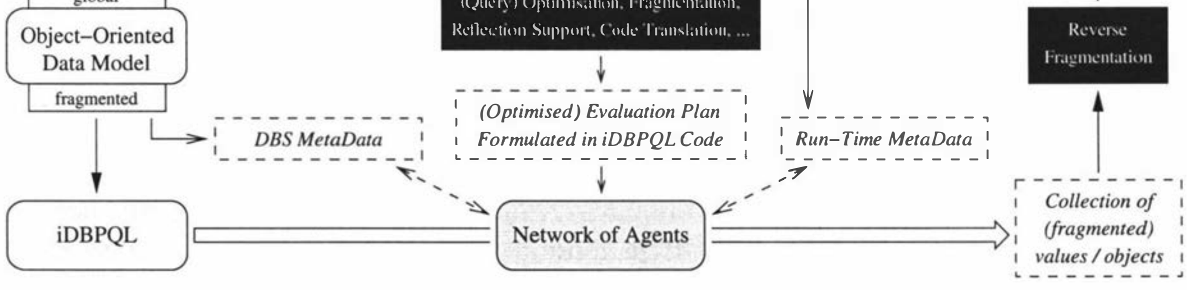

DBPQL Module Interface
DBPQL Module Interface
DBPQL
Imported DBPQL Modules
DBPQL Module
Collection of (global) values / objects
Code Analysis, Type Checking, Code Optimisation, Target Code Generation, (Query) Optimisation, Fragmentation, Reflection Support, Code Translation, ...
global
Object-Oriented Data Model
fragmented
Reverse Fragmentation
DBS MetaData
(Optimised) Evaluation Plan Formulated in iDBPQL Code
Run-Time MetaData
iDBPQL
Network of Agents
Collection of (fragmented) values / objects

**Fig. 3.3.** Relationship between User Requests, Data Models and iDBPQL.

Figure 3.3 provides a more abstract view of the relationship between DBPQL, iDBPQL, conceptual data models and associated processes. A high-level user is ex-

[3] An agent (or database agent or software agent) can be regarded as a piece of software (most commonly realised as a thread) that performs one or more relatively simple tasks to fulfil one or more given requests. Agents may work independently or cooperatively.

posed only to a few of these elements (refer to dotted rectangles in Figure 3.3), which are as follows:

- A programmer codes a new *DBPQL Module* together with its *DBPQL Module Interface(s)*. Modules are units that can be compiled separately. Thus, they can be considered as compile-time abstractions that advocate the development of large-scale programs through the support of import and export (by means of module interfaces) of services. Hence, information hiding is supported naturally.
- Within a module, *DBPQL Module Interfaces* of existing *DBPQL Modules* can be imported. A database schema can be regarded as just another module interface with (possibly) a reduced degree of encapsulation. Following our discussions in Section 1.1.3, we may have regular module interfaces that strictly follow the traditional PL-interpretation of encapsulation while database schemata expose both structural properties as well as behaviour. Thus, desired support for ad-hoc querying can be provided.
- Modules return results as outlined in the corresponding module interface(s). DBPQL statements and expressions may produce *collections of (global) values and / or objects* as defined in imported database schemata and local (transient) type and class definitions.

When processing user requests (i.e. DBPQL modules) code has to be parsed, analysed, type checked, optimised, fragmented, rewritten etc. As mentioned before, we will not consider such concepts in detail but only assume that at the end of these processes the following results are achieved:

- The DBPQL top-level function (i.e. the `MAIN` function or method) that initiates the execution is transformed into an optimised evaluation plan, which we refer to as the *Main Evaluation Plan*. An *Evaluation Plan* may have an *Initialisation Block* (which permits the initialisation of global and local elements before the start of the evaluation), it has an evaluation block and it has a number of associated metadata structures as indicated below.
  An *Evaluation Block* consists of iDBPQL control flow statements, assignments, expressions (e.g. queries), method calls (i.e. references to other evaluation plans), and sub-blocks. Concepts such as modules, interfaces, type definitions and their implementations, classes (as code structuring primitives), class definitions and their implementations etc. have been removed. For instance, the linker (part of the black box) has merged the user's DBPQL module with all DBPQL modules part of the corresponding import graph.
- One or more schemata from the DBS metadata catalogue are associated with the main evaluation plan. The *DBS MetaData catalogue* is a collection of compiled database schemata together with additional information about the location of schema fragments, replica management etc.
  All schema imports in the user's DBPQL module result in such associations. Due to fragmentation, a single DBPQL schema import may result in a number of associated iDBPQL schemata (i.e. schema fragments).
  Schema imports that originate from other modules that the user's DBPQL module imports may or may not be associated with the main evaluation plan. They may

only be attached to a single or multiple evaluation plans implementing a particular type operation or method.
- One or more entries from the run-time metadata catalogue are associated with each evaluation plan. The *Run-Time MetaData catalogue* is a collection of type and class definitions introduced in the source code of the user's DBPQL module or its imported modules. While DBS metadata catalogue entries describe persistent, shared data, run-time metadata catalogue entries relate to transient, non-shared data.

Besides the main evaluation plan, additional evaluation plans are associated with every non-abstract behaviour specification. Thus, evaluation plans are associated with type operation signatures and method signatures (static or non-static). Classes may have static variables that are declared outside of any method. Such declarations are captured in an initialisation block associated with the class. When invoking a method of that class, the corresponding evaluation plan is executed. At first, the evaluation plan's initialisation block, which may include a reference to the initialisation block of its class, is processed. Subsequently, the evaluation block is executed.

An abstract syntax of iDBPQL code that describes metadata entries is presented in Section 4.2. Section 4.3 will introduce an abstract syntax of iDBPQL constructs that make up evaluation plans. In addition, an *iDBPQL library* exists that provides definitions and implementations (i.e. evaluation plans) for all built-in features (e.g. primitive iDBPQL types, the structured iDBPQL type, iDBPQL type constructors and the `NULL`able iDBPQL type together with their type operations as well as built-in iDBPQL class definitions) that form part of iDBPQL as proposed in this thesis.

Finally, we want to underline some important properties of evaluation plans. They differ significantly from the original user requests, in particular in the following ways:

- iDBPQL code refers to schema fragments (i.e. DBS metadata entries) or transient types, classes etc. (i.e. run-time metadata entries). Original (high-level) code fragments have been amended in a way that references to higher-level features (e.g. class and schema constraints, generic operations etc.) have either been removed or replaced by macros (e.g. in case of generic operation support) formulated in iDBPQL code.
- A user program is translated into one main evaluation plan together with associated DBS and run-time metadata entries. Metadata entries and references to iDBPQL library features may have further evaluation plans (again with corresponding metadata entries) associated. Thus, the execution of the user program results in an evaluation of the main evaluation plan together with all evaluation plans that are encountered during its evaluation.
  In fact, every behaviour definition in the DBS and run-time metadata catalogues is defined in terms of an evaluation plan. Invoking such an evaluation plan from a remote DBS node will result in:
  - The evaluation plan (with some of its corresponding metadata entries) being transfered (i.e. replicated) to the remote node. The evaluation may then take place on the remote node; or, alternatively

- The evaluation plan is evaluated locally with a subsequent replication of the results to the remote node.

The exact procedure is determined during run-time based on the nature of behaviour involved, annotations by the code / query optimiser and some run-time properties.

- Definitions of types, sub-type relations, classes etc. are removed from the evaluation plans. These definitions are now part of the metadata catalogues. Every identifier found in the evaluation plan and every statement has a type annotation. The former has its type associated while the latter has the result type of its execution associated. Binding (which is executed as late as possible – as typical for object-oriented languages) is then based on those annotations. Instantiations (of objects, type parameters etc.) are also deferred to as late as possible.
- iDBPQL statements and expressions are allocated to DBS nodes (i.e. as annotations) indicating where their evaluation is to be done. It is assumed that this information is added by a code / query optimiser.
- Evaluation plans consist of blocks. `DO` ... `ENDDO` blocks are used to group statements together, describe atomic steps consisting of a number of iDBPQL statements, model local and distributed transactions, declare blocks that may be processed independently of others or concurrently with others etc. It is assumed that such block declarations are specified in evaluation plans.
  For instance, a high-level DBPQL module may have an implicit support for transactions. It may consider every invocation of a method detailed in a class definition of a database schema as a transaction. Horizontal fragmentation may result in the replacement of a single method call by multiple method calls where results are unified subsequently. The corresponding sequence of iDBPQL statements has to be grouped in a `DO TRANSACTION` ... `ENDDO` block. Otherwise, user transactions may be lost. Example 3.4 considers such a scenario in more detail.
- Indices and other information used to optimise processing are added (i.e. as annotations) to support the evaluation of iDBPQL statements more efficiently. Such annotations are described in more detail when discussing the execution of evaluation plans in Chapter 5.

We will conclude this chapter with a small example. Our aim is to demonstrate the general principle of the usage of iDBPQL. We will do so using a high-level syntax originating from our imagination. The iDBPQL syntax will be introduced in the subsequent chapter in more detail.

EXAMPLE 3.4. Let us use the `University` schema defined in Example 3.2. We will execute a simple request consisting of an import of the `University` schema and a selection on class `AcademicC`. We extract a set of academic staff members that specialise on the subject 'Database Systems'.

```
RUNNABLE MODULE FirstExample {

  IMPORT SCHEMA University;

  MAIN {
    SET ( AcademicC ) rslt;
    rslt = AcademicC WHERE ( specialisation == "Database Systems" );
08    }
09  }
```

Mapping this user module to the fragmented schemata from Example 3.3 results in:

- A main evaluation plan, named `FirstExample`, as detailed in lines 10 to 18.
- Associated DBS metadata entries for iDBPQL schemata `University`$_{CC}$, `University`$_{LS}$ and `University`$_{TO}$.
- Associated run-time metadata entries that correspond to types of intermediate results.
- Associated run-time metadata entries defining the final result type.

```
10  EVALPLAN FirstExample {
11    DO TRANSACTION tr1               // a transaction object is created implicitly
12      rslt1 = AcademicC$_{CC}$ WHERE ( specialisation == "Database Systems" );
13      rslt2 = AcademicC$_{LS}$ WHERE ( specialisation == "Database Systems" );
14      rslt3 = AcademicC$_{TO}$ WHERE ( specialisation == "Database Systems" );
15      rslt = ( rslt1.union ( rslt2 ) ).union ( rslt3 );
16      tr1.commit ( );                                  // explicit transaction commit
17    ENDDO;                                    // the transaction object is destroyed
18  }
```

Let us assume that the user request is received by the ODBS node $N_{CC}$. Thus, only AcademicC$_{LS}$ and AcademicC$_{TO}$ definitions refer to remote locations. A minimal set of corresponding annotations is as follows:

- Line 11: Transaction `tr1` is marked as read-only and distributed.
- Line 12: `rslt1` has a type annotation referring to the DBS metadata entry `University`$_{CC}$`.AcademicC`$_{CC}$.
- Line 13: `rslt2` has a type annotation referring to the DBS metadata entry `University`$_{LS}$`.AcademicC`$_{LS}$.
  `WHERE` has a processing annotation referring to node $N_{LS}$.
  `AcademicC`$_{LS}$ has a location annotation referring to node $N_{LS}$.
- Line 14: `rslt3` has a type annotation referring to the DBS metadata entry `University`$_{TO}$`.AcademicC`$_{TO}$.
  `WHERE` has a processing annotation referring to node $N_{TO}$.
  `AcademicC`$_{TO}$ has a location annotation referring to node $N_{TO}$.
- Line 15: `rslt` has a type annotation referring to run-time metadata entries specifying types

```
UNION ( University$_{CC}$.AcademicC$_{CC}$, University$_{LS}$.AcademicC$_{LS}$ ) AS _i1
// The UNION constructor creates a new super-class which corresponds to the
// union set of both specified classes. Since both classes are of identical
// structure, so is the resulting super-class.

UNION ( _i1, University$_{TO}$.AcademicC$_{TO}$ ) AS _i2
// Same as above. One can say that the resulting super-class with all its
// objects corresponds to the University.AcademicC class as outlined in the
// global schema in Example 3.2.
```

types, collection types (including `BAG`, `SET` and `LIST`) and the `NULL`able type. Types are later extended to include reference-types and the `UNION`-type supporting the unification of identical or similar objects. Sub-typing is structural (order, types and names are considered). While behaviour is not inherited, a sub-type can utilise its super-type's behaviour through type mapping. Classes are used to group more complex structures. The structure of a class of objects is defined over existing types, unnamed types (i.e. types without a behaviour that are defined in the class structure itself) and existing classes (either as inheritance or as reference). Classes are templates for creating objects, expose structural properties, allow for the definition of (reverse) references, may have associated behaviour (i.e. instance methods, class methods and object constructors – all of which are represented as an evaluation plan), support multiple inheritance, and may have associated, system-maintained collections through which access to all objects of a class and its sub-classes is possible.

Evaluation plans consist of control flow statements, assignments, expressions, method calls, and sub-evaluation blocks. Common programming abstractions (e.g. object creation statements, assignments, conditional statements, various loops and sub-routines), and query language constructs (e.g. selection, projection, navigation, join, and order-by) are supported. The integration of both concepts mainly evolves around collections. Evaluation blocks are used to group statements together, form atomic execution units, model local and distributed transactions, support independent or multi-threaded processing etc. Simultaneous processing is utilised to enhance performance. While the transaction management system allows different transactions to execute simultaneously (i.e. inter-transaction concurrency), iDBPQL also supports two expressions that explicitly request simultaneous execution. The latter may occur at the transaction-level (i.e. inter-transaction concurrency) or at the operation-level (i.e. intra-transaction concurrency).

In addition, there exists an *iDBPQL library*, which contains definitions and implementations for all built-in iDBPQL features.

## 4.2 Basic Language Concepts

In this section, we outline the fundamental concepts of the iDBPQL language. Rudimentary language elements, values, types, objects, classes, object-oriented concepts, and schemata are introduced. Initial integration issues have to be faced, e.g. the distinction of types and classes, the support of adequate types, a class-as-collection approach, inheritance, persistence etc.

First, we outline the main challenges addressed in this section in greater detail. Subsequently, an abstract syntax of the data model-related part of the iDBPQL language is proposed. When designing the language, particular attention has been paid to ensure that readers, who are familiar with modern object-oriented languages, find it easy to comprehend. While it is not common to have a syntactically rich, intermediate-level language, the final version of iDBPQL will be less richer than outlined in this thesis. However, our efforts will not be wasted. Instead, most concepts will find their way into a corresponding high-level language (which we referred to as DBPQL in Chapter 3) once the envisioned ODBS environment has been finalised.

### 4.2.1 Challenges

In this section, we will examine and address the following challenges:

- *Types vs. classes or objects vs. values*: There is no common agreement in the OOPL community on how to support or distinguish between types and classes. Numerous approaches have been proposed. The same applies to existing DBPLs. For instance, $O_2$ supports both values and objects with types and classes as their respective structuring primitives. In contrast, TIGUKAT considers everything to be an object, which has a class (providing a structural view of its objects) and multiple types (representing more abstract concepts, i.e. interfaces).
  We follow the approach advocated in [15] that can be found in $O_2$ and the OODM [114]. The concept of interfaces only appears on a higher DBS layer, where the envisioned DBPQL language resides.
- *Support of collection types*: Programming languages mainly include structural types while query languages commonly advocate collections. It is vital that a good mix of pre-defined structure types and type constructors is provided. In addition, users should be permitted to define their own types, in particular type constructors. This is necessary since no language designer can predict which types are desired to satisfy programmers' needs.
  iDBPQL supports common PL types such as structures and arrays as well as collection types such as sets, lists and bags.
- *Inclusion of `NULL` values*: DBSs commonly support `NULL` values for all their supported types including atomic types. This is not the case in PLs[1], which only support `NULL` values for reference-types.
  Similar to C#, we provide a `NULLABLE < _x >` type constructor that adds the `NULL` value to any existing type (that does not already support `NULL` values) specified as its type argument.
- *Relating types*: Types are typically arranged in a hierarchy. Creating such hierarchies may be name-based (i.e. users must explicitly relate types), structure-based (i.e. a type's structure defines its place in the hierarchy), behaviour-based (i.e. only the type's behaviour defines its place in the hierarchy) or a combination of the aforementioned approaches.
  We support structural sub-typing. Behaviours associated with other types may be utilised using type mapping.
- *The class-as-collection approach*: OOPLs enable object access through references only. DBSs, however, have always supported name-based access to all objects that belong to a particular entity, e.g. a relation or a class. While the latter approach is desired, it does not come without its challenges. Not only does it require the system to implicitly maintain collections associated with classes but it also requires concepts to be refined, e.g. garbage collection. Corresponding implementation issues are discussed later. Here, we are more concerned about the effects on the design of iDBPQL. While collections are associated with classes per default, programmers are given the choice to deactivate this feature.
  iDBPQL supports three types of classes, which are abstract classes, concrete classes and collection-classes. The latter is the default. Concrete classes are classes in the

[1] It should be noted that C# (version 2) has recently added this feature into its language.

more traditional OOPL sense, have the same properties as collection-classes apart from associated collections. Thus, objects of concrete classes may only be accessed by reference.

- *Multiple inheritance and its ambiguities*: Inheritance is a concept that has resulted in numerous discussions. Not only are there different types of inheritance, but also different ways of supporting a chosen approach.
  iDBPQL interprets inheritance as specialisation and supports multiple inheritance. While multiple inheritance is rarely supported in modern OOPLs, it is a strongly desired property of ODBSs. A class can be regarded as a specialisation of one or more existing classes, such as a student who is a person, an academic staff member who is also a person or an academic staff member who is also studying (i.e. is both a student and a staff member).
  Supporting multiple inheritance, however, does not come without its difficulties. The basic person-student-staff example already causes ambiguities. For instance, an academic staff member who is also studying inherits the properties of the person class over two paths. Any language supporting multiple inheritance must address related issues. iDBPQL supports primitives that enable programmers to resolve ambiguities in various ways.
- *Provision of a `UNION`-type*: Distribution in database systems is achieved by means of fragmentation. While iDPBQL operates on a DBS layer that is only aware of fragmented schemata and class and type definitions, it must support higher DBS layers to merge results of computations on fragmented schemata. Such results may then correspond to more global data abstractions, e.g. a global schema.
  iDBPQL provides a `UNION`-type that supports the unification of identical or similar objects. The resulting (set-union) type behaves like their least common super-type.
- *Inclusion of domain and entity constraints*: Database systems advocate the support of a number of (static) constraints such as `NOT NULL`, `CHECK` and `UNIQUE` constraints. Static means that the respective constraint can be checked by inspecting the most recent database state. Alternatively, if a sequence of database states has to be evaluated to verify a constraint, we have a dynamic constraint. Due to high performance overload, commercial database systems do not usually provide a general consistency enforcement mechanism.
  In iDBPQL, constraints transpire in one of the following two forms: Domain constraints (i.e. `NOT NULL` and `CHECK` constraints) and entity constraints (i.e. `UNIQUE` constraints). Enclosing data manipulation statements into atomic blocks (as introduced later) will result in a delay of the point in time where constraints are verified.
- *Transparent persistence*: Treating transient and persistent data uniformly is a desired property of every persistence mechanism. In addition, it should be possible that any data entity irrespective of its type may persist.
  iDBPQL does not distinguish between persistent and transient values, types, objects or classes. Persistence is supported simply by adding a class definition to a schema or by creating a new object on a persistent class. Persistence is accomplished by means of reachability.

### 4.2.2 Conventions

To further ease readability, we will adhere to the following conventions:

– All iDBPQL keywords are upper-case;
– Comments appear in italic;
– Names identifying type and class definitions start with an upper-case letter and end with either an upper-case `T` (for type definitions) or an upper-case `C` (for class definitions); and
– Variable names, type operation name, method names, label etc. start with a lower-case letter.

### 4.2.3 Literals, Names and Other Rudimentary Language Elements

Code written in iDBPQL can be regarded as sequences of keywords, literals, identifiers, names and operators. First, we introduce fundamental concepts such as literals and identifiers together with comments and names in more detail. Keywords and operators are discussed in subsequent sections.

Appendix A.1 contains a summary of the corresponding lexical iDBPQL syntax.

**Literals.** *Literals* are representations of values of primitive iDBPQL types, the `STRING` type and the `NULLable` type.

The `BOOLEAN` type has two values, denoted by the literals `TRUE` and `FALSE`.

The `CHARACTER` type has values as defined by the ASCII[2] standard. Character literals are enclosed in single quotes such as `'a'`, `'Q'`, `'\n'` etc.

The `STRING` type has values that correspond to sequences of character literals enclosed in quotation marks.

The `NATURAL` and `INTEGER` types have corresponding numerical literals expressed in decimal or hexadecimal. Decimal literals are sequences of digits either beginning with a non-zero digit or consisting of the single zero digit. Hexadecimal literals are sequences of digits with prefix `0x` or `0X`. For instance, the decimal literal `255` corresponds to the hexadecimal literals `0xFF`, `0XFF`, `0xff`, and `0Xff`.

The `REAL` type has values that correspond to sequences of digits containing a decimal point and, optionally, an exponent indicator.

The `NULLable` type extends any existing type with the `NULL` literal.

**Identifiers.** An *identifier* is an unlimited sequence of letters, digits and underscores. Identifiers must not start with a digit or two underscores. Identifiers beginning with a single underscore are reserved for type parameters. Identifiers are case sensitive.

**Comments.** *Comments* are line-oriented. A comment begins with a pair of slashes `//` and extends to the end of the line.

**Names.** A *name* denotes a variable, type operation, class, method, constraint etc. It can be simple or qualified. A *simple name* is an identifier. A *qualified* name is a sequence of identifiers separated by periods such as `obj.Name.firstName`, where `obj` is

[2] ASCII code [7] is only supported for the ease of prototyping. Unicode [135] will be used once the prototyping stage has been completed. Such a transition is easy to accomplish since the first 128 Unicode characters are the same as the ASCII characters, but with an extra leading **zero** byte in-front of them.

an instance of class PersonC. In addition to identifiers, qualified names may also contain any of the following keywords: THIS and SUPER (refer to Section 4.2.5 for corresponding details).

### 4.2.4 Types and Values

*Types*[3] structure values. Type definitions are used to define the common structure and the behaviour (i.e. type operations) of all values of a particular type. *Sub-typing* is structural. Thus, behaviour is not inherited. Nevertheless, a sub-type can utilise the behaviour specified for any of its super-types through type mapping. Let us consider these type-related concepts in more detail starting with primitive types and their built-in behaviour.

**Primitive Types.** The primitive types of iDBPQL are: BOOLEAN (abbreviated as BOOL), CHARACTER (abbreviated as CHAR), INTEGER (abbreviated as INT), NATURAL (abbreviated as NAT), and REAL. As indicated in Figure 4.1 (on page 71), INT, NAT and REAL are considered *numeric types*. Numeric types together with the CHAR type are *ordered types*. In addition, INT, NAT and CHAR are also *discrete types*. External representations of values of primitive types are literals as discussed in Section 4.2.3. Table 4.1 details domains of values, default values and examples of literals for each of the primitive types of iDBPQL.

| Primitive Type | Allowed Values | Default | Literals |
|---|---|---|---|
| BOOLEAN | TRUE, FALSE | FALSE | TRUE, FALSE |
| CHARACTER | any ASCII character | '\0' | 'a', 'A', '\n', ... |
| NATURAL | any non-negative Integer value | 0 | 0, 1, 6, 321, 14542, ... |
| INTEGER | any positive Natural number, their negatives and the number zero | 0 | -343, -12, -1, 0, 1, 542, ... |
| REAL | any floating-point number | 0.0 | -10.6E4, .5E-3, 3.1415, ... |

**Table4.1.** The Primitive Types of iDBPQL.

**Definition 4.1.** A binary SUBTYPE relation is defined on the set of primitive types. It is the smallest relation which is reflexive, transitive and has the following properties:

1. SUBTYPE ( CHAR, INT );
2. SUBTYPE ( NAT, INT ); and
3. SUBTYPE ( INT, REAL ).

□

SUBTYPE( $A$, $B$ ) means that $A$ is sub-type of $B$. Reflexive means that SUBTYPE ( $A$, $A$ ) holds for each primitive type $A$. Transitive means that, if SUBTYPE ( $A$,

[3] Types constructed over values, as considered in this section, are also referred to as *value-types*. In Section 4.2.5 we introduce objects and classes. Types containing references to objects are not considered value-types, they are referred to as reference- or object-types.

$B$ ) and SUBTYPE ( $B$, $C$ ) hold, then also SUBTYPE ( $A$, $C$ ). For example, each Natural number may be used anywhere a floating-point number can be used. Thus, an implicit type conversion will be applied to convert the Natural number to type REAL. It also means that type operations of type REAL can be applied to any value of type NAT.

Each primitive type has a number of associated type operations. These are as outlined in Table 4.2. Refer to your preferred book on the programming language C [59] or C++ [128] for a more detailed introduction into the meaning, side-effects and priorities of these operations.

**The Record Type.** Besides primitive types, iDBPQL also supports the specification of structured values through the record type constructor. Similar to C-like languages, the keyword STRUCTURE (or abbreviated as STRUCT) is used.

A structure consists of a list of members whose storage is allocated in an ordered sequence. Each structure definition creates a unique structured type within the respective scope. As outlined in Syntax Snapshot 4.1, the STRUCTURE keyword is optionally followed by an identifier, which gives a name to the structured type. The identifier can then be used with the STRUCTURE keyword to declare variables of that type without repeating a long definition.

**Syntax Snapshot 4.1** ***(The iDBPQL Record Type)***

```
StructuredType    = ( "STRUCT" | "STRUCTURE" ), [ Id ],
                    ( ( '{', { StructMemberDecl }, '}' ) | StructuredType ),
                    [ '&', StructuredType, [ "WITH", '{', { RenamingExpr }, '}' ] ];

StructMemberDecl  = StructuredType | VariableDecl;
VariableDecl      = ScopeModifierDecl, Type, Id;
ScopeModifierDecl = [ "PRIVATE" | "PUBLIC" | "READONLY" ];
```

□

Let us consider an initial example.

EXAMPLE 4.1. We intend to define a type that holds date values. We can define the structure of a corresponding myDate type as follows:

```
01  STRUCTURE myDate {
02    NATURAL day;
03    NATURAL month;
04    INTEGER year;
05  };
```

Now, we are able to create a variable, say bDay, of our new structured type and assign its value:

```
10    ...
11    bDay = ( 13, 4, 1976 );                              // record type assignment
12    ...
```

Alternatively, we may create the same structured value by assigning values to individual members (refer to lines 21 to 23):

| Operator | Description | BOOLEAN | CHAR | NATURAL | INTEGER | REAL |
|---|---|---|---|---|---|---|
| (unary operators) | | | | | | |
| + (prefix) | Plus (positive number) | No | No | Yes | Yes | Yes |
| - (prefix) | Minus (negative number) | No | No | Yes | Yes | Yes |
| ++ (prefix) | Unary preincrement | No | Yes | Yes | Yes | Yes |
| ++ (postfix) | Unary postincrement | No | Yes | Yes | Yes | Yes |
| -- (prefix) | Unary predecrement | No | Yes | Yes | Yes | Yes |
| -- (postfix) | Unary postdecrement | No | Yes | Yes | Yes | Yes |
| (arithmetic operators) | | | | | | |
| + | Addition | No | Yes | Yes | Yes | Yes |
| - | Subtraction | No | Yes | Yes | Yes | Yes |
| * | Multiplication | No | Yes | Yes | Yes | Yes |
| / | Division | No | Yes | Yes | Yes | Yes |
| % | Modulus | No | Yes | Yes | Yes | No |
| (assignment operator) | | | | | | |
| = | Assignment | Yes | Yes | Yes | Yes | Yes |
| (mixed arithmetic and assignment operators) | | | | | | |
| += | Addition assignment | No | Yes | Yes | Yes | Yes |
| -= | Subtraction assignment | No | Yes | Yes | Yes | Yes |
| *= | Multiplication assignment | No | Yes | Yes | Yes | Yes |
| /= | Division assignment | No | Yes | Yes | Yes | Yes |
| %= | Modulus assignment | No | Yes | Yes | Yes | No |
| (equality and relational operators) | | | | | | |
| == | Equal to | Yes | Yes | Yes | Yes | Yes |
| != | Not equal to | Yes | Yes | Yes | Yes | Yes |
| < | Less than | No | Yes | Yes | Yes | Yes |
| <= | Less than or equal to | No | Yes | Yes | Yes | Yes |
| > | Greater than | No | Yes | Yes | Yes | Yes |
| >= | Greater than or equal to | No | Yes | Yes | Yes | Yes |
| (logical operators) | | | | | | |
| && | Logical *AND* | Yes | Yes | Yes | Yes | Yes |
| \|\| | Logical *OR* | Yes | Yes | Yes | Yes | Yes |
| ! | Logical *NOT* | Yes | Yes | Yes | Yes | Yes |
| (bit-manipulating operators) | | | | | | |
| & | Bitwise *AND* | No | Yes | Yes | Yes | Yes |
| \| | Bitwise *OR* | No | Yes | Yes | Yes | Yes |
| ^ | Bitwise *XOR* | No | Yes | Yes | Yes | Yes |
| << | Bitwise shift left | No | Yes | Yes | Yes | Yes |
| >> | Bitwise shift right | No | Yes | Yes | Yes | Yes |
| ~ | Bitwise complement | No | Yes | Yes | Yes | Yes |
| (mixed bit-manipulating and assignment operators) | | | | | | |
| &= | Bitwise *AND* assignment | No | Yes | Yes | Yes | Yes |
| \|= | Bitwise *OR* assignment | No | Yes | Yes | Yes | Yes |
| ^= | Bitwise *XOR* assignment | No | Yes | Yes | Yes | Yes |
| <<= | Bitwise shift left assignment | No | Yes | Yes | Yes | Yes |
| =>> | Bitwise shift right assignment | No | Yes | Yes | Yes | Yes |

**Table4.2.** Primitive Types and Their Supported Operations.

```
20    ...
21    bDay.day = 13;
22    bDay.month = 4;
23    bDay.year = 1976;           // now, we have the same structured value as above
24    ...
```

□

Records only have a limited number of associated operations. These are as follows:

- Member access through the `.` (dot) operator (refer to Example 4.1, lines 21 to 23). Each member can then be treated according to its type.
- Assignment of a structured value of this record type through the '`=`' (assignment) operator (refer to Example 4.1, line 11).
- Concatenation of two record types through the `&` operator. This operation declares a new sub-type implicitly. Concatenations may cause naming clashes. Respective conventions are outlined in Example 4.2.

EXAMPLE 4.2. Let us define two structured types that can be used together to represent a machine's Internet Protocol (version 4) address. A class B network address (in decimal representation) may be defined as follows:

```
01  STRUCTURE myClassBNetworkID {
02    NATURAL byte1;
03    NATURAL byte2;
04  };
```

Within the network, two more bytes are available to distinguish local machines. Such a local host identifier (in decimal representation) may be defined as follows:

```
10  STRUCTURE myLocalHostID {
11    NATURAL byte1;
12    NATURAL byte2;
13  };
```

Based on the structure definitions above, we may now define a structure representing the Internet Protocol (version 4) address. This can be done as follows:

```
20  STRUCTURE myIPAddress STRUCTURE myClassBNetworkID & STRUCTURE myLocalHostID;
```

However, there are two obvious naming clashes. Without resolving them, the following code segment is likely to refer to the wrong member of the concatenated structure (assume variable `ip` is of type `STRUCTURE myIPAddress`).

```
30    ...
31    ip = ( 156, 17, 0, 250 );            // metadata: STRUCTURE myIPAddress ip;
32    ...
33    if ( ip.byte1 < 128 ) {                      // Which byte1 do we refer to?
34      ...
```

To avoid naming conflicts, explicit renaming of members is necessary. An example is shown next:

```
STRUCTURE myIPAddress STRUCTURE myClassBNetworkID & STRUCTURE myLocalHostID
WITH {
  myLocalHostID.byte1 AS byte3;         // refer to Section 4.3.5 where renaming
  myLocalHostID.byte2 AS byte4;     // expressions are introduced in more detail.
};
```

The corresponding segment of code must then appear as follows:

```
  ...
  ip = ( 156, 17, 0, 250 );                // metadata: STRUCTURE myIPAddress ip;
  ...
  if ( ip.byte3 < 128 ) {
    ...
```

Thus, naming clashes must be resolved by the programmer.

As a second example, let us consider again the `myDate` type from Example 4.1. Assume, we want to define a `myPerson` type consisting of the person's name and date of birth. This can be done as follows:

```
STRUCTURE myPerson {
  STRING name;
} & STRUCTURE myDate;
```

`myPerson` has four members, which are `name`, `day`, `month`, and `year`. The resulting type is identical to

```
STRUCTURE myPerson {
  STRING name;
  NATURAL day;
  NATURAL month;
  INTEGER year;
}
```

As a last example, we concatenate the `myDate` type with itself to form a `myDuration` structured type. This will involve renaming at least one type definition name (e.g. the second component as shown below) and at least half of the members of the new type. An example is shown next:

```
STRUCTURE myDuration STRUCTURE myDate & STRUCTURE myDate AS myToDate WITH {
  myDate.day     AS fromDay;
  myDate.month   AS fromMonth;
  myDate.year    AS fromYear;
  myToDate.day   AS toDay;
  myToDate.month AS toMonth;
  myToDate.year  AS toYear;
}
```

□

As already indicated in Table 4.1, primitive types have associated default values. So do record types. A record type's default value is defined recursively based on the default value of each of its members.

Sub-typing can be extended to include record types as follows:

**Definition 4.2.** For record types, the binary SUBTYPE relation is the smallest relation which is reflexive, transitive and has the following properties:

1. SUBTYPE ( STRUCT { type$_1$ a$_1$; ... type$_i$ a$_i$; ... type$_n$ a$_n$; }, STRUCT { type'$_1$ a$_1$; ... type'$_i$ a$_i$; ... type'$_n$ a$_n$; } ) if SUBTYPE ( type$_j$, type'$_j$) for all $j = 1, ..., n$;
2. SUBTYPE ( STRUCT $A$ & STRUCT $B$, STRUCT $A$ ); and
3. SUBTYPE ( STRUCT $A$ & STRUCT $B$, STRUCT $B$ ).

Where $A$ and $B$ are identifiers, type$_i$ member types and a$_i$ member names. □

**Type Definitions.** Before considering more complex types, we will introduce a means of specifying new types, i.e. user types. Type definitions are used to define the common structure and the common behaviour (i.e. type operations) of all values of a particular user type. Each type has a unique (within the particular scope) type identifier. Type parameters are supported in order to enable generic definitions. Details about such type parameters are discussed below. A type's structure definition is given in the form of an unnamed record type specification. This may be followed by a list of type operations. The corresponding syntax portion is detailed in Syntax Snapshot 4.2.

**Syntax Snapshot 4.2** ***(iDBPQL Type Definitions)***

```
TypeDefinition       = ScopeModifierDecl, ( UserTypeDecl | TypeSynonymDecl );

UserTypeDecl         = "TYPEDEF", Id, [ '<', TypeParameter-List, '>' ], [ "WITH",
                       '{', { TypeParaConstrClause }, '}' ], '{', StructuredType,
                       [ "BEHAVIOUR", { TypeOpSignature } ], '}';
TypeSynonymDecl      = ScopeModifierDecl, "TYPEDEF", NoneVoidType, Id;
TypeParaConstrClause = "SUBTYPE", '(', TypeParameter, ',', TypeId-List, ')', ';'
```
□

In Syntax Snapshot 4.2, TypeOpSignature refers to the specification of the signature of a (public or private) type operation. It consists of a type operation name, a list of formal input parameters and a formal output parameter. A type definition may not declare two type operations with the same signature.

Types with behaviour (i.e. one or more associated type operations) or a non-default default value (refer below) have an associated evaluation plan, which we will discuss in more detail in Section 4.3.

Notes:

1. Type definitions are based on the record type. While record types only cover value-types so far, their definition is later extended to include reference-types as well (refer to Section 4.2.5).
2. The keyword TYPEDEF has a second purpose. It is used to introduce synonyms for types which could have been declared some other way. The new type name becomes equivalent to the original type, i.e. they are sub-types of one another.

   EXAMPLE 4.3. Let us consider two examples. First, in line 01 an enumeration is given a synonym PassGradesT. Secondly, in line 02 the INTEGER type is given the synonymous name MyIntType.

```
TYPEDEF ENUM ( "A+", "A", "A-", "B+", "B", "B-", "C+", "C" ) PassGradesT;
TYPEDEF INTEGER MyIntType;
```

□

Sub-typing can be extended to include user-defined types as follows:

**Definition 4.3.** For user-defined types, the binary SUBTYPE relation is the smallest relation which is reflexive, transitive and has the following properties:

1. (for type definitions that are based on the structured type): SUBTYPE ( TYPEDEF $A$, TYPEDEF $B$ ) if SUBTYPE ( $a$, $b$ ); and
2. (for type synonym definitions of the form TYPEDEF $C$ $D$): SUBTYPE ( $C$, $D$ ) and SUBTYPE ( $D$, $C$ ).

Where $A$, $B$, $C$, and $D$ are identifiers, and $a$ and $b$ are the corresponding underlying record types of type definitions $A$ and $B$, respectively. □

**Type Parameters.** Type parameters are used to define *generic types*. Such generic types are instantiated to form *parameterised types* by providing actual *type arguments* that replace the formal type parameters. Let us consider an example.

EXAMPLE 4.4. First, we define a generic type Tcouple < _x, _y > where _x and _y are type parameters.

```
TYPEDEF Tcouple < _x, _y > {
  STRUCTURE {
    PRIVATE _x first;
    PRIVATE _y second;
  }
  BEHAVIOUR {
    first   ( )                            : _x;
    second  ( )                            : _y;
    isEqual ( Tcouple < _x, _y > couple2 ) : BOOLEAN;
    INIT    ( _x val1, _y val2 );
  }
};
```

Once a generic type is defined, it can be instantiated:

```
  ...
  Tcouple < PersonT, PersonT > married;
  Tcouple < NameT, INTEGER >   productQuantity;
  ...
```

Instantiations, such as Tcouple < PersonT, PersonT > or Tcouple < NameT, INTEGER > are called parameterised types, and PersonT, NameT and INTEGER the respective actual type arguments. □

Type parameters used above are *unconstrained*, i.e. every possible type may be passed to either one of the two type parameters of the generic type Tcouple. This introduces a number of challenges. For instance, it puts the ability to provide strong typing at risk. In iDBPQL, we follow a similar approach to current OOPLs such as Java and C#. For unconstrained type parameters, say `MyType < T >`, the only type operations available on values of type `T` are those defined for every type (as introduced further below). Since those type operations always exist, we can guarantee at compile-time that any type operation requested will succeed (at run-time).

In addition to unconstrained type parameters, iDBPQL also supports *constrained type parameters*. This is done by including a `WITH` expression (refer to Syntax Snapshot 4.2) into the type definition. An example is as follows:

EXAMPLE 4.5. Let us redefine the generic type from Example 4.4. Type parameters _x and _y are now constrained to the type `PersonT` and any of its sub-types.

```
TYPEDEF Tcouple < _x, _y > WITH {
  SUBTYPE ( _x, PersonT );
  SUBTYPE ( _y, PersonT );
} {
  STRUCTURE {
    PRIVATE _x first;
    PRIVATE _y second;
  }
  BEHAVIOUR {
    first   ( )                          : _x;
    second  ( )                          : _y;
    isEqual ( Tcouple < _x, _y > couple2 ) : BOOLEAN;
    INIT    ( _x val1, _y val2 );
  }
};
```

Considering the instantiations from Example 4.4 again, `Tcouple < PersonT, PersonT > married;` will still be valid while `Tcouple < NameT, INTEGER > productQuantity;` is not. □

**Collection Types.** Besides primitive types, the structured type and user-defined types, iDBPQL also contains the following built-in collection types[4]:

- `BAG` with `SET` and `EMPTYSET` as specialisations.
- `LIST` with `ARRAY`, `EMPTYLIST`, `STRING`, `ENUM`, and `SUBRANGE` as specialisations.

The syntax portion specifying built-in collection types is illustrated in Syntax Snapshot 4.3.

**Syntax Snapshot 4.3** ***(Built-In iDBPQL Collection Types)***

```
CollectionType = ( "BAG", '<', NoneVoidType, '>' ) |
                 ( "SET", '<', NoneVoidType, '>' ) |
                 ( "LIST", '<', NoneVoidType, '>' ) | ( "STRING" ) |
                 ( "ARRAY", '<', NoneVoidType, '>', '[', [ NaturalValue ], ']' ) |
                 ( "ENUM", '(', StringValue, { ',', StringValue }, ')' ) |
                 ( "SUBRANGE", '<', NumericType, '>', "FROM", NumericValue,
                   "TO", NumericValue );
```

[4] Values of collection types are also referred to as *collection values* whenever it is important to distinguish between the different types of values supported by iDBPQL. A collection value may be of a value-type or a reference-type.

□

Properties of these collection types can be summarised as follows:

- A BAG (also referred to as multi-set) is an unordered collection of elements in which the elements can have duplicate values. A BAG with no elements is called the EMPTYSET, which is the default BAG-value. For example, a BAG < INTEGER > (i.e. a multi-set of Integer values) might contain the collection { 12, -3, 3, 12, -3, 12, 1 }, which has duplicate elements.
- A SET is an unordered collection of elements in which each element is unique. A SET with no elements is called the EMPTYSET, which is the default SET-value.
- A LIST is an ordered (but not sorted) collection of elements that allows duplicate values. It differs from a BAG collection type in that each element in a LIST has an ordinal position in the collection. The order of the elements in a LIST corresponds to the order in which values are inserted. A LIST with no elements is called the EMPTYLIST, which is the default LIST-value.
- An ARRAY is an ordered (but not sorted) collection of elements of a fixed number. These collections may contain duplicate values and can be accessed by an index. An ARRAY with no elements is called the EMPTYLIST, which is the default ARRAY-value.
- A STRING is a sequence of CHARACTER values. Strings may be regarded as LIST < CHARACTER >. A STRING consisting of no characters has the same default value as a character value.
- An ENUM (i.e. enumeration) is an ordered collection of unique elements of the STRING type. Enumerations must have at least one element. The value of the first enumeration element is taken as the enumeration's default value.
- A SUBRANGE is a sorted collection of unique elements of a numeric iDBPQL type. The FROM value is used as the default, which might be the maximum (in case of a descending subrange) or the minimum (in case of an ascending subrange).

Figure 4.1 provides an overview of all (built-in) types of iDBPQL. Built-in type constructors have the usual type operations (e.g. refer to [88]).

Sub-typing can be extended to include collection types as follows:

**Definition 4.4.** For collection types, the binary SUBTYPE relation is the smallest relation which is reflexive, transitive and has the following properties:

1. SUBTYPE ( BAG < _x >, BAG < _y > ) if SUBTYPE ( _x, _y );
2. SUBTYPE ( SET < _x >, SET < _y > ) if SUBTYPE ( _x, _y );
3. SUBTYPE ( LIST < _x >, LIST < _y > ) if SUBTYPE ( _x, _y );
4. SUBTYPE ( ARRAY < _x > [ $n$ ], ARRAY < _y > [ $n$ ] ) if SUBTYPE ( _x, _y );
5. SUBTYPE ( SET < _x >, BAG < _x > );
6. SUBTYPE ( EMPTYSET, BAG < _x > );
7. SUBTYPE ( EMPTYSET, SET < _x > );
8. SUBTYPE ( ARRAY < _x > [ $n$ ], LIST < _x > );
9. SUBTYPE ( EMPTYLIST, LIST < _x > );
10. SUBTYPE ( EMPTYLIST, ARRAY < _x > [ $n$ ] );

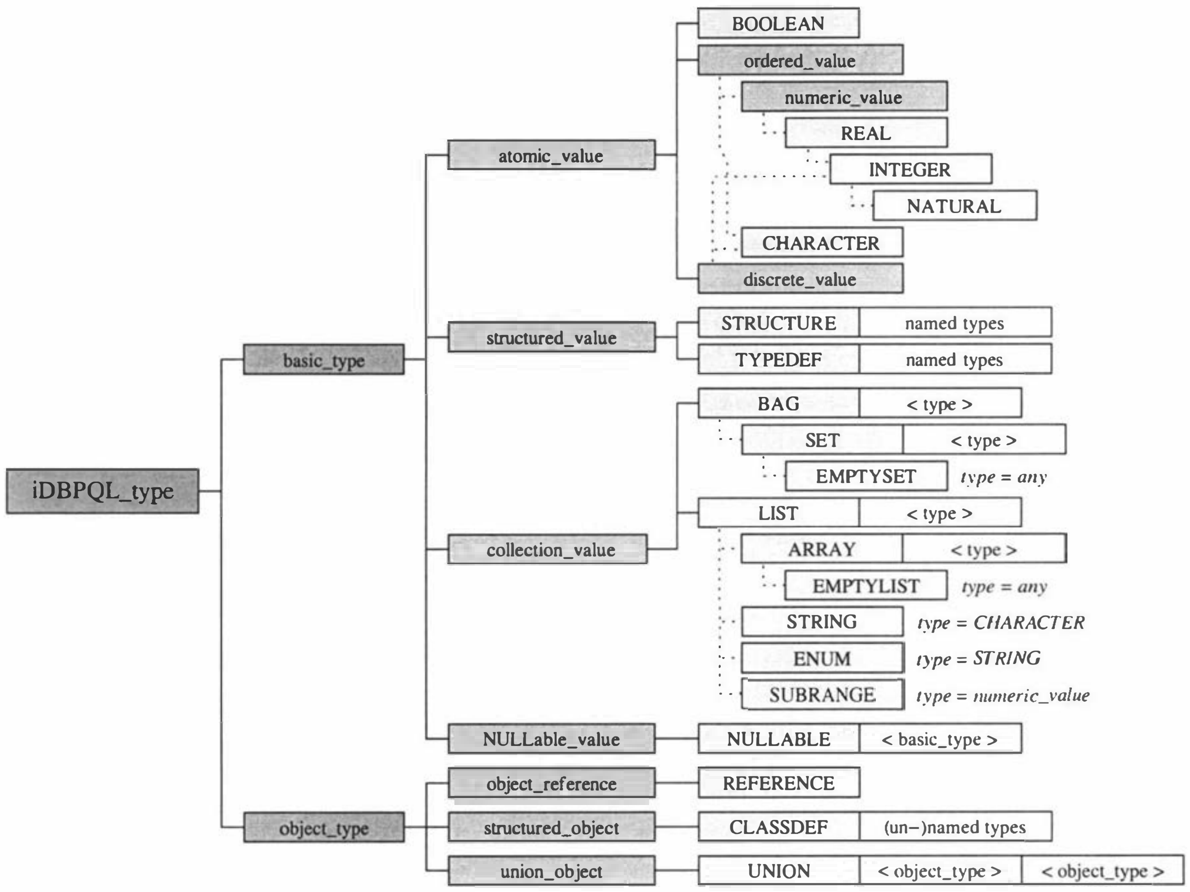


**Fig. 4.1.** The Type System of iDBPQL.

11. `SUBTYPE ( STRING, LIST < CHARACTER > );`
12. `SUBTYPE ( CHARACTER, STRING );`
13. `SUBTYPE ( ENUM, LIST < STRING > );`
14. `SUBTYPE ( SUBRANGE < _z >, LIST < _z > );` and
15. `SUBTYPE ( SUBRANGE < _z >, _z ).`

Where _x, _y and _z are type parameters with _z being restricted to numeric types, and $n$ is a Natural number. □

**NULLable Types.** All types, which we considered so far, have a common property, i.e. a default value. In databases, however, it is also desired to capture the fact that a type's value is not known. Thus, we require a mechanism to extend all value-types to include the NULL value. To do so, we define a NULLABLE type constructor, which has also recently found its way into the C# programming language [117].

*NULLable types* represent value-types whose variables can be assigned the NULL value. Otherwise, NULLable types can be used in the same way as the respective type argument. Thus, we can easily include NULLable types into iDBPQL's sub-type hierarchy. Note: A new super-type is defined! This implies that the conversion of an ordinary type's value to its corresponding `NULL`able type can be done implicitly. However, the same is not true vice versa. A cast expression, an explicit call of the default **`getValue`** type operation, or an explicit call of the default **`getValueOrDefault`** type operation must be included.

**Definition 4.5.** For `NULL`able and corresponding non-`NULL`able types, the binary `SUBTYPE` relation is the smallest relation which is reflexive, transitive and has the following properties:

1. `SUBTYPE ( NULLABLE < _x >, NULLABLE < _y > )` if `SUBTYPE ( _x, _y );`
2. `SUBTYPE ( _x, NULLABLE < _x >);` and
3. `SUBTYPE ( NULL, NULLABLE < _x >).`

Where _x and _y are type parameters. □

The default value of a `NULL`able type is, of course, `NULL`. This, however, requires to refine how pre-defined operations behave in the presence of `NULL` values. In iDBPQL, we adopt respective conventions from SQL [36]. For instance, this means that arithmetic operations return `NULL` if one of its operands is `NULL`. The logical $AND$ (i.e. `&&`) operator is extended to return `FALSE` if one operand evaluates to `FALSE` and the other to `NULL`, and it returns `NULL` if one operand evaluates to `NULL` and the other to either `TRUE` or `NULL`. Similarly, the logical $OR$ (i.e. `||`) operator is extended to return `TRUE` if one operand evaluates to `TRUE` and the other to `NULL`, and it returns `NULL` if one operand evaluates to `NULL` and the other to either `FALSE` or `NULL`.

**Value Initialisation and Default Values.** Default values for primitive types, structured types, built-in collection types, and `NULL`able types have already been outlined above. We can, thus, summarise as follows: Each value-type has an implicit default type initialiser (i.e. `INIT ( );`) that sets the default value of that type. However, it is not always desired to use the implicit default. For instance, think of a date value where day and month values are based on the primitive type `NATURAL` and year is based on `INTEGER`. We desire to initialise each date variable with a valid date value such as `( 1, 1, 0 )` (i.e. 1st of January 0). However, the implicit default for types `NATURAL` and `INTEGER` is 0 resulting in a default date value `( 0, 0, 0 )`. This can be avoided by specifying an explicit type initialiser.

In iDBPQL, types always have an implicit type initialiser but at most one explicit type initialiser. In the event that an explicit type initialiser exists (i.e. a type operation named `INIT` with no result type), no-one but the implementation of that explicit type initialiser can invoke the implicit type initialiser.

Value initialisation is then realised by invoking the type's (implicit or explicit) initialiser. Let us consider an example to demonstrate both approaches.

EXAMPLE 4.6. We will continue with Example 4.1. Consider the following code:

```
01    ...
02    bDate = ( 13, 4, 1976 );                        // metadata: STRUCT myDate bDate;
03    ...
```

First, a new variable (named `bDate`) is initialised. As we can see in Example 4.1, there is no explicit type initialiser specified. Thus, when reaching line `02`, the `bDate` value is `( 0, 0, 0 )`. This (implicit) default value has been obtained as follows: `bDate` is a value of a structured type. Structured types initialise values of each member to the default of their respective types. `bDate.day` and `bDay.month` are of type `NATURAL` and, thus, initialised as `0` (the default value of type `NATURAL`). `dDate.year` is of type `INTEGER` and, thus, initialised as `0` (the default value of type `INTEGER`).
Let us define another date type, one which has an explicit initialiser.

```
TYPEDEF myNewDate {
  STRUCTURE {
    NATURAL day;
    NATURAL month;
    INTEGER year;
  }

  BEHAVIOUR {
    age ( ) : NATURAL;
    INIT ( );
  }
}
```

The implementation of the explicit initialiser will override the default value assigned by the implicit initialiser. For instance, it could set the default date value to `( 1, 1, year.INIT ( ) )`. Thus, the defaults for values of `day` and `month` are set explicitly while the default for the `year` value is derived from its associated type (i.e. the implicit default mechanism is used). □

**Sub-typing and Type Conversion.** *Sub-typing* is structural (order, types and names are considered). Sub-type relations for primitive types, structured types (i.e. record types and, thus, more general type definitions), collection types (with type parameters) and `NULL`able types are specified in Definitions 4.1, 4.2, 4.3, 4.4, and 4.5 respectively. Type definitions that do not expose any structural properties have to be sub-typed explicitly using iDBPQL syntax as outlined in Syntax Snapshot 4.4. This approach has already been used for, e.g. built-in collection types.

### Syntax Snapshot 4.4 *(iDBPQL Sub-Type Declarations)*

```
SubTypeDecl = "SUBTYPE", '(', TypeId, ',', TypeId-List, ')';
```

□

Behaviour is not inherited. Nevertheless, a sub-type can utilise the behaviour specified for any of its super-types through type mapping.

EXAMPLE 4.7. Let us consider type definitions outlined in Examples 4.1 and 4.6. Recall that the following holds:

`SUBTYPE ( STRUCT myDate, myNewDate )` and
`SUBTYPE ( myNewDate, STRUCT myDate )`.

Thus, type operations specified for `myNewDate` can be applied to any (converted) `STRUCT myDate` value and vice versa. Such sub- and super-type relations are given implicitly. □

Sub-typing can be extended to include explicit sub-type specifications as follows:

**Definition 4.6.** For explicit sub-type specifications, the binary `SUBTYPE` relation is the smallest relation which is reflexive, transitive and has the following property:

$$\left.\begin{array}{l} \texttt{SUBTYPE ( } A\texttt{, } B_1 \texttt{ )} \\ \qquad\vdots \\ \texttt{SUBTYPE ( } A\texttt{, } B_n \texttt{ )} \end{array}\right\} \text{ if } \texttt{SUBTYPE ( } A\texttt{, [ } B_1\texttt{, ..., } B_n \texttt{ ] ).}$$

Where $A$, $B_1$, ..., $B_n$ are identifiers. □

Based on sub-typing, we can then define the set of valid type conversions.

**Definition 4.7.** A type conversion from type $S$ to type $T$ is considered to be *valid* iff `SUBTYPE (` $S$`,` $T$ `).` □

Valid conversions are either identity conversions (i.e. a conversion from a type to the same type) or widening conversions (i.e. a conversion from a type to its super-type).

**Variables.** The term variable has been used frequently without further explanation. A *variable* is declared by specifying its scope, the type of the variable and an identifier that names the variable. Syntax Snapshot 4.1 (on page 63) details the corresponding syntax portion.

A variable of a type, as considered in this section, contains a value of the specified type. This will be different for reference-types that are introduced later. In addition, the way variables are declared will also be extended.

As of now, we distinguish between the following three types of variables:

- *Type variables*, which are declared in a structure definition. In a user-type definition, they are considered structural members. Their scope may be `PUBLIC` (i.e. same as the type) or `PRIVATE` (i.e. only visible from 'within' the type).
- *Local variables*, which are declared in the body of a type operation (i.e. an evaluation plan).
- *Parameter variables*, which are declared in a type operation's or initialiser's parameter list. It can be considered as a local variable that is initialised with an argument value at the time the corresponding type operation or initialiser is invoked.

**Default Type Operations.** iDBPQL defines a number of default type operations that may be applied to any existing value-type. These default type operations include (where `_x` is an unconstrained type parameter):

- `equals ( _x val ) : BOOL;` ... determines whether or not the given value `val` is equal to the value of the variable on which the `equals` type operation[5] has been invoked;

[5] The `equals` type operation and the `==` type operator can be used synonymously for atomic values, String values and `NULL`able values.

- `getValue ( ) : _x;` ... returns the currently assigned value of the variable on which the `getValue` type operation has been invoked;
- `getValueOrDefault ( ) : _x;` ... returns the currently assigned value if it is not the `NULL` value. Otherwise, the underlying type's default value is returned;
- `hasValue ( ) : BOOL;` ... determines whether or not a variable contains a value. If so, `TRUE` is returned. Only in the case that a variable carries the `NULL` value, `FALSE` is returned;
- `init ( ) : VOID;` ... initialises the value of the variable on which the `init` type operation has been invoked;
- `reInit ( ) : VOID;` ... reinitialises the value of the variable on which the `reInit` type operation has been invoked; and
- `setValue ( _x val ) : VOID;` ... assigns the provided value `val` to the variable of type `_x` on which the `setValue` type operation has been invoked.

Type operations `hasValue` and `getValueOrDefault` are introduced to support `NULL`able types.

**Type Definitions for the University Application.** Let us conclude this subsection on types with a summary of all type definitions that form a part of the university application as detailed in Example 3.3.

EXAMPLE 4.8. We will restrict ourselves to types defined in the `SCHEMA` $\texttt{University}_{CC}$. All type definitions that appear in $\texttt{University}_{LS}$ and $\texttt{University}_{TO}$ schema fragments are replicas.

```
// synonym types
TYPEDEF ENUM ( "City Centre", "Lake Side", "The Oval" ) CampusT;
TYPEDEF INTEGER YearT;
TYPEDEF ENUM ( "first", "second", "double" ) SemesterCodeT;
TYPEDEF NATURAL MonthT;
TYPEDEF NATURAL DayT;
TYPEDEF ENUM ( "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ) WeekDayT;
TYPEDEF ENUM ( "A+", "A", "A-", "B+", "B", "B-", "C+", "C" ) PassGradesT;
TYPEDEF ENUM ( "D", "E" ) FailGradesT;
TYPEDEF ENUM ( "DNC", "Withdrawn" ) MiscGradesT;
TYPEDEF ENUM ( PassGradesT, FailGradesT, MiscGradesT ) GradesT;

// user-defined types
TYPEDEF NameT {
  STRUCTURE {
    NULLABLE < LIST < STRING > > titles;
    NULLABLE < STRING >          firstName;
    STRING                       lastName;
  }
}

TYPEDEF StreetT {
  STRUCTURE {
    NULLABLE < STRING > name;
    NULLABLE < STRING > numb;
  }
}

TYPEDEF AddressT {
  STRUCTURE {
    NULLABLE < StreetT > street;
    NULLABLE < STRING >  city;
    NULLABLE < NATURAL > zipCode;
  }
}

TYPEDEF PersonT {
  STRUCTURE {
    NATURAL               personId;
    NameT                 name;
    NULLABLE < AddressT > addr;
  }
}

TYPEDEF CourseT {
  STRUCTURE {
    STRING cNumb;
    STRING cName;
  }
}

TYPEDEF RoomT {
  STRUCTURE {
    CampusT campus;
    STRING  building;
    STRING  numb;
  }
}

TYPEDEF SemesterT {
  STRUCTURE {
    YearT     year;
    SemesterT sCode;
  }
}

TYPEDEF PhoneT {
  STRUCTURE { NULLABLE < STRING > phone; }
}

TYPEDEF DepartmentT {
  STRUCTURE {
    STRING                      dName;
    CampusT                     location;
    NULLABLE < SET < PhoneT > > phones;
  }
}
```

```
TYPEDEF StudentT {
  STRUCTURE { NATURAL studentId; }
}

TYPEDEF AcademicT {
  STRUCTURE { NULLABLE < STRING > specialisation; }
}

TYPEDEF DateT {
  STRUCTURE {
    NULLABLE < YearT >  year;
    NULLABLE < MonthT > month;
    NULLABLE < DayT >   day;
  }
}

TYPEDEF ProjectT {
  STRUCTURE {
    NATURAL            projectId;
    STRING             title;
    DateT              begin;
    NULLABLE < DateT > end;
  }
}

TYPEDEF TimeT {
  STRUCTURE {
    NULLABLE < NATURAL > hour;
    NULLABLE < NATURAL > minute
  }
}

TYPEDEF LectureTimeT {
  STRUCTURE {
    WeekDayT           weekDay;
    TimeT              start;
    NULLABLE < TimeT > end;
  }
}

TYPEDEF LectureT {
  STRUCTURE { NULLABLE < LectureTimeT > time; }
}

TYPEDEF EnrolmentT {
  STRUCTURE { NULLABLE < DateT > date; }
}

TYPEDEF RecordT {
  STRUCTURE { NULLABLE < GradesT > result; }
}
```

□

### 4.2.5 Classes and Objects

Types are used to represent simple concepts, such as structures with few members whose values are mutable. *Classes*, on the other hand, are meant to be used to group larger (i.e. more complex) structures or real-world objects. The structure of a class of objects is defined over existing types, unnamed types (i.e. types without a behaviour that are defined in the class structure itself) and existing classes (either as specialisation, i.e. IsA-relationships / inheritance, or as reference). Each object is uniquely identified by a hidden, globally unique object identifier (of a hidden, internal type `__OID`). We will now start to consider how classes are specified.

**Structure of a Class.** Classes can be seen as templates for creating objects. They specify the features that all objects of a class have in common. These *features* (also referred to as *class members*) include named constants, variables, methods, and simple constraints. Syntax Snapshot 4.5 outlines the relevant portion of the iDBPQL syntax.

**Syntax Snapshot 4.5** ***(iDBPQL Class Definitions)***

```
ClassDefinition         = ClassModifierDecl, "CLASSDEF", Id,
                          [ '<', ClassParameter-List, '>' ], [ "IsA", ClassId-List ],
                          [ "WITH", '{', { ( ClassParaConstrClause |
                            PrecedenceClause | RenamingExpr ), ';' }, '}' ],
                          '{',
                           [ StructuredType ],
                           [ "BEHAVIOUR", '{', { MethodSignature }, '}' ],
                           [ ConstraintDeclaration ],
                          '}';
// typically, the structured type is not assigned a unique identifier. In this case,
// we implicitly associate the identifier of the class also with the structured
// type. This is later used when typing query expressions.

ClassParaConstrClause = ClassParameter, "IsA", ClassId-List;
PrecedenceClause        = ( Id, "IS ACCEPTED FROM", Id ) |
                          ( Id, "COMBINES", Id, { "AND", Id } );
VariableDecl            = VarModifierDecl, Type, Id, [ "REVERSE", Id ];
                                                                  // extended version

MethodSignature         = MethodModifierDecl, Id, '(', [ Parameter-List ], ')', ':',
                          ResultType;

ConstraintDeclaration = "CONSTRAINT", [ Id ], '{' { DomainConstraint |
                            EntityConstraint }, '}';
DomainConstraint        = CheckConstraintDecl | NotNullConstraintDecl;
EntityConstraint        = UniqueConstraintDecl;
CheckConstraintDecl     = "CHECK", '(', Expression, ')';
NotNullConstraintDecl = "NOT NULL", '(', Id-List, ')';
UniqueConstraintDecl    = "UNIQUE", '(', Id-List, ')';

ClassModifierDecl       = [ ScopeModifierDecl ], [ StaticModifierDecl ],
                          [ ClassCatModifierDecl ], [ FinalModifierDecl ];
MethodModifierDecl      = [ ScopeModifierDecl ], [ StaticModifierDecl ],
                          [ AbstractModifierDecl | FinalModifierDecl ];
```

```
VarModifierDecl       = [ ScopeModifierDecl ], [ StaticModifierDecl ];

StaticModifierDecl    = "STATIC";
ClassCatModifierDecl  = AbstractModifierDecl | "COLLECTION" | "CONCRETE";
AbstractModifierDecl  = "ABSTRACT";
FinalModifierDecl     = "FINAL";
```

□

The `CLASSDEF` keyword is used to specify classes. While modern object-oriented programming languages strictly enforce the encapsulation property (and, thus, hide all structural properties of classes and their corresponding objects), classes in iDBPQL are not used primarily as information hiding mechanism. In a higher-level version of iDBPQL, concepts such as class interfaces and module interfaces may take over this part as briefly indicated in Section 3.2. While access modifiers such as `PRIVATE` and `PUBLIC` are supported, classes are meant to expose structural properties. In fact, the `PUBLIC` access modifier is the default for all class definitions and class members. Thus, all public properties of a class are accessible wherever the respective class is accessible. The `PRIVATE` scope modifier restricts member visibility. However, in contrast to conventional OOPLs, `PRIVATE` does not limit the view to the class. Instead, it only limits the view to the class where the member is declared and all its sub-classes (i.e. private and public properties are inherited). This is in line with the '*No Paranoia Rule*' as advocated in [132] and also similarly adopted in the object-oriented programming language Theta [82] that is designed for the object base Thor [83]. Having a reduced information hiding mechanism at the class-level but a strong information hiding mechanism at the module-level allows for more efficient implementations of closely related parts within a module. In particular, this strategy is advantageous when object-orientation is only taken to medium granularity[6] – as iDBPQL does. It should be noted that iDBPQL does not fully adhere to the '*No Paranoia Rule*'. Private properties of classes are hidden within the respective class/sub-class hierarchy. This addresses the criticism of the '*No Paranoia Rule*' that it is too open, especially for large modules.

Let us consider some initial examples:

EXAMPLE 4.9. We will define classes based on type definitions for the university application as outlined in Example 4.8. First, the `PersonC` class is defined. It is based on the previously defined `PersonT` value-type as follows:

```
CLASSDEF PersonC {
  STRUCTURE { PersonT; }
}
```

Alternatively, we could specify the same class without reusing the existing value-type `PersonT` (i.e. the `PersonC` class is defined over an unnamed structure type):

```
CLASSDEF PersonC {
  STRUCTURE {
    NATURAL               personId;
```

[6] Medium granularity of object-orientation means that there is a distinction between objects and values. Objects correspond to larger abstractions such as real-world objects, whereas values represent Integer values, or simply structured values.

```
     NameT                 name;
     NULLABLE < AddressT > addr;
  }
}
```

However, the underlying unnamed type is identical to the PersonT value-type. Thus, both class definitions are identical. □

All structural class members considered so far are associated with class instances, i.e. they define the state of objects. We refer to such structural class members as *instance variables*. In addition, classes may also contain *class variables*. There exists only one copy of a class variable that serves all objects of the class, not one per object as for instance variables. Per default, variables are instance variables. Class variables must be declared explicitly using the STATIC keyword.

While classes are regarded as templates for creating objects, *objects* themselves are considered *instances of classes*. They are created using the NEW keyword followed by the name of the class that the object will be based upon. Object creation results in a reference to the object being returned (not the actual object). In iDBPQL, objects are never accessed or returned directly. Object access is possible only through references or class-collections that are introduced further below. Each object has an associated value[7] that identifies the object. The type of a (reference-)value is *reference-to-$x$* where $x$ is the class of the object. Such types are referred to as *reference-types* or *object-types* (Figure 4.1 on page 71 details all iDBPQL types including reference-types). In contrast to value-types, all reference-types have a common, unique default value, which is the NULL reference (of type *reference-to-nothing*). The NULL value does not denote an object rather its absence.

**Variables and Reference-Types.** In Section 4.2.4 three types of variables have already been introduced. Above, instance variables and class variables have been added. Having introduced reference-types, the concept of a variable has to be extended to cover such types. Syntax Snapshot 4.5 details an extended means of declaring variables. Variables of a primitive type still contain a value of the specified type. In contrast, variables of a reference-type contain either the NULL value or a value that references an instance of the specified class or an instance of a sub-class of the specified class.

Variables of a reference-type may have a corresponding reverse variable. Per default, all variables are unidirectional (i.e. no such reverse variable exists). However, variables of a reference-type may be bidirectional. Thus, a corresponding reverse variable exists. A reverse variable is of reference-type *reference-to-$x$* where $x$ is the class in which the bidirectional variable is defined in or a collection over *reference-to-$x$*. A bidirectional variable must be declared in both classes that refer to one another. Each of the two declarations must contain the REVERSE keyword followed by the name of the corresponding reverse variable. While unidirectional variables of a reference-type can only represent one-to-one and one-to-many associations between classes, bidirectional variables can represent one-to-one, one-to-many and many-to-many associations.

[7] Values associated with objects are also referred to as *reference values*.

Before considering an example consisting of such associations, it has to be noted that definitions of the structured type, type parameters and the built-in type constructors (all introduced in Section 4.2.4) are extended to support variables of both value-types and reference-types.

EXAMPLE 4.10. Again, let us consider the university application. A class representing departments may be defined as follows:

```
CLASSDEF DepartmentC {
  STRUCTURE {
    DepartmentT;                                          // existing type definition
    PersonC             director;                         // unidirectional reference
    SET < StudentC >    majorStudents REVERSE major;      // bidirectional reference
    SET < StudentC >    minorStudents REVERSE minor;      // bidirectional reference
    SET < AcademicC_CC > staff REVERSE staffMemberOf;     // bidirectional reference
  }
}
```

This class definition contains members resembling both unidirectional and bidirectional references. By default, such references are initialised to refer to the NULL value. □

**Methods.** Besides structural properties, classes contain methods. A *method* corresponds to a block of iDBPQL statements, i.e. an evaluation plan. As outlined in Syntax Snapshot 4.5, methods are declared by specifying method modifiers, the method name, any method parameters (surrounded by parentheses), a colon, and the result type. Method names need not to be unique, however, the method name together with its method parameters (in number or types) must be.

iDBPQL classes may contain four types of methods. These are:

- *Instance methods*, which are associated with class instances, i.e. objects. Such methods rely on the state of the specific object instance, i.e. the behaviour that the method invokes relies upon its instance's state. Per default, methods are instance methods.
- *Class methods* (also referred to as *static methods*), which are associated with classes and not their instances. Such methods do not access instance variables of any object of the class they are defined in. They only access static members. Most commonly they take all their data only from method parameters and perform computations from those parameters, with no reference to instance variables.
  Class methods must be declared explicitly using the STATIC keyword.
- *Object constructors*, which are special instance methods (with no result type) that are called automatically upon the creation of an instance of a class. Once invoked, object constructors initialise instance variables (in fact, instance variables inherited from base-classes as well as instance variables defined in this class). Similar to other PLs (e.g. Java, C++ and C#), object constructors are distinguished from instance methods by having the same name as the class they are associated with.
  The object constructor with an empty argument list must always exist. If not specified explicitly, it is generated automatically. A system-generated default constructor is specified as follows:

```
EVALPLAN myNewClassC ( VOID ) {
  SUPER ( );                  // respective invocation semantics are outlined
                                                      // later in Example 4.21
  RETURN ( VOID );
}
```

**(Multiple) Inheritance.** *Inheritance* is a mechanism by which a class (which is then called *sub-class*) automatically inherits all the features (except for object constructors) of one or more other classes (which are then called *super-class(es)*). Inheritance may be interpreted as class extension mechanism (e.g. derive a class modelling females from a class modelling males) or as class specialisation mechanism (e.g. the female class and the male class are both derived from a class modelling persons). The specialisation approach is often considered more natural, more intuitive and easier to comprehend, e.g. refer to [136]. It can be viewed as a vehicle for conceptual modelling, which is the interpretation of inheritance as desired by ODBSs. Thus, classes represent concepts, which may then be specialised in sub-classes by adding new instance variables, adding new methods and / or overriding existing methods to support the new instance variables.

Inheritance in iDBPQL is name-based and must be specified when defining a class using the `IsA` keyword as indicated in Syntax Snapshot 4.5. The `IsA` keyword is followed by one (i.e. *single inheritance*) or more (i.e. *multiple inheritance*) identifiers of existing classes (which are also known as the *base classes* of the newly defined class[8]).

A sub-class can *override* an inherited method, i.e. it provides an alternative implementation for that method (both of which have the identical signature). A class that overrides a method can invoke the overridden method using the ***SUPER*** keyword. Corresponding details are discussed further below in Section 4.3.5.

As discussed in Section 1.1.3 and in contrast to most modern OOPLs, DBPLs require support for *multiple inheritance*. That is, a sub-class may inherit features of more than one super-class. Multiple inheritance, however, causes various semantic ambiguities that have to be addressed. Among others, [116] discusses such ambiguities. Multiple inheritance may result in (two copies / replicas of) a base-class being inherited over two different paths in the inheritance graph. Thus, naming clashes arise. This type of multiple inheritance is also known as *replicated inheritance* and can be dealt with by forcing the programmer to rename at least one set of features of the doubly derived base class. iDBPQL supports renaming of class features in a similar manner as it has already been introduced for record members in Section 4.2.4. More challenging ambiguities arise when the same base class is inherited over two different paths in the inheritance graph. This *shared inheritance* may result in a method of the base class being overridden in both inheritance paths. Thus, when calling the method on an object of the new class, it cannot be determined which of the two overridden methods should be invoked. While some PLs such as Eiffel [89] provide elaborate mechanisms for controlling such ambiguities, other PLs such as Loops [20] and CLOS [56] compute precedence orders for super-classes. Originally, iDBPQL followed the latter approach and interpreted the list of super-classes as a priority list according to which the respective overridden method

[8] Inheritance results in all of the class members defined for base-classes to become part of the new class as well. Because a base-class may itself inherit from another class, which inherited from another class, and so on, a class may end up with many base-classes. The terms base-class and super-class can be used synonymously.

is selected. However, this approach is very restrictive since programmers have to prefer one super-class over another. In line with [119], a more refined approach has been adopted. Precedence orders may be defined for particular class members. Let us look at the various ambiguities and the ways iDBPQL offers to address them (refer to the class definition's `WITH` clause as outlined in Syntax Snapshot 4.5 on page 78).

EXAMPLE 4.11. First, we consider replicated inheritance. Example 4.9 outlined a simple definition of the `PersonC` class. We will use this class to define the `StudentC` class representing students and a class `CoupleC` capturing the relationship between a student and another person. These two new classes may be defined as follows:

```
CLASSDEF PersonC {
  STRUCTURE { PersonT; }

  BEHAVIOUR {
    getAddress ( ) : AddressT;                          // returns home address
    PersonC ( NameT name );                              // object constructor
    PersonC ( NameT name, AddressT addr );               // object constructor
  }
};

CLASSDEF StudentC IsA PersonC {
  STRUCTURE {
    StudentT;
    AddressT campusAddr;
  }

  BEHAVIOUR {
    getAddress ( ) : AddressT;                        // returns campus address
    StudentC ( NameT name, AddressT homeAddr, StudentT stud,
               AddressT uniAddr );                       // object constructor
  }
};

CLASSDEF CoupleC IsA StudentC, PersonC {

  BEHAVIOUR {
    CoupleC ( ... );                                     // object constructor
  }
};
```

The definition of class `CoupleC` is not free of ambiguities. This is a result of the class `PersonC` being inherited over two different paths, i.e. as base-class of `StudentC` and explicitly as direct super-class. As it is not intended to refer to the same `PersonC` object, all definitions that stem from the `PersonC` class are replicated causing naming clashes. The programmer can either rename the properties inherited via the direct super-class `PersonC`, rename common properties inherited through the `StudentC` class or rename both sets of properties. The former may be done as follows resulting in a definition of class `CoupleC` that is free of ambiguities:

```
CLASSDEF CoupleC IsA StudentC, PersonC WITH {
  PersonC.personId   AS partnerId;
032    PersonC.name       AS partnerName;
033    PersonC.addr       AS partnerAddr;
034    PersonC.getAddress AS getPartnerAddress;
035  } {
036
037    BEHAVIOUR {
038      CoupleC ( ... );                                    // object constructor
039    }
040  };
```

Secondly, there are the ambiguities associated with shared inheritance. Let us illustrate the problem by adding two more classes, a class representing academic staff members and a class representing persons who are both students and academic staff members. Not only will class PersonC be inherited via two paths, but also will the getAddress method be overridden by both the StudentC class and the AcademicC$_{CC}$ class:

```
050  CLASSDEF AcademicC_CC IsA PersonC {
051    STRUCTURE {
052      AddressT workAddr;
053    }
054
055    BEHAVIOUR {
056      getAddress ( ) : AddressT;                        // returns work address
057      AcademicC_CC ( NameT name, AddressT homeAddr, AddressT workAddr );
058                                                          // object constructor
059    }
060  };
061
062  CLASSDEF StudentAcademicC_CC IsA StudentC, AcademicC_CC {
063    BEHAVIOUR {
064      StudentAcademicC_CC ( ... );                        // object constructor
065    }
066  }
```

Obviously, class StudentAcademicC$_{CC}$ inherits the getAddress method from both its super-classes (but also from its base class PersonC). Assume, we have a method returning the address of a StudentAcademicC$_{CC}$ *staca*:

```
070    ...
071    return ( staca.getAddress ( ) );    // metadata reference: StudentAcademicC_CC
072  }
```

Line 071 causes a problem. The question is which getAddress method has to be invoked at run-time? Is it the one inherited from the base class PersonC, from the direct super-class StudentC or from the direct super-class AcademicC$_{CC}$? Similar to replicated inheritance, iDBPQL forces the programmer to specify his/her precedence(s):

```
080  CLASSDEF StudentAcademicC_CC IsA StudentC, AcademicC_CC WITH {
081    getAddress IS ACCEPTED FROM AcademicC_CC;
082  } {
083    BEHAVIOUR {
084      StudentAcademicC_CC ( ... );                        // object constructor
085    }
086  }
```

Now, the method call described in line 071 will invoke the getAddress method as specified for the AcademicC$_{CC}$ class. The WITH clause permits both renaming and prioritising of inherited class members. However, the same class member cannot be renamed and prioritised in the same WITH clause.
Prioritising one method over another is not always applicable or desired. For instance, we may think of two equals methods that cover all static class members in their respective classes but not in the sub-class. This is likely the case if replicated inheritance is mixed with shared inheritance. Choosing one equals method over the other will not result in a logically correct equals method for the sub-class. To gather such cases, iDBPQL allows to combine inherited class members (something that is not found in traditional OOPLs) or override them. The former falls into the category of *method combination* and is only possible when considering method prioritisation instead of class prioritisation. Let us outline a simple example defining one-, two- and three-dimensional points:

```
090  CLASSDEF PointC {
091    STRUCTURE {
092      INT x;
093    }
094    BEHAVIOUR {
095      equals ( INT xVal ) : BOOL;
096    }
097  }
098
099  CLASSDEF Point2DC IsA PointC AS XPointC, PointC AS YPointC WITH {
100    y AS YPointC.x;                        // renaming, i.e. replicated inheritance
101    equals ( INT xVal, INT yVal ) COMBINES XPointC AND YPointC;
102  } { }
103
104  CLASSDEF Point3DC IsA Point2DC, PointC WITH {
105    z AS PointC.x;                         // renaming, i.e. replicated inheritance
106    equals ( INT xVal, INT yVal, INT zVal ) COMBINES Point2DC AND PointC;
107  } { }
```

Two-dimensional (2D) points are defined based on two incarnations of the PointC class representing one-dimensional (1D) points. Line 91 defines the 2D-equals method as a combination of the 1D-equals method: First, the 1D-equals method is applied to the xVal value and then to the yVal value; the result of the 2D-equals method is a logical *AND* of the results of the two 1D-equals invocations. Similarly, a three-dimensional (3D) point is defined through the combination of a 2D point and a 1D point. The corresponding equals method combines the equals method from the Point2DC class (arguments 1 and 2) and the equals method from the PointC class (argument 3).
The COMBINES mechanism can be applied to methods with the same name and return type. However, the latter is restricted to BOOL and VOID. Arguments of all methods that are combined will be concatenated (i.e. preserving the order). Thus, the number of arguments of the resulting combined method is equal to the sum of all arguments of all methods listed after the COMBINED keyword.
This form of method combination can also be achieved by defining a new equals method that explicitly calls the corresponding super-class methods, collects their results

and determines the return parameter of the method. This overriding approach is the alternative approach to prioritising and combining. The rationale behind introducing method combination is the fact that it can be utilised during the evaluation. It presents an opportunity to utilise concurrent evaluation as it will be outlined later in Section 5.3.9. □

To summarise, multiple inheritance offers programmers three different ways of dealing with possible ambiguities. Renaming can be applied to include features of replicated classes, prioritisation supports shared inheritance, while, in the absence of renaming and prioritisation, identical features inherited via two paths can be treated as if they would have been inherited only once. In the latter case, it is required that inherited features have not been overridden along their respective inheritance paths. In the absence of adequate renaming, prioritisation specifications and the identity property, iDBPQL code suspectable to any multiple inheritance ambiguities must be rejected. Despite the fact that multiple inheritance adds to the complexity of coding, it also offers a powerful tool to design and implement applications. The following example will outline how the different ways of dealing with possible ambiguities can be combined:

EXAMPLE 4.12. Let us temporarily extend the `University` schema. We still have students and academic staff members who are also persons as well as a class modelling individuals that are both at the same time students and academic staff members. This leaves us with the situation outlined in Example 4.11. Now, we add two additional classes. Firstly, there is the `UniMemberC` class that associates a unique ID and an email account with each student and each staff member. Secondly, the university offers a complimentary insurance cover for its students and staff members. Class `UniInsuranceClientC` models this service. We can define a corresponding schema fragment as follows:

```
01  CLASSDEF PersonC {
02    STRUCTURE { PersonT; }
03
04    BEHAVIOUR {
05      getAddress ( ) : AddressT;                          // returns home address
06      PersonC ( NameT name );                              // object constructor
07      PersonC ( NameT name, AddressT addr );               // object constructor
08    }
09  };
10
11  CLASSDEF UniInsuranceClientC {
12    STRUCTURE {
13      NAT policyId;
14    }
15  }
16
17  CLASSDEF UniMemberC {
18    STRUCTURE {
19      NAT id;
20      STRING email
21    }
22  }
23
```

```
CLASSDEF StudentC IsA PersonC, UniMemberC, UniInsuranceClientC {
  STRUCTURE {
    StudentT;
    AddressT campusAddr;
  }
}

CLASSDEF AcademicC_CC IsA PersonC, UniMemberC, UniInsuranceClientC {
  STRUCTURE {
    AddressT workAddr;
  }

  BEHAVIOUR {
    getAddress ( ) : AddressT;                          // returns work address
    AcademicC_CC ( NameT name, AddressT homeAddr, AddressT workAddr );
                                                        // object constructor
  }
};

CLASSDEF StudentAcademicC_CC IsA StudentC, AcademicC_CC WITH {
  AcademicC_CC.id AS staffId;
  AcademicC_CC.email AS staffEmail;
  getAddress IS ACCEPTED FROM AcademicC_CC;
} {
  BEHAVIOUR {
    StudentAcademicC_CC ( ... );                        // object constructor
  }
}
```

Considering the StudentAcademicC$_{CC}$ class definition, various ambiguities arise that are being dealt with. Details are as follows:

- Structural features of the PersonC base class are inherited via both paths, in fact, they are identical. Thus, only one set of these features is retained, i.e. the features inherited via the StudentC path.
  The same is true for all features inherited from the UniInsuranceClientC class.
- The getAddress method from class PersonC, however, is refined along both inheritance paths. Thus, a renaming expression or a prioritisation clause is expected. The latter is the case. Thus, invoking the getAddress method on an instance of class StudentAcademicC$_{CC}$ will result in executing the method inherited via the AcademicC$_{CC}$ path.
- Two different sets of class members of the UniMemberC class are inherited. It is important that both sets of members form a part of the StudentAcademicC$_{CC}$ class. Hence, only renaming is applicable as specified above.

As a result, each StudentAcademicC$_{CC}$ object is a person, a student, an academic staff member and has one complimentary insurance policy. The getAddress method returns the work address by default. In addition, the student identifier, student email address, staff identifier and staff email address are inherited. □

As mentioned earlier, object constructors are not inherited. Instead, each class must define its own object constructors. Once invoked, object constructors initialise instance variables inherited from base-classes as well as instance variables defined in the class itself. Thus, at first, object constructors from all super-classes must be invoked (explicitly). For this, the `SUPER` mechanisms is used. Example 4.21 will later demonstrate two alternatives of such explicit object constructor invocations.

The `IsA` keyword in class definitions defines a direct $IsA$ relation between classes. For a class definition `CLASSDEF` $A$ `IsA` $B_1$`,` ...`,` $B_n$ we say $B_1$, ..., $B_n$ are direct super-classes of $A$ or $A$ is a direct sub-class of $B_1$, ..., $B_n$ and define $A\ IsA\ B_1$, ..., $B_n$.

Based on this direct $IsA$ relation, sub-typing can be extended to include reference-types as follows:

**Definition 4.8.** For reference-types, the binary `SUBTYPE` relation is the smallest relation which is reflexive, transitive and has the following properties:

1. `SUBTYPE (` $A$`,` $B$ `)` if $A\ IsA\ B$;
2. `SUBTYPE ( NULL,` $A$ `)`; and
3. `SUBTYPE (` $A$`, OBJECT )`.

Where $A$ and $B$ are reference-types, `OBJECT` is the default super-class of any class (but itself) that has no `IsA` clause, and `NULL` is the default sub-class of any other class. □

Thus, inheritance by means of specialisation can be seen as being parallel to sub-typing. Hence, `IsA`-specialisations imply sub-typing. However, the same is not true vice versa.

The $IsA$ relation is also used to define the inheritance relation. The *inheritance relation* $\leq_{IsA}$ is defined as the transitive closure over $IsA$ relations (i.e. reachable by a finite number of $IsA$ steps) which is also reflexive. The inheritance relation forms a directed acyclic graph (DAG). This graph has classes as nodes and $IsA$ relationships as links. Using the inheritance relation, we can say that $A$ is a sub-class of $B$ or $B$ is a super-class of $A$ iff $A \leq_{IsA} B$ (i.e. there is a directed path from $A$ to $B$ in the corresponding DAG).

**Variables, Types, Objects, and Classes.** In a nutshell, variables have types and objects have classes. Every object belongs to the class on which it was created (i.e. *its class*). An object is considered an instance of its class and of all super-classes of its class.

Types restrict possible values of variables and expressions. While a variable's type is always declared (i.e. known at compile-time), the type of an expression is derived only at run-time. In the event that an expression's run-time value has an associated reference-type, the value is a reference to an object. This object has a class, which must be compatible with the corresponding compile-time type (which is then also a reference-type). Compatible, here, means that there must be a valid type conversion between the two reference types (refer to Definition 4.7).

**The `FINAL` and the `ABSTRACT` Modifiers.** In addition to scope modifiers and the `STATIC` modifier, two more modifiers are supported by iDBPQL. These are the `FINAL` and the `ABSTRACT` modifiers.

The *ABSTRACT modifier* can precede the CLASSDEF keyword in a class definition indicating that this class cannot be instantiated. Such *abstract classes* may only have other abstract classes as super-classes. However, its sub-classes may or may not be abstract. Abstract classes are commonly used to specify features that their sub-classes must implement. Classes that are not abstract are known as *concrete classes*.

In addition, the ABSTRACT modifier may appear in a method signature preceding the method name. Such *abstract methods* have no associated evaluation plan. Concrete sub-classes must provide their own implementation for such abstract methods. Every class definition that contains at least one abstract method must be declared ABSTRACT itself.

Finally, the *FINAL modifier* may appear in-front of a CLASSDEF keyword in a class definition, in a variable declaration of a class definition or in a method signature. Any concrete class may be declared to be FINAL (i.e by specifying this modifier in-front of the CLASSDEF keyword) implying that it cannot be sub-classed. When the FINAL keyword appears in a method signature (i.e. preceding the method name), it implies that the corresponding method cannot be overridden in any sub-class. Otherwise, if the FINAL modifier appears in a variable declaration, the corresponding variable becomes a named constant.

**The UNION-Type.** *UNION-types* are introduced as a means of specifying super-types explicitly based on existing class definitions. Union types for object-oriented programming have been introduced, among others, in [53]. We adopt this proposal in our iDBPQL language (with minor modifications).

The UNION-type is applicable to classes and values of reference-types only. Among other things, it supports the unification of objects of (horizontally) fragmented classes so that the resulting union corresponds to a global class as it has been defined in a higher-level, global database schemata (as later demonstrated in Example 4.26).

**Syntax Snapshot 4.6** ***(The iDBPQL UNION-Type)***

```
UnionType = ScopeModifierDecl, "UNIONDEF", [ Id ] , '<', RefType, ',', RefType, '>';
```
□

Syntax Snapshot 4.6 outlines the syntax of the UNION-type. It can be considered as a set-union of all objects of two classes (identified by their class name or as a reference value) and it behaves as their least common super-type.

UNION-types can be constructed from any two classes, say $A$ and $B$. The UNIONDEF < $A$, $B$ > denotes the set-union of values of $A$ and $B$. Thus, it includes only values of $A$ or $B$ and nothing else. In contrast to classes, UNION-types have no explicit implementation (i.e. evaluation plan) associated. However, it allows to utilise those methods that both $A$ and $B$ have in common.

UNION-types only have two associated operations: Case analysis and member access. These are defined as follows:

- *Case analysis* is a conditional construct that branches according to the run-time type of the value. For instance, the following code section invokes std.getAddress ( ); since p holds a value of type StudentC.

```
...
p = NEW StudentC ( ... );   // metadata: UNIONDEF < StudentC, AcademicC_CC >;

SWITCH ( p ) {
  CASE StudentC AS std  : { std.getAddress ( ); }
  CASE AcademicC_CC AS aca : { aca.getAddress ( ); }
}
...
```

– *Member access* allows direct access to common features. However, access is restricted to variables with the same name and compatible type, and methods that have identical signatures.

Let us consider an example unifying collections of student and academic staff member objects.

EXAMPLE 4.13. Consider `StudentC` and `AcademicC`$_{CC}$ class definitions from Example 4.11. Assume, we have defined the union over all `StudentC` and all `AcademicC`$_{CC}$ objects. The resulting union type allows for the following member accesses (refer to lines 02 to 04 and 06):

```
01    ...          // local variable p is of type UNIONDEF < StudentC, AcademicC_CC >
02    p.personId
03    p.name
04    p.addr
05    ...
06    p.getAddress ( );
07    ...
```

The only shared static features of both classes, `StudentC` and `AcademicC`$_{CC}$, are those inherited from their common direct super-class `PersonC`. The only shared behavioural feature is the `getAddress` method that is defined not only in the common direct super-class `PersonC` but also overridden in both classes. □

Sub-typing can be extended to include `UNION`-types as follows:

**Definition 4.9.** For `UNION`-types, the binary `SUBTYPE` relation is the smallest relation which is reflexive, transitive and has the following properties:

1. `SUBTYPE` ( $B$, `UNIONDEF` < $B$, $C$ > );
2. `SUBTYPE` ( $C$, `UNIONDEF` < $B$, $C$ > );
3. `SUBTYPE` ( `UNIONDEF` < $B$, $C$ >, $A$ ) if `SUBTYPE` ( $B$, $A$ ) and `SUBTYPE` ( $C$, $A$ ); and
4. `SUBTYPE` ( `UNIONDEF` < $D$, $E$ >, `UNIONDEF` < $B$, $C$ > ) if `SUBTYPE` ( $D$, $B$ ) and `SUBTYPE` ( $E$, $C$ ).

Where $A$, $B$, $C$ and $D$ are reference-types. □

Figure 4.2 provides an example of corresponding sub-typing and inheritance hierarchies. $A$, ..., $G$ are classes where $B$, $C$ and $D$ are sub-classes/-types of $A$; $E$ and $F$ are sub-classes/-types of $B$; and $G$ is a sub-class/-types of $C$. In addition, `UNIONDEF`

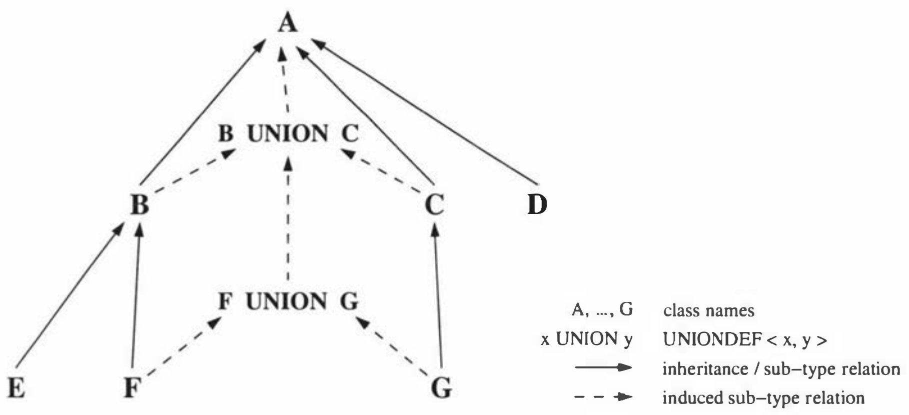

A
B UNION C
B
C
D
F UNION G
E
F
G
A, ..., G class names
x UNION y UNIONDEF < x, y >
inheritance / sub–type relation
induced sub–type relation

**Fig. 4.2.** Union Types, Sub-typing and Inheritance ([53, Figure 1]).

`<` $B$`,` $C$ `>` is a sub-type of $A$ and a super-type of $B$ and $C$; and `UNIONDEF <` $F$`,` $G$ `>` is a sub-type of `UNIONDEF <` $B$`,` $C$ `>` and a super-type of $F$ and $G$. Assume, we have another `UNION`-type defined as follows: `UNIONDEF newUnion <` $A$`,` $B$ `>`. This is a special case since $B$ is a sub-class/-type of $A$. As a result, the `UNION`-type `newUnion` denotes the same set of instances as $A$.

**Special Pre-Defined Classes.** iDBPQL has a number of pre-defined, special-purpose classes. The most important of these are class `Object`, class `Class`, class `CollectionClass`, and class `Transaction`.

First, there is the previously mentioned class `Object`, which is a super-class of every class but itself. Having such a default super-class allows programmers to write generic code that deals with objects of any type. The class `Object` is defined as follows:

```
CONCRETE CLASSDEF Object {
  BEHAVIOUR {
    clone ( ) : Object;                // creates and returns a copy of this object
    equals ( Object obj ) : BOOLEAN;           // determines whether object obj is
                                                              // 'equal to' this one
    getClass ( ) : Class;              // returns the run-time class of this object
    isInstance ( Class class ) : BOOLEAN;   // determines whether this object is
                                                       // an instance of class class
    Object ( );                                                // object constructor
    Object ( STRING strObj );      // object constructor creating the object from
                                                       // its String representation
    toString ( ) : STRING;           // returns the object's String representation
  }
}
```

In addition, there are some implicitly defined operators on objects. These are: Member access through the '`.`' operator and casting. Member access may either correspond to class / instance variable access or method invocation.

Secondly, there is the class `Class` that has instances representing iDBPQL classes within the system. It can be regarded as an (internal) means of keeping track of all class properties. Since `Class` objects are constructed automatically, the class `Class` has only an internal, hidden constructor. The non-hidden portion of class `Class` is defined as follows:

```
CONCRETE CLASSDEF Class {
  BEHAVIOUR {
    getSuperClasses ( ) : LIST < Class >;            // returns a list of direct
                                                                // super-classes
    hasSubClass ( Class class ) : BOOLEAN;      // determines whether class class
                                                  // is a direct sub-class of this class
    hasSuperClass ( Class class ) : BOOLEAN;   // determines whether class class
                                                // is a direct super-class of this class
  }
}
```

Thirdly, there is the final class `CollectionClass` that has instances representing iDBPQL classes, which are also collection-classes. This class extends the class `Class` and adds functionality to maintain collection-classes explicitly. The non-hidden portion of class `CollectionClass` is defined as follows:

```
CONCRETE FINAL CLASSDEF CollectionClass IsA Class {
  BEHAVIOUR {
    contains ( Object obj ) : BOOLEAN;        // determines whether object obj is
                                             // an object of this collection-class
    equals ( Object obj ) : BOOLEAN;          // determines whether object obj is
                                               // 'equal to' this collection-class
    isEmpty ( ) : BOOLEAN;               // determines whether the collection-class
                                                            // has any instances
    remove ( Object obj ) : BOOLEAN;             // removes object obj from this
                                                             // collection-class
    size ( ) : NATURAL;                      // returns the number of instances of
                                                        // this collection-class
  }
}
```

Class `Transaction` is used to model transactions. It only has an internal object constructor, which is automatically invoked whenever a transaction block (refer to Section 4.3.3) is encountered.

```
CONCRETE CLASSDEF Transaction {
  STRUCTURE {
    PRIVATE TransId tid;                // globally unique transaction identifier
  }
  BEHAVIOUR {
    abort ( ) : VOID;                      // triggers the abort of the transaction
    commit ( ) : VOID;                                   // commits the transaction
    getTransId ( ) : TransId;               // returns the transaction's identifier
    isActive ( ) : BOOLEAN;        // determines whether the transaction is active
    isAborted ( ) : BOOLEAN;      // determines whether the transaction is aborted
    rollback ( ) : VOID;                              // rolls back the transaction
  }
}
```

**Class-Collections.** In Section 1.1.3, we have already indicated that DBSs desire to have a means of access to all objects of a particular class. To meet this requirement, iDBPQL supports class-collections. Per default, each concrete class has an associated, system-maintained collection through which access to all objects of this class and its sub-classes is possible.

EXAMPLE 4.14. Let us consider the `PersonC`, `StudentC`, `AcademicC`$_{CC}$, and `StudentAcademicC`$_{CC}$ class definitions from Example 4.11. All four classes are collection-classes by definition. Thus, accessing class `StudentAcademicC`$_{CC}$ by name will return a collection of all instances of that class. In turn, accessing class `PersonC` will not only return a collection of all instances of class `PersonC` but also all instances of its sub-classes, i.e. `StudentC`, `AcademicC`$_{CC}$ and `StudentAcademicC`$_{CC}$. The return type of this collection would be class `PersonC`. □

However, this means of access is not always desired. For instance, assume we have defined a class `EnrolmentIdC` of objects uniquely identifying enrolments of a particular type. This `EnrolmentIdC` class may then later be used by classes representing actual enrolment objects. While it is likely that access to all enrolments of a particular type is desired (i.e. such classes would be collection-classes), a class-as-collection type access to the `EnrolmentIdC` class is not necessary, in fact, in most situations even undesired. To gather such scenarios, iDBPQL introduces concrete-only classes, which are concrete classes in the traditional PL-sense. That is, concrete-only classes have no associated, system-maintained collections. Whenever it will be necessary to distinguish between the two types of concrete classes, we will refer to them as *collection-classes* and as *collection-less classes* respectively. While collection-classes are created by default, collection-less classes are defined using the `CONCRETE` keyword as indicated in Syntax Snapshot 4.5 (on page 78).

Since iDBPQL does not use classes as information hiding mechanism, the decision of whether or not to use a collection-class is mainly a performance issue. Maintaining collections is costly with respect to processing time. Hence, the default mechanism of associating collection-classes should be overridden whenever the additional functionality of direct, name-based access to all class and sub-class instances is not required. As a rule-of-thumb, classes that are designed to be queried should be defined as collection-classes relieving the programmer from maintaining such collections explicitly. Classes that are designed to model more general concepts (e.g. a class representing a hash-table), are used across different application domains or are meant only to support a particular collection-class should be defined as collection-less classes. Examples will appear throughout the remainder of this thesis.

Earlier in this section, on page 88, we have already outlined how abstract and concrete classes appear in the inheritance graph. Having two types of concrete classes requires us to revisit and refine how different types of classes my appear in the inheritance graph. Abstract classes may only have other abstract classes as super-classes, but can be sub-classed by all types of classes. Collection-less classes may have abstract classes or collection-less classes as super-classes and can be sub-classed by both collection-less classes and collection-classes. Collection-classes may have all types of classes as super-classes, but must only be sub-classed by collection-classes. Thus, considering the inheritance graph, we can summarise that abstract classes always appear at the top of the hierarchy, collection-less classes at the centre and bottom of the hierarchy, and collection-classes generally at the bottom of the hierarchy.

The rationale behind those restrictions should be obvious. Assume that a collection-less class, say $Y$, would be allowed to sub-class a collection-class, say $X$. When accessing class $X$ through the class-as-collection mechanism, we require access to all instances of the class itself and also to those of its sub-classes (i.e. $Y$). While $X$ has an associated collection, $Y$ has none. This would result in an inability to fulfil the request.

**Constraints.** While constraints are not common in programming languages, they are an essential component of database languages. iDBPQL supports two types of explicit constraints. These are:

- *Domain constraints*, which only require to validate instance variables of the particular object. iDBPQL supports the following three domain constraints:
  - The *`NOT NULL` constraint*. Variables may be declared to take on a value that is not the `NULL` value. Only the support of `NULL`able types makes it possible that variables of a value-type may be assigned that value. The `NOT NULL` constraint applies to reference values more natural. However, variables of a reference-type may be of reverse nature. Thus, it is possible that both ends of a reference expect a non-`NULL` value. This introduces difficulties during object creation. To circumvent such problems, iDBPQL supports atomic blocks, which delay the point in time where constraints are verified and newly created objects become visible. Corresponding details are discussed in Section 4.3.3.
  - The *`CHECK` constraint*, which can be used to ensure that a variable's value stays within a given range, is an element of a pre-defined collection of values (i.e. the `IN` clause), contains a given pattern (i.e. the `LIKE` clause, or preserves a particular relation to a constant value or another variable (of the same or a references class).
- *Entity constraints*, which can only be verified when considering all objects that are instances of the particular class.
  iDBPQL supports the *`UNIQUE` constraint*, which is a class-level constraint. One or more structural class members may be declared to be unique.

Constraints are inherited. In contrast to SQL, `UNIQUE` implies `NOT NULL`. Thus, a sub-class may be missing `NOT NULL` constraints for instance variables that are added to the class's `UNIQUE` constraint.

Let us consider a sample class definition that contains domain and entity constraints.

EXAMPLE 4.15. Based on the `EnrolmentT` type definition, we specify a class capturing properties of course enrolments. This may be done as follows:

```
CLASSDEF EnrolmentC_CC {
  STRUCTURE {
    LectureC_CC lecture;
    StudentC   student;
    EnrolmentT;
  }

  BEHAVIOUR {
    verifyEnrolment ( VOID ) : BOOLEAN;                          // public method
    PRIVATE checkCrsPreRequisites ( VOID ) : BOOLEAN;           // private method
    EnrolmentCCC ( StudentC std, LectureCCC lect );         // object constructor
  }

  CONSTRAINT {
    UNIQUE ( lecture, student );                              // entity constraint
  }
}
```

Above, we have two explicit constraints. In addition, the `UNIQUE` constraint implies that values of instance variables `lecture` and `student` are not `NULL`. Thus, when creating an enrolment object, corresponding lecture and student objects must be already known. The `UNIQUE` constraint implies that those lecture and student objects must not only be known but also represent a unique pair in the corresponding class-collection (including class-collection of all sub-classes). □

**Database Schemata and Classes.** A database schema can be regarded as a collection of class definitions. This collection must be *closed* in the sense that all referenced classes, all super-classes and all non-standard type definitions are also part of the collection. In addition, all non-abstract behaviour specifications must have associated evaluation plans describing the implementation of the respective behaviours. Those evaluation plans must also be closed as defined in Section 4.3. Let us consider a more formal definition for the term schema:

**Definition 4.10.** A *schema* $S$ corresponds to a collection of type definitions $T$ and class definitions $C$ where the following properties are met:

- The class-collection $C$ is initialised with all classes classC$_1$, ..., classC$_n$ that are added explicitly by a user (using schema manipulation commands of the high-level language DBPQL).
- If class `classC`, defined as `CLASSDEF classC IsA supClassC`$_1$, ..., supClassC$_n$, is in $C$ so are all its super-classes supClassC$_1$, ..., supClassC$_n$.
- If class `classC`, with a structure definition containing value-typed variables valType$_1$ var$_1$; ...; valType$_n$ var$_n$;, is in $C$ then all types valType$_1$, ..., valType$_n$ must be iDBPQL system types or must be defined in $T$.
- If class `classC`, with a structure definition containing reference-typed variables classC$_1$ var$_1$; ...; classC$_n$ var$_n$; or `COLLECTION <` classC$_1$ `>` var$_1$; ...; `COLLECTION <` classC$_n$ `>` var$_n$;, is in $C$ so are all referenced class definitions classC$_1$, ..., classC$_n$.
- Each non-abstract behaviour specification part of a type definition in $T$ or a class definition in $C$ must have an associated evaluation plan, which is closed within $S$.
- If class `classC`, with a behaviour specification $B$, is in $C$ then all type definitions and all class definitions that appear in $B$ must be in $T$ and $C$ respectively.
- If class `classC` contains `CHECK` constraints that refer to other classes class$_1$, ..., class$_n$ then class$_1$, ..., class$_n$ must also be in $C$.

– If type `typeT`, with a structure definition containing value-typed variables `valType`$_1$ `var`$_1$`;` ...; `valType`$_n$ `var`$_n$`;`, is in $T$ then all types `valType`$_1$, ..., `valType`$_n$ must be iDBPQL system types or must also be defined in $T$.
– If type `typeT`, with a behaviour specification $B$, is in $C$ then all type definitions that appear in $B$ must also be in $T$.
– $C$ and $T$ must not contain any class definitions or type definitions respectively that do not meet any of the above criteria.

□

Syntax Snapshot 4.7 outlines the corresponding syntax portion that is used subsequently to refer to schema definitions. As indicated, schemata represent shared data that is maintained persistently. A `SchemaBlock` consists of a list of `IMPORTS` that are either used to import whole database schemata (in case of `IMPORTS SCHEMA`) or individual type and class definitions (in case of `IMPORTS Id`$_1$`.Id`$_2$, where `Id`$_1$ identifies a schema and `Id`$_2$ refers to a class or type definition in schema `Id`$_1$).

**Syntax Snapshot 4.7** ***(Definition of iDBPQL DBS MetaData Units)***

```
DBSMetaDataUnit   = Schema;
Schema            = "SCHEMA", Id, '{', SchemaBlock, '}';
SchemaBlock       = { ImportDeclaration }, { SchemaDefinition };
ImportDeclaration = "IMPORTS", [ "SCHEMA" ], Id, [ '.', Id ], [ "AS", Id ];
SchemaDefinition  = ClassDefinition | ConstantDeclaration | TypeDeclaration;
```

□

A sample schema definition (without behaviour specifications) can be found below at the end of this section. This definition is later extended with behaviour specifications and their associated evaluation plans.

**Run-Time MetaData Catalogue Entries.** Besides DBS metadata entries, there are type and class definitions that are associated with evaluation plans. The life-time of such entries is bound to the particular evaluation plan or even a smaller unit within. Syntax Snapshot 4.8 outlines the corresponding iDBPQL syntax portion.

**Syntax Snapshot 4.8** ***(Definition of iDBPQL Run-Time MetaData Units)***

```
RunTimeMetaDataUnit = EvalPlanAnnotation | EvalAnnotation;
EvalPlanAnnotation  = ClassDefinition | ConstantDeclaration | TypeDeclaration;
EvalAnnotation      = LocalDeclaration;
LocalDeclaration    = ConstantDeclaration | TypeDeclaration | VariableDecl;
```

□

Similar to the definition of schemata, each transient type or class definition must be closed in the particular scope (i.e. an enclosing context). While the notion of scope will only be discussed in more detail later in Section 5.3.2, we will briefly outline what the closed constraint implies. Given a run-time class definition `CLASSDEF classC`, we refer to it as *closed* if the following conditions hold:

– If class `classC` is defined as `CLASSDEF classC IsA supClassC`$_1$`,` ..., `supClassC`$_n$ then all its super-classes `supClassC`$_1$, ..., `supClassC`$_n$ must be visible from the current scope and also be closed themselves.

- If class `classC` has a structure definition containing value-typed variables `valType`$_1$ `var`$_1$`;` ...; `valType`$_n$ `var`$_n$`;` then all types `valType`$_1$, ..., `valType`$_n$ must be iDBPQL system types or must be visible from the current scope and also be closed themselves.
- If class `classC` has a structure definition containing reference-typed variables `classC`$_1$ `var`$_1$`;` ...; `classC`$_n$ `var`$_n$`;` then all referenced class definitions `classC`$_1$, ..., `classC`$_n$ must be visible from the current scope and also be closed themselves.
- Each non-abstract behaviour specification in class `classC` must have an associated evaluation plan. In turn, this evaluation plan is likely to have further associated run-time and DBS metadata entries. Each of them must be closed within their own scope.
- If class `classC` contains `CHECK` constraints that refer to another classes `class`$_1$`,` ..., `class`$_n$ then `class`$_1$`,` ..., `class`$_n$ must be visible from the current scope and also be closed themselves.

Given a run-time type definition `TYPEDEF typeT`, we refer to it as *closed* if the following conditions hold:

- If type `typeT` has a structure definition containing value-typed variables `valType`$_1$ `var`$_1$`;` ...; `valType`$_n$ `var`$_n$`;` then all types `valType`$_1$, ..., `valType`$_n$ must be iDBPQL system types or must also be visible from the current scope and also be closed themselves.
- Each non-abstract behaviour specification in class `classC` must have an associated evaluation plan. In turn, this evaluation plan has associated run-time and, possibly also, DBS metadata entries associated. Each of them must be closed within their own scope.

Corresponding examples are outlined throughout this thesis. For instance refer to Examples 4.8, 4.17 and Section 4.5.

**Persistence.** In accordance with the definitions of the '*closed*' property of DBS and run-time metadata entries, we can summarise that:

- Classes (and their objects) and types (and their values) maintained in the DBS metadata catalogue are persistent;
- Classes (and their objects) and types (and their values) defined in the run-time metadata catalogue are transient;
- Persistent classes cannot sub-class transient classes, but transient classes may sub-class persistent classes.

The only means of specifying a 'class-like' construct over persistent classes at run-time is provided through the `UNION`-type. However, the `UNION`-type does not have associated implementations nor can it have an associated, system-maintained collection.

Apart from the different placement of type and class definitions and the restrictions on the inheritance graph, there are no further differences with respect to the treatment of transient and persistent values and objects. For instance, class-collections can be associated with persistent and transient classes (refer to Example 4.16), a `NOT NULL` constraint may be associated with an instance variable of a transient class, a `UNION`-type may be declared over a persistent and a transient class etc.

EXAMPLE 4.16. Consider Example 4.11 and assume that classes **PersonC**, **StudentC** and **AcademicC** are defined in the DBS metadata catalogues as part of the **University**$_{CC}$ schema. The **StudentAcademicC**$_{CC}$ class shall not be persistent. We will only add it at run-time. As a sub-class of a collection-class, **StudentAcademicC**$_{CC}$ must also be a collection-class. Thus, when accessing all objects of class **Student**, the result will consist of all objects that have been instantiated on class **Student** (retrieved from persistent storage) and on class **StudentAcademicC**$_{CC}$ (i.e. objects that only exist in main memory). However, every other requests performed simultaneously, will not be able to access **StudentAcademicC**$_{CC}$ objects (unless it is a transaction originating from the same main evaluation plan – refer to Section 4.4.2). □

**Class Definitions for the University Application.** Let us conclude this section on classes with a summary of all class definitions that form a part of the university application as detailed in Example 3.3.

EXAMPLE 4.17. Similar to Example 4.8, we will restrict ourselves to classes defined in the SCHEMA University$_{CC}$. All class definitions that appear in University$_{LS}$ and University$_{TO}$ schema fragments are replicas.

```
CLASSDEF PersonC {
  STRUCTURE { PersonT; }
  CONSTRAINT { UNIQUE ( personId ); }
}

CLASSDEF CourseC {
  STRUCTURE {
    CourseT;
    SET < CourseC > prerequisites    REVERSE isPrerequisiteOf;
    SET < CourseC > isPrerequisiteOf REVERSE prerequisites;
  }
  CONSTRAINT { UNIQUE ( cNumb ); }
}

CLASSDEF RoomC_CC {
  STRUCTURE { RoomT; }
  CONSTRAINT { UNIQUE ( campus, building, numb ); }
}

CLASSDEF SemesterC {
  STRUCTURE { SemesterT; }
  CONSTRAINT { UNIQUE ( year, sCode ); }
}

CLASSDEF DepartmentC_CC {
  STRUCTURE {
    DepartmentT;
    PersonC              director;
    SET < StudentC >     majorStudents REVERSE major;
    SET < StudentC >     minorStudents REVERSE minor;
    SET < AcademicC_CC > staff REVERSE staffMemberOf;
  }
```

```
}

CLASS StudentC IsA PersonC {
  STRUCTURE {
    StudentT;
    UNIONDEF < UNIONDEF < DepartmentC$_{CC}$, DepartmentC$_{LS}$ >, DepartmentC$_{TO}$ > major;
    UNIONDEF < UNIONDEF < DepartmentC$_{CC}$, DepartmentC$_{LS}$ >, DepartmentC$_{TO}$ > minor;
    UNIONDEF < UNIONDEF < AcademicC$_{CC}$, AcademicC$_{LS}$ >, AcademicC$_{TO}$ > supervisor
      REVERSE supervises;
  }
  CONSTRAINT {
    UNIQUE ( studentId );
    NOT NULL ( major );
  }
}

CLASSDEF AcademicC$_{CC}$ IsA PersonC {
  STRUCTURE
    AcademicT;
    DepartmentC$_{CC}$    staffMemberOf REVERSE staff;
    SET < UNIONDEF < UNIONDEF < LectureC$_{CC}$, LectureC$_{LS}$ >, LectureC$_{TO}$ >
                  lectures REVERSE lecturer;
    SET < StudentC > supervises REVERSE supervisor;
  }
  CONSTRAINT {
    UNIQUE ( personId, staffMemberOf );
    NOT NULL ( staff );
  }
}

CLASSDEF ProjectC {
  STRUCTURE {
    ProjectT;
    SET < UNIONDEF < UNIONDEF < UNIONDEF < AcademicC$_{CC}$, AcademicC$_{LS}$ >,
      AcademicC$_{TO}$ >, PersonC > > participant;
  }
  CONSTRAINT { UNIQUE ( projectId ); }
}

CLASSDEF LectureC$_{CC}$ {
  STRUCTURE {
    LectureT;
    CourseC           course;
    UNIONDEF < UNIONDEF < AcademicC$_{CC}$, AcademicC$_{LS}$ >, AcademicC$_{TO}$ >
                      lecturer REVERSE lectures;
    SemesterC         semester;
    RoomC$_{CC}$ room;
  }
  CONSTRAINT { UNIQUE ( course, semester ); }
}

CLASSDEF EnrolmentC$_{CC}$ {
  STRUCTURE {
    LectureC_CC lecture;
    StudentC   student;
    EnrolmentT;
  }
  CONSTRAINT { UNIQUE ( lecture, student ); }
}

CLASSDEF RecordC {
  STRUCTURE {
    CourseC  course;
    StudentC student;
    RecordT;
  }
  CONSTRAINT { UNIQUE ( course, student ); }
}
```

The corresponding inheritance graph has only two entries. Class `StudentC` is a sub-class of class `PersonC` and class `AcademicC`$_{CC}$ is also a sub-class of class `PersonC`. □

## 4.3 Evaluation Plans

User programs and implementations of type operations and methods are formulated as evaluation plans. As briefly indicated in Section 3.2, each evaluation plan consists of an optional initialisation block, an evaluation block and has several associated metadata entries. Evaluation blocks contain iDBPQL statements and expressions. Associated metadata entries link this execution unit with respective transient and persistent data definitions. Access to (shared) values and objects that are described in the DBS metadata catalogue must be governed by transactions. Transactions are described in terms of blocks. Besides transaction blocks, iDBPQL supports statement blocks and atomic blocks. The latter delay constraint checking until the end of the block is reached.

Before we discuss evaluation plans and iDBPQL statements and expressions in greater detail, we will briefly summarise the challenges to be faced when designing such units of execution.

### 4.3.1 Challenges

Main challenges encountered in this section include:

– *Provide a means of behaviour specification*: An abstraction that captures the user request's main control flow is desired as much as an abstraction bound to behaviour specifications, which are associated with types and classes. While (high-level) programming languages commonly provide different means of such abstractions (e.g. functions, procedures, class implementations, main routines etc. that may be associated with types, classes, modules, packages or entire programs) to ease a programmer's task, a uniform means of an execution unit is suitable for intermediate-level languages.
  iDBPQL provisions the abstraction of evaluation plans. Evaluation plans form an implicit hierarchy. For each user request, there exists an evaluation plan from which

all processing commences. During the execution of this main evaluation plan, invocations of other behaviours are encountered and their respective evaluation plans are processed.

– *Support of common PL abstractions*: It is desired to include programming language statements and expressions commonly found in current OOPLs.
– *Provide a means to access features of super-classes*: OOPLs such as Java and C# provide a `SUPER` mechanism that simplifies programming significantly. In the presence of multiple inheritance, the definition of such a mechanism must be refined. The invocation `SUPER ( )` now refers to multiple classes. This is different compared to most OOPLs that provide such a `SUPER` mechanism. Most commonly, OOPLs only support single inheritance where `SUPER ( )` invocations refer to the direct super-class.
  iDBPQL contains such a refined `SUPER` mechanism, which is also used to by-pass default behaviours associated with object constructors and multiple inheritance.
– *Support deferred constraint checking*: The presence of constraints always raises performance issues and also complicates object construction and data manipulation. For instance, in iDBPQL both ends of a bi-directional reference may have an associated `NOT NULL` constraint. In this case, object creation is not possible without violating the constraint for at least one object. While relational database languages (e.g. SQL) do not permit such cyclic constraints, a general means of deferring constraint checking is supported.
  In iDBPQL, we introduce a special abstraction, i.e. atomic blocks. Constraint evaluation resulting from object creation and data manipulation statements, which are enclosed in such atomic blocks, is delayed until the end of the corresponding block.
– *Transactions*: Access and manipulation of shared data must be governed by transactions. Similar to atomic blocks, iDBPQL supports a block concept to model transactions.
– *Support of QL constructs*: It is desired to include query expressions commonly found in current QLs. Challenges encompass the integration of query expressions and programming language abstractions, and the inclusion of query expressions into a typed language.
  In iDBPQL, query expressions always return collections of values and/or object references or the `NULL` value. For instance, loop statements have been extended to include a `FOR EACH`-variant to better support collections. Result types of query expressions can either be specified explicitly by defining a corresponding structured collection type or implicitly in the case of some `JOIN` expressions.

### 4.3.2 Components of Evaluation Plans

Evaluation plans serve as evaluation units. As mentioned earlier, they model user requests, behaviour associated with types and classes, and built-in behaviours that form part of the iDBPQL library. An evaluation plan consists of a unique identifier (unique within its scope), an optional initialisation block and an evaluation block. The corresponding syntax portion is outlined in Syntax Snapshot 4.9.

**Syntax Snapshot 4.9** ***(iDBPQL Evaluation Plan Syntax)***

```
EvaluationUnit = iDBPQLProgram;
iDBPQLProgram  = "EVALPLAN", Id, '(', Argument-List, ')', [ ':', ReturnType ],
                 EvalPlanBlock;
EvalPlanBlock  = [ EvalPlanInit ], EvalBlock;
EvalPlanInit   = "INIT", DoBlock;
EvalBlock      = DoBlock;
```

□

An *initialisation block* permits the initialisation of elements of the evaluation plan before the start of the evaluation. For instance, class variables must be initialised. However, since they are shared among all class instances, they must be initialised only once. Corresponding routines can be added to an evaluation plan's initialisation block.

*Evaluation blocks* describe the implementation of a behavioural iDBPQL entity. Such blocks consists of sequences of iDBPQL statements, which may be subdivided into further evaluation blocks as introduced in Section 4.3.3 and later extended in Section 4.4.2.

### 4.3.3 Evaluation Blocks and Their Properties

An evaluation block is a language construct that supports the grouping of a sequence of statements into a (sub-)unit. Such groups may serve special purposes such as modelling atomic steps and transactions. In addition, blocks may have variables associated that are local to the respective block. Syntax Snapshot 4.10 outlines the corresponding iDBPQL syntax portion. Local variable declarations are associated in the form of metadata references.

**Syntax Snapshot 4.10** ***(iDBPQL Evaluation Block Syntax)***

```
DoBlock     = ( ( '{' | ( "DO", [ "ATOMIC" ], [ "TRANSACTION", Tid ] ) ),
              Statements, ( '}' | "ENDDO" ) ) | DoThenBlock;       // later extended

DoThenBlock = "DO", Statements, [ "THEN", DoBlock ], "ENDDO";      // later extended
```

□

{ ... } blocks, `DO` ... `ENDDO` blocks and `DO` ... `THEN` ... `ENDDO` blocks are basic grouping constructs that do not serve any special purpose apart from providing a grouping of statements. This is different for `DO ATOMIC` ... `ENDDO` blocks and `DO TRANSACTION` ... `ENDDO` blocks. The former, i.e. atomic blocks, model atomic execution units. All statements grouped into an atomic block are executed as if they correspond to a single computational step. As a result, constraint checking associated with any statement inside an atomic block is delayed until the corresponding `ENDDO` keyword is encountered. The same applies to the visibility of effects of object creation and other data manipulation statements.

The latter, i.e. transaction blocks, are used to model transactions as briefly indicated in Section 3.2. A transaction block has an associated identifier that uniquely identifies the transaction. Transaction identifiers support the specification of multiple transactions within the same evaluation block. The first appearance of a transaction identifier associates a new transaction object with the respective block. The transaction object is implicitly associated with every statement encountered until the block's

ENDDO keyword (refer to line 13 in Example 4.18 below) or an explicit commit ( ) or abort ( ) invocations is encountered (refer to line 16 in Example 4.18 below). In the event that there is a sub-block, which corresponds to another transaction block with a different identifier, the current transaction is suspended and a new transaction object is associated with the respective block (refer to line 05 in Example 4.18 below). The first transaction is continued if the ENDDO keyword of the sub-block is encountered (refer to lines 10 and 13 in Example 4.18 below) or another sub-block that carries the same identifier as the first transaction block (refer to line 08 in Example 4.18 below) is specified. In the latter case, the second transaction is suspended.

EXAMPLE 4.18. This example demonstrates how two transactions can be specified within the same evaluation plan.

```
01    ...
02    DO TRANSACTION tr1                        // creates a new transaction object
03      doSomething;                            // operations of transaction tr1
04
05      DO TRANSACTION tr2                   // creates a second transaction object
06        doSomething;                          // operations of transaction tr2
07
08        DO TRANSACTION tr1                   // continues with transaction tr1
09          doSomething;                        // operations of transaction tr1
10        ENDDO;
11
12        doSomething;                          // operations of transaction tr2
13      ENDDO;                                        // commits transaction tr2
14
15      doSomething;                            // operations of transaction tr1
16      tr1.commit( );                      // explicit commit of transaction tr1
17
18      doSomething;                          // operations are non-transactional
19    ENDDO;
20    ...
```

At the moment, this specification results in a forced (or static) interleaving of two transactions. However, this can be disastrous since forced interleavings would require a means of guaranteeing conflict freeness (which has to be provided by the programmer, compiler or optimiser). The specification of truly concurrent transactions that originate from the same evaluation plan is introduced later in Section 4.4.2 (refer to Example 4.24). The rationale behind supporting interleavings as presented in this example will then become more obvious. □

Assume that an evaluation plan with transaction blocks is invoked from within a transaction block defined in another evaluation plan. In such an event, the outermost transaction block defines the transaction. Transaction blocks encountered during the evaluation of other associated evaluation plans are demoted to sub-transactions (as defined for multi-level transactions [16, 141, 142]).

The following additional properties must be preserved when accessing persistent objects and values: Evaluation plans associated with DBS metadata entries must be invoked only from within a transaction block. This is necessary to ensure that access to shared data is serialised by the transaction management system. The same restriction applies to DBS metadata references. On the contrary, references to and invocations of entities located in the run-time metadata catalogue may be enclosed in transaction blocks but this is not required.

### 4.3.4 Statements

A *statement* can be interpreted as a unit of execution. As in most PLs, iDBPQL statements do not return results (with some exceptions where expressions are permitted to appear as statements) and are executed solely for their side effects. This is in large contrast to their internal components, i.e. expressions. *Expressions* (refer to Section 4.3.5) always return a value and often do not have side effects.

**Syntax Snapshot 4.11** ***(iDBPQL Statements)***

```
Statements      = [ Statement, { ';', Statement } ];

Statement       = ControlFlowStmt | DoBlock | ExpressionStmt;

ControlFlowStmt = BreakStmt | ConditionStmt | LabelStmt | LoopStmt | ReturnStmt |
                  SwitchStmt | WaitStmt;
ExpressionStmt  = ( AssignmentExpr | CreationExpr | MethodCallExpr |
                    TypeOpCallExpr ), ';';
```

□

As indicated in Syntax Snapshot 4.11, iDBPQL distinguishes between the following types of statements:

- The *empty statement* ;, which has no effect;
- *Assignment statements*, which set or reset the value assigned to a variable;
- *Block statements*, which group together a sequence of statements as already discussed in Section 4.3.3;
- *Control flow statements*, which regulate the order in which statements are executed; and
- *Expressions with side-effects* (e.g. method invocations that create new objects, which are accessible by means other than object references).

Assignment statements, control flow statements and expressions with side-effects that form part of the iDBPQL language are discussed in more detail next.

**Assignment Statements.** An *assignment statement* assigns the value of an expression to a variable. Syntax Snapshot 4.12 outlines four types of assignments. These are: Regular assignments, compound assignments (i.e. arithmetic assignments and bit-manipulating assignments), pre-incrementation and pre-decrementation, and post-incrementation and post-decrementation. Only regular assignments are applicable to both types of variables, i.e. value-type variables and reference-type variables. The latter three types are applicable to value-type variables only.

**Syntax Snapshot 4.12** ***(iDBPQL Assignment Statements)***

```
AssignmentExpr    = ( Expression, '=', Expression ) |
                    ( Expression, CompoundAssignOp, Expression ) |
                    ( Expression, InDeCrementOp ) | ( InDeCrementOp, Expression );
CompoundAssignOp = "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=";
InDeCrementOp     = "++" | "--";
```

□

In a regular assignment statement, the left-hand side of the assignment operator must be classified as a variable, while the expression on the right-hand side of the assignment operator must be classified as a value. Thus, the type of the right-hand expression must be implicitly convertible to the type of the variable.

Compound assignment statements can be converted easily into regular assignment statements where the left-hand side corresponds to a variable and the right-hand side can be classified as a value. A compound assignment statement is of the form: `Expression`$_1$`, ( "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=" ), Expression`$_2$ which can be converted into the following regular assignment statement: `Expression`$_1$`, '=', Expression`$_1$` ( '+' | '-' | '*' | '/' | '%' | "<<" | ">>" ), Expression`$_2$.

Pre-incrementation and pre-decrementation statements are of the form: `( "++" | "--" ), Expression`. Post-incrementation and post-decrementation statements are of the form: `Expression, ( "++" | "--" )`. Considering these two types of in-/decrementations as individual statements, they can be converted into the same regular assignment expression which is: `Expression, '=', Expression, ( '+' | '-'), '1'`. Pre- and post-de-/incrementations only behave differently when being passed as arguments or being returned as results.

**Control Flow Statements.** The evaluation of iDBPQL statements is usually performed in a serial manner resulting in a sequential flow of control. However, similar to most other PLs, iDBPQL has control flow statements which allow variations in this sequential order. Syntax Snapshot 4.13 provides on overview of these statements. In addition, Section 4.4 will discuss another means of altering the flow of control.

### Syntax Snapshot 4.13 *(iDBPQL Control Flow Statements)*

```
ControlFlowStmt = BreakStmt | ConditionStmt | LabelStmt | LoopStmt |
                  ReturnStmt | SwitchStmt | WaitStmt;

BreakStmt       = "BREAK", [ LabelId ], ';';

ConditionStmt   = "IF", '(', Expression, ')', DoBlock,
                  { "ELSEIF", '(', Expression, ')', DoBlock }, [ "ELSE", DoBlock ];

LabelStmt       = "LABEL", LabelId, ':', Statement;

LoopStmt        = DoWhileLoop | ForEach | LoopLoop | WhileLoop;
DoWhileLoop     = DoBlock, "WHILE", '(', BooleanExpr, ')', ';';
ForEach         = "FOR EACH", Expression, DoBlock;
LoopLoop        = "LOOP", DoBlock;
WhileLoop       = "WHILE", '(', BooleanExpr, ')', DoBlock;
```

```
ReturnStmt      = "RETURN", '(', [ Expression ], ')', ';';

SwitchStmt      = "SWITCH", '(', Expression, ')', SwitchBlock;
SwitchBlock     = '{', { CaseBlock }, [ "DEFAULT", ':', DoBlock ], '}';
CaseBlock       = "CASE", Expression, ':', DoBlock;

WaitStmt        = "WAIT", [ LabelId ], ';';
```

□

iDBPQL control flow statements have the following properties:

- The ***LABEL statement*** may add a prefix to any statement. Such a prefix consists of the keyword LABEL followed by an identifier followed by a colon. Labels correspond to reference points which can be utilised by BREAK and WAIT statements.
- The *conditional IF-THEN-ELSE statement* specifies an execution choice based on a given condition (i.e. a boolean expression). If this expression is evaluated to TRUE the THEN block is processed. Otherwise, the ELSE block is executed. *IF-THEN-ELSEIF-ELSE statements* are also supported. Instead of only one alternative, it is now possible to specify multiple alternatives (each with its own condition).
- The *SWITCH statement* allows the value of a variable or expression to control the execution flow. The SWITCH block consists of labelled CASE blocks. Execution continues with the block following the label that matches the SWITCH value. If no label matches, execution continues at the DEFAULT label. In case such a DEFAULT label is missing, execution of the SWITCH statement terminates.
- The following loop or iteration-type statements are supported in iDBPQL:
  - The *WHILE-loop statement* and the *DO ... WHILE-loop statement* are control flow statements that allow code to be executed repeatedly based on a given condition. Only difference being the point in time where the condition is evaluated. WHILE-loops are only entered in case the condition is evaluated to TRUE. In contrast to this, DO ... WHILE-loops only evaluate the condition at the end of each iteration.
  - The *LOOP-loop statement* also allows code to be processed repeatedly. However, it does not specify a terminal condition. Instead, the loop must contain a BREAK statement ending the cycle of iterations.
  - The *FOR EACH statement* supports the traversal of values or objects in a collection. In contrast to more traditional FOR-loops as known from procedural PLs such as Pascal, FOR EACH-loops do not specify the order in which the values or objects are considered. Such loops terminate as soon as the last member of the collection has been evaluated.
- The *BREAK statement* which may appear within a loop statement or a SWITCH statement. It consists of the keyword BREAK optionally followed by a label and terminated by a semicolon. A BREAK statement without a label terminates the innermost loop or SWITCH statement. A BREAK statement with a label terminates the corresponding labelled statement (which it must form a part of).
- The *RETURN statement* which terminates the execution of an evaluation plan. It consists of the keyword RETURN followed by a possibly empty expression enclosed in parentheses terminated by a semicolon. Termination means that the execution should return to the calling evaluation plan. If a non-empty expression is specified, the RETURN statement reports the expression's value to the calling evaluation plan.

– The *WAIT statement* which synchronises simultaneous execution flows. Section 4.4 will introduce corresponding means of specifying simultaneous processing. A `WAIT` statement without a label causes the current execution flow to wait until all concurrent execution flows that originated from the current flow have terminated. A `WAIT` statement with a label only results in a waiting period until the corresponding labelled simultaneous execution flow has terminated.

Examples can be found throughout the remaining part of this chapter, e.g. refer to Examples 4.22 and 4.24 as well as Section 4.5.

**Type Operation Invocation, Method Calls and Object Creation.** Behaviour may be associated with both type definitions and class definitions. The invocation of such behaviours constitutes a statement. As outlined in Syntax Snapshot 4.14, a *type operation invocation statement* consists of a type operation identifier followed by a possibly empty argument list (enclosed in parentheses) followed by a semicolon. Analogously, a *method call statement* consists of a method identifier followed by a possibly empty argument list (enclosed in parentheses) followed by a semicolon. Considering both types of behaviour invocation, arguments must correspond to type operation parameters or method parameters respectively in both type and number.

**Syntax Snapshot 4.14** ***(iDBPQL Expression Statements)***

```
ExpressionStmt = ( AssignmentExpr | CreationExpr | MethodCallExpr |
                   TypeOpCallExpr ), ';';

CreationExpr   = "NEW", ClassId, '(', [ Argument-List ], ')';
MethodCallExpr = MethodId, '(', [ Argument-List ], ')';
TypeOpCallExpr = TypeOpId, '(', [ Argument-List ], ')';
```

□

A behaviour invocation results in the evaluation of the corresponding evaluation plan implementing the particular type operation or method. Its execution first initialises the behaviour's parameters with the provided argument values. Subsequently the evaluation plan's initialisation block is evaluated followed by the evaluation block.

In addition to type operation invocation and method call statements, *object creation statements* fall into the class of behaviour invocation statements. An object creation statement consists of the reserved keyword `NEW` followed by a class identifier, a possibly empty argument list (enclosed in parentheses) and a semicolon. While such an usage is uncommon in traditional OOPLs (typically object creation expressions are found as the right-hand expression in assignment statements), object-oriented DBPLs encourage it. This can be justified by having collections associated with classes. Thus, object access does not solely rely on object references.

Behaviour invocation statements are the only type of statements that may return results. For instance, an object creation statement always returns a reference to the newly created object. However, this reference is discarded at the end of the execution of the statement. The object will still be available through the corresponding class-collection (if it exists).

A special invocation statement is the SUPER call. It consists of the SUPER keyword followed by a possibly empty argument list (enclosed in parentheses) and a semicolon. The only arguments that are allowed are class identifiers. Section 4.3.5 considers such calls in more detail. SUPER call statements may appear at the beginning of object constructor implementations invoking object constructors of all direct super-classes.

### 4.3.5 Expressions

An *expression* can be thought of as a combination of literals, identifiers, values, objects, variables, operators, and behaviours. Expressions are iDBPQL code segments that perform computations and produce values. Syntax Snapshot 4.15 outlines the expressions of iDBPQL, which we will discuss in more detail next.

**Syntax Snapshot 4.15** ***(iDBPQL Expressions)***

```
Expression = AssignmentExpr | BinaryTypeOpExpr | BooleanExpr | CastExpr |
             CreationExpr | Identifier | Literal | MethodCallExpr | QueryExpr |
             RenamingExpr | TypeOpCallExpr | UnaryTypeOpExpr |
             ( '(', Expression, ')' );
```

□

**Simple Expressions.** Literals, variable names, class names and constant names are considered as simple expressions. A literal simply evaluates to the value it denotes. A variable name denotes the value it stores or references respectively. A class name denotes the collection associated with it (if any). A named constant denotes the value it is initialised with.

**Parenthesised Expressions.** Any expression can be enclosed in parentheses as specified in Syntax Snapshot 4.16.

**Syntax Snapshot 4.16** ***(iDBPQL Parenthesised Expressions)***

```
Expression = '(', Expression, ')';
```

□

**Assignment Expressions.** Refer to Section 4.3.4 where assignment statements are introduced. In short, an *assignment expression* is an assignment statement (without the terminating semicolon) that can be used wherever an expression may be used.

**Type Operation Invocation, Method Calls and Object Creation.** Refer to Section 4.3.4 where type operation invocation, method calls and object creation are introduced. In short, respective expressions for each corresponding statement (without the terminating semicolon) can be used wherever an expression may be used.

**Renaming Expressions.** A *renaming expression* associates an identifier with an expression. This identifier can then be used to refer to the value of the expression or its sub-expressions. As indicated in Syntax Snapshot 4.17 two types of renaming expressions are supported.

**Syntax Snapshot 4.17** ***(iDBPQL Renaming Expression)***

```
RenamingExpr = Expression, [ "GROUP" ], "AS", Id;
```

□

The *AS renaming expression* associates auxiliary identifiers with non-collection values. In the event that an expression denotes a collection value, the assigned identifier moves into the collection identifying the collected values. For instance, the renaming expression `StudentC AS std` consists of a class name denoting the collection of all objects of class `StudentC`. The assigned identifier `std` is not associated with the collection rather it is associated with each `StudentC` objects in the collection. Such a naming mechanism has many useful applications including: Cursors in `FOR EACH` statements, iteration variables in loops and queries, variables bound by quantifiers etc. On the other hand, in case the expression denotes a non-collection value the identifier is associated with the value.

A *GROUP AS renaming expression* simply associated an identifier with an expression (no matter whether it is a collection or not). This identifier can then be used in place of the expression. Thus, the identifier has the same value associated as the original expression.

Note: `AS` and `GROUP AS` renaming expressions have only different semantics for collection values.

**Boolean Expressions.** A *boolean expression* is an expression that produces a value of type `BOOLEAN`. Syntax Snapshot 4.18 outlines the types of boolean expressions supported in iDBPQL.

**Syntax Snapshot 4.18** ***(iDBPQL Boolean Expressions)***

```
BooleanExpr       = EqualityExpr | InCollectionExpr | InheritanceExpr |
                    InstanceOfExpr | LogicalExpr | NULLExpr |
                    QuantifierExpr | RelationalExpr;

EqualityExpr      = Expression, ( "==" | "!=" ), Expression;
InCollectionExpr = Expression, "IN", Expression;
InheritanceExpr   = Expression, ( "ISSUBTYPEOF" | "ISSUBCLASSOF" ), Expression;
InstanceOfExpr    = Expression, "ISINSTANCEOF", Expression;
LogicalExpr       = ( BooleanExpr, ( "&&" | "||" ), BooleanExpr ) |
                    ( '!', BooleanExpr );
NULLExpr          = Expression, "IS", [ "NOT" ], "NULL";
QuantifierExpr    = ( "EXISTS" | "FOR ANY" ), Expression, '(', BooleanExpr, ')';
RelationalExpr    = ( Expression, ( '<' | "<=" | ">=" | '>' | "LIKE" ), Expression );
```

□

An *equality expression* compares the values of two expressions of compatible types for equality (i.e. ==) or inequality (i.e. !=).

A *relational expression* compares the values of two expressions of compatible, ordered types. Available comparison operators are: Less than (i.e. <), less than or equal to (i.e. <=), greater than (i.e. >), and greater than or equal to (i.e. >=).

A *logical expression* either negates the boolean value of the expression using the logical *NOT* operator (i.e. !) or combines two boolean expressions. The latter can be achieved through the logical *AND* (i.e. `&&` or `&`), *OR* (i.e. `||` or `|`) and *XOR* (i.e. `^`) operators.

A ***NULL** expression* tests whether the value of the expression is (i.e. `IS`) or is not (i.e. `IS NOT`) equal to `NULL`.

An ***IN**-collection expression* determines whether the value of the left-hand expression forms a part of the collection resulting from evaluating the right-hand expression.

A *quantifier expression* consists of the keyword `EXISTS` (i.e. a generalised *OR* operator) or `FOR EACH` (i.e. a generalised *AND* operator) followed by an expression that denotes a collection value followed by a condition. The `EXISTS` quantifier expression is evaluated to `TRUE` if the condition evaluates to `TRUE` for at least one collection member. In contrast, the `FOR EACH` quantifier expression evaluates to `TRUE` if the condition evaluates to `TRUE` for each member of the collection denoted by the expression.

A *sub-type expression* tests whether or not the type produced by the left-hand side expression is a sub-type of the type resulting from the evaluation of right-hand side expression.

An *inheritance expression* determines whether or not the class produced by the left-hand side expression is a sub-class of the class resulting from the evaluation of the right-hand side expression.

An ***INSTANCEOF** expression* tests whether the (referenced) object produced by the left-hand side expression is an instance of the class (or any of its sub-classes) named in the right-hand side expression.

**Query Expressions.** Syntax Snapshot 4.19 outlines query expressions as supported by iDBPQL. These query expressions are based on SBQL queries [131]. Queries range from very simple expressions (e.g. a projection of a collection of values of a structured type to a collection of one member of the original structured type) to very complex and deeply nested expressions. Below, we will consider each iDBPQL query expression in more detail.

### Syntax Snapshot 4.19 *(iDBPQL Query Expressions)*

```
QueryExpr         = JoinExpr | OrderByExpr | ProjectionExpr | SelectionExpr |
                    UniquenessExpr;

JoinExpr          = Expression, [ "NAVIGATIONAL" | "INNER" |
                    ( [ "NATURAL" ], [ "LEFT" | "RIGHT" ], [ "OUTER" ] ) ], "JOIN",
                    Expression, [ "ON", ( PathExpr | BooleanExpr ) ];
OrderByExpr       = Expression, "ORDER BY", Expression, [ "ASC" | "DESC" ];
ProjectionExpr    = Expression, '.', ProjectComponent;
ProjectComponent  = Expression |
                    ( '(', ProjectComponent, { ',', ProjectComponent }, ')' );
SelectionExpr     = Expression, "WHERE", BooleanExpr;
UniquenessExpr    = ( "DISTINCT" | "UNIQUE" ), Expression;
```

□

Query expressions most commonly return collection values, e.g. collections of Natural values, collections of instances of class `PersonC` etc. These collections may contain other collection values, structured values, reference values or calculated values. The respective collection type may be any collection type supported by iDBPQL. Less commonly, query expressions return a single value which may be a structured value, a reference value or a calculated value.

Query expressions supported by iDBPQL have the following associated properties:

- A *(conditional) selection query expression* consists of an expression followed by the keyword `WHERE` and a condition. First, the expression is evaluated. For each resulting value of that expression, it is tested whether or not the condition evaluates to `TRUE`. If so, the value is added to the resulting collection.
- A *projection query expression* consists of an expression followed by the '`.`' (dot) operator followed by a single projection expression or a list of projection expressions. First, the expression is evaluated. For each resulting value of that expression, a projected value is generated. If a single (right-hand) projection expression is encountered, a collection of values of the type of the projected value is produced. Otherwise, if there is a list of projection expressions, a collection of a structured type (that corresponds to the projection list) is computed.
  Note: Projection may result in navigation in case a sequence of projection expressions (i.e. a path expression) is specified.
- iDBPQL supports various types of join operations. These are the navigational join, the natural join, the inner join, the left outer join, the right outer join and the (full) outer join. All types of join operations have a common structure. A *join query expression* consists of a left-hand side expression, one or more keywords specifying the join type followed by a right-hand side expression. In addition, some join query expressions may have a condition. Navigational joins and natural joins do not have such a condition whereas all other join types must have it. The condition specifies how both expressions are to be joined, i.e. how the results of the evaluation of both the left-hand side and the right-hand side expressions are to be merged into one collection value.
  The *navigational join* spans two interconnected classes using path navigation. It produces a collection of pairs of objects where the second object is reachable from the first object. In contrast to all other join types, the navigational join consists of a right-hand side expression that corresponds to a path expression. In all other cases, the evaluation of both the left-hand side expression and the right-hand side expression results in a collection of a structured type (e.g. a reference value, a class name, a projected value, a union value or a join value).
  The *natural join* joins two structured collections where all identically named instance variables have matching values.
  The *(inner) join* merges two structured collections so that the specified condition (a path expression or boolean expression) is evaluated to `TRUE`.
  All outer joins evaluate the specified condition but differ in the way the result is produced. The *left outer join* contains entries from the left-hand side expression's collection whether or not they had matches in the right-hand side expression's collection.
  The *right outer join* contains entries from the right-hand side expression's collection whether or not they had matches in the left-hand side expression's collection.
  The *(full) outer join* combines the results of the left and right outer joins.

- An *order-by expression* consists of a left-hand side expression denoting a collection value followed by the keyword `ORDER BY` followed by possibly many right-hand side expressions identifying the instance variables on which the sorting is based upon. Each of these right-hand side expressions may be followed by the keyword `ASC` (i.e. ascending which is the default) or `DESC` (i.e. descending) specifying the sort order. If more than one right-hand side expression is specified, the resulting collection is ordered first by values that correspond to the first expression, then by values that correspond to the second expression etc.
- Any query expression can be prefixed with the `DISTINCT` keyword or the `UNIQUE` keyword. If such a keyword is specified only those values / objects are selected that are distinct or unique respectively. `DISTINCT` and `UNIQUE` can be used synonymously for value-types. However, their associated meaning is different for reference-types, i.e. objects. An expression prefixed with the `DISTINCT` keyword only returns objects that have different values on selected variables. `UNIQUE` means that only those objects that are different with respect to an associated uniqueness constraint or that represent different objects are selected (i.e. have different internal object identifiers).

Let us consider some initial examples of query expressions.

EXAMPLE 4.19. Again, we consider the schema of the university application. First, we outline simple selection expressions that also include navigation, projection, behaviour invocations, `DISTINCT` expressions, and `UNIQUE` expressions.

```
01   ...                    // metadata reference: NULLABLE < AcademicC_CC > myProf;
02   myProf = ( AcademicC_CC WHERE ( ( staffMemberOf.dName ==
03     "Department of Information Systems" ) && ( name.( firstname, lastname ) ==
04     ( "Klaus-Dieter", "Schewe" ) ) ) ).getMember ( );
05
06   ...                    // metadata reference: NULLABLE < SET < StudentC > > res1;
07   res1 = StudentC WHERE ( supervisor == myProf );
08
09   ...          // metadata reference: NULLABLE < BAG < STRING > > res2, res3, res4;
10   res2 = ( StudentC WHERE ( supervisor == myProf ) ).addr.city;
11   res3 = UNIQUE ( StudentC WHERE ( supervisor == myProf ).addr.city );
12   res4 = DISTINCT ( StudentC WHERE ( supervisor == myProf ).addr.city );
13
14   ...                    // metadata reference: NULLABLE < LIST < StudentC > > res5;
15   res5 = UNIQUE ( StudentC WHERE ( major.dName ==
16     "Department of Information Systems" ) ).append ( StudentC WHERE (
17     minor.dName == "Department of Information Systems" ) );
18
19   ...                    // metadata reference: NULLABLE < SET < StudentC > > res6;
20   res6 = ( StudentC WHERE ( major.dName == "Department of Information Systems" )
21     ).union ( StudentC WHERE ( minor.dName ==
22    "Department of Information Systems" ) );
23
24   ...
```

The first selection expression in lines 02 to 04 retrieves the object (i.e. a reference to the object) that represents the specified academic staff member. This object is used subsequently (refer to line 07) to retrieve the set of students the staff member supervises.

In lines 10 to 12, a similar query expression is executed. First, all objects representing students who have the staff member as their supervisor are selected. Subsequently, a projection is performed only retaining the `city` field of the students' `address` values. Differences are:

- Line 10 is evaluated without a uniqueness expression. This expression returns a bag of `STRING` values of city names or, in case the staff member does not supervise any students, the `NULL` value. In the former case, there are as many result values as student objects that have the staff member as their supervisor.
- Lines 11 and 12 apply a `UNIQUE` or `DISTINCT` expression respectively. As a result, the number of returned city names decreases. Only unique city names are returned, i.e. the resulting bag corresponds to a set.

While these first two applications of uniqueness expressions resulted in the same return value, this will be different when objects are concerned. There might be two objects of identical values but different internal identifier. In such a case, the `DISTINCT` expression would return only one of these objects but the `UNIQUE` expression would return both. The final two selection expressions utilise type operations associated with collection types. In lines 15 to 17, results of two selections are first concatenated and then filtered to remove duplicates. While filtering is explicit, it is done implicitly in lines 20 to 22.

Next, we consider join expressions. While navigational joins return collections of pairs of objects, inner and outer join operators produce results of a new structured type. A special means of defining structured types over classes, which eases the ways to how corresponding result types of joins are specified, is proposed.

```
30   ...           // metadata reference: NULLABLE < SET < STRUCT { AcademicC$_{CC}$ aca;
31                                                   // DepartmentC dept; } > > res7;
32   res7 = AcademicC$_{CC}$ GROUP AS aca NAVIGATIONAL JOIN staffMemberOf GROUP AS dept;
33
34   ...         // metadata reference: NULLABLE < SET < STRUCT STRUCT AcademicC$_{CC}$ &
35                                                    // STRUCT DepartmentC > > res8;
36   res8 = AcademicC$_{CC}$ AS aca INNER JOIN Department ON aca.staffMemberOf;
37
38   ...               // metadata reference: NULLABLE < SET < STRUCT STRUCT PersonC &
39                 // STRUCT PersonC AS person2 WITH { person2.personId AS person2Id;
40          // person2.name AS person2Name; person2.addr AS person2Addr; } > > res9;
41   res9 = PersonC AS p1 INNER JOIN PersonC AS p2 ON p1.addr == p2.addr;
42
43   ...           // metadata reference: NULLABLE < SET < STRUCT { AcademicC$_{CC}$ aca;
44                                                          // NAT stdNumb; } > > resA;
45   resA = AcademicC$_{CC}$ GROUP AS aca NAVIGATIONAL JOIN (
46     ( SET < StudentC > ) supervises ).count AS stdNumb;
47
48   ...                  // metadata reference: NULLABLE < SET < STRUCT { NameT name;
49                                 // STRING specialisation; NAT stdNumb; } > > resB;
50   resB = ( AcademicC$_{CC}$ GROUP AS aca LEFT OUTER JOIN ( ( SET < StudentC > )
51     supervises ).count AS stdNumb ).( name, specialisation, stdNumb );
52
53   ...
```

The first join expression (i.e. line 32) represents a simple navigational join. The result is a set of pairings of references of academic staff members and the departments they work in. Next, in line 36, the same result is computed, but this time the INNER JOIN operator is used. Instead of a set of pairings of object references, we now obtain a set of structured values. The structure of values is determined by concatenating the structures underlying both classes that are joined. The third join expression also computes an INNER JOIN. However, instead of specifying a boolean expression as join condition, a path expression is given. Thus, the inner join is similar to a navigational join but a collection of structured values is computed instead of pairs of object references.
The second set of query expressions demonstrates how grouping can be achieved. Analogous to SBQL, there is no explicit GROUP BY clause included in iDBPQL. Lines 45 and 46 generate a set of references to staff objects where each are paired with the number of students they supervise. The last query expression shown in lines 50 and 51 contains the same grouping. However, this time the LEFT OUTER JOIN operator is used. The intermediate result corresponds to the following metadata reference: NULLABLE < SET < STRUCT STRUCT AcademicC$_{CC}$ & STRUCT { NAT stdNumb; } > >. Finally, a projection is executed extracting values of the three indicated variables only. □

Further examples are found throughout the remainder of this chapter.

**Cast Expressions.** A *cast expression* consists of a parenthesised type or class identifier followed by an expression (as indicated in Syntax Snapshot 4.20). A type cast converts a non-reference-valued expression to a sub- or super-type. A class cast converts a reference-valued expression to a sub- or super-class.

**Syntax Snapshot 4.20** ***(iDBPQL Cast Expressions)***

```
CastExpr = '(', ( TypeId | ClassId ), ')', Expression;
```

□

**The SUPER Expression and Keywords THIS and SUPER.** Instance methods and object constructors may have associated parameter variables or local variables that have the same name as an instance member. When using such a name within the instance method or object constructor, its occurrence refers to the parameter variable or local variable. The instance member is not visible directly but may be accessed using the prefixed THIS keyword (followed by a period). The THIS keyword denotes a value, that is a reference to the object for which the instance method was invoked or to the object being constructed. The type of THIS is *reference-to-X* where *X* is the class in which the keyword THIS occurs.

EXAMPLE 4.20. Let us consider the PersonC class from Example 4.9 again. We add behaviour specifications as follows:

```
01  CLASSDEF PersonC {
02    STRUCTURE { PersonT; }
03
04    BEHAVIOUR {
05      equals ( PersonC other ) : Boolean;
06      PersonC ( NameT name, AddressT addr );                   // object constructor
07    }
08  };
```

The class PersonC implements a method equals, which compares two persons. It may be implemented as follows:

```
10  EVALPLAN equals ( PersonC other ) : Boolean {
11
12    if ( THIS == other ) {
13      return ( TRUE );
14    }
15    else {
16      // perform comparison tests
17      ...
18    }
19
20    return ( TRUE );
21  }
```

If the other person is the same PersonC object as the one for which the equals method was invoked (i.e. determined by comparing the reference to the other object to THIS), comparisons tests can be skipped.

The PersonC class also implements an object constructor that takes two arguments. The corresponding two parameter variables have the same names as two instance variables. Accessing those parameter and instance variables in the implementation of the object constructor is carried out as demonstrated in the following evaluation plan:

```
30  EVALPLAN PersonC ( NameT name, AddressT addr ) {
31
32    personId = ...                              // assign unique person identifier
33    THIS.name = name;
34    THIS.addr = addr;
35
36    return ( VOID );
37  }
```

In line 33, THIS.name refers to the instance variable while name refers to the parameter variable. In line 32, however, personId refers to the instance variable. Since there is no parameter or local variable with the same name, it is not necessary to say THIS.personId but the programmer may choose to do so. □

Similarly, the THIS keyword can also be used to directly access members in a super-class in the exact same situations described above. However, while there is only one instance member that may have the same name as a parameter variable or local variable, a class can have multiple direct super-classes. Thus, a class may override an instance member in each of its super-classes. To identify the particular super-class, the THIS keyword must be specified with a preceding cast expression. For instance, consider the expression ( ( PersonC ) THIS ).name located in an evaluation plan implementing a method of a sub-class of Person. The expression refers to the instance member named `name` of the current object, but with the current object viewed as an instance of the super-class `PersonC`. Thus it can access the instance member named `name` that is visible in class `PersonC`, even if that class member is hidden by a declaration with the same name in the current class.

While accessing an instance variable of a super-class may be achieved through the `THIS` keyword with a preceding cast expression, the same does not apply to method invocations. Casting does not change the method that is invoked because the instance method to be invoked is chosen according to the run-time class of the object referred to by `THIS`. Casting only checks that the class is compatible with the specified type. Instead an overridden instance method of a super-class may be accessed by using the `SUPER` keyword (optionally with the name of the super-class as first argument).

EXAMPLE 4.21. Let us revisit Example 4.11 that demonstrated how ambiguities related to multiple inheritance are to be dealt with. Of particular interest are means to:

– Bypass inheritance mechanisms that override a super-class's method or prioritise a method of one super-class over a method of another super-class; and
– Invoke object constructors in the presence of multiple super-classes.

We start considering the latter first. Assume we want to implement an object constructor for class `StudentAcademicC`$_{CC}$. Since object constructors are not inherited, they must be invoked explicitly at the beginning of the implementation of each new constructor. A `SUPER ( );` call results in the invocation of the object constructor (with an empty argument list) of each direct super-class in the order as the respective super-classes appear in the `IsA` clause.
In the event that invocation in the `IsA` order is not desired, there must be a sequence of `SUPER (` *`className`* `);` calls, where *className* is the name of a direct-superclass. This sequence must not be broken by any other statements and has to consist of exactly one `SUPER` invocation for each direct super-class. `SUPER (` *`className`* `);` invokes the object constructor of class *className* that has an empty argument list.
Analogous, object constructors with argument list may be invoked. However, the first argument of a `SUPER` call must always be a class name. The first argument passed to the super-class's constructor will be the second argument of the `SUPER` call and so on. The default constructor of class `StudentAcademicC`$_{CC}$ is as follows:

```
EVALPLAN StudentAcademicC_CC {
  SUPER ( );

  RETURN (VOID);
}
```

`SUPER ( );` first invokes the object constructor of class `StudentC` followed by an invocation of the object constructor of class `AcademicC`$_{CC}$. Note: Only class `StudentC` will invoke the object constructor of class `PersonC`.
Assume, we want to create the underlying `PersonC` object through a constructor of class `AcademicC`$_{CC}$. In addition, we pass some arguments. This can be done as follows:

```
EVALPLAN StudentAcademicC_CC ( NameT name, AddressT homeAddr, AddressT workAddr )
{
  SUPER ( AcademicC_CC, name, homeAddr, workAddr );
  SUPER ( StudentC );

  ...
  RETURN ( VOID );
}
```

In a similar manner, methods of super-classes may be invoked. Examples are as follows:

```
  ...
  SUPER ( ).getAddress ( );            // invokes getAddress from class StudentC
  SUPER ( StudentC ).getAddress ( );   // invokes getAddress from class StudentC
  SUPER ( AcademicC_CC ).getAddress ( );        // invokes getAddress from class
                                                               // AcademicC_CC
  SUPER ( PersonC ).getAddress ( );     // invokes getAddress from class PersonC
  ...
```

Thus, by using the SUPER keyword, default mechanism associated with object constructors and multiple inheritance may be overridden. □

## 4.4 Simultaneous Processing

Concurrency and, to some degree, also parallelism are supported by the iDBPQL language. While there are various types of simultaneous processing, e.g. multi-programming, multi-processing, control parallelism, process parallelism, data parallelism, multi-threading, distributed computing etc. (some of which refer to the same processing type), only some of them are supported explicitly, but others are utilised implicitly.

Before considering the supported types of simultaneous processing in greater detail, we will briefly outline some advantages and disadvantages of their support and define their meaning in greater detail.

First, let us consider a more general notion of concurrency, which also covers parallelism. Edsger Dijkstra already defined it as follows: '*Concurrency occurs when two or more execution flows are able to run simultaneously*'. Thus, it encompasses computations that execute overlapped in time, and which may permit the sharing of common resources between those overlapped computations. This results in a number of potential advantages, which include:

– Reduction in run-time, increase in throughput and decrease in response time;
– Increase in reliability through redundancy;
– Potential to reduce duplication in code; and
– Potential to solve more real-world problems than with sequential computations alone.

As promising as these advantages may sound, they do not come without potential drawbacks. The use of shared resources, added synchronisation and communication requirements lead to a number of disadvantages, which include:

- Run-time is not always reduced, i.e. careful planning is required;
- Concurrent computations can be more complex than sequential computations;
- Shared data can be corrupted more easily; and
- Communication between tasks is needed.

As it is already suggested above, simultaneous processing can be achieved in various ways. For instance, processing may take place at multiple nodes where each node utilises its own processing powers or at a single node with a multi-processor machine or even at a single node with only a single-processor machine where CPU time and other resources are shared (i.e. tasks of different programs are interleaved). This leads to a variety of different types of simultaneous processing. Neglecting distribution for a while (which will be considered separately in Section 5.3.10), we may utilise concurrent or parallel processing when:

- Executing multiple programs (i.e. evaluation plans) on a single CPU. This (rudimentary) form of simultaneous processing is commonly referred to as *multi-programming* or *time sharing* and results in interleaved (i.e. concurrent) execution of two or more programs.
- Executing one or more programs by two or more CPUs within a single computer system. This form of simultaneous processing is commonly referred to as *multi-processing* and results in multiple CPUs working on the same program at the same time (i.e. in parallel).

A system can be both multi-processing and multi-programming, only one of the two, or neither of the two.

Multi-programming, in its rudimentary form, allows context switches of programs (e.g. when one program reaches an instruction waiting for a peripheral), but does not give any guarantee that a program will run in a timely manner. When computer usage evolved so did multi-programming. As a result, *time sharing* (i.e. multi-tasking and multi-threading) emerged. This allows the computer system to guarantee each process (or thread[9] in case of multi-threading) a regular 'slice' of operating time.

When multiple programs, processes or threads are present in memory, an ill-behaved execution unit may (inadvertently or deliberately) overwrite memory belonging to another execution unit. Thus, it is important that:

- The memory accessible to the running program or process is restricted; or
- Accesses to the memory accessible to multiple running threads are synchronised.

Execution units that are entirely independent are not difficult to program. Most of the complexity in time sharing systems comes from the need to share computer resources between tasks and to synchronise the operation of co-operating tasks.

Multi-processing can take place in various ways. All CPUs may be equal, or some may be reserved for special purposes. Systems that treat all CPUs equally are called

[9] Threads are basically processes that run in the same memory context. Thus, switching between threads that run in the same memory context, can be achieved more efficiently since it does not involve changing the memory context.

*symmetric multi-processing (SMP) systems*. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including *asymmetric multi-processing (ASMP)*, *non-uniform memory access (NUMA) multi-processing*, and *clustered multi-processing*.

In multi-processing, multiple processors can be used within a single system to execute multiple, independent sequences of instructions in multiple contexts (i.e. *multiple-instruction, multiple-data (MIMD)* processing); a single sequence of instructions in multiple contexts (i.e. *single-instruction, multiple-data (SIMD)* processing); and multiple sequences of instructions in a single context (i.e. *multiple-instruction, single-data (MISD)* processing).

MIMD multi-processing is suitable for a wide variety of tasks in which completely independent and parallel execution of instructions touching different sets of data can be put to productive use. Processing is divided into multiple threads, each with its own state, within a single process or within multiple processes. MIMD does raise issues of deadlock and resource contention. However, threads may collide in their access to resources in an unpredictable way that is difficult to manage efficiently.

SIMD multi-processing is well suited to parallel processing, in which a very large set of data can be divided into parts that are individually subjected to identical but independent operations. A single instruction stream directs the operation of multiple processing units to perform the same manipulations simultaneously on potentially large amounts of data. However, applications must be carefully and specially written to take maximum advantage of this type of multi-processing, and often special optimising compilers designed to produce code specifically for this environment must be used. Some compilers in this category provide special constructs or extensions to allow programmers to directly specify operations to be performed in parallel (e.g. `DO FOR ALL` statements in the version of FORTRAN used on the ILLIAC IV, which was a SIMD multi-processing supercomputer [90]).

MISD multi-processing offers mainly the advantage of redundancy, since multiple processing units perform the same tasks on the same data, reducing the chances of incorrect results if one of the units fails. Apart from the redundant and fail-safe character of this type of multi-processing, it has few advantages, and it is very expensive.

iDBPQL mainly utilises time sharing (i.e. concurrency in terms of multi-tasking and multi-threading) as well as MIMD and SIMD multi-processing (i.e. true parallelism) [68]. While the transaction management system allows different transactions to execute simultaneously (i.e. inter-transaction concurrency or parallelism), iDBPQL also supports two expressions that explicitly request simultaneous execution. The former is discussed in Section 4.4.1. The latter may happen on the transaction-level (i.e. inter-transaction concurrency) or the operation-level (i.e. intra-transaction concurrency). Section 4.4.2 discusses the support of simultaneous processing in greater detail. Finally, individual operations may be implemented in ways that simultaneous processing is utilised (i.e. intra-operation concurrency). However, this form of processing is discussed in detail only in Section 5.3.9.

### 4.4.1 Implicit Inter-Transaction Concurrency

As typical for DBSs, independent transactions are executed simultaneously whenever possible. The two most commonly considered properties, affecting the degree of interleaving, are serialisability (that is conflict-serialisability) and recoverability. This, of course, is also the case for transactions in iDBPQL that stem from different user programs, i.e. originate from different main evaluation plans. The TMS scheduler determines the degree of interleaving of operations of such transactions. Section 5.2.2 will later introduce a corresponding prototype. Transactions that belong to the same main evaluation plan may be executed serially or interleaved. The programmer has a greater influence over the mode of execution. Corresponding details are outlined next.

### 4.4.2 Support for Explicit Concurrency

Inter- and intra-transaction concurrency may be specified explicitly. iDBPQL provides two different control flow expressions that imply concurrency.

On one hand, it can be specified that a block of statements may be executed independently from its surrounding statements (i.e. time sharing or MIMD). Thus, different, independent statements are processed at the same time. When specified within a loop statement, this time sharing or MIMD approach is mixed with SIMD. The general syntax for such a specification is as follows:

**Syntax Snapshot 4.21** ***(Independent iDBPQL Blocks)***

```
IndependentDoBlock = "INDEPENDENT", "DO", Statements, "ENDDO";
```

□

Whether or not independent execution results in multi-tasking, single-threaded or multi-threaded execution is indicated by the compiler and / or decided at execution time.

Example 4.22. Let us consider some examples. First, we consider the simultaneous execution of two independent statements, which belong to the same transaction (i.e. utilising intra-transaction concurrency).

```
01  PUBLIC EVALPLAN ViewProfile ( ) {     // implementation of a persistent method of
02    doSomething;                                      // class University_{CC}.StudentC
03
04    LABEL i1 : INDEPENDENT DO
05      doSomethingIndependently;
06    ENDDO;
07
08    doSomethingElse;
09
10    WAIT i1;    // synchronisation of main execution stream with independent stream
11
12    doMore;
13  }
```

From line 01 to line 04 execution has been serial. In line 04, an independent execution block is declared. Together with this execution block declaration (i.e. INDEPENDENT

DO) a label is specified. This label is used later to synchronise the independent execution stream (i.e. the doSomethingIndependently block) with the main execution stream (i.e. doSomethingElse). Serial execution continues from line 12 onwards.

Secondly, we consider this type of processing when executing two transactions that belong to the same request / evaluation plan (i.e. explicit inter-transaction concurrency).

```
20 {
21   ...                  // consider an evaluation plan using the University schema
22   INDEPENDENT DO TRANSACTION tr1
23     stNumb = LectureC.countStudents ( ) WHERE ( course.cNumb == "157.*" );
24     tr1.commit ( );
25   ENDDO;
26
27   DO TRANSACTION tr2
28     FOR EACH CourseC AS x {
29       selectedPapers.add ( x WHERE x.cNumb == "157.*" );
30     }
31     tr2.commit ( );
32   ENDDO;
33   ...
34 }
```

Transactions tr1 and tr2 are executed concurrently. No synchronisation command has been specified. Thus, both execution streams are only synchronised at the end of the corresponding block, i.e. line 34.

Finally, we will utilise the repetitive simultaneous execution of a block of independent statements on a common data set. In this example, we intend to process all statements of one iteration in a for each loop simultaneously with all statements of the next iteration etc.

```
40   ...
41   FOR EACH StudentC AS x INDEPENDENT DO
42     doSomething;
43   ENDDO;
44   ...
```

The doSomething-block is invoked independently for each object in class StudentC. Thus, we could also write:

```
50   ...
51   LABEL i1: INDEPENDENT DO
52     x = StudentC.first ( );
53     doSomething;
54   ENDDO;
55
56   LABEL i2: INDEPENDENT DO
57     x = StudentC.next ( );
58     doSomething;
59   ENDDO;
60
```

```
LABEL i3: INDEPENDENT DO
  x = StudentC.next ( );
  doSomething;
ENDDO;
...

LABEL in: INDEPENDENT DO
  x = StudentC.last ( );
  doSomething;
ENDDO;

WAIT i1, i2, i3, ..., in;
...
```

□

On the other hand, it can be specified that a block of statements is executed concurrently while preserving the indicated ordering. This type of processing is particularly useful when processing collections. Let us consider an example:

EXAMPLE 4.23. Assume, we have a database that keeps track of student enrolments. Students have to apply for course enrolments. They will be approved into a particular course only if they meet all course pre-requisites. Before the beginning of a new semester we like to execute a routine that automatically approves applications which meet all respective course pre-requisites. Subsequently, we have to contact all students whose applications could not be approved automatically.
This procedure can be done in two subsequent steps. Alternatively, we could utilise concurrency (to be more precise, pipelining). This may be achieved as follows:

```
ForEach student Do in parallel
  Automatically approve all course applications.
Then
  Compile a list of students which have at least one unapproved application.
EndDo
```

Obviously, it is vital that the execution ordering is preserved. □

In iDBPQL, we support this type of execution. The general syntax for such a specification is based on the FOR EACH loop statement:

**Syntax Snapshot 4.22** ***(Concurrent iDBPQL Blocks)***

```
ConcurrentForEachBlock = "FOR EACH", Expression, "CONCURRENT", "DO", Statements,
                           { "THEN", "DO", Statements, "ENDDO", ';' },
                         "ENDDO", ';'
```

□

The evaluation of the expression results in a collection value. Members of this collection are made available to the DO-statement first. Once those values have been processed they are pipelined to the THEN DO-statement. Thus, both statements may execute simultaneous while preserving execution ordering.

EXAMPLE 4.24. Let us consider some examples. First, we will call two methods on a collection of objects. The second method may be invoked on each object on which the first method has been executed successfully, i.e. objects are pipelined from the first invocation statement to the second invocation statement.

```
01    ...
02    FOR EACH myCollection CONCURRENT DO
03      myCollection.sort ( );   // sorts all members in the collection, e.g. a list
04      THEN DO
05        myCollection.eliminateDuplicatesSorted ( );          // based on a sorted
06            // collection, duplicates are eliminated by considering 'neighbouring'
07      ENDDO;                                                  // collection members
08    ENDDO;
09    ...
```

Line 02 indicates the start of a block of statements that is to be executed concurrently. The two statements in lines 03 and 05 operate on the same collections and, thus, may execute concurrently as long as the execution order is preserved per collection object. So, every object that the `myCollection.sort ( );` statement releases (i.e. adds to its associated result queue) is passed on (i.e. pipelined) to the `myCollection.eliminateDuplicates ( );` statement. As a result, the execution of both statements may overlap (in a controlled manner).

Another means of specifying order-preserving, concurrent execution is within a `for`-loop. Within a `for each`-loop the pipelined object is selected explicitly. In addition to the `CONCURRENT DO`, we also have an `INDEPENDENT DO` in this loop.

```
10    ...
11    FOR EACH mydb.SalaryC AS x CONCURRENT DO
12      x.addBonus ( 2,000 );
13      THEN DO
14
15        Label i1: INDEPENDENT DO
16          x.printPaymentSlip ( );
17        ENDDO;
18
19        extMail.add ( x ) WHERE x.hasInHouseMailAddress ( ) == FALSE;
20
21        WAIT i1;
22
23      ENDDO;
24      THEN DO
25        x.salaryProcessed ( date.today ( ), time.now ( ) );
26      ENDDO;
27    ENDDO;
28    ...
```

For each object `x` in the collection-class `mydb.SalaryC`, the `addBonus` method is invoked. Subsequently, this object is pipelined to both the execution unit invoking the `printPaymentSlip` method as well as to the execution unit that maintains a set of salary objects that do not satisfy the boolean `hasInHouseMailAddress` method. Once, both invocations have been completed (refer to the `WAIT` in line 21), we can pipeline the object `x` to the fourth execution unit invoking the `salaryProcessed` method. While both order-preserving examples outlined above only refer to operations within a single transaction, the same concept can be applied across transactions. Let us demonstrate this next:

```
...
DO TRANSACTION tr1
  salCol = mydb.SalaryC;

  DO TRANSACTION tr2
    FOR EACH salCol AS salObj CONCURRENT DO TRANSACTION tr1
      salObj.addBonus ( 2,000 );

      THEN DO TRANSACTION tr2              // implies ordering: tr1 THEN tr2
        salObj.printPaymentSlip ( );
      ENDDO;
    ENDDO;
  ENDDO;
ENDDO;
...
```

Here, transaction `tr1` obtains the `SalaryC` collection and adds all bonifications. After each bonus is added, the objects is handed over to transaction `tr2`, which prints all payment slips. □

### 4.4.3 Implications

Supporting explicit concurrency has implications on other components of the DBMS and on the syntax of iDBPQL itself. We will look at the different types of simultaneous processing and outline their implications:

- Firstly, there is the `INDEPENDENT DO` statement that applies to sequences of iDBPQL statements inside a single transaction or, in the event that no transaction is defined, the entire evaluation plan. Since access to shared data must be from within a transaction, the latter case can be neglected since its evaluation will not involve the transaction management system (only local, non-shared data is accessed). Considering the former case, the simultaneous execution only applies to operations within the same transaction. This, however, will not have implications for other DBMS components, in particular the transaction management system. According to the ACID principle, data consistency has to be preserved by each transaction, when run in isolation, and the programmer is responsible for ensuring this property.
- Secondly, there is the `INDEPENDENT DO` statement that results in the simultaneous execution of multiple transactions originating from the same evaluation plan, i.e. same main execution stream. From the transaction management system's point of view, these transactions are serialised as any other concurrent transactions (i.e. those discussed in Section 4.4.1). However, some syntactical constraints must be observed when using this type of explicit concurrency. Multi-threaded transactions must commit or abort before rejoining the main evaluation plan and synchronisation commands (i.e. `WAIT` statements) must not form cyclic waiting conditions.

– Thirdly, there is the CONCURRENT DO statement that applies to operations on collections either across transactions, within a single transaction or, in case no transaction is defined, the entire evaluation plan. Operations are synchronised implicitly by pipelining objects that the first operation has released to the second operation etc. Thus, concurrent access to the same collection object is enabled while access to the collection members is synchronised. This form of cooperation is different from the modes of processing traditional DBMSs usually perform. As a consequence, the transaction management system must not be only able to distinguish serial and independent (i.e. of operations allowing shared access or operations operating on unrelated objects) execution of operations within the same transaction but also the coordinated execution of possibly conflicting operations on the same collection. Corresponding extensions to transaction models and correctness criteria are proposed in [6]. The suggested transaction model distinguishes between two partial orderings, which are weak order (i.e. data flow takes place through DBS objects) and strong order (i.e. external flow of information between operations or transactions). While the former enables simultaneous processing, the latter implies serial execution. The corresponding correctness criteria, stack-conflict consistency, permits parallel execution of weakly ordered operations given that their serialisation graph is preserved.

While the support of explicit simultaneous execution requires a more sophisticated transaction management system, it offers the potential to significantly increase system performance. For instance, assume that we have two subsequent operations accessing the same collection. This collection may be of a size that does not fit into the available main memory. Serial evaluation is likely to result in two consecutive scans, which degrades system performance. On the contrary, the CONCURRENT DO block enables the evaluation of both operations with one scan.

## 4.5 Examples

EXAMPLE 4.25. First, we will consider a very simple request that does not involve the run-time or DBS metadata catalogues nor does it invoke any other evaluation plan during its execution. The popular '*Hello world!*' example can be formulated in iDBPQL as follows:

```
EVALPLAN HelloWorld ( VOID ) : STRING {
  RETURN ( "Hello world!" );
}
```

The only metadata reference that is associated with this evaluation plan identifies the argument of the RETURN statement as a STRING value. □

While this first example does not have any associated metadata catalogue references, the majority of evaluation plans do. In iDBPQL, one can code requests that do not involve any persistent data, use persistence only to keep track of the state of objects (e.g. a user's profile) or take full advantage of the integrated processing and querying capabilities on non-shared transient and shared persistent objects. While the former two usages correspond to more traditional PL programs, the latter utilises advantages that result from the integration of DBS languages and OOPLs. While the next Example 4.26 only outlines a more complete specification of one particular class of the university application, Example 4.27 contains a few transient requests that access this schema.

EXAMPLE 4.26. As a second example, we outline the behaviour specification of a class definition in a database schema. In fact, we will continue Example 4.17 and refine the definition of class `EnrolmentC`$_{CC}$. Behaviour specifications are added and their corresponding evaluation plans are proposed.
First, let us consider the extended `EnrolmentC`$_{CC}$ class definition:

```
CLASSDEF EnrolmentC_CC {
  STRUCTURE {
    LectureC_CC lecture;
    StudentC   student;
    READONLY EnrolmentT;     // the date value can be viewed from the outside of
  }                        // this class but the same does not hold for modifications

  BEHAVIOUR {
    PRIVATE checkCrsPreRequisites ( VOID ) : BOOLEAN;
    verifyEnrolment ( VOID ) : BOOLEAN;
    EnrolmentC_CC ( LectureC_CC lect, StudentC std );        // object constructor
  }

  CONSTRAINT {
    UNIQUE ( lecture, student );                           // uniqueness constraint
  }
}
```

For each behaviour specification, there exists an associated evaluation plan, which has the same modifiers, name and arguments as the behaviour specification. We will consider corresponding evaluation plans next:

First, let us consider the evaluation plan that describes the implementation on the `verifyEnrolment` method:

```
EVALPLAN verifyEnrolment ( VOID ) : BOOLEAN {
  // check whether or not the student has already completed the
  // corresponding course successfully
  IF ( EXISTS ( RecordC WHERE ( ( THIS.student == student ) &&
       ( THIS.lecture.course == course ) ) )
       ( result.isValueOf ( PassGradesT ) ) ) {
    // the student already has completed the course successfully
    RETURN ( FALSE );
  }

  // check whether or not the student meets all course pre-requisites
  IF ( ! checkCrsPreRequisites ( VOID ) ) {
    // at least one pre-requisite is not met
    RETURN ( FALSE );
  }

36    // all pre-requisites are met
37    RETURN ( TRUE );
38  }
```

While this evaluation plan is associated with a DBS metadata entry, the plan itself has further DBS and run-time metadata entries associated. These are as follows:

**line 23:** `RecordC` → `DBSMetadata.University`$_{CC}$`.RecordC`,
`THIS.student` → `DBSMetadata.University`$_{CC}$`.StudentC`, and
`student` → `DBSMetadata.University`$_{CC}$`.StudentC`;
**line 24:** `THIS.lecture.course` → `DBSMetadata.University`$_{CC}$`.Course` and
`course` → `DBSMetadata.University`$_{CC}$`.Course` and
**line 25:** `result` → `NULLABLE < DBSMetadata.University`$_{CC}$`.GradesT >` and
`PassGradesT` → `DBSMetadata.University`$_{CC}$`.PassGradesT`.

Next, there is the private evaluation plan `checkPrerequisites` that evaluates whether or not the student meets all course pre-requisites:

```
40  PRIVATE EVALPLAN checkPrerequisites ( VOID ) : BOOLEAN {
41    // for each pre-requisite check whether or not the student has a pass
42    // grade
43    FOR EACH THIS.lecture.course.prerequisites AS preRequCrs DO
44      IF ( ! EXISTS ( RecordC WHERE ( ( preRequCrs == course ) &&
45             ( THIS.student == student ) ) )
46             ( result.isValueOf ( PassGradesT ) ) ) {
47        // at least one missing prerequisite
48        RETURN ( FALSE );
49      }
50    ENDDO;
51
52    // all pre-requisites are met
53    RETURN ( TRUE );
54  }
```

Associated metadata entries are as follows:

**line 43:** `THIS.lecture.course.prerequisites` →
`SET < DBSMetadata.University`$_{CC}$`.CourseC >`;
**line 44:** `RecordC` → `DBSMetadata.University`$_{CC}$`.RecordC`,
`preRequCrs` → `DBSMetadata.University`$_{CC}$`.CourseC` and
`course` → `DBSMetadata.University`$_{CC}$`.CourseC`;
**line 45:** `THIS.student` → `DBSMetadata.University`$_{CC}$`.StudentC` and
`student` → `DBSMetadata.University`$_{CC}$`.StudentC`; and
**line 46:** `result` → `NULLABLE < DBSMetadata.University`$_{CC}$`.GradesT >` and
`PassGradesT` → `DBSMetadata.University`$_{CC}$`.PassGradesT`.

Finally, there is the non-default object constructor `EnrolmentC`$_{CC}$, which is implemented as follows:

```
60  EnrolmentC_CC ( LectureC_CC lect, StudentC std ) {
61    date.today ( );
62    lecture = lect;
63    student = std;
64
65    RETURN ( VOID );
66  }
```

Associated metadata entries are as follows:

**line 61:** date $\rightarrow$ NULLABLE < DBSMetadata.University$_{CC}$.DateT >;
**line 62:** lecture $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC and
lect $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC; and
**line 63:** student $\rightarrow$ DBSMetadata.University$_{CC}$.StudentC and
std $\rightarrow$ DBSMetadata.University$_{CC}$.StudentC.

Definitions and implementations that are outlined in this example are persistent and may be shared between applications. The next example will demonstrate this for a few simple (transient) user requests, which will also access data that is held in the University database. □

Example 4.27. Let us continue the previous example. First, a simple request is issued that accesses enrolment objects through its associated class collection.

```
01  EVALPLAN calcEnrolNumb ( LectureC_CC lect ) : NATURAL {
02    // count the number of students enrolled in the given lecture lect
03    numb = EnrolmentC_CC WHERE ( lecture == lect ).student.COUNT ( ) );
04    RETURN ( numb );
05  }
```

Associated metadata entries are as follows:

**line 01:** $\rightarrow$ DBSMetadata.EnrolmentC$_{CC}$ and
numb $\rightarrow$ NAT;
**line 03:** lecture $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC,
lect $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC and
student $\rightarrow$ DBSMetadata.University$_{CC}$.StudentC and
**line 04:** numb $\rightarrow$ NAT.

The first unnamed metadata reference corresponds to a schema import. The numb reference represents a local variable while all other metadata references refer to classes in the imported schema.

The second request creates a transient class, which provides a more tailored service to its users. A simple student manager that is designed to enable students to access their some relevant information more easily is modelled.

```
10  CLASSDEF StudentMgrC {
11    STRUCTURE {
12      University$_{CC}$.StudentC          myDetails;
13      SET < University$_{CC}$.CourseC$_{CC}$ > shortList;
14    }
15    BEHAVIOUR {
16      getSelectedCourses ( VOID ) : SET < University$_{CC}$.CourseC$_{CC}$ >;
17      compileCrsList ( VOID ) : VOID;
18      compileCrsList ( DepartmentC ) : VOID;
19      enrol ( SemesterC sem ) : VOID;
20      StudentMgrC ( University$_{CC}$.StudentC.studentId myId );
21    }
22  }
```

The class definition contains several references into the university database. The object constructor may be implemented as follows:

```
30  EVALPLAN StudentMgrC ( NAT myId ) {                        // object constructor
31    // retrieve the student's StudentC object from the database
32    myDetails = ( StudentC WHERE ( studentId == myId ) ).getValue ( );
33    shortList = THIS.compileCrsList ( VOID );        // compile list of recommended
34  }                                                                   // courses
```

Associated metadata entries are as follows:

**line 30:** → DBSMetadata.University$_{CC}$.EnrolmentC$_{CC}$,
myDetails → DBSMetadata.University$_{CC}$.StudentC and
shortList → SET < DBSMetadata.University$_{CC}$.CourseC$_{CC}$ > and
**line 33:** studentId → NAT and
myId → NAT.

The statement in line 32 must be executed as a transaction since persistent data is accessed. If no such transaction blocks are specified, each statement is executed as an individual transaction. That is, the evaluation plan is automatically transformed into:

```
40  EVALPLAN StudentMgrC ( NAT myId ) {                        // object constructor
41    // retrieve the student's StudentC object from the database
42    DO TRANSACTION tr1
43      myDetails = ( StudentC WHERE ( studentId == myId ) ).getValue ( );
44    ENDDO;
45    shortList = THIS.compileCrsList ( VOID );        // compile list of recommended
46  }                                                                   // courses
```

As a second implementation, we consider the enrol method, which will contain a user-defined transaction block and utilise simultaneous processing:

```
50  EVALPLAN enrol ( SemesterC sem ) : VOID {                     // enrolment method
51    DO TRANSACTION enrolTrans
52      // retrieve all matching lecture objects
53      lects = Lecture WHERE ( ( course IN shortList ) AND ( semester == sem ) );
54
55      FOR EACH lects AS lect CONCURRENTLY DO
56        // enrol into all lectures
57        lect.enrol ( myDetails );
58
59        THEN DO
60          // remove each lectures after enrolment was successful
61          lects.remove ( lect );
62        ENDDO;
63      ENDDO;
64    ENDDO;
65
66    shortList.discard ( VOID );
67  }
```

Associated metadata entries are as follows:

**line 50:** $\rightarrow$ DBSMetadata.University$_{CC}$.EnrolmentC$_{CC}$ and
lects $\rightarrow$ SET < DBSMetadata.University$_{CC}$.LectureC$_{CC}$ >;
**line 53:** lects $\rightarrow$ SET < DBSMetadata.University$_{CC}$.LectureC$_{CC}$ >,
course $\rightarrow$ DBSMetadata.University$_{CC}$.CourseC$_{CC}$,
shortList $\rightarrow$ SET < DBSMetadata.University$_{CC}$.CourseC$_{CC}$ >,
semester $\rightarrow$ DBSMetadata.University$_{CC}$.SemesterC, and
sem $\rightarrow$ DBSMetadata.University$_{CC}$.SemesterC;
**line 55:** lects $\rightarrow$ SET < DBSMetadata.University$_{CC}$.LectureC$_{CC}$ > and
lect $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC$_{CC}$;
**line 57:** lect $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC$_{CC}$ and
myDetails $\rightarrow$ DBSMetadata.University$_{CC}$.StudentC; and
**line 61:** lects $\rightarrow$ SET < DBSMetadata.University$_{CC}$.LectureC$_{CC}$ > and
lect $\rightarrow$ DBSMetadata.University$_{CC}$.LectureC$_{CC}$.

The implementation of the remaining methods is similar to those presented in this chapter. The same application can also be implemented across all three university schema fragments. Actually, this may be more realistic and would leave the student a larger range of available courses to choose from. □

Chapter 5

# On the Implementation of iDBPQL

The implementation of the intermediate-level integrated database programming and querying language iDBPQL is the main concern of this chapter. Section 5.1 outlines the internal representation of metadata catalogues, evaluation plans and annotations associated with evaluation units. Apart from capturing all properties of iDBPQL entities, internal representations have to also support the evaluation of user requests in a concurrent and distributed database environment. Main challenges result from the requirements of efficient run-time evaluation, orthogonal persistence, concurrency and distribution. DBS components that support the evaluation process are introduced in Section 5.2. A persistent object store, a multi-level transaction management system and a remote communication mechanism are proposed. Subsequently, the functionality of these components is utilised. Section 5.3 discusses the processing of evaluation plans in the concurrent and distributed database computing environment. The evaluation follows a similar idea as the SBA approach [131]. However, a more sophisticated run-time environment that significantly enhances the capabilities and performance of the evaluation procedure is proposed. Operational semantics are discussed for the majority of iDBPQL statements and expression. Finally, Section 5.4 concludes this chapter by briefly discussing ways on how to apply code and query optimisation techniques as known from conventional PLs and relational QLs.

## 5.1 Internal Representation of MetaData Catalogues, Objects and Evaluation Plans

In this section, we propose internal representations of entries that are held in metadata catalogues, objects and values and evaluation plans together with their annotations. It is our aim to model these concepts in a way that the evaluation procedure, mappings to and from persistent storage, and distribution of data and/or processing are supported efficiently.

First, we will discuss corresponding challenges in greater detail. Subsequently, we present the internal representation of metadata units using pseudo-structures formulated in a C-like syntax [59]. To underline the difference between iDBPQL syntax and internal representations, we prefix all internal identifiers and names with two underscores '__'. In a similar manner, we then outline how objects and values are depicted. The representation of evaluation plans is most challenging. Such plans have to link all previously introduced concepts and also capture different means of processing. In addition, evaluation plans must be very flexible with respect to how different operators may be combined to allow for an efficient execution. The latter is important particularly for DBSs since query optimisation processes traditionally consider a large number of possible evaluation plans and only select the most appropriate one for execution. Finally, we introduce the variety of annotations that may be associated with evaluation graphs.

### 5.1.1 Challenges

While outlining internal representations of iDBPQL concepts the following challenges have to be met:

- *Capture all properties of metadata entries*: It must be ensured that all properties of pre-defined types, user-defined types, type synonyms, class definitions, and schemata are preserved. This includes the *closed* property as outlined in Section 4.2.5.
- *Find a suitable internal representation of evaluation plans*: Not only is it required to find a suitable representation that can capture all properties of evaluation plans as outlined in Section 4.3, but it must also include provisions for:
  - Specifying different styles of processing. These should include serial, concurrent and distributed execution of (portions of) evaluation plans.
  - Processing statements and expressions efficiently. This includes the possibility to select the most suitable machine instruction from the list of all available implementations.
  - Supporting code and query expression optimisation processes. A certain degree of flexibility and modularity that enable code and query optimisers to consider multiple evaluation plans should be supported.
  - Linking evaluation blocks with entries in DBS and run-time metadata catalogues. This must also include a means of capturing the declaration and initialisation of local variables.

  A graph-like representation is used to provide the necessary degree of flexibility and modularity. Various types of annotations are introduced as a means of linking evaluation plans and metadata entries and specifying additional information utilised during the evaluation process.
- *Efficient mapping to and from persistent storage*: The support of orthogonal persistence implies that every iDBPQL entity may also persist. This should be achieved in a transient manner. Having an in-memory representation that can easily be reflected on persistent storage (and vice versa), will assist with meeting these objectives.
- *Support distributed processing*: Being able to relocate metadata entries and (portions of) evaluation plans as efficiently and effortlessly as possible is desired to minimise corresponding effects during the processing of evaluation plans.

### 5.1.2 MetaData Entries and Associated Information

Metadata entries are either located in the DBS metadata catalogue or the Run-Time metadata catalogue. We consider DBS metadata entries first since they do not refer to run-time entries. The same does not hold vice versa.

DBS metadata entries are internally represented in the following format:

```
__dbsMetaDataCatalogue {       // DBS metadata catalogue as collection of schemata
  long         __schemaCount;                               // number of schemata
  __schemaInfo __schemata[__schemaCount];             // array of schema information
}
```

Each value in the `__schemata` array must be a `__schemaInfo` structure that provides a complete description of a schema in the DBS metadata catalogue:

```
__schemaInfo {            // a schema is a collection of type and class definitions
  char *         __name;                               // (simple) valid schema name
  long           __typeSynCount;               // number and array of type synonym ...
  __typeSynInfo __typeSyn[__typeSynCount];               // ... declaration information
  long           __typeCount;                          // number of type definitions
  __typeInfo     __types[__typeCount];     // array of type definition information
  long           __classCount;                       // number and array of class ...
  __classInfo    __classes[__classCount];                   // definition information
  __dag          __isaRelation;         // inheritance relation (introduced below)
}
```

`__name` uniquely identifies the schema. Each value in the `__typeSyn` array must be a `__typeSynInfo` (refer below), which provides a complete description of the type synonym declaration. Each value in the `__types` array must be a `__typeInfo` structure (refer below), which describes the type definition. Accordingly, a `__classes` array value must be a `__classInfo` structure (refer below), which provides a complete description of the class definition.

In contrast to DBS metadata entries, run-time metadata entries are not grouped explicitly. Instead, they are associated with evaluation blocks. This association defines the visibility (i.e. scope) of the particular metadata entry. Thus, it is only natural to organise run-time metadata entries according to their association, which results in the following internal format:

```
__rtMetaDataCatalogue {          // the Run-Time metadata catalogue as collection
                                           // of run-time scope extension entries
  long           __rtEntryCount;               // number and array main run-time ...
  __rtEntryInfo __rtEntries[__rtEntryCount];                // ... entry information
}
```

Each value in the `__rtEntries` array must be a `__rtEntryInfo` structure that describes run-time entries. These run-time entries in the `__rtMetaDataCatalogue` are associated with evaluation blocks. The corresponding `__rtEntryInfo` structure is defined as follows:

```
__rtEntryInfo {                         // run-time entries as collection of type and
                                          // class definitions, and local symbols
  long           __typeSynCount;               // number and array of type synonym ...
  __typeSynInfo __typeSyn[__typeSynCount];               // ... declaration information
  long           __typeCount;                          // number of type definitions
  __typeInfo     __types[__typeCount];     // array of type definition information
  long           __classCount;                       // number and array of class ...
  __classInfo    __classes[__classCount];               // ... definition information
  long          __symbCount;                        // number of local symbols
  __symbInfo    __symbols[__symbCount];              // array of local symbols
  __dag *       __isaRelation;         // inheritance relation (introduced below)
}
```

Analogous to the `__schemaInfo` structure, each value in the `__typeSyn` array must be a `__typeSynInfo` (refer below), which provides a complete description of the type synonym declaration. Each value in the `__types` array must be a `__typeInfo` structure (refer below), which describes the type definition. Accordingly, a `__classes` array value must be a `__classInfo` structure (refer below), which describes the class definition. In addition, each value in the `__localSymbols` array must be a `__symbInfo` structure that completely describes the particular local symbol. A local symbol corresponds to a variable or constant declaration. At the beginning of the evaluation of a particular block, the block's local symbols are loaded (onto the environment stack, which we will only introduce in Section 5.3) and initialised. This results in local symbols being in the innermost scope. The internal structure of collections of local symbols is as follows:

```
__symbInfo {
  char *       __name;                       // (simple) valid name of the symbol
  __descriptor __symbolDescriptor;                     // valid symbol descriptor
  long         __attribCount;                 // number and array of associated ...
  __attribInfo __attributes[__attribCount];        // ... attributes, e.g. CONSTANT
}
```

`__name` uniquely identifies the symbol within the particular evaluation block.

**Representing Type Information.** Information about types, which are defined in the DBS metadata catalogue or at run-time, is represented internally in two different structures. Firstly, there is the structure capturing type synonyms. Its format is as follows:

```
__typeSynInfo {
  byte         __modFlag;                  // modifier flags as outlined in Table 5.1
  char *       __name;                           // (simple) valid type synonym name
  __descriptor __typeSynDescriptor;               // valid type synonym descriptor
}
```

`__name` uniquely identifier the type synonym in the respective scope. Details about the `__descriptor` structure are discussed further below.

Secondly, there are type definitions that are represented internally in the following format:

```
__typeInfo {
  byte         __modFlag;                  // modifier flags as outlined in Table 5.1
  char *       __name;                                 // (simple) valid type name
  __descriptor __typeDescriptor;                 // valid type (parameter) descriptor
  long         __fieldCount;                           // number of type variables
  __fieldInfo  __fields[__fieldCount];                  // array of type variables
  long         __typeOpCount;                         // number of type operations
  __typeOpInfo __typeOps[__typeOpCount];                // array of type operations
}
```

__name and __typeDescriptor uniquely identify a type. Each value in the __fields array must be a __fieldInfo structure that provides a complete description of a type variable. No two fields in a type may have the same __name and __fieldDescriptor. The format of the __fieldInfo structure is as follows:

```
__fieldInfo {
  byte         __modFlag;                    // modifier flags as outlined in Table 5.1
  char *       __name;                              // (simple) valid variable name
  __descriptor __varDescriptor;                        // valid variable descriptor
  long         __attribCount;                  // number and array of associated ...
  __attribInfo __attributes[__attribCount];         // ... attributes, e.g. CONSTANT
}
```

Details about the __descriptor and __attribInfo structures are discussed further below.

| Value Mask | Applies To | Interpretation |
|---|---|---|
| $x$ $y$ 0 | __typeSynInfo, __typeInfo, __fieldInfo __typeOpInfo, __classInfo, __methodInfo | PUBLIC modifier |
| $x$ $y$ 1 | __fieldInfo | READONLY modifier |
| $x$ $y$ 2 | __typeSynInfo, __typeInfo, __fieldInfo __typeOpInfo, __classInfo, __methodInfo | PRIVATE modifier |
| $x$ 1 $z$ | __classInfo, __methodInfo | ABSTRACT modifier |
| $x$ 2 $z$ | __classInfo | CONCRETE modifier |
| $x$ 4 $z$ | __classInfo | COLLECTION modifier |
| 1 $y$ $z$ | __classInfo, __methodInfo | FINAL modifier |
| 2 $y$ $z$ | __fieldInfo, __methodInfo | STATIC modifier |
| 4 $y$ $z$ | __methodInfo | FINAL STATIC modifiers |

Where $x \in \{ 0, 1, 2, 4 \}$, $y \in \{ 0, 1, 2, 4 \}$, and $z \in \{ 0, 1, 2 \}$.

**Table5.1.** Modifier Flags and Their Interpretation.

In addition to the __fields array, the __typeInfo structure also contains the __typeOps array. Its values must be __typeOpInfo structures, which provide complete descriptions of each of the corresponding type operations. No two operations in a type may have the same __name and __typeOpDescriptor. The format of the __typeOpInfo structure is as follows:

```
__typeOpInfo {
  byte         __modFlag;                    // modifier flags as outlined in Table 5.1
  char *       __name;                          // (simple) valid type operation name
  __descriptor __typeOpDescriptor;                  // valid type operation descriptor
  long         __attribCount;                  // number and array of associated ...
  __attribInfo __attributes[__attribCount];    // ... attributes, e.g. INITIALISER
}                                                                     and EVALPLAN
```

**Representing Class Information.** Similarly, information about classes, defined in the DBS metadata catalogue or at run-time, is represented internally in the following format:

```
__classInfo {
  byte               __modFlag;         // modifier flags as outlined in Table 5.1
  char *             __name;                         // (simple) valid class name
  __descriptor       __classDescriptor;     // valid class (parameter) descriptor
  long               __supClassCount;                    // number and array of ...
  __classInfo        __supClasses[__supClassCount];    // ... direct super-classes
  long               __fieldCount;                       // number and array of ...
  __fieldInfo        __fields[__fieldCount];    // ... instance and class variables
  long               __methodCount;                          // number of methods
  __methodInfo       __methods[__methodCount];                // array of methods
  long               __constrCount;                      // number and array of ...
  __classConstrInfo __classConstrs[__constrCount]; // ... class-level constraints
}
```

Each value in the __fields array, __methods array or __classConstrs array must be a __fieldInfo structure (refer above), __methodInfo structure (refer below) or __constrInfo structure (refer below) respectively. The corresponding structure provides a complete description of an instance / class variable, method or class constraint. Considering instance and class variables first, no two field entries may have the same __name and __fieldDescriptor in a class. Internally, the __fieldInfo structure is used to represent structural members of both types and classes. The only difference is that fields, which belong to a class definition, may be declared STATIC.

In addition to the NOT NULL constraint, class definitions may also contain UNIQUE and CHECK class-level constraints. To capture these properties, the __classInfo structure contains a __classConstrs array. Its values must be __classConstrInfo structures, which provide complete descriptions of the corresponding collection of class-level constraints. The format of the __classConstrInfo structure is as follows:

```
__classConstrInfo {
  char *        __name;                   // (simple) valid constraint name or NULL
  long          __constrCount;           // number and array of NOT NULL, UNIQUE ...
  __constrInfo __constraints[__constrCount];          // ... and CHECK constraints
}
```

__name uniquely identifies the class-level constraint within the class definition. There may be one class-level constraint without a name. Each value in the __constraints array must be a __constrInfo structure that provides a complete description of the collection of NOT NULL, UNIQUE and CHECK constraints for a particular set of class-level constraints.

```
__constrInfo {
  char          __type;         // either 0 ≅ NOT NULL, 1 ≅ UNIQUE, or 2 ≅ CHECK
  long          __fieldCount;                // number of fields in the constraint
  __fieldInfo * __fields[__fieldCount];        // array of pointers to the fields
                                                    // that the constraint affects
  long          __attribCount;            // number and array of associated ...
  __attribInfo  __attributes[__attribCount];      // ... attributes, e.g. EVALPLAN
}
```

Besides static class members and class-level constraints, behaviour specifications have to be captured. As usual, no two class methods may have the same __name and __methodDescriptor. The format of the __methodInfo structure is as follows:

```
__methodInfo {
  byte          __modFlag;              // modifier flags as outlined in Table 5.1
  char *        __name;                                // (simple) valid method name
  __descriptor __methodDescriptor;                        // valid method descriptor
  long          __attribCount;                // number and array of associated ...
  __attribInfo __attributes[__attribCount];   // ... attributes, e.g. CONSTRUCTOR
}                                                                  // and EVALPLAN
```

**Representing Descriptors.** A descriptor is a String representation of the type in a type synonym definition, a constraint or unconstrained type / class parameter specification, the type of a variable, or the signature of a behaviour specification.

Descriptors for type synonyms and variables have the same format. The type of the synonym type or variable is encoded in internal form.

Behaviour signatures (i.e. `__typeOpDescriptor` from structure `__typeOpInfo` and `__methodDescriptor` from structure `__methodInfo`), on the other hand, are of the following format:

`( __parameterDescriptor[] ) __returnDescriptor`

where `__parameterDescriptor[]` represents a possibly empty array of parameters passed to a type operation or method (using the same format as for variable descriptors) and `__returnDescriptor` represents the type of the corresponding value that is returned. In the event that the descriptor represents the signature of an object constructor, there is no `__returnDescriptor` present.

A type or class parameter has the following format:

`( __parameterDescriptor[] ) __constraintDescriptor[]`

where `__parameterDescriptor[]` represents an array of types or classes (using the same format as for variable descriptors) and `__constraintDescriptor[]` represents a possibly empty array of types or classes that constraint the parameter (again, using the same format as for variable descriptors). If the latter array is empty, we have an unconstrained type parameter.

**Representing Other Attributes.** Additional attributes may be associated with the following structures: `__symbInfo`, `__fieldInfo`, `__typeOpInfo`, `__constrInfo`, and `__methodInfo`. The format of the `__attribInfo` structure, together with all pre-defined attribute types, are as follows:

```
__attribInfo {
  enum           __attribType;         // pre-defined values are: CHECK, CONSTANT,
                                // CONSTRUCTOR, EVALPLAN, INITIALISER, and NOT NULL
  char *         __name;                          // optional (simple) valid name
  __iDBPQLvalue __value;                                // optional iDBPQL value
  __evalPlan *  __code;                // optional reference to an evaluation plan
}
```

Whether or not `__name`, `__value` and / or `__code` are used depends on the corresponding attribute type. The `NOT NULL` attribute type represents the variable-level

`NOT NULL` constraint. This attribute type does not use any of the three optional structure members. The type `CONSTANT` only has an associated constant value. All other pre-defined types use the `__code` member, which refers to the evaluation plan that implements the corresponding behaviour. This behaviour corresponds to a user-defined method, a type operation, a user-defined constructor, the default constructor, a user-defined type initialiser, or the default type initialiser.

In a future release of iDBPQL, an `EXCEPTION` attribute will be added. The Java Virtual Machine [80] uses a similar, less modular, but more complex internal representation. It already demonstrates how exceptions can be supported.

**Inheritance Relations.** While inheritance relations are already given implicitly through the `__classInfo` structures, there is also a corresponding in-memory graph, a DAG, that is maintained to access sub- and super-class information more efficiently. With each DBS metadata entry, there is an associated `__isaRelation` structure member of type `__dag`. The internal structure of this directed graph is not of particular interest. Instead, we only require the following operations to be defined on this implicitly maintained graph:

```
__classInfo[] * getSubClasses ( __classInfo * class );
                       // returns an array of all  direct sub-classes of class class
__classInfo[] * getSuperClasses ( __classInfo * class );
                      // returns an array of all direct super-classes of class  class
boolean isSubClassOf ( __classInfo * class1, __classInfo * class2 );
                        // tests whether class class1 is a sub-class of class class2
boolean isSuperClassOf ( __classInfo * class1, __classInfo * class2 );
                      // tests whether class class1 is a super-class of class class2
```

During run-time, a corresponding DAG is maintained in-memory and associated with the corresponding run-time metadata entries. It is initialised before the processing of a request's main evaluation plan commences. Subsequently, the DAG is associated with any run-time metadata entry that is encountered during the request's evaluation. If a run-time metadata entry has new class definitions associated, the DAG is updated accordingly. Updates result in adding new leaves or `UNION`-types. Similarly, as the execution of an evaluation plan terminates, class definitions that are local to this block are removed from the run-time DAG. Such updates result in pruning operations or removal of `UNION`-types.

### 5.1.3 The Representation of Objects and Values

In addition to metadata entries, we have to specify how objects and values are represented. While we outline the internal representation of objects, the representation of values is not dictated. Instead, it is implementation dependent. Most likely, it is influenced by the underlying persistent object store (refer to Section 5.2.1). While an abstract notation would be sufficient, we, however, will consider a more physical representation. By doing so, we can demonstrate more easily how the evaluation component is linked to an underlying object store. Before we consider values in greater detail, the internal representation of objects is introduced:

```
__object {
  __OID            __oid;   // the object's unique and immutable object identifier
  __classInfo *  __class;         // reference to a __classInfo structure held in
          // either the run-time metadata catalogue or the DBS metadata catalogue
  char *           __name;                                      // an external name
  __iDBPQL_value __value;     // the object's value; its structure is determined
                                           // by the object's associated class definition
```

An object's unique, internal identifier is assigned by the persistent object store (refer to Section 5.2.1) if the object is created on a class that resides in the DBS metadata catalogue. Otherwise, the run-time environment (i.e. the REE component) will assign an identifier. The rationale behind this approach is simply a more effective means of using the available pool of OIDs. Objects that are not made persistent have a relatively short life-span – so do their OIDs. While it would be desirable (from a theoretical point of view) not to re-use an OID, the size of the OID pool, however, is always restricted in practical systems (e.g. by the number of bytes reserved for the internal representation of the type `__OID`). Thus, we will be able to release previously allocated (in-memory) OIDs when it is safe to do so.

An `__iDBPQL_value` is either an atomic iDBPQL value or a complex iDBPQL value. In the case of the latter, object references may be included. They are represented persistently using the `__OID` type. In main memory, references by OID are replaced by in-memory pointers to speed up object access. Corresponding pointer swizzling techniques have been detailed in, among others, [57].

With respect to values, we only require an additional system routine that determines the value's current type. The signature of such a routine is defined as follows:

```
( __typeInfo || __classInfo ) * typeOf ( __iDBPQLvalue val );
```

The `typeOf` routine returns a pointer to the value's `__typeInfo` structure if `val` holds a simple, structured, collection-type, or `NULL`able value. Otherwise, if we deal with a reference-type value, a pointer to the referenced object's associated `__classInfo` structure is returned.

In order to relate values to their types, a possible internal representation may be a pair that consists of an `__iDBPQL_value` value and a pointer to its corresponding run-time type.

EXAMPLE 5.1. Let us consider a fragment of a user request importing all classes of the University$_{CC}$ schema as outlined in Example 4.17. The relevant fragment is as follows:

```
...
NEW StudentC ( ( [ "Mr." ], "Robin", "Steward" ), ( ( "Main Street", "50A" ),
                "Palmerston North", 4412 ), DepartmentC_CC WHERE
                ( dName == "Department of Information Systems" ), NULL, NULL );
...
p = (PersonC) PersonC WHERE (
      ( name.( lastname, firstname ) == ( "Steward", "Robin" ) ) );
...
```

After executing the object creation statement in lines 02 to 04, the corresponding internal object is represented as follows (variable names in the form of cast information have been added to ease readability):

```
10 __oid      = (__OID) 87
11 __class    = (__classInfo *) DBSMetadata.UniversityCC.StudentC
12 __name     = "StudentC"
13 __value    = (personId) 433, (name) ( [ "Mr." ], "Robin", "Steward" ),
14              (addr) ( ( "Main Street", "50A" ), "Palmerston North", 4412 ),
15              (studentID) 65978462, (major __OID) 3, (minor) NULL,
16              (supervisor) NULL );
```

The `(major __OID)` 3 entry from line 15 implies that the value of structure member `major` is a reference value, which is represented as an object identifier of value 3. Subsequently, the selection statement in lines 06 and 07 is evaluated. As a result, a collection of type SET ¡ PersonC ¿ is returned. It is likely to contain only one value: `(__OID) 87` or its corresponding main memory representation. □

The example above already indicates how inherited class members are added into the object structure. Let $A$ and $B$ be distinct super-classes of $C$ with `CLASSDEF` $C$ `IsA` $A$, $B$. Class members inherited from $A$ appear first, followed by class members of $B$ and, finally, followed by all local class members.

In the presence of renaming expressions, prioritisation clauses or identical inherited features, the internal representation is not as straightforward. For instance, let us revisit Example 4.12 (on page 86). Here, instances of the `StudentC`, `AcademicC`$_{CC}$ and `StudentAcademicC`$_{CC}$ classes have the internal structure as outlined in Table 5.2.

class `StudentC`

| | |
|---|---|
| `PersonT` | `__iDBPQLvalue` |
| `id` | `__iDBPQLvalue` |
| `email` | `__iDBPQLvalue` |
| `policyId` | `__iDBPQLvalue` |
| `StudentT` | `__iDBPQLvalue` |
| `campusAddr` | `__iDBPQLvalue` |
| `getAddress ( )` | `__methodInfo` |

class `AcademicC`$_{CC}$

| | |
|---|---|
| `PersonT` | `__iDBPQLvalue` |
| `id` | `__iDBPQLvalue` |
| `email` | `__iDBPQLvalue` |
| `policyId` | `__iDBPQLvalue` |
| `workAddr` | `__iDBPQLvalue` |
| `getAddress ( )` | `__methodInfo` |

class `StudentAcademicC`$_{CC}$

| | |
|---|---|
| `PersonT` | `__iDBPQLvalue` |
| `id` | `__iDBPQLvalue` |
| `email` | `__iDBPQLvalue` |
| `policyId` | `__iDBPQLvalue` |
| `StudentT` | `__iDBPQLvalue` |
| `campusAddr` | `__iDBPQLvalue` |
| `PersonT` | *ptr to the* `PersonT` *value above* |
| `staffId` | `__iDBPQLvalue` |
| `staffEmail` | `__iDBPQLvalue` |
| `policyId` | *ptr to the* `policyId` *value above* |
| `workAddr` | `__iDBPQLvalue` |
| `getAddress ( )` | `__methodInfo` (*from AcademicC*$_{CC}$) |

**Table5.2.** Internal Object Structure of Instances of Classes Presented in Example 4.12.

Considering Table 5.2, it should be evident how the internal object structure reflects programmer's decisions dealing with ambiguities that arise in the presence of multiple inheritance.

### 5.1.4 The Representation of Evaluation Plans

An evaluation plan can be regarded as a quadruple consisting of modifiers, a flag indicating the type of the evaluation plan, an external name providing a unique identifier, and an evaluation graph. In terms of our structure-like notation, an evaluation plan has the following format:

```
__evalPlan {
  byte        __modFlag                   // modifier flags as outlined in Table 5.1
  char        __type;       // pre-defined values are: M ≅ main evaluation plan,
                                        T ≅ type operation, I ≅ type initialiser,
                                          F ≅ method, and C ≅ object constructor
  char *      __name;                                  // valid (qualified) name
  __evalGraph __evaluation;       // evaluation graph as defined in Definition 5.1
}
```

The value of `__evaluation` must be an `__evalGraph` structure, which provides a complete description of the evaluation of the respective behaviour. The `__evalGraph` structure is specified using a more visual graph representation. This will not only reduce the complexity of subsequent examples, but will also allow for an easier understanding of how such behaviour implementations are evaluated and linked to other internal representations (as introduced previously in this section).

**Definition 5.1.** An *evaluation graph* `__evalGraph` is a quadruple of the form ( *ROOTnode*, { *EVALnode* }, { *subEVALedge* }, { *ctrlFLOWedge* } ), where:

- *ROOTnode* is the root node of the evaluation graph from which all processing commences.
- { *EVALnode* } is a set of evaluation plan nodes. Each of which describes the proposed implementation of a statement or expression of the iDBPQL language. Evaluation nodes may have one or more internal handles $h_1, ..., h_n$, a set of rules $r_1, ..., r_m$ governing the relationships among the handles and various annotations $an_1, ..., an_k$ including processing annotations, location annotations and metadata references. Handles, rules and annotations are defined as follows:
  - A handle $h_i$ associates another evaluation node $n$ with the current node. The associated node $n$ may either precede the local evaluation (i.e. evaluate a sub-expression) or succeed the local evaluation. Evaluation nodes associated with preceding handles must be processed while nodes associated with succeeding handles may be processed. The set of rules $r_1, ..., r_m$ determines whether or not a particular succeeding evaluation is carried out.
  - A rule $r_i$ links one or more preceding handles with one or more succeeding handles. The following rules are supported[1]:
    - **Rule 1:** $r_i$: $h_{j_1}$, ..., $h_{j_r}$ **then** $h_{j_s}$ where $h_{j_s}$ must not be a preceding node. $h_{j_1}$, ..., $h_{j_r}$ must evaluate first before the evaluation of $h_{j_s}$ commences. This first rule models serial evaluation.

[1] Assigning rules to evaluation nodes is an intermediate step of the optimisation process. It enables the capturing of the flow of result values more easily and assists with the decision of which intermediate result values are pipelined, returned at once or even materialised. In addition, the degree of multi-threading is determined during this step. Subsequently, machine codes are assigned that support the determined result passing and processing characteristics.

  * **Rule 2:** $r_i$: $h_{j_1}$, ..., $h_{j_r}$ `then` $h_{j_s}$ `else` $h_{j_t}$ where $h_{j_s}$ and $h_{j_t}$ must not be preceding nodes. As a result, only the node associated with $h_{j_s}$ or the node associated with $h_{j_t}$ is evaluated, but never both. This second rule models conditional, serial evaluation.
  * **Rule 3:** $r_i$: $h_{j_1}$, ..., $h_{j_r}$ `pipe` $h_{j_s}$ where $h_{j_s}$ must not be a preceding node. $h_{j_1}$, ..., $h_{j_r}$ start evaluating first. As results become available they are pipelined to $h_{j_s}$. Once $h_{j_s}$ has sufficient input values, it commences its evaluation while $h_{j_1}$, ..., $h_{j_r}$ continue to forward further results. This third rule models concurrent evaluation that is synchronised using pipelines.
  * **Rule 4:** $r_i$: $h_{j_1}$, ..., $h_{j_r}$ `pipe` $h_{j_s}$ `else pipe` $h_{j_t}$ where $h_{j_s}$ and $h_{j_t}$ must not be preceding nodes. This rule combines the second and third rule and models conditional, concurrent evaluation that is synchronised using pipelines.
  * **Rule 5:** $r_i$: $h_{j_1}$ `&&` ... `&&` $h_{j_r}$. This rule overrides the serial processing mode of $h_{j_1}$, ..., $h_{j_r}$. Instead, these handles will be invoked simultaneously, i.e. sparking new evaluation threads. The evaluation of $h_{j_1}$, ..., $h_{j_r}$ is considered successful once all $h_{j_x}$ have been terminated.
  * **Rule 6:** $r_i$: $h_{j_1}$ `||` ... `||` $h_{j_r}$. This rule overrides the serial processing mode of $h_{j_1}$, ..., $h_{j_r}$. Instead, these handles will be invoked simultaneously, i.e. sparking new evaluation threads. The evaluation of $h_{j_1}$, ..., $h_{j_r}$ is considered successful as soon as one $h_{j_x}$ has been terminated.

- An annotation $an_i$ is either a processing annotation, a location annotation, a metadata reference, a machine instruction or a label annotation annotation as discussed in Section 5.1.5.

Per default, all handles are treated as preceding handles. This only changes in case the handle appears on the right-hand side of a rule. Then, the handle is promoted to a succeeding handle. No handle can be both preceding and succeeding.
If an evaluation node has no handle defined, it represents a terminal node (i.e. a leave) in the evaluation graph.

– { *subEVALedge* } is a set of bidirectional evaluation plan edges. Each edge connects an *EVALnode*'s preceding handle with another *EVALnode*. This edge represents a parent-child relationship where the child node assists with the implementation of the parent node (i.e. a sub-evaluation is described). The parent ensures that all necessary variables are in scope while the child returns the results of its execution. The *subEVALedge* may have processing annotation in forward direction informing the child how to return result values. In backward direction, there must be an annotation in the form of a `__returnDescriptor`.

– { *ctrlFLOWedge* } is a set of directed evaluation plan edges. Each edge connects the *ROOTnode* with an *EVALnode* or two *EVALnode*s. The edge assists with controlling the flow of the evaluation.
An evaluation graph may have multiple control flows that are evaluated simultaneously. *ctrlFLOWedge*s are also used to synchronise such control flows when applicable.

□

Let us consider an initial example of the internal representation of an evaluation graph.

EXAMPLE 5.2. We will revisit Example 4.26. Figure 5.1 depicts an evaluation graph that corresponds to the `verifyEnrolment ( VOID )` evaluation plan.

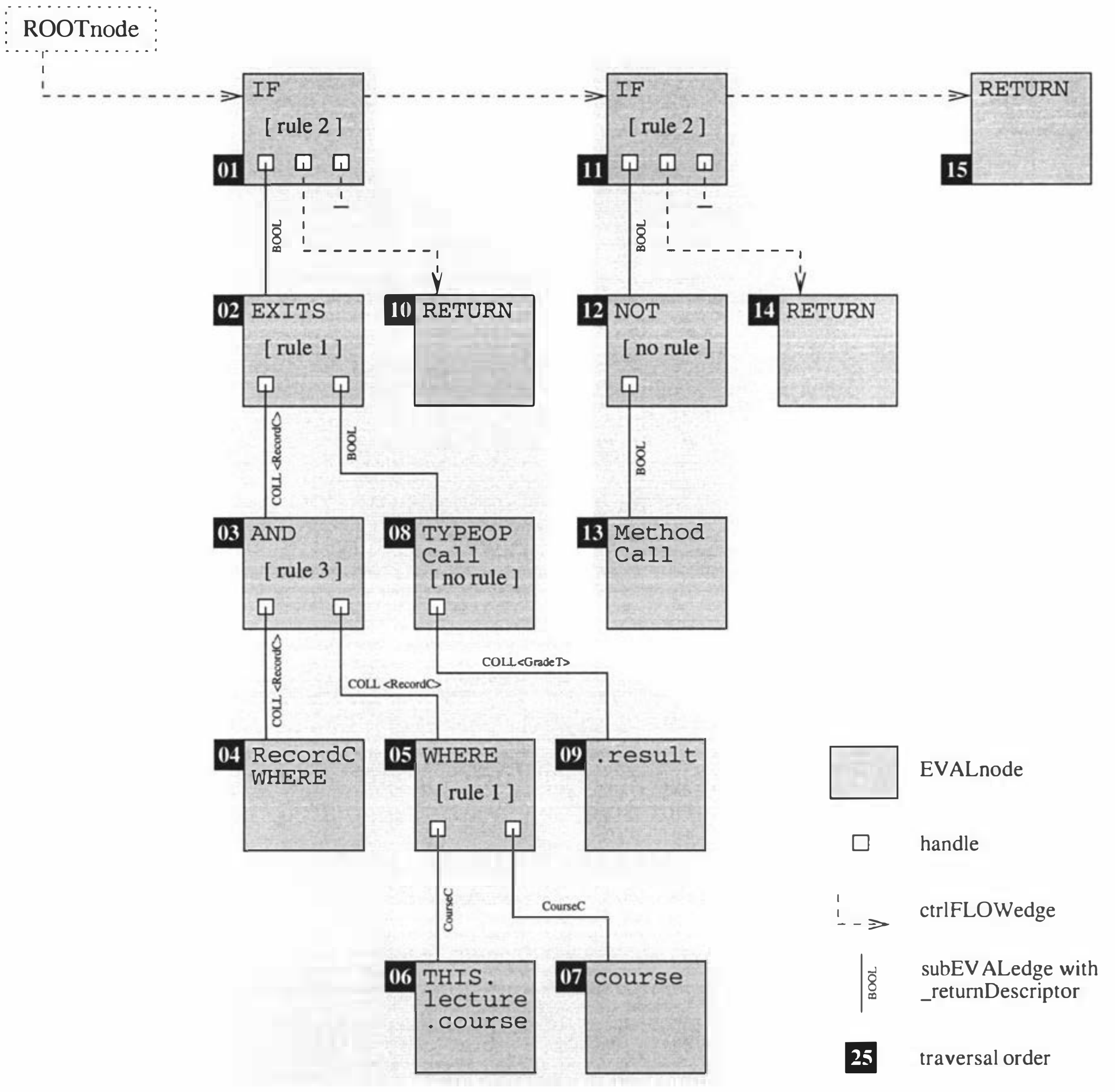


**Fig. 5.1.** Sample Evaluation Graph for the `verifyEnrolment` Method.

The evaluation of the first `IF` ... `THEN` statement is described by ten evaluation nodes. Five of these nodes (i.e. nodes with traversal orderings 04, 06, 07, 09, and 10) correspond to leave nodes. Such nodes do not have handles and rules associated. Non-leave nodes, i.e. nodes with at least one sub-evaluation node, have their corresponding rules, handles and outgoing evaluation and control flow edges associated. While rules 1 and 2 only indicate serial processing, the node with traversal ordering 03 utilises the third rule, which implies multi-threaded processing and pipelining. The second `IF` ... `THEN` statement is less complex to describe. While sub-evaluation edges have their expected return types associated, additional annotations are outlined next. □

### 5.1.5 Overview of Annotations

Nodes and edges of evaluation plans may have one or more of the following types of annotations attached: *Metadata references*, *processing annotations*, *descriptor annotations*, *machine instruction annotations*, *location annotations*, and *label annotations.*

Metadata references are associated with *ctrlFLOWedges*. A *ctrlFLOWedge* that is attached to the *ROOTnode* or a handle of an *EVALnode*, which opens a new evaluation block, may have an annotation referring to a `__rtEntryInfo` structure. The structure contains type synonyms, type definitions, class definitions and a list of all declarations that are local to the evaluation block. These declarations are used later to initialise the local environment on the respective frame on the evaluation stack before subsequent statements and expressions are processed.

Each *subEVALedge* has the following associated annotations:

- In forward direction, a processing annotation that informs the child about the means by which to return result values may be associated. Per default, results are returned at once after the evaluation of the child has been terminated. Alternatively, pipelining may be requested. In the latter event, a result queue will be created enabling the child to push (blocks of) intermediate result values or object references to the parent as they are computed.
- In backward direction, there must be an annotation in the form of a `__returnDescriptor` that outlines the type of the expected return value.

An *EVALnode* may have a machine instruction annotation. The implementation of iDBPQL offers a number of alternatives for most supported operators. During the optimisation of evaluation plans, the most suitable machine instruction is associated with an evaluation node.

In addition to metadata references, *ctrlFLOWedges* may also have a processing annotation associated. Per default, i.e. in the event that no such annotation is present, processing proceeds within the same execution unit (i.e. thread) in serial manner. Alternatively, one of the following processing annotations may be specified:

- *Serial*, which is the default as mentioned above.
- *Multi-threaded*, which splits the current execution stream into two or more concurrent execution streams, which are evaluated on the same processing unit (i.e. different threads but same process).
- *Distributed*, which implies a relocation of the evaluation to a remote ODBS instance. If this is the case, there will be another annotation, a location annotation, associated with the edge.

A location annotation is associated with *ctrlFLOWedges* if the processing of parts of the evaluation plan is distributed to a remote node. In addition, the local evaluation may require access to values or objects that are stored remotely. In such an event, a location annotation is attached to the corresponding metadata reference. A location annotation consists of a reference to the ODBS instance, which is supposed to continue with the processing of a portion of the evaluation plan, or holds the desired data entity.

Finally, label annotations are associated with evaluation nodes. Per default, the label annotation is `NULL`. However, if a `LABEL` statement is encountered, the corresponding identifier is attached to the respective node as label statement. Subsequently, the label specification may be removed from the evaluation plan.

## 5.2 Interface Definitions of Related Database System Components

Before we discuss the evaluation procedure in more detail, we still have to define how the evaluation component (i.e REE) interacts with other DBMS components. In this section, we will introduce the general functionality and service interfaces of corresponding DBS prototypes that are required during the implementation of the proposed iDBPQL language. Prototypes include:

- A persistent object store [73] that supports storage, access and maintenance of persistent database objects. Section 5.2.1 will discuss the corresponding prototype in more detail.
- A multi-level transaction management system, which ensures that the evaluation of user requests is (conflict-)serialisable and recoverable. Section 5.2.2 will discuss the corresponding prototype in more detail.
- A remote communication mechanism, which is based on the agent communication language DBACL [67]. Section 5.2.3 will discuss the corresponding prototype in greater detail.

All prototypes are implemented in the programming language C [59]. References to more detailed documentations are included in the corresponding sections.

### 5.2.1 A Persistent Object Store

Object stores are primarily popular for their support of persistence and different types of data access. Atkinson et al. [10] discusses relevant concepts and issues that arise with respect to object stores, DBSs, DBPLs and persistence.

Research work that has influenced our proposal include the HiPOS system [146]. Researchers have also focused on object stores that support ODBMSs and persistent programming languages. An abstract object storage model consisting of a pair of sets $(O, R)$ where $O$ is a set of objects and $R$ is a set of references between objects is proposed. Our research extends this model by introducing indices and different types of references into the object model. Accordingly, the object store architecture, internal concepts and also the service interface differ between both approaches.

The Persistent Objects Store (POS) maintains storage objects and provides a service interface that enables access to and storage of these objects to higher-level DBS components.

A storage object consists of a list of attributes with a globally unique storage object identifier. Storage objects are classified internally. We distinguish between collections and regular storage objects. Collections are introduced to enable higher-level modules to classify storage objects according to their (logical) structure. In addition to using collections to related storage objects and attributes referring to other storage objects (what we call embedded references), explicit references from one storage object to another storage object can be specified. Explicit references between two storage objects are not stored with the storage objects themselves, but in additional structures, e.g. used for navigation between storage objects or access through indices. These references can be added to relate storage objects that are commonly accessed together. Considering ODBMS, embedded references correspond to object references and explicit references include $IsA$ relationships reflecting the inheritance property.

The remainder of this section is organised as follows: First, we provide definitions of all relevant concepts. Subsequently, different types of accesses are discussed. Being familiar with the basic POS-concepts, we then consider the architecture of an object store. Finally, we introduce the service interface of POS, which consists of operation signatures that enable the storage of and access to storage objects.

**An Object Model for POS.** The (abstract) object model of the persistent object store can be defined as follows:

**Definition 5.2.** The *object store* POS is a triple

$$\text{POS} = (O, R, I), \text{ where } O = \{O_1, ..., O_n\},$$

$$R = < \{R_{11}, ..., R_{1k_1}\}, ..., \{R_{m1}, ..., R_{mk_m}\} >$$

$$\text{and } I = \{O_1, ..., O_j\}$$

where $O$ is a set of *storage objects*, $R$ is a bag of references, $I$ is a set of collection storage objects that represent indices and $n, k_i, m, j$ are positive Integer values denoting cardinalities of the sets and the bag respectively. □

The separation of $O$ and $I$ (which is different to other approaches such as HiPOS [146]) offers several advantages. Introducing $I$ into POS allows it to independently maintain indices. The only action required by a high-level DBS component is to create an index and associate it with a collection (e.g. representing an iDBPQL collection-class). Thus, manipulating such indexed collections allows POS to updates corresponding indices automatically. Without $I$, the high-level DBS component must explicitly maintain indices. Treating indices in the same way as other collections enables code and query optimisers to utilise direct and index-based accesses more uniformly.

Each storage object $O_i$ consists of a list of attributes with a globally unique storage object identifier $OID$. Each attribute, in turn, is a quadruple of the form ( *type* : *card* : *name* : *value* ), where *type* is the object's type, *card* stores the number of sub-objects in case the type is a collection type, *name* is the (external) name of the object, and *value* holds the (nested) object value. More details about identifiers, supported types etc. can be found in Section 5.2.1.

Storage objects are classified internally. We distinguish between collections and *regular storage objects*. The physical representation of collections and regular storage objects is identical. Collections (e.g. indices, a collection containing identifiers of all storage objects that correspond to DB collections, or a collection containing identifiers of all storage objects that correspond to instances of this class) are introduced to enable higher level modules to classify storage objects according to their (logical) structure.

**Definition 5.3.** A *collection* $C = ( OID, \{ OID_1, ..., OID_n \} )$ is a pair of a collection identifier $OID$ – just another storage object identifier – and a set of storage object identifiers $OID_i$ in which all storage objects are of the same (logical) structure. □

In addition to using collections to related storage objects, references from one storage object to another storage object can be specified. The object store distinguishes between two types of references. These are embedded references and explicit references. They are defined as follows:

**Definition 5.4.** Let POS $= (O, R, I)$ be an object store as introduced in Definition 5.2. A *reference* $Ref_j \in R$ corresponds to a triple of the following form:

$$Ref_j = (name, OID_k, OID_l),$$

where $name$ is the name of the reference, $OID_k$ is the identifier of the initial and $OID_l$ the identifier of the terminal storage object ($\in O$) of the reference. The reference name $name$ is not required to be unique. However, there should not be any two references from $OID_k$ to $OID_l$ with the same reference name.
An *embedded reference* $Ref^{emb}$ from a regular storage object $O_k$ (with identifier $OID_k$) to another regular storage object $O_l$ (with identifier $OID_l$) is a reference that exists in both $O$ and $R$. In $O$, $Ref^{emb}$ is represented as an attribute of $O_k$ whose value is the object identifier of $O_l$. In $R$, $Ref^{emb}$ is represented as the triple $(attribute\_name, OID_k, OID_l)$.
An *explicit reference* is a reference that exists only in $R$. □

Embedded references form a part of the object structure in contrast to explicit references. However, both reference-types are represented in $R$. Explicit references are added to link storage objects that are commonly accessed together. They are maintained in $R$ separately from other references.

EXAMPLE 5.3. Once again, let us consider the `University` schema fragments from Example 3.3. To demonstrate the POS concepts, we restrict ourselves to the fragment on ODBS node $N_{CC}$ and its following five classes: `PersonC`, `StudentC`, `AcademicC`$_{CC}$, `StudentAcademicC`$_{CC}$, and `DepartmentC`$_{CC}$. The remaining classes as well as references to them are omitted. Furthermore, we assume that there are existing instances for each of the considered classes (with cardinalities as indicated below).
The persistent object store $POS = (O, R, I)$ that captures the considered schema fragment may be comprised of the following objects $O$, references $R$ and indices $I$:

– $O$ will include at least the `ROOT` collection containing references to all local (persistent) schemata. This includes the `University`$_{CC}$ collection object, which holds all objects representing schema classes. According to our assumptions, there will be at least five such collection objects. Each of these represents a collection of instances of the respective class. Class instances correspond to the largest proportion of storage objects, i.e. regular storage objects.

```
O = { $O_0$, $O_1$, $O_2$, $O_3$, $O_4$, $O_5$, $O_6$, $O_7$, $O_{h_1}$, ..., $O_{h_p}$, $O_{i_1}$, ..., $O_{i_q}$, $O_{j_1}$, ..., $O_{j_r}$,
      $O_{k_1}$, ..., $O_{k_s}$, $O_{l_1}$, ..., $O_{l_t}$ }

// the ROOT collection, which contains all schema collections, is the only
// POS object that has a pre-allocated OID, i.e. $O_0$
$O_0$ = { $O_1$ }

// schema collections -- each contains further collection objects that represent
// all class-collections of the particular schema; the only schema collection
// shown here corresponds to the University$_{CC}$ fragment
$O_1$ = { $O_2$, $O_3$, $O_4$, $O_5$, $O_6$ }

// class-collections -- each contains all objects that are instances of the
// particular class; the five classes listed correspond to PersonC, StudentC,
// AcademicC$_{CC}$, StudentAcademicC$_{CC}$, and DepartmentC$_{CC}$ respectively
$O_2$ = { $O_{h_1}$, ..., $O_{h_p}$ }
$O_3$ = { $O_{i_1}$, ..., $O_{i_q}$ }
$O_4$ = { $O_{j_1}$, ..., $O_{j_r}$ }
$O_5$ = { $O_{k_1}$, ..., $O_{k_s}$ }
$O_6$ = { $O_{l_1}$, ..., $O_{l_t}$ }

// regular storage objects: PersonC instances
$O_{h_1}$ = ( ... )
...
$O_{h_p}$ = ( ... )
// regular storage objects: StudentC instances
$O_{i_1}$ = ( ... )
...
$O_{i_q}$ = ( ... )
// regular storage objects: AcademicC$_{CC}$ instances
$O_{j_1}$ = ( 584, ( [ "Prof.", "Dr." ], "Klaus-Dieter", "Schewe" ),
         ( ( "PN 311, Massey University, Private Bag 11 222", "" ),
         "Palmerston North", 4412 ), "Database Concepts", $O_{l_1}$,
         { $O_{i_{13}}$, $O_{i_{43}}$, $O_{i_{84}}$, $O_{i_{134}}$, $O_{i_{332}}$, $O_{k_1}$, $O_{k_{33}}$ } )
$O_{j_2}$ = ( ... )
...
$O_{j_r}$ = ( ... )
// regular storage objects: StudentAcademicC$_{CC}$ instances
$O_{k_1}$ = ( 653, ( [ "Mr." ], "Markus", "Kirchberg" ), ( ( "Rugby Street", "78" ),
         "Palmerston North", 4412 ), 99003525, $O_{l_1}$, $O_{l_{12}}$, $O_{j_1}$, "Database Systems",
         $O_{l_1}$, { $O_{i_{23}}$, $O_{i_{53}}$, $O_{i_{112}}$ } )
$O_{k_2}$ = ( ... )
...
$O_{k_s}$ = ( ... )
// regular storage objects: DepartmentC$_{CC}$ instances
$O_{l_1}$ = ( "Department of Information Systems", "City Centre",
         { "0800 DEPT IS", "06 350 5799" }, $O_{j_1}$, { ... }, { ... }, { ... } )
$O_{l_2}$ = ( ... )
...
$O_{l_t}$ = ( ... )
```

– $R$ will be comprised of at least two sets of references: $R = < R_1, R_2 >$ where $R_1$ is a set of embedded references between storage objects and $R_2$ is a set of explicit references between collection storage objects only. While the former captures object references, the latter models the inheritance forest.

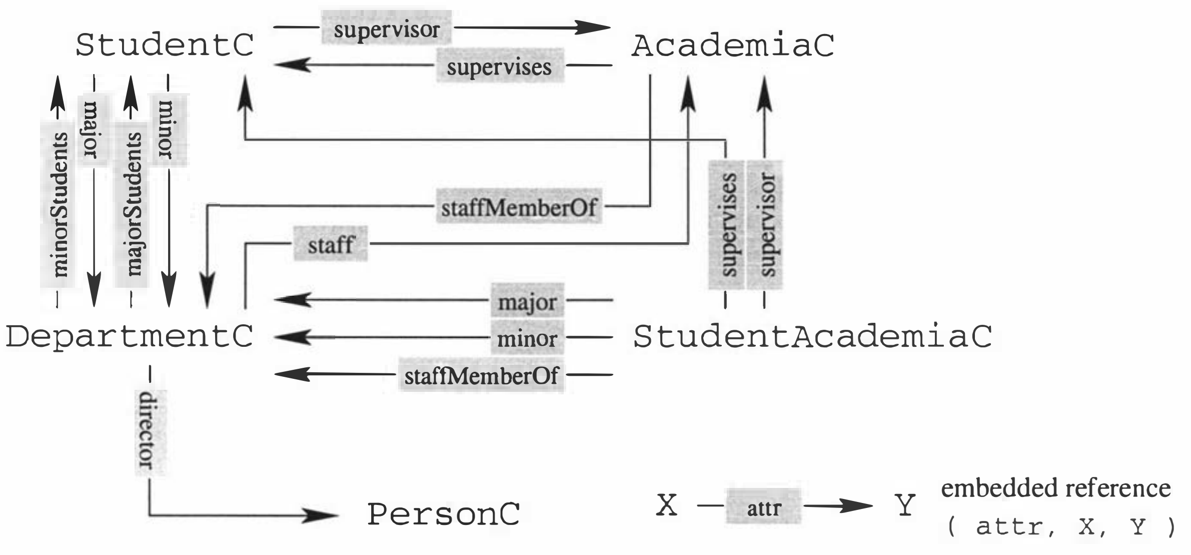


**Fig. 5.2.** Overview of Embedded References Between Instances of Classes of the $\texttt{University}_{CC}$ Schema Fragment as Considered in Example 5.3.

```
$R_1$ = { ... refer to Figure 5.2; it indicates which object references are
        captured in this set ... }

$R_2$ = { ( "IsA", StudentC, PersonC ), ( "IsA", AcademicC$_{CC}$, PersonC ),
        ( "IsA", StudentAcademicC$_{CC}$, StudentC ),
        ( "IsA", StudentAcademicC$_{CC}$, AcademicC$_{CC}$ ) }
```

- $I$ may be empty or include associative index structures defined on this schema fragment. Let us assume that we only have one index, i.e. $I = \{O_7\}$. The indexed class is `PersonC` with index key = ( `addr.city`, `addr.street` ). To capture all corresponding objects, all instances of its sub-classes must be included in the index too. $R_2$ can be used to determine the respective sub-classes.

```
// Dense index on all instances of PersonC (and its sub-classes)
$O_7$ = { $O_{h_1}$, ..., $O_{h_p}$, $O_{i_1}$, ..., $O_{i_q}$, $O_{j_1}$, ..., $O_{j_r}$, $O_{k_1}$, ..., $O_{k_s}$ }
```

We will later refine this example by outlining how storage objects (together with type information and additional cardinalities) are mapped to internal data structure. This is required to demonstrate how simple operations, such as selections involving base types only, may be passed to POS. On the contrary, the internal representation and organisation of index structures are not of interest to this thesis. A well-defined interface utilising these structures will be sufficient.

□

**Access Methods.** Apart from storing storage objects and retrieving storage objects by identifier (i.e. direct access), POS offers associative and navigational access to storage objects.

*Associative Access.* Associative access allows to retrieve a subset of storage objects that satisfy a certain condition from a collection. Conditions are applied to attribute values and include the following basic algebraic operators: <, <=, ==, ! =, >=, and >. As a result, a set of references to all storage objects in the collection satisfying the condition is returned.

Considering ODBMSs, associative access structures are similar to those commonly used in ORDBMSs. Examples include Single-Class (SC) and Class-Hierarchy (CH) indices, H-tree, hierarchy class Chain (hcC) tree, Class-Division (CD) index, nested, path and multi indices, access support relations, and Nested-Inherited index (NIX). The majority of these approaches are modifications of ORDBMS indices (such as B+ trees and join indices) and can be used efficiently to implement associative access methods.

*Navigational Access.* Navigational access allows the retrieval of storage objects that are connected to a given storage object via references or inverse references. For instance, if there is a reference from storage object $O_1$ to storage object $O_2$ or vice versa, we say that $O_1$ and $O_2$ are connected. If $O_1$ is connected to $O_2$ and $O_2$ is connected to $O_3$, $O_1$ is also connected to $O_3$. If there is a reference from $O_1$ to $O_2$, we call it a direct reference. Each direct reference defines an inverse reference implicitly, e.g. from $O_2$ to $O_1$.

Navigation can also be restricted by specifying a condition. Conditions contain path expressions specifying a minimum and / or maximum number of (direct or inverse) references (i.e. the navigation depth) to be followed or a list of allowed or disallowed paths to be or not to be followed.

Examples for ODBMSs include navigation index, ring and spider structures, join index hierarchies, triple-node hierarchies, and Matrix-Index Coding (MIC). In [70], it is suggested that the navigation index and MIC can be generalised by making them independent from the coding technique used, i.e. any appropriate coding technique (e.g. data compression techniques) can be selected to guarantee optimal performance in different prevailing data and request patterns.

**The Architecture of POS.** POS implements the basic services outlined above while meeting the general requirements of data persistence, concurrent access support, support of multi-level transactions, and support of checkpointing and recovery procedures. The basic architecture of POS is shown in Figure 5.3. A request manager is instantiated for each higher-level request. The collection manager maintains access structures that are required for associative access. The navigation manager maintains structures that support efficient navigation. Both managers are singletons, which are shared by all request manager instances.

Request managers use the services of the caching module's record interface (i.e. to access DB objects), while the collection manager and the navigation manager use the services of the caching module's page interface (i.e. to ensure persistence for associated and navigational access structures). Before proceeding with an operation, a request

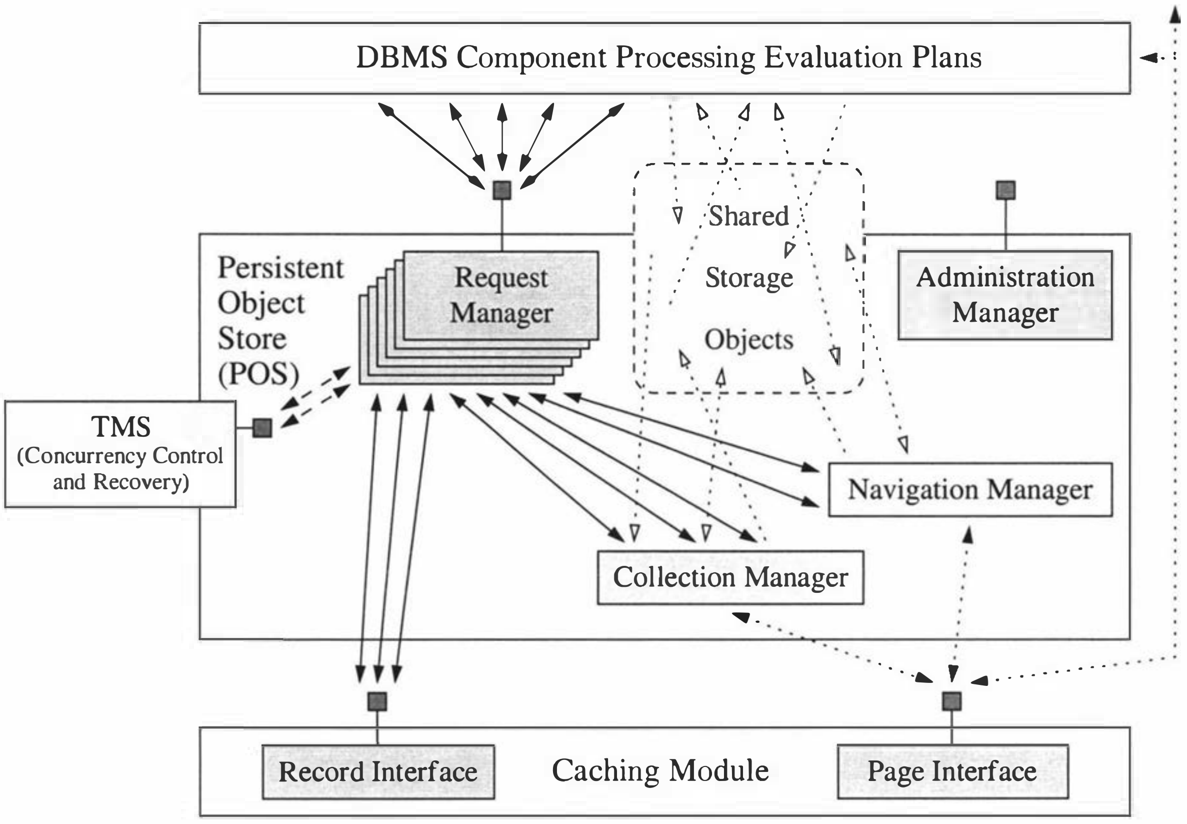


**Fig. 5.3.** Architecture of the Persistent Object Store.

manager consults with the transaction management system to ensure that (conflict-)serialisability and recoverability is ensured. The operation proceeds only if permission is granted.

Object access results in a collection of in-memory references being returned to the higher-level requester. Persistent, shared objects are made available in the shared memory area. To do so, each request is accompanied by an *Object Store Access Control Block (OSACB)*. This OSACB control block (refer to Section 5.2.1) contains the transaction identifier assigned to the high-level operation, a pointer to an empty, uninitialised POS collection object and variables used for request and object access synchronisation. Upon completion of a request, the corresponding request manager attaches the result to the OSACB control block's result collection and signals success (or failure).

The administration manager assists with checkpointing, recovery and maintenance of associative and navigational access structures.

**The Service Interface.** Higher-level DBMS components communicate with POS through a well-defined service interface. This interface exposes operations that support all three types of object access. In addition, POS supports the evaluation of simple expressions. A description of the supported operations and related concepts is provided next.

| Operator | Operand $p$ | Operand $q$ | Description |
|---|---|---|---|
| fwdNavLen | min NAT | max NAT | specifies minimum and maximum forward navigation depths |
| bckNavLen | min NAT | max NAT | specifies minimum and maximum backward navigation depths |
| navLen | max NAT | max NAT | specifies maximum navigation depths; $p$ and $q$ refer to forward navigation and backward navigation respectively |
| fwdNavPath | max NAT | $\in$ PATH[] | specifies a maximum forward navigation depth and restricts the paths to be followed |
| bckNavPath | max NAT | $\in$ PATH[] | specifies a maximum backward navigation depth and restricts the paths to be followed |
| path | $\in$ PATH[] | $\notin$ PATH[] | restricts the paths to be followed; $p$ and $q$ refer to lists of allowed path names and disallowed path names respectively |

**Table5.3.** (Binary) Path-Operators that are Supported by POS.

*Types.* POS supports primitive types (i.e. CHAR, BOOL, NAT, INT and REAL), a type OID that generates globally unique identifiers for storage objects OBJ, the reference-type REF, the record type REC and various collection types (including LIST, SET and BAG).

*Algebraic Operators.* A number of basic algebraic operators are supported by POS. These are: <, <=, ==, ! =, >=, and >. Such operators are defined in the usual manner for any primitive data type. In addition, == and ! = are also defined for the OID type.

*Path-Operators.* A PATH is a list of String values. POS supports a number of basic path-operators. Table 5.3 summarises these path expressions.

*The OBJ and REF Data Structures.* POS accepts objects of type OBJ and always returns collections of in-memory references to objects of type OBJ). The prototype system of the POS component is implemented in the programming language C. Corresponding data type definitions are as follows:

- A storage object is implemented as a doubly-linked list. The first element identifies the list and refers to the first attribute. All subsequent elements of the list correspond to attributes.

```
typedef struct objBody_struct {
  struct objBody_struct * prev;
  POS_ObjType             type;             // type of this 'value' or sub-object
  int                     card; // cardinality; used for collection objects only
  char *                  name;                                  // external name
  union {
    CHAR *                  charAtom;
    NAT                     natAtom;
    INT                     intAtom;
    REAL                    realAtom;
    BOOLEAN                 boolAtom;
    OID                     oid;
```

```
      struct objBody_struct * attr;
    } value;                                         // storage object value
    struct objBody_struct * next;
  } ObjBody;

  typedef struct objHead_struct {
    OID         oid;                  // unique, immutable object identifier
    POS_ObjType type;               // refers to the original collection type
    int         card;                                        // cardinality
    char *      name;                                       // external name
    ObjBody *   value;    // (nested) object value that matches the specified type
  } OBJ;
```

– A reference between two storage objects is defined as follows:

```
  typedef struct ref_struct {
    char * name;                                          // reference name
    OID    oid1;                   // identifier of the starting storage object
    OID    oid2;                   // identifier of the terminal storage object
  } REF;
```

*Additional Data Structure.* Two special-purpose data structures are supported by POS, which are defined as follows:

– POS supports the evaluation of simple conditions. Such conditions have to be specified in the following format:

```
  typedef struct cond_struct {
    Operator op;  // algebraic operator; OR matching semantics for operand arrays
    Operand  val1[];                                 // array of 1st operands
    Operand  val2[];        // array of 2nd operands or (null) if unary operator
  } Condition;

  typedef union operand_def {
    CHAR * name;                              // String value or String pattern
    INT    pos;                                // position, e.g. 2nd attribute
    CHAR * const;                                          // constant value
    PATH * path;                                                     // path
  } Operand;
```

– An OSACB control block has the following format:

```
  typedef struct osAccCB_struct {
    TRANSID        transId;                          // transaction identifier
    OBJ *          result;                 // reserved pointer for result value
    POS_RequStatus status;                       // status of request execution
    MUTEX          mutex;
    COND_VAR       condVar;
  } OSACB;
```

EXAMPLE 5.4. Let us revisit the object store presented in Example 5.3. Regular storage objects and collections storage objects are represented internally as follows:

```
// the ROOT collection, which contains all schema collections; it is the only
// POS object that has a pre-allocated OID, i.e. $O_0$
$O_0$ = 0:SET:1:"ROOT" → OID:0:"University$_{CC}$":1
     ^1 ^2 ^3 ^4    ^5 ^6  ^7 ^8               ^9

// ^1 to ^5 refer to members of the OBJ structure, while ^6 to ^9 correspond to
// the ObjBody structure. Details are as follows:
//     ^1 ≅ OID          oid                   ^6 ≅ POS_ObjType type
//     ^2 ≅ POS_ObjType type                   ^7 ≅ INT         card
//     ^3 ≅ INT          card                  ^8 ≅ CHAR *      name
//     ^4 ≅ CHAR *       name                  ^9 ≅ OID         value.oid
//     ^5 ≅ ObjBody *    value
// ↔ (refer below) represents a doubly linked list

// schema collections -- each contains further collection objects that represent
// all class-collections of the particular schema; the only schema collection
// shown here corresponds to the University$_{CC}$ fragment
$O_1$ = 1:SET:5:"University$_{CC}$" → OID:0:"PersonC":2 ↔ OID:0:"StudentC":3 ↔
      OID:0:"AcademicC$_{CC}$":4 ↔ OID:0:"StudentAcademicC$_{CC}$":5 ↔
      OID:0:"DepartmentC$_{CC}$":6

// class-collections -- each contains all objects that are instances of the
// particular class; the five classes listed correspond to PersonC, StudentC,
// AcademicC$_{CC}$, StudentAcademicC$_{CC}$, and DepartmentC$_{CC}$ respectively
$O_2$ = 2:SET:$p$:"PersonC" → OID:0:"PersonC":$h_1$ ↔ ... ↔ OID:0:"PersonC":$h_p$
$O_3$ = 3:SET:$q$:"StudentC" → OID:0:"StudentC":$i_1$ ↔ ... ↔ OID:0:"StudentC":$i_q$
$O_4$ = 4:SET:$r$:"AcademicC$_{CC}$" → OID:0:"AcademicC$_{CC}$":$j_1$ ↔ ... ↔ OID:0:"AcademicC$_{CC}$":$j_r$
$O_5$ = 5:SET:$s$:"StudentAcademicC$_{CC}$" → OID:0:"StudentAcademicC$_{CC}$":$k_1$ ↔ ... ↔
      OID:0:"StudentAcademicC$_{CC}$":$k_s$
$O_6$ = 6:SET:$t$:"DepartmentC$_{CC}$" → OID:0:"DepartmentC$_{CC}$":$l_1$ ↔ ... ↔
      OID:0:"DepartmentC$_{CC}$":$l_t$

// regular storage objects: We restrict ourselves to the two instances fully
// detailed in Example 5.3
$O_{j_1}$ = $j_1$:REC:6:"AcademicC$_{CC}$" → NAT:0:"personId":584 ↔ REC:3:"name" →
      LIST:2:"titles" → CHAR:6::"Prof." ↔ CHAR:4::"Dr." ↔
      CHAR:13:"firstname":"Klaus-Dieter" ↔ CHAR:7:"lastname":"Schewe" ↔
      REC:3:"addr" → REC:2:street →
      CHAR:46:"name":"PN 311, Massey University, Private Bag 11 222" ↔
      CHAR:0:"number": ↔ CHAR:17:"city":"Palmerston North" ↔
      NAT:0:"zipcode":4412 ↔ CHAR:18:"specialisation":"Database Concepts" ↔
      OID:0:"staffMemberOf":$O_{l_1}$ ↔ SET:7:"supervises" → OID:0:"StudentC":$O_{i_{13}}$ ↔
      OID:0:"StudentC":$O_{i_{43}}$ ↔ OID:0:"StudentC":$O_{i_{84}}$ ↔ OID:0:"StudentC":$O_{i_{134}}$ ↔
      OID:0:"StudentC":$O_{i_{332}}$ ↔ OID:0:"StudentAcademicC$_{CC}$":$O_{k_1}$ ↔
      OID:0:"StudentAcademicC$_{CC}$":$O_{k_{33}}$

$O_{k_1}$ = $k_1$:REC:10:"StudentAcademicC$_{CC}$" → NAT:0:"personId":653 ↔ REC:3:"name" →
      LIST:1:"titles" → CHAR:4::"Mr." ↔ CHAR:7:"firstname":"Markus" ↔
      CHAR:10:"lastname":"Kirchberg" ↔ REC:3:"addr" → REC:2:street →
      CHAR:46:"name":"Rugby Street" ↔ CHAR:0:"number":"78" ↔
      CHAR:17:"city":"Palmerston North" ↔ NAT:0:"zipcode":4412 ↔
      NAT:0:"studentId":99003525 ↔ OID:0:"major":$O_{l_1}$ ↔ OID:0:"minor":$O_{l_{12}}$ ↔
      OID:0:"supervisor":$O_{j_1}$ ↔ CHAR:17:"specialisation":"Database Systems" ↔
```

`OID:0:"staffMemberOf":`$O_{l_1}$ $\leftrightarrow$ `SET:3:"supervises"` $\rightarrow$ `OID:0:"StudentC":`$O_{i_{23}}$ $\leftrightarrow$ `OID:0:"StudentC":`$O_{i_{53}}$ $\leftrightarrow$ `OID:0:"StudentC":`$O_{i_{112}}$

The representation of the remaining storage objects is analogous. References are represented using the `REF` structure as outlined above. □

*Interface Signatures.* POS enables higher-level modules to access, insert, update, and delete storage objects and corresponding references through the following collection of operations:

- `void Retrieve ( OSACB * oac, OID oid, BOOL isa )` ... locates the storage object that corresponds to the given storage object identifier `oid`.
  If the `isa` value is `TRUE` (which only makes sense for collection storage objects), not only the given object identifier is considered but also its associated sub-collections. Thus, explicit references are utilised.
  In addition, each attribute (with a value not equal to `(null)`) that forms a part of an embedded reference is checked against $R$. If there is no corresponding reference in $R$ (i.e. the referenced object has been deleted previously) the attribute value is set to `(null)`.

  **Arguments:**

  `oac` - a pointer to an OSACB control block in POS's shared memory area.
  `oid` - the storage object identifier of the storage object to be retrieved.
  `isa` - a boolean value indicating whether or not the inheritance (i.e. sub- and super-class information added as explicit references) relation should be taken into consideration.

  **Effects:**

  The storage objects with identifier `oid` wrapped in a collection object or, in the event that no such object exists, the `(null)` pointer is assigned to the OSACB control block's result pointer.

- `void FindFromCollection ( OSACB * oac, OID cid, BOOL isa, Cond objCond );` ... determines a subset of objects that belong to the collection type storage object with identifier `cid`. All objects in the subset must meet the condition `objCond`.
  First, the collection type storage object (say $O_2$ from Example 5.3) with identifier `cid` is located. If the `isa` value is `TRUE`, all objects that can be reached using only forward navigation over explicit references (i.e. $O_3$, $O_4$ and $O_5$ in Example 5.3) are also considered. Each of these collection type storage objects contain attributes of type `OID`, i.e. all attribute values correspond to storage object identifiers. Subsequently, the condition `objCond` is applied to all storage objects that belong to the located collections (i.e. $O_2$, $O_3$, $O_4$, and $O_5$). Objects that satisfy the condition are added to the result list. If no condition is specified all objects that belong to these collections are added to the result list.

  **Arguments:**

  `oac` - a pointer to an OSACB control block in POS's shared memory area.

`cid` - the identifier of a (collection type) storage object.
`isa` - a boolean value indicating whether or not the inheritance (i.e. sub- and super-class information added as explicit references) relation should be taken into consideration.
`objCond` - the condition to be applied to each member of the collection.

**Effects:**

A collection of all storage objects directly referenced by the collection with identifier `cid` that meet the condition `objCond` is assigned to the OSACB control block's result pointer.

– `void FindEnclosure ( OSACB * oac, OID oid, Cond pathCond, Cond objCond );` ... determines a subset of objects that can be reached by following references in accordance with the path condition `pathCond` while satisfying the storage object condition `objCond`.
First, the storage object (e.g. $O_{k_1}$ from Example 5.3) with identifier `oid` is located. Subsequently, a list of storage objects consisting of all objects 1) that are reachable from $O_{k_1}$ by adhering to the path condition `pathCond`; and 2) that meet the storage object condition `objCond` is generated.
The path condition adds restrictions to the path of forward and/or backward references to be followed. If no path condition is specified all references are followed.
The storage object condition adds restrictions to the storage objects to be returned. If no storage object condition is specified, all objects obtained while adhering to the path condition, are added to the result list.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`oid` - the identifier of a storage object that represents the starting point of the computation of the enclosure.
`pathCond` - the path condition to be evaluated while following references during the computation of the enclosure.
`objCond` - the storage object condition to be evaluated while computing the enclosure.

**Effects:**

A collection of all storage objects that satisfy both conditions, i.e. the path condition `pathCond` and the storage object condition `objCond`, are assigned to the OSACB control block's result pointer.

– `void AddNewObject ( OSACB * oac, OBJ newObj, REF[] expRefs );` ... assigns a storage object identifier to the given object `newObj` and adds the object to the object store. For each embedded reference in *newObj* a corresponding entry is added to $R$.
In addition, a list of (explicit) references `expRefs` may be specified. This is the case only if the new object is a member of a collection type storage object. For instance, the object *newObj* may represent a class-collection. Thus, explicit references correspond to *IsA*-relationships. Accordingly, POS will update its structure(s)

maintaining super- and sub-class relationships.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`newObj` - the new storage object to be added to the object store.
`expRefs` - a list of explicit references.

**Effects:**

The storage object identifier assigned to the newly created object wrapped in a collection object is assigned to the OSACB control block's result pointer.

– `void InsertObject ( OSACB * oac, OID colOid, CHAR * name, OID stOid );` ... adds a storage object with an optional external name `name` and an identifier `stOid` to the collection type object identified by `colOid`. In addition, the storage object with identifier `stOid` is added to any index associated with the collection `colOid`.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`colOid` - the identifier of an existing collection type storage object to which an storage object is to be added.
`name` - a (optional) name of the object to be added to a collection type storage object (default is `(null)`).
`stOid` - the identifier of an existing storage object that is to be added to the collection type object with identifier `colOid`.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the insertion was successfully and `false` otherwise.

– `void AddReference ( OSACB * oac, REF ref );` ... adds an embedded reference to structures maintaining relationships between storage objects.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`ref` - the reference to be added.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the reference was added successfully and `false` otherwise.

– `void DeleteObject ( OSACB * oac, OID oid );` ... deletes the storage object with identifier `oid` from the object store. In addition, all references to and from this object are deleted too.
This non-cascading approach is to be supported by a service routine, which periodically deletes objects (and corresponding references) that are no longer used (i.e. not referenced and not member of any collection).

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`oid` - the identifier of the storage object to be deleted.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the storage object was deleted successfully and `false` otherwise.

– `void RemoveObject ( OSACB * oac, OID colOid, OID stOid );` ... removes the storage object with identifier `stOid` from the collection type storage object with identifier `colOid`. In addition, the storage object with identifier `stOid` is removed from any index associated with the collection `colOid`. Furthermore, all associated explicit references are deleted.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`colOid` - the identifier of an existing collection type storage object from which an storage object is to be removed.
`stOid` - the identifier of an existing storage object that is to be removed from the collection type object with identifier `colOid`.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the storage object was removed successfully and `false` otherwise.

– `void DeleteReference ( OSACB * oac, REF ref );` ... deletes an existing (embedded) reference from $R$.

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`ref` - the references to be deleted.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the reference was deleted successfully and `false` otherwise.

– `void UpdateObject ( OSACB * oac, OID oid, OBJ newObj )` ... replaces the existing storage object with identifier `oid` with the new storage object `newObj`. The storage object identifier and the object structure remain unchanged. In addition, it has to be ensured that all index entries referring to this object are updated.
For each embedded reference, the corresponding references in $R$ need to be updated (e.g. delete and add).

**Arguments:**

`oac` - a pointer to an OSACB control block in POS's shared memory area.
`oid` - the identifier of the storage object to be updated.
`newObj` - the new storage object that replaces the existing storage object with identifier `oid`.

**Effects:**

A boolean value wrapped in a collection object is assigned to the OSACB control block's result pointer. The boolean value is `true` in the event that the update was successful and `false` otherwise.

**POS as a Platform for iDBPQL.** iDBPQL distinguishes types and values from classes and objects. Schemata are defined over classes which expose both structure and behaviour. Values are only found 'inside' classes (i.e. as object values). Thus, they never persist independently. Hence, POS and higher-level DBS components always exchange objects or collections of objects but never values. Also, iDBPQL associates system-maintained collections with classes through which access to all objects of this class and its sub-classes is possible. POS enables to capture such concepts as follows:

– Schemata correspond to collections of class objects. A special `ROOT` collection, which keeps track of all existing schemata, is maintained (refer to object $O_0$ in Example 5.4).
– Classes correspond to collections of storage objects. The `InsertObject` and `RemoveObject` operations allow to maintain such collections.
– iDBPQL objects are mapped to regular storage objects. Variables of value types are represented as POS attributes of some POS type. Variables of a reference-type are represented as POS attributes of type OID as well as embedded references in $R$.
– Inheritance hierarchies are maintained as explicit references. This is supported by `AddNewObject` (explicitly) and `DeleteObject` (implicitly) operations.
– An object's value-type and reference-type variables may be updated through the `UpdateObject` operation.

Object access takes place through iDBPQL expressions, in particular query expressions as outlined in Syntax Snapshot 4.19 (on page 110). POS enables direct access through the `Retrieve` operation, associative access through the `FindFromCollection` operation, and the `FindEnclosure` operation supports a mixture of both navigational and associative access. Query expressions that directly affect POS are selections, projections (using index-only accesses) and navigational joins. The remaining operations will be implemented by the operational DBS component and only access data using those three simple operations. We will outline some initial examples next.

EXAMPLE 5.5. Let us consider some iDBPQL requests and their affects on POS. Assume, requests stem from a program that imports the `University`$_{CC}$ schema fragment from Examples 5.3 and 5.4.

1. `FOR EACH StudentAcademicC`$_{CC}$ `AS staca DO` ... all objects of class `StudentAcademicC`$_{CC}$ (which does not have any sub-classes) have to be retrieved. Without any hints from the optimiser this can be achieved as follows):

```
// determine the corresponding schema collection object
result₁ = FindFromCollection ( (OSACB *) oac, (OID) 0, FALSE,
        (Cond) ( '==', "name", "University_CC" ) );          // returns: (OID) 1

// determine the corresponding class-collection object
result₂ = FindFromCollection ( (OSACB *) oac, (OID) result₁, FALSE,
        (Cond) ( '==', "name", "StudentAcademicC_CC" ) );    // returns: (OID) 5

// retrieve all objects from the class-collection
result₃ = Retrieve ( (OSACB *) oac, (OID) result₂, TRUE );
                                               // returns: (OID) k₁ ... (OID) k_s

// subsequently, with each iteration, the next object from the class-collection
// is retrieved directly (i.e. via Retrieve)
```

2. `PersonC WHERE ( DepartmentC`$_{CC}$`.director == THIS )` ... retrieve all `PersonC` objects that are directors of a department in the university. This request utilises backward references since there is no reverse iDBPQL reference defined (assume the OID of class `PersonC` is already known):

```
result = FindFromCollection ( (OSACB *) oac, (OID) 2, TRUE,
        (Cond) ( BckNavPath, 1, (PATH) "director" ) );
```

First, POS locates object $O_2$. Subsequently, all objects that can be reached from $O_2$ via explicit forward references in $R_2$ are also located. As an intermediate result, we now have collection storage objects $O_2$, $O_3$, $O_4$, and $O_5$. Next, the condition is applied to all objects in those four collections. According to the condition, we add any object to the result that has an associated, embedded (backward) reference with label "director". The result will not be empty as it will contain at least object $O_{j_1}$.
Note: Despite using a navigation condition, a `FindFromCollection` call never results in navigational access. Instead, it is only tested whether or not the corresponding reference exists in $R$.

3. `PersonC WHERE ( addr.city == "Palmerston North" )` ... selects all objects from `PersonC` (or any of its sub-classes) that live in "Palmerston North". Since there is an index defined on `PersonC`, we can execute an index only scan to retrieve all qualifying OIDs.

```
result = FindFromCollection ( (OSACB *) oac, (OID) 7, FALSE,
        (Cond) ( ==, (PATH) "addr.city", "Palmerston North" ) );
```

4. `std.supervisor.supervises WHERE ( std.major == major )` ... navigates first from the current student object `std` to the instance of class `AcademicC` that represents the student's supervisor and then to all `StudentC` objects that are supervised by the same academic staff member as student `std`. Subsequently, a selection that only chooses students, which are majoring in the same department as student `std`, is executed. Assuming that `std` corresponds to the object with OID $k_1$ and the `std.major` projection has been evaluated, POS may execute the request as follows:

```
result = FindEnclosure ( (OSACB *) oac, (OID) k₁,
        (Cond) ( path, (PATH) "supervisor.supervises", NULL ),
        (Cond) ( ==, (PATH) "major", "Department of Information Systems" ) );
```

5. Assume, a new class is added into the schema. For instance, the `StudentAcademicC`$_{CC}$ class does not appear in the original schema as introduced in Example 3.3. Let us consider the corresponding POS calls that would add this new class into the persistent object store:

```
result₁ = AddNewObject ( (OSACB *) oac,
          (OBJ) (null):SET:0:"StudentAcademicC_CC" → (null),
          (REF[]) CHAR:"IsAStudentC":OID:(null):OID:3 →
          CHAR:"IsAAcademicC_CC":OID:(null):OID:4 );        // assigns a unique OID,
      // i.e. 5, creates the POS object, add the explicit reference to R₂ (after
        // entering the OID in the respective field) and returns the assigned OID

// next, insert the new class-collection storage object into the corresponding
// collection of all schema classes
result₂ = InsertObject ( (OSACB *) oac, 1, "StudentAcademicC_CC", 5 );
```

6. `NEW StudentAcademicC`$_{CC}$ `staca = ( 653, ( [ "Mr." ], "Markus", "Kirchberg" ), ( ( "Rugby Street", "78" ), "Palmerston North", 4412 ), 99003525,` $O_{l_1}$, $O_{l_{12}}$, $O_{j_1}$, `"Database Systems",` $O_{l_1}$, `{` $O_{i_{23}}$, $O_{i_{53}}$, $O_{i_{112}}$ `} )` ... inserts a new `StudentAcademicC`$_{CC}$ object. In fact, the object with OID $k_1$ from Example 5.4 is inserted. Creating this new object will result in nine POS calls. The first creates the regular storage object, the second to eighth calls add embedded references into $R_1$, and the ninth call adds the object into the corresponding collection object associated with the StudentAcademicC$_{CC}$ class.

```
result₁ = AddNewObject ( (OSACB *) oac,
          (OBJ) (null):REC:10:"StudentAcademicC_CC" → NAT:0:"personId":653 ↔
          REC:3:"name" → LIST:1:"titles" → CHAR:4::"Mr." ↔
          CHAR:7:"firstname":"Markus" ↔ CHAR:10:"lastname":"Kirchberg" ↔
          REC:3:"addr" → REC:2:street → CHAR:46:"name":"Rugby Street" ↔
          CHAR:0:"number":"78" ↔ CHAR:17:"city":"Palmerston North" ↔
          NAT:0:"zipcode":4412 ↔ NAT:0:"studentId":99003525 ↔
          OID:0:"major":l₁ ↔ OID:0:"minor":l₁₂ ↔ OID:0:"supervisor":j₁ ↔
          CHAR:17:"specialisation":"Database Systems" ↔
          OID:0:"staffMemberOf":l₁ ↔ SET:3:"supervises" →
          OID:0::i₂₃ ↔ OID:0::i₅₃ ↔ OID:0::i₁₁₂,
          (REF[]) (null) );                          // assigns a unique OID, i.e. k₁,
                                      // creates the POS object and returns the assigned OID

result₂ = AddReference ( (OSACB *) oac, (REF) CHAR:"major":OID:1:OID:l₁ );

result₃ = AddReference ( (OSACB *) oac, (REF) CHAR:"minor":OID:1:OID:l₁₂ );

result₄ = AddReference ( (OSACB *) oac, (REF) CHAR:"supervisor":OID:1:OID:j₁ );

result₅ = AddReference ( (OSACB *) oac, (REF) CHAR:"staffMemberOf":OID:1:OID:l₁ );

result₆ = AddReference ( (OSACB *) oac, (REF) CHAR:"supervises":OID:1:OID:i₂₃ );

result₇ = AddReference ( (OSACB *) oac, (REF) CHAR:"supervises":OID:1:OID:i₅₃ );

result₈ = AddReference ( (OSACB *) oac, (REF) CHAR:"supervises":OID:1:OID:i₁₁₂ );
```

$result_9$ = `InsertObject ( (OSACB *) oac, 5, "StudentAcademicC`$_{CC}$`",` $k_1$ `);`

When inserting the newly created object, POS also updates all indices that are associated with the respective collection – in the considered example, this results in object $O_7$ being updated accordingly.

□

### 5.2.2 A Multi-Level Transaction Management System

From an internal point of view, users access databases in terms of transactions. Fast response times and a high transaction throughput are crucial issues for all database systems. Hence, transactions are executed concurrently. The transaction management system ensures a proper execution of concurrent transactions. It implements concurrency control and recovery mechanisms to preserve the well-known ACID principles. A further increase in both response time and transaction throughput can be achieved by employing a more advanced transaction management system, e.g. a system that is based on the multi-level transaction model [16, 141, 142]. This model is counted as one of the most promising transaction models. It schedules operations of transactions based on information that is obtained from multiple levels. Since there are usually less conflicts on higher levels, lower-level conflicts[2] can be ignored. Hence, their detection increases the rate of concurrency. For instance, assume that two higher-level operations $op_1$ and $op_2$, which belong to different transactions, increment Integer values $A$ and $B$, respectively. Furthermore, let $A$ and $B$ reside on the same physical page. An increment will be executed as a page read followed by a page write. In the event that scheduling only considers operations on pages, $op_1$ will have to wait for $op_2$ or vice versa. Considering both levels, i.e. incrementations and read and write operations, we can allow $op_1$ and $op_2$ to execute concurrently. The information obtained from the higher-level indicates that different portions of the page are accessed. Thus, the corresponding read and write operations do not affect each other. Protecting such individual operations by short-term locks (i.e. latches) is sufficient.

In [142] the execution of concurrent transactions is described by means of a multi-level schedule. Level-by-level (conflict-)serialisability is ensured by employing one-level schedulers, i.e. level-by-level schedulers, on each level. A multi-level schedule is multi-level (conflict-)serialisable, if all level-by-level schedules are (conflict-)serialisable. However, [6] outlines two weaknesses of ensuring level-by-level (conflict-)serialisability, these are:

1. Scheduling requires information (i.e. input orderings) to be passed down from higher-level schedulers to lower-level schedulers. This contradicts independence, i.e. it restricts the modularisation of DBMS components; and
2. The execution of any two conflicting (higher-level) operations cannot be interleaved on lower levels even though that only a few lower-level operations might be in conflict.

[2] Such conflicts are referred to as *pseudo-conflicts*, which are low-level conflicts that do not stem from a higher-level conflict.

These issues are addressed in [6]. A new correctness criteria, which is referred to as *stack-conflict consistency*, is proposed. Stack-conflict consistency supports two correctness criteria: a weak and a strong order criteria. The weak order criteria allows conflicting operations to be executed simultaneously as long as the final result is not affected. The strong order criteria implies serial execution of conflicting operations. For instance, we assume an update of all sub-objects $a$, $b$ and $c$ of object $A$ that is to be followed by a read of values of all sub-objects of $A$. The common approach (i.e. using the strong order criteria) only releases the read operation on $A$ for execution once the update of $A$ has been executed successfully. Using the weak order criteria, read access sub-objects $a$, $b$ and $c$ is allowed as soon as their respective updates have succeeded. Thus, the degree of concurrency is increased without affecting the result of the execution. Instead of a multi-level schedule, a stack schedule is considered. The outputs of one schedule are 'plugged' to the inputs of the next. Only a strong ordering is propagated to all levels, weak ordering might even disappear (i.e. it depends on whether or not conflicting operations are involved). Finally, it should be noted that [6] also proves that level-by-level serialisability is a proper subset of the class of stack-conflict consistent schedules.

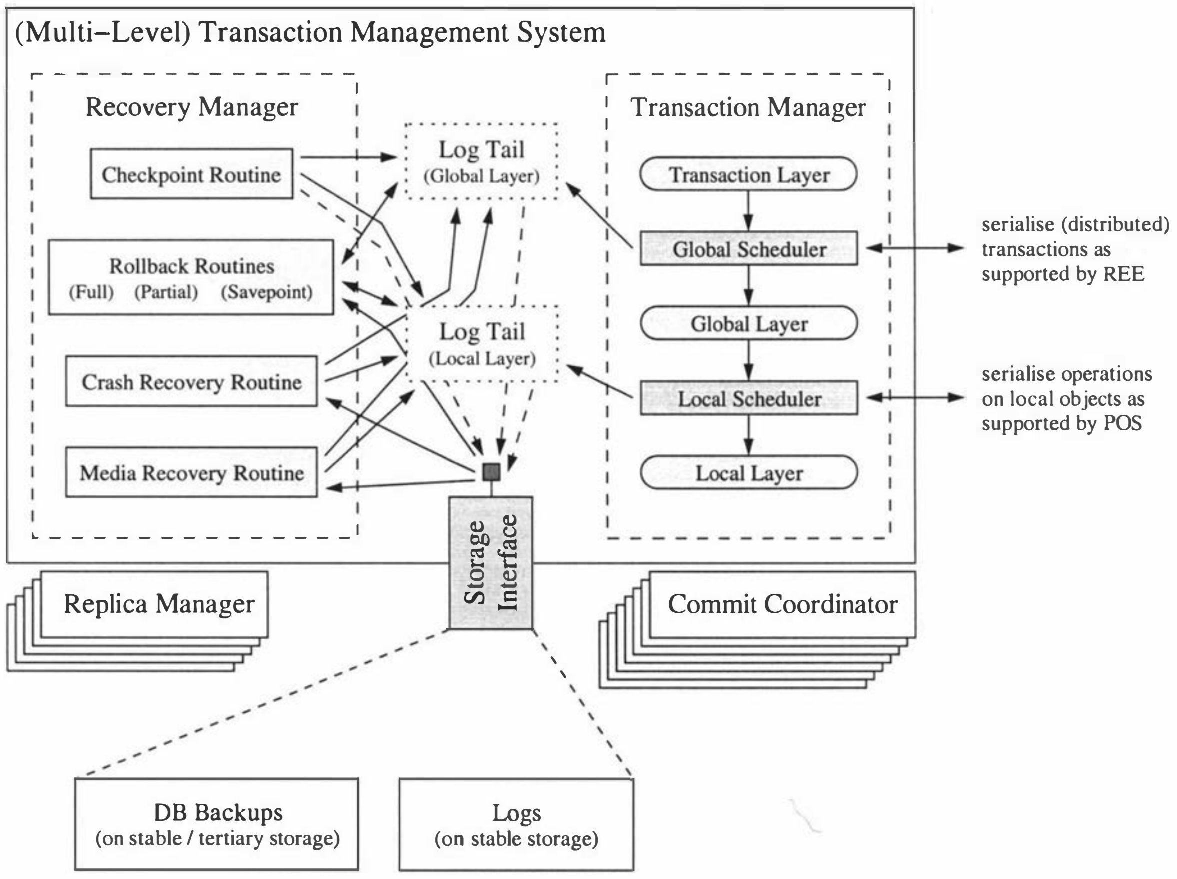


**Fig. 5.4.** Architecture of the Transaction Management System.

The layered system architecture as proposed in Section 3.1.1 allows to use the multi-level transaction model in a straightforward manner. Figure 5.4 shows a corresponding architecture in more detail. Either of the two approaches discussed above (i.e. level-by-level scheduling and stack scheduling) can be supported using this architecture.

Figure 5.4 indicates that the transaction management system controls the execution of operations of the request evaluation engine and POS. Thus, at least a two-level or two-stack scheduler is employed[3]. On each level, a certain concurrency control protocol is executed. For instance, on the highest level, sequences of iDBPQL operations are serialised. Such sequences always corresponds to top-level transactions that may execute only on the local node or across many ODBS instances.

The general approach to concurrency control is the use of locking protocols, especially strict two-phase locking (str-2PL). However, locking suffers from some major problems affecting transaction throughput. Two of those problems are transaction deadlocks and the impossibility to accept all (conflict-)serialisable schedules. A hybrid protocol called FoPL (*forward oriented concurrency control protocol with pre-ordered locking*), which is a provably correct protocol for multi-level transactions, is presented in [111]. FoPL does not suffer from any of the two problems that are mentioned above. A prototype implementation is presented in [65, 71]. It describes first experimental results of the basic FoPL protocol in comparison to str-2PL. It shows that FoPL-based schedulers outperform locking-based schedulers if conflict probabilities do not exceed a certain threshold.

Since we deal with distributed transactions, guaranteeing serialisability as described above is not sufficient. Global serialisability and one-copy serialisability [17] have to be guaranteed when dealing with distributed transactions on replicated objects. In [66], it is outlined how str-2PL and FoPL can be used on the global level. Guaranteeing global serialisability requires the use of a commit protocol, e.g. the optimised two-phase commit protocol [108], to ensure atomicity. Additionally, global deadlock detection mechanisms must be employed if str-2PL is used. FoPL requires only globally unique timestamps to be assigned at the start of the validation phase. Guaranteeing one-copy serialisability requires a certain replication schema to be applied [101]. As far as FoPL is concerned only a few extensions to this replication schema have to be applied. A comparison of str-2PL and FoPL with respect to their efficiency in the presence of distributed data is included in [66]. It is outlined that necessary extensions to FoPL are likely to be much less expensive than extensions to str-2PL.

Since concurrency control may force transactions to abort (either fully or partially), recovery mechanisms are provided. So far, only a few research projects have focused on recovery mechanisms for multi-level transactions [85, 107, 143]. However, none of these approaches comes without major restrictions or disadvantages limiting its practical use.

ARIES/NT [107] is an extension of the popular ARIES algorithm [92] to a very general model of nested transactions [16]. ARIES/NT is not designed to be used with multi-level transactions explicitly. In particular, locks are not released after finishing operations that are not transactions. MLR [85] adapts basic features of ARIES/NT to support both nested and multi-level transactions. It utilises compensation operations,

[3] It is likely that a three-level scheduler is implemented. The third level will serialise operations as supported by a caching module / buffer manager.

but unfortunately assumes them to exist in any case, which is not realistic. A simple, non-ARIES based approach to multi-level recovery is also known as Multi-Level Recovery [143]. It aims to '*reconcile performance, simplicity, and a rigid foundation in a well-understood framework*' [143, page 121] rather than being developed for industrial-grade systems as ARIES is. There are some similarities to ARIES-based approaches. Nevertheless, they are more sophisticated. For instance, ARIES-based approaches use a single technique for all levels whereas the Multi-Level Recovery approach requires level-specific recovery mechanisms and does not avoid rollbacks of undo actions. A single logging technique, however, is superior (i.e. only one log file has to be maintained, no additional communication mechanisms between level-specific recovery modules are necessary etc.).

An alternative approach to multi-level recovery is outlined in [66, 111]. The ARIES/ML algorithms preserves the main features of ARIES, is designed to support multi-level transactions, may be coupled with locking, optimistic or hybrid concurrency control protocols, utilises the existence of compensation operations, supports partial rollbacks and more. Necessary extensions to adapt the ARIES and ARIES/ML algorithms to a distributed computing environment (without relying on perfectly synchronised clocks) are proposed in [122, 123].

**The Service Interface.** The basic service interface of a multi-level transaction management prototype system is outlined below. Starting from the implementation that is presented in [65], a number of extensions have been included in order to provide basic support for concurrent and distributed processing. Interface operations are defined as follows:

```
TransId openTrans ( );          // open a new transaction; this routine returns a
                                            // globally unique transaction identifier
TransId openSubTrans ( TransId parentId );          // open a new sub-transaction;
                                  // parentId identifies the parent (sub-)transaction;
               // this routine returns a globally unique sub-transaction identifier
INT execute ( TransID tid, OID obj, OpId op );              // schedule the specified
    // operation that is to be executed by the (sub-transaction) with identifier
                         // tid; this routine returns one of the following pre-defined
                                              // values: APPROVED, RESTART or MONITOR
INT executeWeak ( TransID tid1, OID obj1, OpId op1,
                  TransID tid2, OID obj2, OpId op2 );       // schedule the second
                          // operation with respect to the weak order criteria; result
                            // values are identical to those of the execute operation
INT join ( TransID tid, NodeId remoteNode );         // the specified node will also
// participate in this transaction; this routine returns an acknowledgement only
INT transferOwner ( TransID tid, OID obj, NodeId fromNode, NodeId toNode );
      // control over the specified object is transfered from one ODBS instance
                                                                       // to another
INT commit ( TransID tid );         // a commit protocol is initiated; this routine
          // returns one of the following pre-defined values: SUCCESS or RESTART
INT abort ( TransID tid );          // a (sub-)transaction abort is initiated; this
                                    // routine acknowledges the success of the abort
INT setSavePoint ( TransID tid );                              // not implemented yet
INT rollback ( TransID tid, SaveID savePt );                   // not implemented yet
INT getStatus ( TransID tid );                       // return the current status of the
```

```
26          // (sub-transaction) that corresponds to the given transaction identifier
```

An additional administrative interface that permits DBS components to register operators and their compatibilities, DBS instances etc. is required.

### 5.2.3 The Remote Communication Module

Object-oriented data communication approaches have come a long way. Microsoft DCOM and .Net, OMG CORBA, Sun Java/RMI and J2EE are used most commonly these days. While they are adequate for current needs of most application developers, more and more attention is drawn to *agent communication languages (ACLs)*. According to [75], ACLs stand a level above CORBA (and similar approaches). Among others, ACLs handle propositions, rules, and actions instead of simple objects with no semantics associated with them. In addition, agents are able to perceive their environment and may reason and act both alone (i.e. autonomy) and with other agents (i.e. interoperability) [120]. ACLs define the type of messages that agents exchange. However, agents do not just exchange messages; they have (task-oriented) 'conversations'. For instance, in a DBS they could first negotiate how to execute a given request most efficiently (i.e. query cost estimation) and then cooperatively process the chosen evaluation plan that implements the request most efficiently. ACLs are meant to equip agents with the ability to exchange more complex objects, such as shared plans and goals. At the technical level, agents still transport messages over the network using a lower-level protocol.

A remote communication module that is based on such an agent-based communication mechanism [67] will enable agents and, thus, different components of a DODBS to interact in order to execute user requests more efficiently.

The processing of evaluation plans involves a number of ODBS components. While the request evaluation engine controls the evaluation process, other components such as transaction management systems also require communication support. For instance, commit protocols and deadlock detection mechanisms are prime examples. Originally, we have intended to rely on a mixture of inter-process and thread communication mechanisms and an extended remote object call mechanism [140]. However, our experiences (with the first prototype system as briefly introduced in Section 6.1) have shown that this level of communication support is not sufficient. For instance, during the optimisation process an evaluation plan has to be selected. In order to do so, a number of potential plans are considered. Having negotiation and voting capabilities built-in the basic communication mechanism, such a process can be realised much more efficiently in contrast to more conventional communication approaches.

**The Database Agent Communication Language (DBACL).** DBACL is similar to KQML (Knowledge Query Manipulation Language) [41] in terms of separating communication aspects from the language or service interface that ODBS components expose or use internally. As a result, the communication between ODBS components is wrapped in a DBACL message. DBACL supports the following types of communication:

- Agent-to-Agent (AtA), i.e. one agent communicates with another agent;
- Agent-to-AgentList (AtAL), i.e. one agent communicates with a list of agents; and
- AgentBroadcast (AB), i.e. a broadcast message is sent to a list of interested agents.

Once the required communication type is known, the type of message transmission is to be decided. DBACL supports the following types of data transmission:

- All-At-Once (AAO), i.e. messages are only transmitted once they are fully assembled; and
- Stream, i.e. messages are transmitted in chunks. DBACL further distinguishes between upstream and downstream transmission.
  - Upstream transmission allows for task specifications to be provided in multiple messages. For instance, some arguments will be forwarded as they become available. The first message contains a description of how the remaining data is provided. Triggers are utilised here. The agent is then given a reference to a message queue that will be used to provide additional messages.
  - Downstream transmission supports that results are returned in chunks. Again, the agent is given a reference to a message queue that will be used to send result values (or other forms of reply messages). When using stream transmission, triggers have to be specified. A trigger tells an agent when to push or pull the next message. Push triggers forward data from the source to the destination. In turn, pull triggers rely on signals from the receiving end.

  (Software) pipelining is one of the main applications of stream-based communication.

Transmission types can be specified both ways, upstream and downstream. For instance, an agent can forward a task without providing any of the operational data. Operational data may then be pulled from the target-agent using triggers. This type of request is very useful for DBSs, e.g. for join operations. A remote REE can be instructed to prepare for the execution of a join operation while corresponding input values are still computed. Those values may then be pulled as required.

There are a number of pre-defined DBACL message types. These include signals, notifications, broadcasts, votes, negotiations, evaluation requests etc. For instance, request, vote and negotiation message types support the specification of pre-condition, post-condition, and completion-condition in addition to the message content. While triggers result in agents taking actions when some set of communication-related conditions are met, pre-conditions, post-conditions, and completion-conditions correspond to instructions that are forwarded in the language of the involved ODBS components. Thus, two levels of condition-based actioning are supported.

During the evaluation process, services of the remote communication module are utilised implicitly. Thus, we will not concern ourselves with a detailed description of this ODBS component.

## 5.3 The Execution of Evaluation Plans

Evaluation plans describe how the processing of user requests is to be performed. The main evaluation routine will examine the corresponding evaluation graphs, follow control flow edges and recursively execute all expressions that make up individual statements. During this process, services provided by other ODBS modules, such as the persistent object store, the transaction management system and the remote communication module are utilised.

In this chapter, we describe the evaluation of user requests that are formulated in the proposed iDBPQL language. First, we summarise the main challenges that have to be dealt with when designing such an evaluation procedure. Subsequently, a corresponding run-time environment is proposed. While we introduce a number of additional operators that mainly target run-time entities, we will also define additional primitives, which utilise functionalities provided by other ODBS modules. After having presented on overview of the evaluation process, we will discuss the processing of individual iDBPQL operators, keywords, expressions, statements, eval blocks and entire evaluation plans in greater details. Such processes or considered for local evaluations first. While we begin with the consideration of serial processing, it will not be long until support for internal multi-threading is added. The provision of a variety of implementation routines for each iDBPQL language construct enables code and query optimisers to better fine tune evaluation plans in a way that overall performance is enhanced. Besides serial and internal multi-threading, the execution of evaluation plans is discussed that involve explicit simultaneous processing as well as distribution.

### 5.3.1 Challenges

Main challenges that are encountered in this section include:

- *Definition of a suitable run-time environment.* Support of concurrent and distributed processing, orthogonal persistence and (distributed) transactions requires a run-time environment can accommodate the processing of multiple tightly coupled, loosely related or independent execution streams. As such, the run-time environment underlying the original SBA approach [131] is not suitable. Instead, such an environment must be adapted to that of modern OOPLs. For instance, we might consider the Java Virtual Machine environment [80]. However, run-time environments of OOPLs are no perfect match either since traditional database concepts such as transaction support and data persistence are not among the core issues that these languages are concerned with.
- *Refinements of the structure of stacks and their associated operations.* Enhancing performance capabilities by supporting different styles of processing affects the structure and usage of stacks that are assist with naming, scoping and binding, the storage of intermediate results and the passing of (intermediate) result values. If, for instance, we consider the desired support of pipelining, we must find a way to enable two simultaneous execution units to access both the head-end and the tail-end of a result pipeline. Such patterns of access are not common to stacks. To overcome this problem, we opted to refine the structure and usage of result stacks. Stacks no longer store result values directly. Instead, they hold queues of results, which may be accessed from both ends.
- *Linkage with other ODBS components.* In order to utilise services of other system components, additional primitives must be defined that hide certain concepts that are relevant to the particular ODBS components but not to the evaluation process.
- *Garbage collection.* Simultaneous processing requires that access to objects can be shared. Thus, we must maintain them in a central location and only place object

references onto stacks and queues. As a result, the run-time system has to provide a garbage collection mechanism, which ensures that valuable main memory space is not taken up by objects that are no longer required.
As previously mentioned, support for name-based access to all class instances complicates this garbage collection process. Only considering the existence of object references from environment stacks and result queues and other objects is not longer sufficient. However, the existence of name binders in addition with shallow and deep class extents allows us to refine the approach to garbage collection.
- *A proposal of operational semantics for iDBPQL constructs.* In chapter 4, we have proposed the syntax of iDBPQL. For each construct of this language corresponding evaluation routines have to be defined. While semantics for some basic operators, keywords and expressions can be derived from the SBQL proposal [131], iDBPQL is a much more complex language that processing in a concurrent and distributed database environment.
Support of user-types and a multitude of collection types, object-oriented concepts, implicit and explicit concurrency, deferred constraints, local and distributed transactions, object migration etc. only include a few of those concepts that have to be taken into account during the processing of user requests.

### 5.3.2 The Run-Time Environment

The simple two-stack-based run-time environment underlying the SBA approach [131] is not suitable for an environment supporting simultaneous processing. Instead, a shared memory area, *the heap*, that consists of the following components is used:

- *REE Stack Areas*: There exists exactly one REE stack area per user request. Within this stack area, the processing of the request's corresponding evaluation plans is performed. Each individual execution thread has an associated sub-area. At the beginning of the processing, the main evaluation plan will be associated with the first sub-area. Further sub-areas will be created as specifications for simultaneous processing are encountered or inter-operation concurrency is utilised. An REE stack area is discarded once all sub-areas are destroyed.
- A *Main-Memory Object Store*, which holds objects that are currently present in main memory. Such objects always correspond to transient objects. Local persistent objects, migrated objects or remote objects are accessible only through the embedded shared memory areas that is maintained by the persistent object store and the remote communication module, respectively.
- An *Evaluation Plan Area*, which is shared among all REE stack areas. It holds the collection of evaluation plans that are associated with behaviour specifications located in the run-time metadata area or evaluation plans that are associated with DBS metadata behaviour specifications. In the latter case, evaluation plans have been loaded from persistent storage and prepared for processing.
- A *Run-Time MetaData Area* that holds the run-time metadata catalogue.
- A *DBS MetaData Area* that holds the DBS metadata catalogue.

Figure 5.5 provides on overview of the composition of the local heap together with embedded shared memory area. Examples of values and objects and their references are outlined:

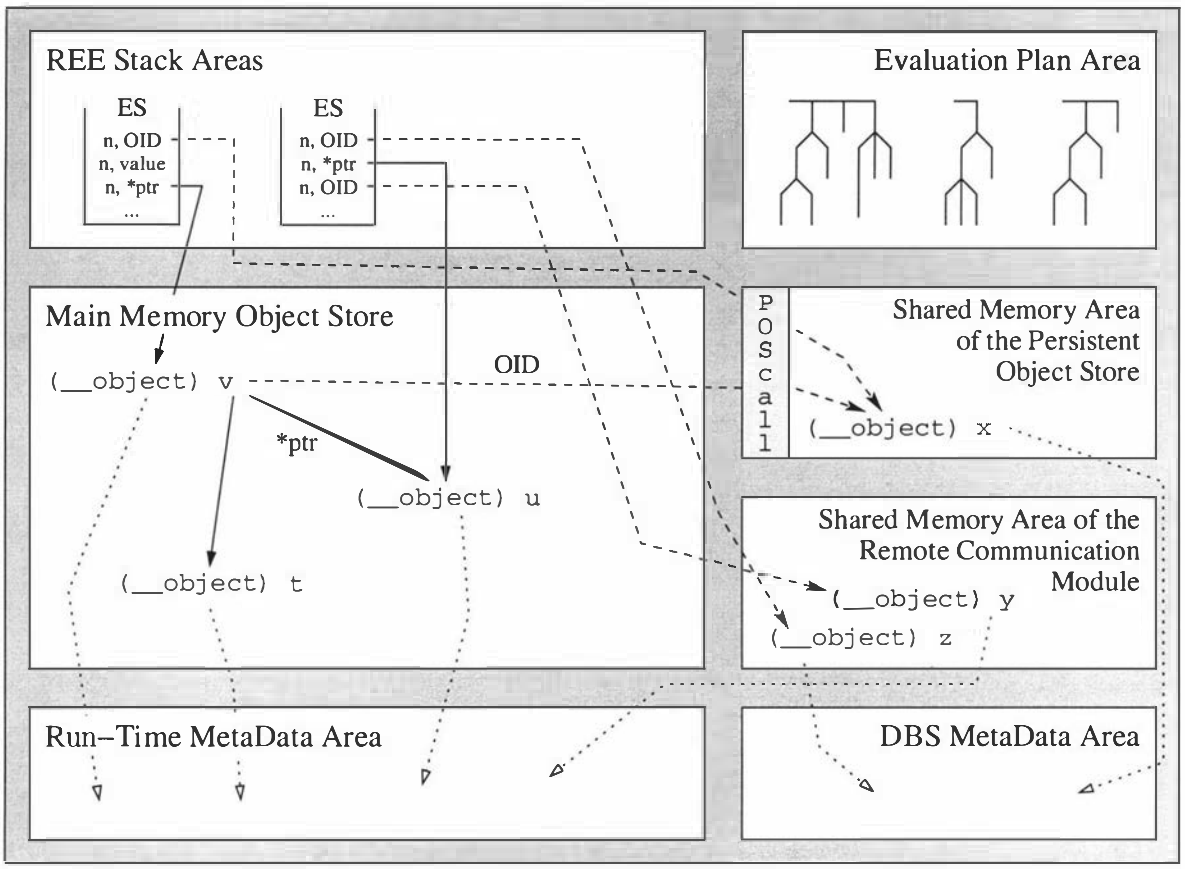


**Fig. 5.5.** Local Heap with Embedded POS and RCM Shared Memory Areas.

- Values reside on stacks and queues that are maintains in REE stack areas.
- Objects t, u and v are physically located in the local main memory object store and have associated run-time metadata entries. Object v is a local transient object, which contains three references: One to the transient object t, one to the transient object u and one to the persistent object x.
  References to objects, which are held in the main memory object store, from stacks and queues in local REE stack areas and references between objects in the main memory store are represented as main memory pointers.
- Object x is a local persistent object and made accessible in the shared memory area of the persistent object store. References appear in the form of object identifiers. Special POScall primitives, which facilitates access to persistent objects, are introduced further below.
- Objects y and z reside in the shared main memory area of the remote communication module, i.e. these objects are 'on loan' from remote ODBS instances. References appear in the form of object identifiers. Access to loaned transient and persistent

objects is enabled through the remote communication module. When objects are retrieved from other nodes, so are their metadata information. Object `y` is a transient object. Thus, its metadata information is added to the local run-time metadata catalogue. In contrast, object `z` is a persistent object that has its associated metadata information in the DBS metadata catalogue – a new `__schema` entry, which consists only of the necessary information that are required to process all objects on loan from this particular schema.

**The REE Stack Area.** As a new main evaluation plan is ready for processing, the first REE stack sub-area is created. This sub-area is the run-time component of our approach that is most similar to the two-stack abstract machine as defined in the SBA approach. Assuming that we only have one execution stream, the evaluation of iDBPQL code that makes up the main evaluation plan and all evaluation plans, which are invoked during processing, are executed in this sub-area. However, it is more common that simultaneous execution is utilised. Thus, multiple sub-areas exist within each REE stack area.

Every sub-area contains an *environment stack (ES)*, which consists of *frames* (in SBA they correspond to *sections*). As its name suggests, the environment stack represents the environment in which an evaluation plan is executed. Scoping and binding are the two main tasks performed on this stack.

The environment stack can be regarded as a collection of *name binders*. These binders can either appear as singletons or as collections. They associate external names with transient or persistent entities. A binder is a triple $(n, rt, e)$, where $n$ is an external name, $rt$ is its associated run-time type or class, and $e$ is a transient or persistent iDBPQL entity (e.g. an object (reference), a value, a variable, an evaluation plan etc.). If we deal with a transient entity, $e$ is a pointer to a memory area within the heap (recall the use of pointer swizzling techniques to enhance performance). In contrast, local persistent entities, which are made accessible in the shared memory area of the persistent object store, and migrated objects or objects located on remote ODBS instances are referenced using object identifiers (i.e. $e$ is a value of type `__OID`).

**Definition 5.5.** A *name binder* (of internal type `__binder`) is a triple $(n, tr, e)$ with the following properties:

- $n$ is a String value of type `(char *)`. It represents an external name, which is bound to the entity $e$.
- $tr$ is a reference a `__typeInfo` or `__classInfo` structure that identifies the run-time type or class, respectively, of the entity $e$.
- $e$ is an iDBPQL entity. The internal type and the value of this entity is as follows:

$$e = \left\{\begin{array}{lll} \texttt{\_\_OID} & \texttt{\_\_oid} & \text{if referencing a persistent or remote object;} \\ \texttt{\_\_object *} & \texttt{\_\_mmObject} & \text{if referencing an object in the heap; and} \\ \texttt{\_\_iDBPQLvalue} & \texttt{\_\_value} & \text{if holding a simple, structured, collection-type,} \\ & & \quad \text{or } \texttt{NULL}\text{able value.} \end{array}\right.$$

□

Access to information that is captured by name binders is possible through the . (dot) operator. The identifying name, run-time type or bound entity of the top-most name binder on ES may be retrieved by executing `top ( ES ).n`, `top ( ES ).tr` or `top ( ES ).e`, respectively.

The environment stack is divided into frames and sub-frames, which help with scoping and binding. With every behaviour invocation, a new frame is created. Similarly, whenever a new evaluation block is encountered, a new sub-frame is created. Frames and sub-frames group all run-time entities that are local to the respective behaviour implementation or evaluation block, respectively. During the process of binding, the top-most sub-frame (i.e. the most local (sub-)environment) is considered first. In the event that a name binder is not found, the next sub-frame or frame[4] is visited. This approach is continued until the bottom of the stack, which describes the global environment, is reached. The evaluation of any request that is formulated in iDBPQL will always be able to locate a binder on ES. Otherwise, the evaluation plan together with all annotations and references is not well formed and should have been rejected by the compiler.

Frames have a second stack associated, the *result stack (RS)*. This is different to the SBA approach, which only defines one global result stack that holds results in the form of tables. Having a result stack associated with a particular invocation enables the sharing and re-use of result values more easily. For instance, if we encounter a sequence of identical invocations of a static method, we may perform the computation once, retain the result and then share it among all invocations. Nevertheless, the purpose of RS remains largely unchanged. It stores intermediate results and assists with passing of results between frames. RS stores intermediate results in the form of *result queues (RQs)*. This is necessary to better support pipelining and simultaneous and distributed processing. In addition, supporting RQs also allows for a more refined approach to how results are represented. While the SBA approach restricts results to a representation that corresponds to a table (or bag), we support the storage of results in the form of singletons or different types of collections (i.e. as a bag, set, list or array). Result queues are associated with evaluation steps. Each such step may have one or more corresponding result queues. These queues are used to exchange result values, store intermediate results or synchronise different forms of processing.

**The Environment Stack (ES).** In accordance with traditional programming languages and the SBA approach, (references to) run-time entities that are available at a given point in time during the evaluation procedure are maintained on the environment stack. The availability of these entities is determined by their appearance in the respective evaluation plan and a set of scoping rules. The latter adhere to the following principles:

1. A local name is given priority over an inherited, static or global name;
2. The local context of the implementation of a behaviour specification is hidden from other evaluation plans the former one invokes; and
3. Nesting of run-time entities is not restricted.

[4] While the next sub-frame is always the previous sub-frame, the same is not true for frames. To facilitate the hiding of local contexts, certain frames are skipped. Corresponding details are outlined below.

The first principle outlines that name binding results in a search starting from the top of ES. In the event that the name is located in the top sub-frame of ES, binding terminates successfully. Thus, the most local entity is bound to the given name. If the name is not found, the search continues with the next sub-frame and so on until all sub-frames of the top-most frame are considered. While sub-frames are never skipped, the same does not apply to frames themselves. The second principle advocates that entities, which are local to a particular implementation, must be hidden from the view of other implementations. This relieves programmers from knowing details of implementations of behaviour specifications that they utilise. As a result, frames are linked by ***prevScope** pointers*, which chain those frames together that may access one another's local variables. In the event that a name cannot be located in a particular frame, the search continues in the next frame encountered along the chain formed by the associated prevScope pointers.

Frames and sub-frames on the environment stack are nested according to block and behaviour invocation specifications in all evaluation plans that are encountered during the processing of a user request. The third principle implies that there is no restriction on the depth of the nesting of these block specifications and behaviour invocations.

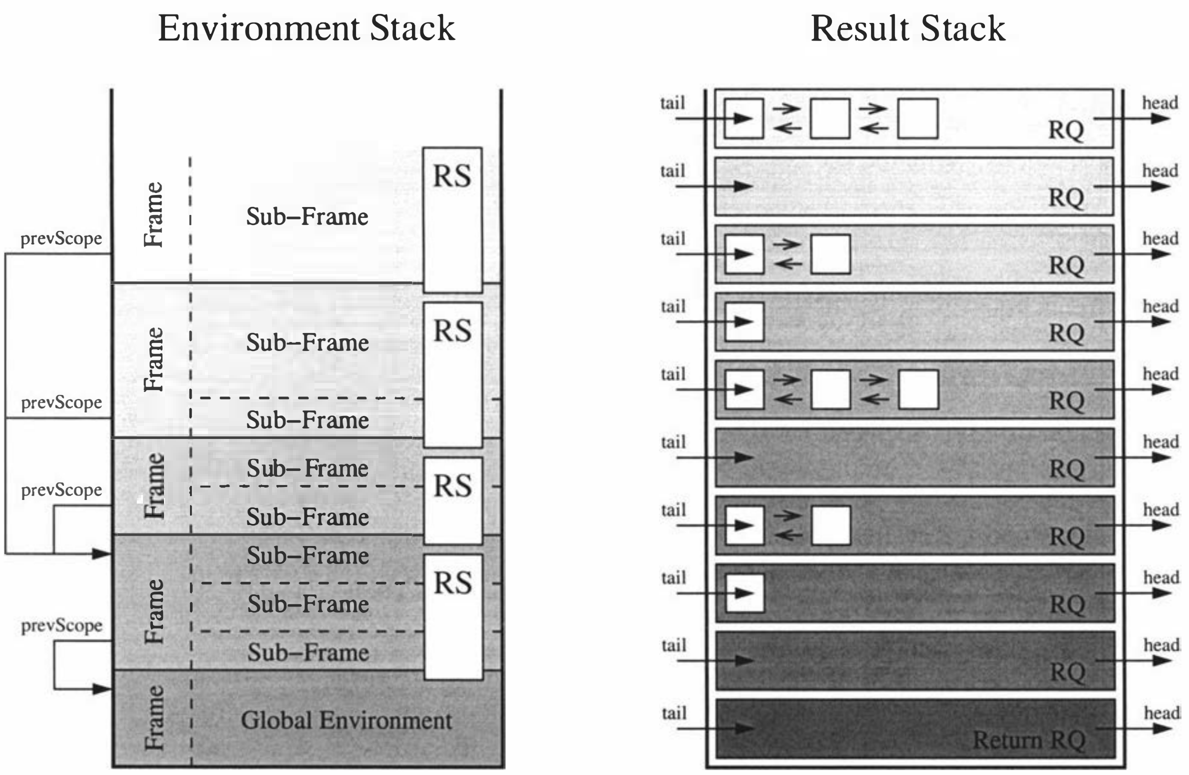


**Fig. 5.6.** Logical View of the Environment and Result Stacks.

Figure 5.6 (left-hand side) provides a logical view of the composition of the environment stack. At the bottom of the stack, there is the global environment. This includes definitions that are imported from the DBS metadata catalogue, type and class definitions associated with the run-time metadata catalogue and other definitions that are global to the respective user request. The frame above the global environment corresponds to the main evaluation plan. The third frame from the bottom holds entities that are local to the evaluation plan, which has been invoked during the processing of the main evaluation plan. Similarly, the fourth and fifth frames (counting from the bottom) hold entities that are local to the corresponding evaluation plan, which has been invoked during the processing of the behaviour implementation described in the frame below. In accordance with the second principle, `prevScope` pointers, which ensure that local implementation entities remain hidden, are outlined.

Additional pointers are associated with frames. For instance, a pointer that keeps track of the corresponding `THIS` object will be added. Corresponding additions are motivated and outlined throughout the remainder of this chapter. The common rationale behind the usage of additional pointers is mainly related to performance considerations.

As mentioned earlier, a new frame is created with every behaviour invocation. That is, a new scope is opened. Parameters supplied during the invocation procedure are maintained in the frame itself. During the evaluation process, new sub-frames are established whenever a new evaluation block is encountered. Since every evaluation plan consists of at least one evaluation block, every frame has at least one sub-frame. Variables local to a particular evaluation block are maintained in the respective sub-frame on ES.

**The Result Stack (RS).** A result stack is associated with each frame, i.e. with each behaviour invocation. Intermediate results as well as the behaviour's return value are maintained on RS. In fact, each result stack can be regarded as a stack of result queues as outlined in Figure 5.6 (right-hand side). The result queue at the bottom of RS serves a special purpose. We also refer to it as *return result queue*. It facilitates the exchange of values that are returned as the result of a behaviour's invocation. The return result queue is always present on RS except for frames that correspond to behaviour implementations with no return type (e.g. object constructors) or with the `VOID` return type. All other result queues are associated with the evaluation of individual iDBPQL statements and expressions.

The size of a result queue is dynamic. Memory is allocated from the heap. In order to enable two evaluation procedures to exchange result values, the result stack and its result queues are created, maintained and accessed as follows:

- Upon the invocation of a behaviour implementation, a new scope is opened. That is, a new frame is placed on top of the environment stack. During this process, a result stack is initialised. If the behaviour's return descriptor is empty or of type `VOID`, then there is no return result queue and the initialisation of RS is complete. Otherwise, a return result queue is pushed onto RS[5].
- A new result queue is created implicitly with every sub-evaluation or, in the event that the local evaluation procedure requires storage space for intermediate results, it is created explicitly.

[5] The return result queue is always initialised by the evaluation process that invokes the behaviour. This way, the calling process retains access to the result queue even after the invoked behaviour has terminated.

- Each result queue has two handles that regulate access associated. On one hand, results can be appended to or, in some cases, accessed from the end of the queue, the *tail*. On the other hand, results can be accessed from the front of the queue, the *head*.
  If the result queue was created explicitly, both means of access are available from the current evaluation procedure. Otherwise, the evaluation procedure that invokes a behaviour retains the read-only access to the front of the queue. However, write access by means of appending result values is only available to the procedure that evaluates the behaviour implementation or the sub-evaluation routine, respectively.
- In order to support pipelining of results, a means of synchronisation between the corresponding two evaluation procedures is required. This is enabled by means of a `status` operator and a special result value that marks the end of the pipelining of result values.

As we will see next, result queues may be accessed using both stack operators and queue operators. In the event that a stack operator is invoked upon a result queue, the queue is treated as a stack. For implicitly defined queues, there is only one end of the queue made accessible to the particular evaluation routine. This available end is regarded as the top of the (queue-)stack. Otherwise, if both ends are accessible to the evaluation routine, the tail-end is considered as the top.

RQs have the ability to hold different types of results. For instance, the result of an evaluation process may just be a simple atomic value, the `NULL` value, a structured value, an object identifier, a main memory reference, a bag, a set, a list or an array. In order to determine the type of a result queue, a header is associated with every queue. The format of a RQ header is as follows:

- `type` ... a reference to a `__typeInfo` or `__classInfo` structure held in the run-time or DBS metadata catalogue;
- `pipe` ... a Boolean value that indicates whether or not result values are pipelined;
- `eAccStep` ... a Natural value that corresponds to an offset value. Such offset values are utilised by the `eAccess` array, which speeds up access-by-position to lists and arrays (a value of 0 implies that there is no such support); and
- `eAccess[]` ... an array of pointers to elements held on the result queue. The first pointer refers to the `eAccStep`th element, the second pointer to the $2 *$ `eAccStep`th element and so on. The `eAccess` array is used for lists and arrays to implement access-by-position more efficiently and also to assist with the evaluation of operations such as searching, sorting etc.

Access to information maintained in a RQ's header is supported through the '.' (dot) operator. Queue elements are maintained according to the respective collection type. If, for instance, RQ holds a set of object references, it is ensured that no object is referenced twice from RQ.

**Operations on Stacks and Queues.** ES, RS and RQ support most of the common operations defined on stacks (i.e. `empty`, `pop`, `push` and `top`) and queues (i.e. `head`, `tail`, `prev`, and `next`). Furthermore, queues may be modified through the following operators:

`moveInfront`, `moveBehind`, `swap`, `cutInfront`, `cutBehind`, `append` and `merge`. Details are as follows:

- `void append (` $q_1$`,` $q_2$ `)`, where $q_1$ and $q_2$ are pointers to result queues. The `append` operator simply adds all entities from $q_2$ to the tail of $q_1$.
- $q_2$ `* cutBehind (` $q_1$`,` $e$ `)`, where $q_1$ and $q_2$ are result queues and $e$ is a pointer to an entry on $q_1$. The `cutBehind` operator splits the given queue $q_1$ in two parts: All elements in-front of $e$ remain on $q_1$. Entry $e$ and all following entries are moved to the new queue $q_2$ of identical format. The return value of the `cutBehind` operator is a pointer to the newly created result queue.
- $q_2$ `* cutInfront (` $q_1$`,` $e$ `)`, where $q_1$ and $q_2$ are result queues and $e$ is a pointer to an entries on $q_1$. Similarly to the `cutBehind` operator, the `cutInfront` operator splits the given queue $q_1$ in two parts. This time, however, all elements following entry $e$ stay behind. Entry $e$ and all preceding entries are moved to the new queue $q_2$ and a pointer to this queue is returned as result.
- `boolean empty (` $s$ `)`, where $s$ is either ES, RS or RQ. The `empty` operator tests whether or not the given stack or queue is empty.
- $e$ `head (` $q$ `)`, where $q$ is a result queue and $e$ is an entry on $q$ similar to the `top` operator (with the exclusion of name binders). The `head` operator returns the entry $e$ at the front of $q$.
- $q_3$ `* merge (` $q_1$`,` $q_2$ `)`, where $q_1$, $q_2$ and $q_3$ are result queues. The `merge` operator initialises a new queue $q_3$ with the contents of $q_1$, appends $q_2$ and then returns a pointer to $q_3$.
- `void moveBehind (` $q$`,` $e_1$`,` $e_2$ `)`, where $q$ is a result queue and $e_1$ and $e_2$ are pointers to entries on $q$. The `moveBehind` operator takes the entry $e_1$ from queue $q$ and places it right behind entry $e_2$.
- `void moveInfront (` $q$`,` $e_1$`,` $e_2$ `)`, where $q$ is a result queue and $e_1$ and $e_2$ are pointers to entries on $q$. The `moveInfront` operator takes the entry $e_1$ from queue $q$ and places it right in-front of entry $e_2$.
- $e_2$ `* next (` $q$`,` $e_1$ `)`, where $q$ is a result queue and $e_1$ and $e_2$ are pointers to entries on $q$. The `next` operator returns $e_2$ that is the entry right behind $e_1$.
- $e$ `pop (` $s$ `)`, where $s$ is either ES or RQ and $e$ is an iDBPQL entity. The `pop` operator removes the top-most element from $s$ and returns the associated value $e$.
- $e_2$ `* prev (` $q$`,` $e_1$ `)`, where $q$ is a result queue and $e_1$ and $e_2$ are pointers to entries on $q$. The `prev` operator returns $e_2$ that is the entry right in-front of $e_1$.
- `void push (` $s$`,` $e$ `)`, where $s$ is either ES or RQ and $e$ is a name binder or an iDBPQL entity, respectively. The `push` operator places $e$ on top of $s$.
- `void swap (` $q$`,` $e_1$`,` $e_2$ `)`, where $q$ is a result queue and $e_1$ and $e_2$ are pointers to entries on $q$. As its name suggests, the `swap` operator swaps the two queue entries $e_1$ and $e_2$.
- $e$ `tail (` $q$ `)`, where $q$ is a result queue and $e$ is an entry on $q$ similar to the `head` operator. The `tail` operator returns the entry $e$ at the end of $q$.
- $e$ `top (` $s$ `)`, where $s$ is either ES, RS or RQ and $e$ is an entry on $s$, e.g. a name binder, a collection of name binders, an iDBPQL value, a collection of object identifiers etc. The `top` operator returns the top-most element from $s$.

When invoked on ES, the `top` operator accesses name binders. However, such binders are never returned. Instead, the `top` operator only returns the associated value $e$.

In addition, a number of operators that assist with scoping and binding are defined. The basic `bind`, `openScope` and `closeScope` operators stem from SBA and have been refined to suit the more complex run-time environment. Corresponding operators are as follows:

- $p$ `* bind (` $n$ `)`, where $n$ is a name of type `(char *)` and $p$ is a main-memory pointer to an entry on ES. The `bind` operator searches ES until it locates the first entry that has a name value, which matches $n$. The search commences from the top of ES and is governed by the three scoping principles outlined above, i.e. `prevScope` pointers are followed.
- $p$ `* bindNext (` $n$ `)`, where $n$ is a name of type `(char *)` and $p$ is a main-memory pointer to an entry on ES. The `bindNext` operator complements the `bind` operator. In contrast to the `bind` operator, the `bindNext` operator does not necessarily commence its search from the top of ES. It continues from the most recently bound entry on ES that has a matching name value. In the event that there is no such bound entry, search commences from the top of ES. Again, the search follows the three scoping principles outlined above.
- $p$[] `* bindAll (` $n$ `)`, where $n$ is a name of type `(char *)` and $p$ is an array of main-memory pointers to entries on ES. The `bindAll` operator searches ES until the bottom entry is reached. Finally, it returns pointers to all encountered entries that have a name value, which matches $n$. Same as both previous binding operators, the search follows the three scoping principles outlined above.
- $p$ `* bindCrsNext (` $n$ `)`, where $n$ is a name of type `(char *)` and $p$ is a main-memory pointer to an entry on ES. The `bindCrsNext` operator is similar to the `bindNext` operator but its visibility is restricted to the first collection of entries on ES that has a matching binder. The rationale behind this restriction can be explained by considering the mechanism this operator is meant to support. In order to realise a cursor on a collection object, the `bindCrsNext` operator retrieves one collection member after the other. It terminates once all members of this particular collection have been visited.
  Let us assume that the same collection object is unnested twice and that with each unnesting, the same external name has been assigned. Subsequently, a loop through this collection is evaluated. The `bindCrsNext` operator ensures that only the top-most unnested collection object is considered. As a result it is avoided that each collection member is accessed more than once when implementing a cursor.
- $p$ `openNewScope (` *args*[] `)`, where *args* is an array of arguments supplied to the respective behaviour invocation and $p$ is a main-memory pointer to the head of a return RQ on RS. The `openNewScope` operator places a new frame on top of ES and the `prevScope` pointer is adjusted accordingly. The frame itself is initialised and name binders for the invocation's arguments *args* are pushed onto ES. During this process, a result stack is associated with this frame and a return RQ is initialised as applicable. The result of the `openNewScope` operator is a pointer to the head of

the return RQ or, in the event that no such queue is required, the `(null)` pointer is returned.

- `void closeScope ( void )`. The `closeScope` operator removes the top-most frame from ES. It signals the end of the evaluation of a behaviour implementation. Only the return RQ remains accessible from the frame, which invoked the behaviour that resulted in the creation of the now closed frame.
- `void openNewSubScope ( char * name, char * transFlag )`. The `openNewSubScope` operator places a new sub-frame on top of ES. If a `name` argument is supplied, a named sub-frame is created. Named sub-frames are introduced to better support the evaluation of loop and switch statements in combination with the `BREAK` statement.
  The `transFlag` argument sets a transaction flag to the specified value. Transaction flags assist with the monitoring of operations performed by individual transactions. Initially, the value of this flag must be initialised to `"__none"`, which indicates that no transaction is active. Section 5.3.7 will later introduce how this transaction flag is utilised. In the meantime, all calls of the `openNewSubScope` operator will simply pass on the value that is set for the current sub-frame.
- `void closeSubScope ( void )`. The `closeSubScope` operator removes the top-most sub-frame from ES. It signals the end of the processing of an evaluation block.

Result queues require a number of maintenance operations that address the release and destruction of queues as well as operations that assist with the monitoring of their status:

- `void release (` *q* `)`, where *q* is a RQ. The `release` operator relinquishes the evaluation routine's access rights to *q*. In the event that the queue *q* is not accessible from any evaluation procedure, it is destroyed and its memory is returned to the heap's free memory area.
- `short status (` *q* `)`, where *q* is a RQ. The `status` operator returns a constant value from a pool of pre-defined queue states. The following constants are supported:
  - `FILLED` $\cong$ the result queue contains one or more values.
  - `EMPTY` $\cong$ the result queue is empty but further values may be added.
  - `END` $\cong$ the result queue is empty and no further values will be added, e.g. the sub-evaluation has terminated.
- `waitOn (` *q* `)`, where *q* is a RQ. The `waitOn` operator monitors the specified request queue *q*. It halts processing of the current evaluation procedure until the status of *q* changes from `EMPTY` to any other status.

**Initialising Result Queues.** Due to the more complex nature of result queues, we require a special means of initialisation. As previously mentioned, a metadata reference is associated with every result queue. Accordingly, elements of the queue are maintained. The declaration of the format of a result queue can be achieved using either of the following two alternatives:

- An iDBPQL system type can be specified explicitly; or

– A special primitive `__returnDescriptor rtype ( subEvalEdge * edge )` extracts the expected return type information from the current evaluation plan. Instead of an sub-evaluation edge and a return descriptor, we may also specify the argument in the form of an expression that corresponds to the sub-evaluation edge and receive a pointer to the metadata entry that is synonym to the return descriptor.

**Binding Behaviour Names to Evaluation Plans.** In addition to binding names to values, a mechanism that binds the name and arguments of a behaviour invocation to the corresponding evaluation plan is required. This binding process will involve the environment stack as well as entries in at least one metadata catalogue. The signatures of respective binding routines are as follows:

```
__evalPlan * bindTypeOpEvalPlan ( __typeInfo * rttype, char * name,
  __iDBPQLvalue * args[] );                         // returns the evaluation plan that
          // corresponds to the specified type operation invocation on value val
__evalPlan * bindMethodEvalPlan ( __object * obj, char * name,
  __iDBPQLvalue * args[] );                         // returns the evaluation plan that
                 // corresponds to the specified method invocation on object obj
```

The implementation of the `bindTypeOpEvalPlan` routine is based on the run-time type associated with the value on which the type operation was invoked and the arguments provided during the invocation.

Analogously, the implementation of the `bindMethodEvalPlan` routine considers the run-time class of the object on with the method was invoked and the arguments provided during the invocation. Similar to modern OOPLs, this method invocation mechanism follows the dynamic (single) dispatch approach.

### 5.3.3 The `SYSTEMcall`, `POScall` and `TMScall` Primitives

In addition to stack operations, we utilise three additional primitives during the evaluation process. These primitives allow the invocation of pre-defined routines and unary and binary operators supplied by the underlying system language, service interface operations of the persistent object store (refer to Section 5.2.1) and service interface operations of the transaction management system (refer to Section 5.2.2).

**The `SYSTEMcall` Primitive.** Common type operators such as arithmetic and logical operators and pre-defined routines that operate on additional main-memory structures may be invoked through the `SYSTEMcall` primitive. The signature of this primitive is defined as follows:

`void SYSTEMcall_`*sysOpCode* `( RQ * queue, void * args[] );`

where *sysOpCode* is either a pre-defined procedure name or an operator defined in the underlying system. The argument list `args[]` depends upon the respective procedure or operator that is invoked.

For example, operators that are utilised from the underlying system include those outlined in Table 4.2 (on page 64). A number of pre-defined procedures, which have been introduced earlier in this chapter, may be invoked. Such procedures include the `getSubClasses`, `getSuperClasses`, `isSubClassOf`, and `isSuperClassOf` routines associated with the inheritance graph structure `__dag`.

**The `POScall` Primitive.** The service interface of the persistent object store (refer to Section 5.2.1) supports a variety of operations on storage objects. To utilise any of these available POS operations, we introduce a special primitive:

```
void POScall_posOpCode ( RQ * queue, void * args[] );
```

where `posOpCode` is a service routine name as defined in the POS interface. The number and type of the expected arguments depends on the specified operation code.

The `POScall` primitive abstracts from more physical concepts that have to be dealt with when using operators as outlined in the service interface of POS. For instance, the usage of object store access control blocks and mappings of storage objects to stack / queue objects (and vice versa) are hidden from evaluation procedures. Corresponding details of the implementation of this primitive are omitted.

**The `TMScall` Primitive.** The service interface of the transaction management system (refer to Section 5.2.2) allows for the monitoring of transactions. In particular access, creation and manipulation of shared objects is ensured to be serialisable and recoverable. To utilise any of the TMS interface operations, we introduce a special primitive:

```
void TMScall_tmsOpCode ( RQ * queue, void * args[] );
```

where `tmsOpCode` is a service routine name as defined in the interface of the transaction management system. The number and type of the expected arguments depends on the specified operation code.

Similar to the `TMScall` primitive, the `TMScall` primitive abstracts from a number of conversion tasks. For instance, the `execute` routine defined in Section 5.2.2 expects an operator name and an array of objects on which this operator executes. A corresponding `TMScall` primitive will only require a result queue and an evaluation node as arguments. The result queue is used to return the reply to the request and the evaluation node is used to extract all necessary arguments to translate a `TMScall_execute` call into the corresponding TMS `execute` call. Corresponding details of the implementation of this primitive are omitted.

### 5.3.4 Overview of the Evaluation Process

The evaluation of a user request is based upon the corresponding evaluation graph. Annotations, metadata references and services provided by other ODBS components play an important role during the evaluation process. Before we consider the main evaluation procedure in greater detail, we will first outline the internal format of machine instructions and then consider a routine that supports the unnesting of references, collections and values.

**Machine Instructions.** Each iDBPQL statement and expression has one or more different stack-based implementations associated. These implementations are identified by unique machine instructions or operation codes (of internal type `__opCode`). Machine instructions are specified as annotations (refer to Section 5.1.5) and encode additional information:

- The first two characters refer to the type on which the respective machine instruction may operate. Table 5.4 outlines a mapping of iDBPQL types to their internal two-letter representation.
  For instance, an operation code ov*xxx* implies that it can be applied to all ordered values, i.e. operation codes ch*xxx*, na*xxx*, in*xxx* and re*xxx* are given implicitly (where *xxx* represents the remaining portion of the operator code).
- At the core or centre of the machine instruction, there is the basic operator name. It appears in upper-case and may include an operator symbol.
  For example, the operation code cvWHERE*yyy* corresponds to a selection operator that may be performed upon collections or identifiers representing collections (where *yyy* is the suffix of the operator code). In contrast, a machine instruction that includes an operator symbol is ovUOP++. This code represents the unary post-increment operator that may be applied to any ordered value. The ovUOP++ instruction does not have a suffix.
- An optional suffix (in lower case letters) indicates which particular implementation is to be used.
  For instance, both the machine instructions cvFORANYet and cvFORANYpet refer to the generalised *AND* operator that are applicable to collection values. The latter implementation utilises pipelining (as indicated by the letter p) while the former processes in serial manner.

| **Abstract iDBPQL Type** | **Two-Letter Representation** |
|---|---|
| atomic_value | av |
| collection_value | cv |
| discrete_value | dv |
| numeric_value | nv |
| ordered_value | ov |
| reference_value | rv |
| structured_value | sv |

| **Other Abstract iDBPQL Types** | **Two-Letter Representation** |
|---|---|
| *non*-reference_value | nr |
| nullable values | nu |
| type identifiers | ti |
| class identifiers | ci |
| independent operators | xx |

| **iDBPQL Type** | **Two-Letter Representation** |
|---|---|
| ARRAY < > | a< |
| BAG < > | b< |
| BOOLEAN | bo |
| CHAR | ch |
| CLASSDEF | c< |
| EMPTYLIST | el |
| EMPTYSET | es |
| INT | in |
| LIST < > | l< |
| NATURAL | na |
| NULLABLE < > | n< |
| REAL | re |
| SET < > | s< |
| STRING | st |
| TYPEDEF | t< |
| UNIONDEF | u< |
| VOID | vd |

**Table5.4.** Type Information Mapping to Machine Instructions.

**Unnesting Objects and Values.** As we have outlined previously, the environment stack captures run-time entities that are available at a given point in time during the evaluation procedure. Run-time entities are represented in the form of name binders. While we have introduced concepts to access, manipulate and maintain entities on ES, a mechanism to create name binders is still required. To bridge this gap, the `unnest` operator, which corresponds to the `nested` operator in the SBA approach, is introduced next. This operator pushes object references and values onto ES and ensures that they are easily identifiable and accessible by binding these values and references to names.

In addition to creating name binders, the `unnest` operator enables us to move deeper into an object's structure, extract members from collections or resolve object identifiers / in-memory references. The `unnest` operator is defined for values and object identifiers as follows:

`unnest ( x, xName )` =

| | |
|---|---|
| `EMPTYSET` | *if* `x` *is a value-typed* `__iDBPQLvalue` *that resides on ES*; |
| `( n, rt, v )` | *if* `x` *is an* `__iDBPQLvalue` *of type* `rt` *and* `x` *resides on RS and has an associated name* `n` *(e.g. denoting a remote or persistent object, which is identifier by its OID)*; |
| `( n, rt, (__object *) o )` | *if* `x` *is a reference-type* `__iDBPQLvalue` *of the form* `(__OID) v` *or* `(__object *) v` *that identifies object* `o` *with associated (run-time) class* `n`; |
| ( $\mathtt{n}_1$, $\mathtt{rt}_1$, $\mathtt{v}_1$ ), ..., ( $\mathtt{n}_j$, $\mathtt{rt}_j$, $\mathtt{v}_j$ ) | *if* `x` *is a structured reference-type* `__iDBPQLvalue` *of the form* ( $\mathtt{v}_1$, ..., $\mathtt{v}_j$ ) *with members named* $\mathtt{n}_1$, ..., $\mathtt{n}_j$ *(e.g. denoting the value of* `__object o` *that has members named* $\mathtt{n}_1$, ..., $\mathtt{n}_j$*)*; |
| ? ( n, $\mathtt{rt}_1$, $\mathtt{v}_1$ ), ..., ( n, $\mathtt{rt}_j$, $\mathtt{v}_j$ ) ? | *if* `x` *is a reference-type* `__iDBPQLvalue` *of the form* ? $\mathtt{v}_1$, ..., $\mathtt{v}_j$ ? *where* ? *denotes a bag, set, list or array collection with an associated name* `n` *or, if* `xName != NULL`, `n` *is* `xName`; *and* |
| { ( x, $\mathtt{rt}_1$, (__object *) $\mathtt{o}_1$ ) ..., ( x, $\mathtt{rt}_j$, (__object *) $\mathtt{o}_j$ ) } | *if* `x` *is an identifier of a collection-class with objects* $\mathtt{o}_1$, ... $\mathtt{o}_j$. |

If the `unnest` operator is invoked on a value that is fully unnested and already resides on ES, no action is taken. If the value does not yet reside on ES, a corresponding binder is added. This is necessary since collection values are returned individually when pipelining is utilised. Otherwise, object references, structured values and collections are unnested by placing binders for their corresponding structural members or collection members, respectively on ES. In the event that a collection-class identifier is encountered, a set of binders for all collection members (including those of sub-classes) are added to ES. This means that access to remote ODBS instances, the local persistent object store and deep extents that are associated with run-time classes may be necessary.

Besides unnesting iDBPQL values and identifiers, the support of collection classes requires their class identifiers to be represented on the environment stack. Thus, the `unnest` operator is extended to extract the respective information from entries in the

DBS and run-time metadata catalogues:

```
unnest ( x, xName ) = ( x.classes[1].__name, x.classes[1], r1 ), ...,
      ( x.classes[x.__classCount].__name, x.classes[x.__classCount], rx.__classCount )
```

where x represents a `__schemaInfo` or `__rtEntryInfo` structure, and `xNAME` is `NULL`. The actual number of binders is smaller or equal to `x.__classCount` since binders are added only for collection-classes. Concrete and abstract classes do not require their names to be present on ES.

In addition, the `unnest` operator may be applied to the whole DBS metadata catalogue. Let `x` represent the collection of all persistent database schemata, i.e. the `__dbsMetaDataCatalogue` structure. Then, the unnesting is executed by invoking `unnest ( __dbsMetaDataCatalogue.__schemata, NULL )`. For efficiency reasons, this unnesting step is outsourced to the optimiser. A list of binders that associates each schema's name with an internal reference to the respective `__schemaInfo` structure is maintained. Accordingly, textual references to schema names are replaced by their internal references. Thus, the global environment on ES does not need to include a list of all known schemata, only those it requires during processing.

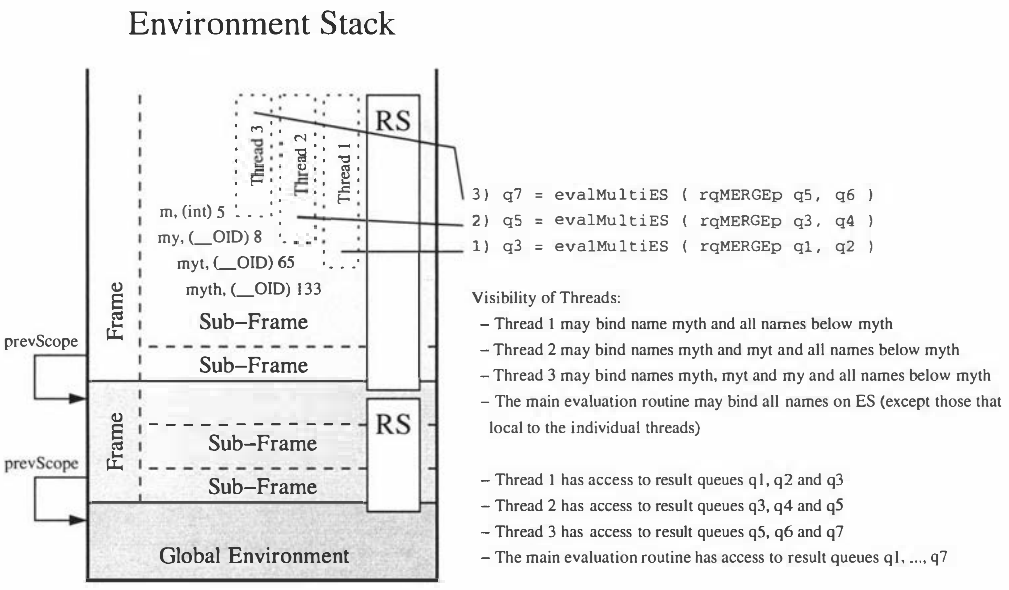
As previously mentioned, there are two internal representations of references. On one hand, we have main memory pointers, which are applicable only to iDBPQL entities that reside in the local heap. On the other hand, object identifiers are used to represent references to objects that do not reside in the local heap. Instead, such objects may reside on an external storage device, in the object store's main memory pool or on a remote ODBS instance. A collection of values of a reference-type may consists of a mixture of both types of references. Accordingly, the implementation of the unnesting routine must be able to deal with the following four cases:

- Access by main memory pointers. This is the most efficient means of access. We only have to follow the pointer to obtain the respective object.
- Access by OID with a corresponding `__classInfo` structure that resides in the local DBS metadata catalogue. Thus, the referenced object resides on a local, external storage device or in the local object store's main memory pool. In either case, object access is governed by the object store and requires the usage of the special `POScall` primitive as introduced above. Access by OID is translated into direct access using POS's `Retrieve` operation.
- Access by OID with a corresponding `__classInfo` structure that originates from a remote ODBS instance. This situation is encountered when accessing distributed objects. Corresponding details are discussed in Section 5.3.10. In short, unnesting a remote object will result in the object being (temporarily) migrated to the local ODBS node. Migrated objects are maintained in a separate object store, which is part of the remote communication module.

**The Main Evaluation Routine.** The semantics of iDBPQL statements, expressions and various keywords can be expressed in a number of ways, e.g. in the form of axiomatic, denotational or operational semantics. Axiomatic semantics are based on a formal logic, in particular the first order predicate calculus. Denotational semantics are based on recursive function theory and can be considered as the most abstract semantics description method. Operational semantics provide meaning of keywords, operators, expressions, statements, type operations and methods in terms of their implementation on a real machine [116, 137]. Having practicability in mind, we adopt the latter approach and present the meaning of iDBPQL statements, expressions and various keywords in the form of operational semantics.

From an operational perspective, the main evaluation routine can be regarded as a recursive execution unit that takes a syntactical iDBPQL entity as its argument. An evaluation routine operates on an evaluation plan and processes according to the information associated with the plan's edges and nodes. As we have seen already, an evaluation plan is given in the form of a graph where directed edges represent the execution flow and undirected edges indicate sub-evaluations. The style of evaluation can be described as depth-first traversal where undirected edges are given priority over directed edges. During evaluation procedures, side effects that affect ES, RS or RQ may occur. While some of these side effects only impact on local transient data, other side effects affect shared data. The latter requires particular attention in a DB environment where data consistency has to be enforced.

iDBPQL supports different styles of processing. Accordingly, different means of evaluation routines are supported. Serial execution, internal multi-threading, user-enforced concurrency and distributed processing are utilised. As a result, there is one main evaluation routine that has three main sub-routines. Corresponding details are as follows:



```
3) q7 = evalMultiES ( rqMERGEp q5, q6 )
2) q5 = evalMultiES ( rqMERGEp q3, q4 )
1) q3 = evalMultiES ( rqMERGEp q1, q2 )
```

Visibility of Threads:
- Thread 1 may bind name myth and all names below myth
- Thread 2 may bind names myth and myt and all names below myth
- Thread 3 may bind names myth, myt and my and all names below myth
- The main evaluation routine may bind all names on ES (except those that local to the individual threads)

- Thread 1 has access to result queues q1, q2 and q3
- Thread 2 has access to result queues q3, q4 and q5
- Thread 3 has access to result queues q5, q6 and q7
- The main evaluation routine has access to result queues q1, ..., q7

**Fig. 5.7.** Logical View of the Effects of `evalMultiES ( )` Evaluation Procedures on ES.

– `void eval ( RQ * rq, EVALnode * eNode )` ... is the main evaluation routine. At the beginning of the processing of a new user request, the root node of the main evaluation plan is passed as second argument. This main evaluation routine performs processing in serial manner but may utilise any of the other three multi-threaded or distributed styles of processing.

- `void evalMultiES ( RQ * rqOut, __opCode opCode, RQ * rqsIn[] )` ... is the only main sub-evaluation routine that is solely used to make internal processing more efficient. Hence, a low-level machine instruction together with an array of arguments (i.e. result queues) are passed as parameters. Once invoked, the specified machine code is processed as a separate execution thread.
  The `evalMultiES` routine does not create a new frame or sub-frame on ES. Instead, it splits the current sub-frame in two parts: One part is used for the main evaluation stream to continue its processing and the other for its own processing. This splitting approach allows both parts of the sub-frame to share the same scope. Figure 5.7 demonstrates this approach. Three different sub-evaluations are performed in the same sub-frame. Among others, operational semantics for the `ORDER BY` expression utilised this style of processing.
  However, sharing the same scope imposes a number of restrictions on the type of sub-evaluations that may be `MultiES`-threaded. Such restrictions include:
  - Service routines not accessing environment variables other than those defined within the routine may safely be `MultiES`-threaded.
  - Modification of global or shared variables is not permitted from within `MultiES`-threads. However, local variables (e.g. a counter of collection elements) defined in the same frame may be modified.

  Thus, `MultiES` threading is mainly utilised to implement a simultaneous version of the particular machine code.
- `void evalThreaded ( RQ * rq, EVALnode * eNode )` ... is a main sub-evaluation routine that branches the processing of an execution stream into multiple execution streams according to processing annotations that are encountered when traversing the evaluation graph. Again, multi-threading is utilised.
  In contrast to `MultiES`-threaded processing, the ES associated with the original execution stream is cloned and a copy is given to each thread. Threads then continue processing independently according to the evaluation graph until they rejoin the original execution stream or terminate.
  The `evalThreaded` routine is used to delegate sub-evaluations or execute multiple statements concurrently as indicated by `INPEDENDENT DO` ... `ENDDO` and `CONCURRENT DO` ... `ENDDO` blocks.
- `void evalDistributed ( RQ * rq, EVALnode * eNode )` ... is the last main sub-evaluation routine. It is similar to `evalThreaded` but processing continues on a remote ODBS node instead of another thread.

In order to demonstrate the recursive nature of the evaluation process, we revisit the sample evaluation graph as shown in Figure 5.1 (on page 143). Figure 5.8 indicates the corresponding traversal order (assuming that both `IF` statements execute the `THEN` clause). *init:* indicates that necessary scopes on ES, result stacks and all required result queues are initialised. Evaluation commences once the first leave is reached. Subsequently, we backtrack until another unvisited sub-evaluation edge or control flow edge is encountered. During backtracking those evaluation nodes, which have no unvisited edges remaining, are processed (as indicated by the *eval:* prefix).

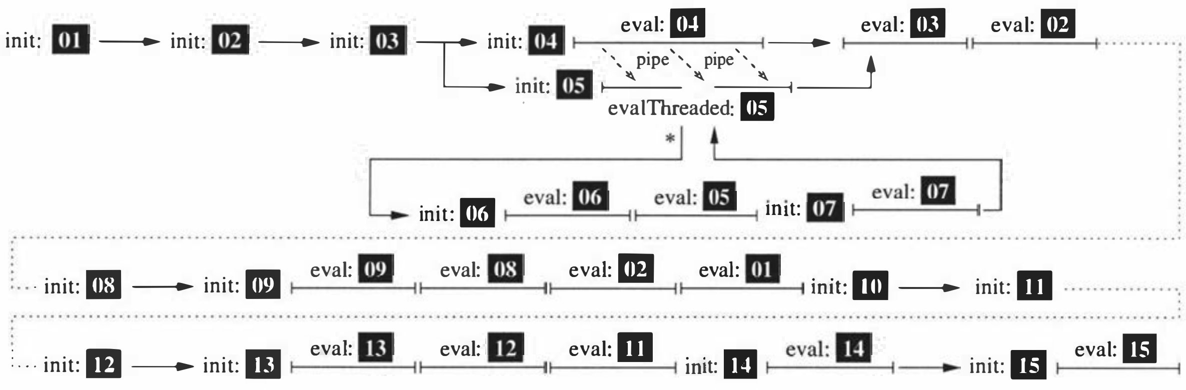


**Fig. 5.8.** Overview of the Evaluation Process of the `verifyEnrolment` Method from Figure 5.1.

### 5.3.5 Maintaining the Run-Time Environment

Garbage collection is the main challenge that has to be addressed when considering the maintenance of the run-time environment. Persistent objects are of little concern. They can always be retrieved from persistent storage if they have been cached out. Transient objects, however, must only be garbage-collected if they are no longer required but as soon as possible to conserve main memory space.

Since name-based access to class instances is permitted, object references alone can no longer be used to determine whether or not an object is still accessible. Instead, in-memory structures that are associated with classes are utilised to determine the point in time at which an object is garbage-collected. iDBPQL supports collection-classes and collection-less classes. Instances of collection-less classes can be garbage-collected in the usual manner since name-based access is not supported. Thus, only instances of collection-classes may cause difficulties. In order to enable efficient access to all instances of a class (and its sub-classes), two special in-memory structures are maintained with each collection-class. These structures are shallow and deep extents. A *shallow class extent* contains references to all instances of the respective class. In turn, a *deep class extent* is a collection of object references of the respective class together with references to all instances of sub-classes (which must also be collection-classes).

Supporting shallow and deep extents alone is not sufficient for garbage collection. In addition, some run-time properties of the stack-based evaluation must be utilised. As mentioned above, the environment of a particular evaluation contains name binders for all collection-classes that are in scope. Such binders are always found at the beginning of a frame on ES. This together with the fact that a recursive evaluation approach is followed, enables us to define a garbage collector. Recursive evaluation means that those name binders at the beginning of a frame are only removed once processing of the corresponding evaluation plan terminates. Thus, garbage collection may be achieved as follows:

- Objects that are instances of collection-less classes are garbage-collected if it is no longer referenced. In addition, an object can be discarded if it is involved in a cycle of references where all involved objects have no other references associated other than those that form the cycle.

– Objects that are instances of collection-classes are garbage collected if no more name binders to the object's class or any of its super-classes exists in the REE stack area.

### 5.3.6 Evaluating Individual Statements and Expressions

First, we consider operational semantics that describe the evaluation procedure for individual iDBPQL statements and expressions. For most statements and expressions that have been introduced in Sections 4.3.4 and 4.3.5, we will start off outlining a basic algorithm. Subsequently, enhanced operational semantics are discussed for a number of machine instructions that utilise multi-threaded processing. It is our intention to demonstrate how to enhance processing by providing alternative implementation approaches for the same iDBPQL statement, expression or keyword. However, a complete description of the language implementation is beyond the scope of this thesis.

**Simple Expressions: Literals and Names.** The evaluation of literals and simple names (i.e. identifiers) does not affect the state of the environment stack. In the event that a literal is encountered, the value is simply pushed to the result queue and the evaluation terminates. The corresponding portion of the evaluation routine is as follows:

```
...
else if ( EVALnode.code is recognised as a literal v ) {    // parse condition
  push ( RQ, v );                                       // return the literal value
}
...
```

While the evaluation of literals does not involve the ES, the evaluation of simple names accesses the ES, but does not modify its state. Evaluating simple names results in a search of the ES until the first binder, which has a matching name, is found. Subsequently, the entity associated with the binder is added to the result queue RQ.

```
...
else if ( EVALnode.code is recognised as a simple name n ) {
  __binder * x;                         // initialise a pointer to a name binder

  x = bind ( ES, n );                                      // bind the given name
  push ( RQ, x.e );                // return the value associated with the binder
}
...
```

If no matching binder is found on the environment stack, an exception occurs.

The evaluation of qualified names is discussed later since it already involves more sophisticated concepts such as navigation, projection, the examination of multiple scopes, knowledge about super-classes and multiple inheritance etc.

**Object Identifiers and Stacks.** Object identifiers are an internal concept. Thus, they may never be returned as part of a result answering a user request. However, this does not mean that all result values / objects have to be present in main memory. Let us assume that a result corresponds to a collection of objects (i.e. a collection of references to objects). Internally, this may be represented as a collection of object identifiers. In the event that an object of this collection is accessed, the run-time environment ensures that the respective object is made available in main memory. In the worst case, this results in object migration or a direct access to the persistent object store, which retrieves the respective object based on its unique OID.

The rationale behind returning object references over objects includes memory consumption considerations, shared object access and efficiency reasons such as the fact that not all operations require access to the object. For instance, equality between two objects can be decided by examining the respective OIDs. Objects are only loaded into main memory when access to its features is required.

**Expressions with Unary and Binary Operators.** iDBPQL contains a large number of pre-defined unary and binary operators over simple expressions. These operators are defined in the iDBPQL library and associated with system types. Considering unary operators, they are of the form: `operator expression` or `expression operator`, where the result of the expression must be compatible with at least one of the types for which the operator is defined. The evaluation of such an expression is shown next:

```
...
else if ( EVALnode.code is recognised as the unary operator op on an
          expression exp ) {
  rtype ( exp ) RQ_1;            // initialise a result queue that can hold the
             // value, which will result from the evaluation of expression exp
  eval ( RQ_1, exp );                                  // evaluate expression exp
  SYSTEMcall_op ( RQ, head ( RQ_1 ) );                 // invoke the unary operator
  release ( RQ_1 );                          // release the auxiliary result queue
}
...
```

Examples of such unary operators (together with their respective machine codes) include pre-incrementation (i.e. `ov++UOP`), post-incrementation (i.e. `ovUOP++`), pre-decrementation (i.e. `ov--UOP`), post-decrementation (i.e. `ovUOP--`) and the logical *NOT* operator (i.e. `boNOT`).

Furthermore, there are a large number of pre-defined binary operators. Their common format is `expression`$_1$ `operator expression`$_2$. Binary operators can be grouped in various ways. On one hand, we can distinguish arithmetic operators, assignment operators, mixed arithmetic and assignment operators, equality and relational operators, logical operators, bit-manipulating operators, and mixed bit-manipulating and assignment operators. On the other hand, binary operators can be organised according to their effects (or lack of it) on the environment stack. Operators with side effects include assignment operators and other operators that are mixed with assignment operators. The remaining types of operators mentioned above do not have side effects.

Independent of the fact whether or not an operator has side effects, the evaluation of binary operators is performed as follows:

```
...
else if ( EVALnode.code is recognised as the binary operator op on
          expressions exp_1 and exp_2 ) {
  rtype ( exp_1 ) RQ_1;
  rtype ( exp_2 ) RQ_2;
```

```

    eval ( RQ1, exp1 );                                     // evaluate expression exp1
    eval ( RQ2, exp2 );                                     // evaluate expression exp2
    SYSTEMcall_op ( RQ, head ( RQ1 ), head ( RQ2 ) );   // invoke binary operator
    release ( RQ2 );
    release ( RQ1 );
  }
  ...
```

| Operation Code | Comment | Operation Code | Comment |
|---|---|---|---|
| avBOP== | == for atomic values | ovBOP* | * for ordered values |
| stBOP== | == for String values | nvBOP* | * for NULLable values |
| nvBOP== | == for NULLable values | ovBOP/ | / for ordered values |
| avBOP!= | != for atomic values | nvBOP/ | / for NULLable values |
| stBOP!= | != for String values | ovBOP% | Modulus for ordered values |
| nvBOP!= | != for NULLable values | nvBOP% | Modulus for NULLable values |
| avBOP< | < for atomic values | avBOP&& | Logical *AND* for atomic values |
| stBOP< | < for String values | nvBOP&& | Logical *AND* for NULLable values |
| nvBOP< | < for NULLable values | avBOP\|\| | Logical *OR* for atomic values |
| avBOP<= | <= for atomic values | nvBOP\|\| | Logical *OR* for NULLable values |
| stBOP<= | <= for String values | avBOP& | Bitwise *AND* for atomic values |
| nvBOP<= | <= for NULLable values | nvBOP& | Bitwise *AND* for NULLable values |
| avBOP>= | >= for atomic values | avBOP\| | Bitwise *OR* for atomic values |
| stBOP>= | >= for String values | nvBOP\| | Bitwise *OR* for NULLable values |
| nvBOP>= | >= for NULLable values | avBOP^ | Bitwise *XOR* for atomic values |
| avBOP> | > for atomic values | nvBOP^ | Bitwise *XOR* for NULLable values |
| stBOP> | > for String values | avBOP<< | Bitwise << for atomic values |
| nvBOP> | > for NULLable values | nvBOP<< | Bitwise << for NULLable values |
| ovBOP+ | + for ordered values | avBOP>> | Bitwise >> for atomic values |
| nvBOP+ | + for NULLable values | nvBOP>> | Bitwise >> for NULLable values |
| ovBOP- | - for ordered values | avBOP~ | Bitwise ~ for atomic values |
| nvBOP- | - for NULLable values | nvBOP~ | Bitwise ~ for NULLable values |

**Table5.5.** Overview of Machine Instructions for Binary Expressions without Side Effects.

Table 5.5 outlines common expressions with binary operators and their respective machine instructions. Only those operators, which do not have side effects, are listed.

Considering, for example, the equality operator, sub-typing permits the comparison of a Natural value with an Integer value. Such a comparison would be invoked using the `inBOP==` operator and two Integer values as arguments. The first argument results from a `NAT-to-INT` type conversion while the second value is already of type `INT`.

In the event that an operator has side effects, the state of ES changes after the `SYSTEMcall` has been executed. For instance, consider the assignment operator. It changes the value of the variable identified by expression $\texttt{exp}_1$ and returns an empty result queue. If this variable holds a persistent value, further actions are taken. In addition, the assignment operator would have to be executed as part of a transaction. Corresponding details are addressed separately when transaction support is discussed

| Assignment Expression | Operation Code | Comment |
|---|---|---|
| exp$_1$ = exp$_2$ | xxBOP= | Assignment operator for all values |
| exp$_1$ += exp$_2$ | ovBOP+= | += operator for ordered values |
| exp$_1$ += exp$_2$ | nvBOP+= | += operator for NULLable values |
| exp$_1$ -= exp$_2$ | ovBOP-= | -= operator for ordered values |
| exp$_1$ -= exp$_2$ | nvBOP-= | -= operator for NULLable values |
| exp$_1$ *= exp$_2$ | ovBOP*= | *= operator for ordered values |
| exp$_1$ *= exp$_2$ | nvBOP*= | *= operator for NULLable values |
| exp$_1$ /= exp$_2$ | ovBOP/= | /= operator for ordered values |
| exp$_1$ /= exp$_2$ | nvBOP/= | /= operator for NULLable values |
| exp$_1$ %= exp$_2$ | ovBOP%= | %= operator for ordered values |
| exp$_1$ %= exp$_2$ | nvBOP%= | %= operator for NULLable values |
| exp$_1$ &= exp$_2$ | ovBOP&= | &= operator for ordered values |
| exp$_1$ &= exp$_2$ | nvBOP&= | &= operator for NULLable values |
| exp$_1$ \|= exp$_2$ | ovBOP\|= | \|= operator for ordered values |
| exp$_1$ \|= exp$_2$ | nvBOP\|= | \|= operator for NULLable values |
| exp$_1$ ^= exp$_2$ | ovBOP^= | ≙ operator for ordered values |
| exp$_1$ ^= exp$_2$ | nvBOP^= | ≙ operator for NULLable values |
| exp$_1$ <<= exp$_2$ | ovBOP<<= | <<= operator for ordered values |
| exp$_1$ <<= exp$_2$ | nvBOP<<= | <<= operator for NULLable values |
| exp$_1$ =>> exp$_2$ | ovBOP=>> | =>> operator for ordered values |
| exp$_1$ =>> exp$_2$ | nvBOP=>> | =>> operator for NULLable values |

**Table5.6.** Overview of Machine Instructions for Assignment Expressions with Side Effects.

below.

Table 5.6 summarises all assignment expressions together with their respective machine instructions.

**Boolean Expressions.** Evaluation procedures for unary and binary operators also apply to boolean expressions as outlined in Syntax Snapshot 4.18 (on page 109). Only difference being the result type. We only refine the evaluation procedure for quantifier expressions and IS*OF expressions. Remaining boolean expressions (as outlined in Table 5.7) are easy to derive from the operational semantics presented in this section.

The EXISTS quantifier is of the form EXISTS exp ( boolExp ). First, the expression exp is evaluated. Subsequently, for each result value, the boolean condition is considered. Evaluation terminates as soon as the first positive match is found. Corresponding operational semantics are as follows:

```
...
// OpCode: cvEXISTSet ≅ boolean EXISTS with early termination
else if ( EVALnode.code is recognised as boolean expression cvEXISTSet exp
          ( boolExp ) ) {
  boolean RQ$_1$;          // initialise a result queue that holds a Boolean value
  rtype ( exp ) RQ$_2$;

  push ( RQ$_1$, FALSE );                                  // default result value
  eval ( RQ$_2$, exp );                          // evaluate the collection expression
```

| Boolean Expression | Machine Instructions | Comment |
|---|---|---|
| `exp`$_1$ `IN exp`$_2$ | `cvISIN` | is-member-of-collection test |
| `exp`$_1$ `ISSUBTYPEOF exp`$_2$ | `tiISSUBTYPE` | is-sub-type-of test |
| `exp`$_1$ `ISSUBCLASSOF exp`$_2$ | `ciISSUBCLASS` | is-sub-class-of test |
| `exp`$_1$ `ISINSTANCEOF exp`$_2$ | `ciISINSTANCE` | is-instance-of test |
| `exp IS NULL` | `nuISNULL` | has `NULL` value test |
| `exp IS NOT NULL` | `nuNOTNULL` | has value other than `NULL` test |
| `EXISTS exp ( boolExp )` | `cvEXISTSet` | generalised *OR* with early termination |
| | `cvEXISTSpet` | generalised *OR* with pipelining and early termination |
| `FOR ANY exp ( boolExp )` | `cvFORANYet` | generalised *AND* with early termination |
| | `cvFORANYpet` | generalised *AND* with pipelining and early termination |
| `exp`$_1$ `LIKE exp`$_2$ | `stLIKE` | pattern matching form String values |

**Table5.7.** Overview of Additional Boolean Expressions and their Machine Instructions.

```
    while ( ( top ( RQ1 ) == FALSE ) && ( empty ( RQ2 ) == FALSE ) ) {
      push ( ES, unnest ( top ( RQ2 ) ) );      // process next collection value
      eval ( RQ1, boolExp );                         // test the boolean condition
      if ( top ( RQ1 ) == TRUE ) {                         // was test successful?
        push ( RQ, TRUE );                // if successful, write result value and
      }                                                    // initialise termination
      pop ( ES );                      // remove current collection value from scope
      pop ( RQ2 );                                   // discard intermediate result
    }
    if ( empty ( RQ ) == TRUE ) {                      // were all test unsuccessful?
      push ( RQ, top ( RQ1 ) );                       // if so, return negative result
    }
    release ( RQ2 );
    release ( RQ1 );
  }
  ...
```

To further enhance performance of the evaluation of the EXISTS quantifier, simultaneous processing can be utilised. Simultaneous execution is synchronised through a pipeline (i.e. a result queue). Corresponding operational semantics are as follows:

```
  ...
  // OpCode: cvEXISTSpet ≅ boolean EXISTS with pipelining and early
  // termination
  else if ( EVALnode.code is recognised as boolean expression cvEXISTSpet exp
            ( boolExp ) ) {
    rtype ( exp ) RQ1;
    boolean RQ2;           // initialise a result queue that holds a Boolean value

    evalThreaded ( RQ1, exp );       // initialise concurrent evaluation; results
                                                       // are returned via RQ1
    push ( RQ2, FALSE );                                  // default result value
    while ( ( top ( RQ2 ) == FALSE ) && ( state ( RQ1 ) != END ) ) {
```

```
    while ( state ( RQ$_1$ ) == EMPTY ) {
      waitOn ( RQ$_1$ );                               // wait for next result(s)
    }
    push ( ES, unnest ( top ( RQ$_1$ ) ) );     // process next collection value
    eval ( RQ$_2$, boolExp );                     // test the boolean condition
    if ( top ( RQ$_2$ ) == TRUE ) {                     // was test successful?
      push ( RQ, TRUE );              // if successful, write result value and
    }                                                 // initialise termination
    pop ( ES );                   // remove current collection value from scope
    pop ( RQ$_1$ );                              // discard intermediate result
  }
  if ( empty ( RQ ) == TRUE ) {                  // were all test unsuccessful?
    push ( RQ, top ( RQ$_2$ ) );                // if so, return negative result
  }
  release ( RQ$_2$ );
  release ( RQ$_1$ );
}
...
```

Similarly, the FOR ANY quantifier is supported by two implementations: One with and the other without pipelining. In both cases, the algorithm terminates as soon as the first boolean expression is evaluated to FALSE. Corresponding operational semantics are as follows:

```
...
// OpCode: cvFORANYet ≅ boolean FOR ANY with early termination
else if ( EVALnode.code is recognised as boolean expression cvFORANYet exp
          ( boolExp ) ) {
  boolean RQ$_1$;
  rtype ( exp ) RQ$_2$;

  push ( RQ$_1$, FALSE );                                  // default result value
  eval ( RQ$_2$, exp );                          // evaluate the collection expression
  while ( ( empty ( RQ ) == TRUE ) && ( empty ( RQ$_2$ ) == FALSE ) ) {
    push ( ES, unnest ( top ( RQ$_2$ ) ) );     // process next collection value
    eval ( RQ$_1$, boolExp );                     // test the boolean condition
    if ( top ( RQ$_1$ ) == FALSE ) {                    // was test successful?
      push ( RQ, FALSE );                       // if not, write result value
    }
    pop ( ES );                   // remove current collection value from scope
    pop ( RQ$_2$ );                        // discard intermediate result value
  }
  if ( empty ( RQ ) == TRUE ) {                  // were all tests successful?
    push ( RQ, TRUE );                          // if so, return positive result
  }
  release ( RQ$_2$ );
  release ( RQ$_1$ );
}

// OpCode: bFORANYpet ≅ boolean FOR ANY with pipelining and early
// termination
else if ( EVALnode.code is recognised as boolean expression bFORANYpet exp
          ( boolExp ) ) {
```

```
    rtype ( exp ) RQ₁;
    boolean RQ₂;

    evalThreaded ( RQ₁, exp );      // initialise concurrent evaluation; results
                                                          // are returned via RQ₁
    push ( RQ₂, FALSE );                                  // default result value
    while ( ( empty ( RQ ) == TRUE ) && ( state ( RQ₁ ) != END ) ) {
      while ( state ( RQ₁ ) == EMPTY ) {
        waitOn ( RQ₁ );                                // wait for next result(s)
      }
      push ( ES, unnest ( top ( RQ₁ ) ) );      // process next collection value
      eval ( RQ₁, boolExp );                       // test the boolean condition
      if ( top ( RQ₂ ) == FALSE ) {                     // was test successful?
        push ( RQ, FALSE );                        // if not, write result value
      }
      pop ( ES );                   // remove current collection value from scope
      pop ( RQ₁ );                           // discard intermediate result value
    }
    if ( empty ( RQ ) == TRUE ) {                  // were all tests successful?
      push ( RQ, TRUE );                         // if so, return positive result
    }
    release ( RQ₂ );
    release ( RQ₁ );
  }
  ...
```

Other boolean expressions include sub-type, sub-class and instance-of tests. The respective operational semantics rely on system routines that examine metadata entries and / or auxiliary structures maintained together with metadata catalogues.

```
  ...
  // OpCode: tiISSUBTYPE ≅ boolean ISSUBTYPEOF
  else if ( EVALnode.code is recognised as boolean expression exp₁ ISSUBTYPEOF
            exp₂ ) {
    rtype ( exp₁ ) RQ₁;
    rtype ( exp₂ ) RQ₂;

    eval ( RQ₁, exp₁ );                      // evaluate left-hand side expression
    eval ( RQ₂, exp₂ );                     // evaluate right-hand side expression
    systemCall_isSubTypeOf ( RQ, head ( RQ₁ ), head ( RQ₂ ) );        // utilise
    release ( RQ₂ );                                        // pre-defined routine
    release ( RQ₁ );
  }

  // OpCode: ciISSUBCLASS ≅ boolean ISSUBCLASSOF
  else if ( EVALnode.code is recognised as boolean expression exp₁ ISSUBCLASSOF
            exp₂ ) {
    rtype ( exp₁ ) RQ₁;
    rtype ( exp₂ ) RQ₂;

    eval ( RQ₁, exp₁ );                      // evaluate left-hand side expression
    eval ( RQ₂, exp₂ );                     // evaluate right-hand side expression
    SYSTEMcall_isSubClassOf ( RQ, head ( RQ₁ ), head ( RQ₂ ) );       // utilise
    release ( RQ₂ );                                        // pre-defined routine
    release ( RQ1 );
  }

  // OpCode: ciISINSTANCE ≅ boolean ISINSTANCEOF
  else if ( EVALnode.code is recognised as boolean expression exp1 ISINSTANCEOF
            exp2 ) {
    rtype ( exp1 ) RQ1;
    rtype ( exp2 ) RQ2;

    eval ( RQ1, exp1 );                         // evaluate left-hand side expression
    eval ( RQ2, exp2 );                        // evaluate right-hand side expression
    SYSTEMcall_isSubClassOf ( RQ, ( (__object *) head ( RQ1 ) ).__class,
                              head ( RQ2 ) );    // utilise pre-defined routine
    release ( RQ2 );
    release ( RQ1 );
  }
  ...
```

**Renaming Expressions.** iDBPQL supports two renaming expressions. These are the AS and the GROUP AS expressions. An exp GROUP AS id expression is processed simply by evaluating the expression exp and then adding a new binder to ES. The binder consists of the identifier as name, the type of the value and the value, which results from exp's evaluation, as entity.

The evaluation of an exp GROUP AS id expression is more complex. For non-collection values, there is no difference between a GROUP AS and an AS expression. However, for collection values, the renaming does not apply to the collection object itself but to its members. Thus, evaluation is performed by unnesting the value denoted by the expression exp and then followed by renaming all collection members. Corresponding operational semantics are as follows:

```
  ...
  // OpCode: xxGROUPAS ≅ renaming
  else if ( EVALnode.code is recognised as renaming expression exp GROUP AS
            id ) {
    rtype ( exp ) RQ1;

    eval ( RQ1, exp );
    push ( ES, (__binder) ( id, typeOf ( head ( RQ1 ) ), head ( RQ1 ) ) );
    release ( RQ1 );
  }

                                                        // OpCode: xxAS ≅ renaming
  else if ( EVALnode.code is recognised as renaming expression exp AS id ) {
    rtype ( exp ) RQ1;

    eval ( RQ1, exp );
    if ( head ( RQ1 ) is a value of a reference-type ) {
      push ( ES, unnest ( head ( RQ1 ), id ) );          // unnest collection;
    }                                                    // assign id as name
    else {
      push ( ES, (__binder) ( id, typeOf ( head ( RQ1 ) ), head ( RQ1 ) ) );
    }
```

```
     release ( RQ_1 );
   }
   ...
```

One usage of the AS renaming expressions is that of a cursor in loops such as the FOR EACH loop. To enable such a loop to access one collection value after the next, the bindCrsNext stack operator is utilised.

**Accessing Data Objects.** There are various ways to how objects and values can be accessed. First, there is a special means of access to all instances of a particular class. This access by class name is based on the previously introduced unnest operator:.

```
...
// OpCode: cvCNAME n ≅ object access through class-collection n
else if ( EVALnode.code is recognised as name-based access to
          class-collection cvCNAME n ) {
  __binder * x;

  x = bind ( ES, n );                                  // find binder on ES
  push ( ES, unnest ( x.e, NULL ) );            // unnest class-collection
  push ( RQ, top ( ES ) );                      // push references to result
  pop ( ES );                                  // return ES to previous state
}
...
```

Similarly, access by reference, i.e. through a variable holding an object reference, or by object identifier is executed in a similar manner:

```
...
// OpCode: rvNAME ≅ object access through reference
else if ( EVALnode.code is recognised as access through a variable of
          a reference-type rvNAME v ) {
    __binder * x;

  x = bind ( ES, v );                                  // find binder on ES
  push ( ES, unnest ( x.e, NULL ) );              // unnest reference value
  push ( RQ, top ( ES ) );                      // push references to result
  pop ( ES );                                  // return ES to previous state
}

// OpCode: rvOID ≅ object access through OID
else if ( EVALnode.code is recognised as access by identifier rvOID oid ) {
  push ( ES, unnest ( oid, NULL ) );                       // unnest object
  push ( RQ, top ( ES ) );                      // push references to result
  pop ( ES );                                  // return ES to previous state
}
...
```

While the first three variations on data access always return all objects within a particular collection, specifying a selection with a WHERE clause returns only those objects that also match the boolean expression. Let us consider operational semantics of a basic selection algorithm:

```
...
// OpCode: cvWHERE ≅ selection
else if ( EVALnode.code is recognised as selection exp cvWHERE boolExp ) {
  rtype ( exp ) RQ_1;
  rtype ( boolExp ) RQ_2;

  eval ( RQ_1, exp );                           // evaluate collection expression
  while ( empty ( RQ_1 ) == FALSE ) {                       // for each value do
    push ( ES, unnest ( top ( RQ_1 ) ) );                    // add value to ES
    eval ( RQ_2, boolExp );                           // test boolean expression
    if ( top ( RQ_2 ) == TRUE ) {                      // if true, add to result
      push ( RQ, top ( RQ_1 ) );
    }
    pop ( RQ_2 );
    pop ( ES );
  }
  pop ( RQ_1 );
}
...
```

A more efficient evaluation may be achieved through pipelining. Based on the approach used for quantifiers, selection can also be achieved by utilising simultaneous processing:

```
...
// OpCode: cvWHEREp ≅ selection with pipelining
else if ( EVALnode.code is recognised as selection exp cvWHEREp boolExp ) {
  rtype ( exp ) RQ_1;
  rtype ( boolExp ) RQ_2;

  evalThreaded ( RQ_1, exp );
  while ( state ( RQ_1 ) != END ) {
   while ( state ( RQ_1 ) == EMPTY ) {
      waitOn ( RQ_1 );                           // wait for next result(s)
    }
    push ( ES, unnest ( top ( RQ_1 ) ) );
    eval ( RQ_2, boolExp );
    if ( top ( RQ_2 ) == TRUE ) {
      push ( RQ, top ( RQ_1 ) );
    }
    pop ( RQ_2 );
    pop ( ES );
  }
  pop ( RQ_1 );
}
...
```

Direct access and selection always return collections of a whole type or class. However, we might only be interested in one or more members of a particular type or class. The projection operator supports such a means of access:

```
...
// OpCode: cvPROJ exp_1.exp_2 ≅ projection or one-step navigation
```

```
092    else if ( EVALnode.code is recognised as projection cvPROJ exp1.exp2 ) {
093      rtype ( exp1 ) RQ1;
094
095      eval ( RQ1, exp1 );
096      while ( empty ( RQ1 ) == FALSE ) {
097        push ( ES, unnest ( top ( RQ1 ) ) );
098        eval ( RQ, exp2 );
099        pop ( ES );
100      }
101      pop ( RQ1 );
102    }
103    ...
```

Let us consider a first example, which also indicates how the evaluation of multiple expressions will be performed.

EXAMPLE 5.6. Again, we consider the university application. Similar to the boolean expression specified in the `IF` statement in Example 4.26 (lines 23 and 24), we will consider the evaluation of a slightly simpler expression:

```
FOR ANY ( RecordC WHERE ( THIS.student == student ) )
  ( result IN { "A+", "A", "A-", "B+", "B", "B-", "C+", "C" } )
```

This expression only evaluates to `TRUE` if the student has passed all his / her associated courses successfully.
Assuming that the environment stack has been built up to contain all necessary binders, the evaluation can be performed as follows:

1. Evaluate `cvFORANYet exp ( boolExp )`: First, only lines 062 to 067 are processed. That is, the result stack and a result queue are initialised. Subsequently, the evaluation of expression `exp` commences.
2. Evaluate `exp cvWHERE boolExp`: Only lines 42 to 45 are processed before another sub-evaluation is initiated. The only effect on the run-time environment is the creation of two new result queues, which will later hold the results the sub-evaluations.
3. Evaluate `cvCNAME RecordC`: This is the first evaluation that is performed without any sub-evaluation. However, the `unnest` routine is likely to involve the persistent object store. First, a search on the environment stack is executed. The top-down search will terminate as the first binder that matches the specified collection identifier (i.e. class name `RecordC`) is located. Subsequently, the corresponding collection is retrieved (either from the heap, if the binder has an in-memory pointer associated, or, otherwise, from the persistent object store). References to all collection objects are pushed to the result queue, which has been set up by the evaluation described in Step 2.
   Let us assume that the binder has an associated object identifier. Thus, the `unnest` routine invokes the `POScall_Retrieve` primitive. The result of this invocation, which is a set of name binders that have identifiers of instances of class `RecordC` as their values, is then pushed onto the environment stack. In contrast, if we assume that the binder has an associated in-memory pointer, the `unnest` routine pushes a set of name binders that have in-memory references, OIDs or even a mixtures of

both as their values. At last, all object identifies or object references, respectively are extracted from those binders and pushed to the evaluation's result queue.

4. Continue the evaluation of `exp cvWHERE boolExp`: As all result values of the `rvNAME` sub-evaluation become available, processing of the `cvWHERE` routine continues. For each object reference in the result queue, the following steps are executed:
   (a) The object reference is pushed to ES. Subsequently, another sub-evaluation is initiated.
   Note: This step is different compared to the SBA approach. Instead of unnesting the object immediately, we will delay this step. It is left to the corresponding sub-routine to decide whether or not an unnesting is required. Since this is not always the case, we are likely to save a few disk I/Os by delaying the unnesting.
   (b) Evaluate $\texttt{exp}_1$ `rvBOP==` $\texttt{exp}_2$: The equality operator for reference-type expressions evaluates both expressions and then tests for equality of their respective (reference-type) values. First, the $\texttt{exp}_1$ expression is processed resulting in another sub-evaluation.
   (c) Evaluate `cvPROJ THIS.student`: The evaluation of the `THIS` keyword would normally result in a search on ES. Since this is a task that is commonly performed, we maintain a special pointer that always keeps track of the `THIS` object within the respective scope. This avoids frequent searches for the current `THIS` object.
   Using the `THIS` pointer, we first unnest the corresponding object. The unnested value is placed on ES. Subsequently, the value of the `student` variable is pushed to the result queue (i.e. by means of projecting to its value) and the top element on ES (i.e. the previously unnested object) is discarded.
   (d) Continue the evaluation of $\texttt{exp}_1$ `rvBOP==` $\texttt{exp}_2$: As the result of evaluating $\texttt{exp}_1$ is received, a second sub-evaluation that of $\texttt{exp}_2$ is initialised.
   (e) Evaluate `cvPROJ student`: The `student` variable relates to the object that resides on top of ES, i.e. the current instance of class `RecordC`. First, the object has to be unnested. The unnested value becomes the new top element on ES. Subsequently, the value of the `student` variable is projected to the respective result queue and the top element on ES is discarded.
   (f) Continue the evaluation of $\texttt{exp}_1$ `rvBOP==` $\texttt{exp}_2$: Now, that result values for both expressions are computed, the equality operator is invoked. This results in pushing the result of $\texttt{exp}_1$ and the result of $\texttt{exp}_2$ onto ES. Unnesting is not required since object equality can be verified by looking at the respective object identifiers. In the event of equality, a `TRUE` value is pushed to the corresponding RQ. Otherwise, a `TRUE` value is returned. Subsequently, the top-two elements are removed from ES and results of the evaluations of $\texttt{exp}_1$ and $\texttt{exp}_2$ are discarded.
   (g) Continue the evaluation of `exp cvWHERE boolExp`: At the end of each iteration step, the result queue $RQ_2$ is emptied and the top element on ES is removed. Continue with Step 4a.
5. Finish the evaluation of `exp cvWHERE boolExp`: If the iteration has completed, the result queue holding all instances of `RecordC` is discarded and the selection terminates.
6. Continue the evaluation of `cvFORANYet exp ( boolExp )`: Now, that the first expression `exp` is evaluated, processing continues on line `065`. For each object reference

in the result queue, the following steps are executed:

(a) The object reference is pushed to ES. Subsequently, the boolean expression is evaluated for the object reference now located on top of the environment stack.

(b) Evaluate `exp`$_1$ `stIN` `exp`$_2$: This boolean expression tests whether the String value of `exp`$_1$ is an element of the collection value of `exp`$_2$. First, `exp`$_1$ is processed.

(c) Evaluate `result`: Processing is similar to Step 4e. The `result` variable relates to the object that resides on the top of ES. First, the object is unnested. Subsequently, the value of the `result` variable is projected to the respective result queue and the top element on ES is discarded.

(d) Continue the evaluation of `exp`$_1$ `stIN` `exp`$_2$: As the result of evaluating `exp`$_1$ is received, a second sub-evaluation that of `exp`$_2$ is initialised.

(e) Evaluate { `"A+", "A", "A-", "B+", "B", "B-", "C+", "C"` }: The expression corresponds to a value-type. Thus, a set of String values is returned.

(f) Finish the evaluation of `exp`$_1$ `stIN` `exp`$_2$: Having results of both expressions, we may invoke the respective machine operation code that implements an is-element-of test. If this test is successful, a `TRUE` value is pushed to the result stack. Otherwise, the `FALSE` value is returned. Finally, both result queues, which have been used to hold result values of sub-evaluations, are destroyed.

(g) Continue the evaluation of `cvFORANYet exp ( boolExp )`: At the end of each iteration step, it is verified whether or not the sub-evaluation of the boolean expression was successful. If not, an early termination can be initialised by pushing the `FALSE` value to RQ. Otherwise, RQ remains empty. Finally, the top element on ES is removed and the result queue RQ$_1$ is emptied. Continue with Step 6a.

7. Finish the evaluation of `cvFORANYet exp ( boolExp )`: As the iteration terminates, it is verified whether or not there was at least one sub-evaluation of the boolean expression that was not successful. If so, RQ is not empty. Otherwise, a `TRUE` value is pushed onto RQ indicating a successful evaluation of the `FOR ANY` expression. Finally, RQ$_2$ and RQ$_1$ are destroyed and the evaluation of the considered expression terminates.

At the end of the evaluation, ES is in the same state as before the evaluation commenced. The same applies to the persistent object store. Similarly, no object in memory has been modified. The only difference is the contents of the result queue, which contains a single Boolean value that corresponds to the result of the evaluation of the `FOR ANY` expression. □

**Accessing Persistent Data: Beyond Direct Access.** In order to utilise some of the additional features of the persistent object store as introduced in Section 5.2.1, we support a number of special evaluation routines. These routines only target locally held data that does not currently reside in the heap and rely on the `POScall` primitive.

First, we consider various types of selections. Selecting objects from a collection by OID or collection name is commonly accompanied by a simple arithmetic or equality (boolean) expression. Corresponding operational semantics are as follows:

```
  ...
  // OpCode: cvWHEREisbex ≅ selection by OID with a simple
  // boolean expression
  else if ( EVALnode.code is recognised as oid cvWHEREisbex ( vcp1 op vcp2 ) ) {
    POScall_FindFromCollection ( RQ, oid, TRUE, op, vcp1, vcp2 );
  }

  // OpCode: cvWHEREcnsbex ≅ selection by name with a simple boolean expression
  else if ( EVALnode.code is recognised as cn cvWHEREcnsbex ( vcp1 op vcp2 ) ) {
    __binder * x;

    x = bind ( ES, cn );                                 // find binder on ES
    POScall_FindFromCollection ( RQ, x.e, TRUE, op, vcp1, vcp2 );
  }
  ...
```

where *vcp* stands for (simple) value, constant or path.

The `TRUE` value in the argument list signals that sub-class objects are retrieved. Example 5.5 (2) outlines a corresponding POS-level call. When invoking the `POScall` primitive, results are pipelined by default. A separate machine code for values of a reference type is not necessary. In the event that a reference is given, it will be in the form of an OID. Otherwise, the collection must already reside in the heap.

The same mechanism may also be applied if the boolean expression consists of a conjunction or disjunction of boolean expressions. Before we consider such selections in greater details, we outline how indices may further enhance performance. Example 5.5 (3) has already indicated that indices correspond to collections. In the event that an optimiser determines that the processing of a selection is to be performed with the help of an index, the corresponding OID takes the place of the original collection name or expression evaluating to a collection name. Thus, no additional evaluation procedures for selections that utilise indices are required.

Returning to selections with conjunctions and disjunctions, we may now take advantage of rearranged query expressions that:

- For selections with conjunctions and an index for each conjunct, execute a `cvWHERE*` evaluation for every conjunct (simultaneously) and then perform an intersection[6] of all results in the run-time environment;
- For selections with conjunctions, execute a `cvWHERE*` evaluation for the most selective conjunct and then evaluate the remaining conjuncts in the run-time environment; and
- For selections with disjunctions and an index for each disjunct, execute a `cvWHERE*` evaluation for every disjunct (simultaneously) and then perform a union[7] of all results in the run-time environment.

Analogously, navigation and navigation with selections that are mapped to the `FindEnclosure` operation of the POS service interface may be evaluated in one step:

```
  ...
```

[6] Intersections are defined for all collection types.
[7] A union operator is defined for each collection type.

```
  // OpCode: rvNAV ≅ navigation starting from a reference value
  else if ( EVALnode.code is recognised as rvNAV exp.path ) {
    rtype ( exp ) RQ1;

    eval ( RQ1, exp );
    POScall_FindEnclosure ( RQ, head ( RQ1 ), path );
  }

  // OpCode: rvNAVWHEREsbex ≅ navigation followed by a selection
  // with a simple boolean expression
  else if ( EVALnode.code is recognised as rvNAVsec exp.path WHERE (
           vcp1 op vcp2 ) ) {
    rtype ( exp ) RQ1;

    eval ( RQ1, exp );
    POScall_FindEnclosure ( RQ, head ( RQ1 ), path, op, vcp1, vcp2 );
  }
  ...
```

First, we have operational semantics, which only result in navigation from a referenced object to one or more objects that can be reached over the given path `path`. The second operation code extends the former by restricting the resulting values to those that also meet the specified boolean selection expression.

**More Query Expressions.** Operational semantics for the `ORDER BY` expression are provided first for general collections. This is a more difficult case as there are numerous algorithms defined for array-like collections. The basic idea for the serial version of the `ORDER BY` evaluation is as follows:

1. Obtain the collection, which is to be ordered. As a result, we receive a queue of collection values.
2. Project the values in the sorting key. In order to ensure that these projected values can be related to their original collection values, this process outputs a queue of pairs < *key value*, *collection value* >.
3. Sort the queue of pairs on the *key value* field:
   (a) Scan the queue from both the tail end and the head end.
   (b) Compare the first tail entry with the first head entry, then the second tail entry with the second head entry and so on.
   (c) Rearrange queue entries in a way that we maintain an ordered section of small values at the head of the queue and an order section of large values at the tail of the queue.
   (d) Once each queue entry has been visited, i.e. when the two scans meet in the middle of the queue, create three partitions:
      – A partition of ordered entries at the head of the queue;
      – A partition of ordered entries at the tail of the queue; and
      – A partition of remaining unordered entries in the middle of the queue.
   (e) Append the tail partition to the head partition.
   (f) Restart the sorting on the unordered partition. If the unordered partition is empty or consists only of one entry, terminate the sorting part of this algorithm.

(g) At the end of the sorting phase, we obtain $k$ ordered partitions. Subsequently, a merging phase that creates an ordered queue of pairs < *key value*, *collection value* > is initiated.

4. Discard all *key values*.
5. Return a queue of ordered *collection values*.

Operational semantics for the evaluation of the ascending version of an `ORDER BY` expressions is outlined next:

```
...
// OpCode: cvORDERBYa ≅ ascending order-by expression
else if ( EVALnode.code is recognised as exp1 cvORDERBYa exp2 ) {
  rtype ( exp1 ) RQ1;          // hold results of the left-hand side expression
  struct { rtype ( exp2 ) key; rtype ( exp1 ) val; } RQ2, RQ2[];
  struct { rtype ( exp2 ) key; rtype ( exp1 ) val; } *dw, *up, *cand;
  rtype ( exp2 ) RQ3;

  push ( ES, (__binder) ( __cnt, INT, -1 ) );                  // counters for ...
  push ( ES, (__binder) ( __j, INT,  -1 ) );                 // ... queue partitions
  eval ( RQ1, exp1 );          // evaluate collection expression, i.e. step 1
  while ( empty ( RQ1 ) != FALSE ) {      // for each collection value, project
    push ( ES, unnest ( head ( RQ1 ) ) );    // to their key values and create
    eval ( RQ3, exp2 );              // pairs of keys with values, i.e. step 2
    push ( RQ2, ( head ( RQ3 ), head ( RQ1 ) ) );
    pop ( RQ3 );
    pop ( RQ1 );
  }
  release ( RQ3 );
  release ( RQ1 );

  dw = dw_old = head ( RQ2 );        // next, sort RQ2 on the key value fields
  up = up_old = tail ( RQ2 );
  while ( ( dw != up_old ) && ( up != dw_old ) ) {                     // step 3a
    if ( dw.key <= up.key ) {                                   // steps 3b and 3c
      if ( dw.key >= tail ( RQ2 ).key ) {
        move ( RQ2, dw, TAIL );
        move ( RQ2, up, TAIL );
      }
      else if ( up.key <= head ( RQ2 ).key ) {
        move ( RQ2, up, HEAD );
        move ( RQ2, dw, HEAD );
      }
    }
    else {
      if ( ( dw.key >= tail ( RQ2 ) ) && ( up.key <= head ( RQ2 ) ) ) {
        move ( RQ2, dw, TAIL );
        move ( RQ2, up, HEAD );
      }
      else if ( ( dw.key >= tail ( RQ2 ) ) && ( up.key > head ( RQ2 ) ) ) {
        move ( RQ2, dw, TAIL );
      }
      else if ( ( dw.key < tail ( RQ2 ) ) && ( up.key <= head ( RQ2 ) ) ) {
        move ( RQ2, up, HEAD );
      }
      else {
        swap ( RQ2, dw, up );
      }
    }
    // check whether dw is close or equal to up, i.e. step 3d
    if ( ( dw == up ) || ( ( dw = nextDown ( RQ2, dw ) ) == up ) ) {
      // separate top-most and bottom-most partitions and concatenate, i.e.
      RQ2[++bind ( __cnt ).e] = merge ( cutInfront ( RQ2, dw-old ),
                                  cutBehind ( RQ2, up-old ) );      // step 3e
      // reinitialise dw, up, dw_old and up_old
      dw = dw_old = head ( RQ2 );
      up = up_old = tail ( RQ2 );
    }
  }
  if ( empty ( RQ2 ) == FALSE ) {                        // add last partition
    RQ2[++bind ( __cnt ).e] = RQ2;
  }

  cand = NULL;
  while ( TRUE ) {              // browse partitions and merge, i.e. step 3g
    bind ( __j ).e = 0;
    while ( bind ( __j ).e <= bind ( __cnt ).e ) {      // init 1st candidate
      if ( empty ( RQ2[bind ( __j ).e] ) == FALSE ) {
        cand = head ( RQ2[bind ( __j ).e] );
        break;
      }
      bind ( __j ).e++;
    }
    if ( cand == NULL ) {                         // check terminal condition
      break;
    }
    while ( bind ( __j ).e <= bind ( __cnt ).e ) {   // compare head elements
      if ( empty ( RQ2[bind ( __j ).e] ) == FALSE ) {     // of result queues
        if ( head ( RQ2[bind ( __j ).e] ) < cand ) {
          cand = RQ2[bind ( __j ).e];
        }
      }
      bind ( __j ).e++;
    }
    push ( RQ, cand.val ); // merge smallest candidate with existing results,
    pop ( stack to which cand points );                  // i.e. steps 4 and 5
    cand = NULL;
  }
  pop ( ES );                                         // discard __cnt counter
  pop ( ES );                                           // discard __j counter
  release ( RQ2[] );
  release ( RQ2 );
}
...
```

Analogously, a descending ORDER BY expression is implemented. Its associated operation code is cvORDERBYd.

Operational semantics presented above can be made more efficient in various ways. For instance, the machine instructions `cvORDERBYath` and `cvORDERBYdth` outsource the merging phase to a simultaneously executing thread. Merging commences as soon as two intermediate queues are available. If one queue is exhausted, the other queue is appended to the intermediate result of the merging process. This further reduces the number of comparisons. Corresponding operational semantics are as follows:

```
...
// OpCode: cvORDERBYath ≅ ascending order-by
// expression with multi-threading
else if ( EVALnode.code is recognised as exp1 cvORDERBYath exp2 ) {
  rtype ( exp1 ) RQ1;
  struct { rtype ( exp2 ) key; rtype ( exp1 ) val; } RQ2, RQ2[];
  struct { rtype ( exp2 ) key; rtype ( exp1 ) val; } *dw, *up;
  rtype ( exp2 ) RQ3;

  push ( ES, (__binder) ( __cnt, INT, -1 ) );      // queue partition counter
  eval ( RQ1, exp1 );                             // evaluate collection expression
  while ( empty ( RQ1 ) != FALSE ) {           // project to all key values; create
    push ( ES, unnest ( head ( RQ1 ) ) );         // pairs of keys with values
    eval ( RQ3, exp2 );
    push ( RQ2, ( head ( RQ3 ), head ( RQ1 ) ) );
    pop ( RQ3 );
    pop ( RQ1 );
  }
  release ( RQ3 );
  release ( RQ1 );

  dw = dw_old = head ( RQ2 );                     // sort RQ2 on the key value fields
  up = up_old = tail ( RQ2 );
  while ( ( dw != up_old ) && ( up != dw_old ) ) {
    if ( dw.key <= up.key ) {
      if ( dw.key >= tail ( RQ2 ).key ) {
        move ( RQ2, dw, TAIL );
        move ( RQ2, up, TAIL );
      }
      else if ( up.key <= head ( RQ2 ).key ) {
        move ( RQ2, up, HEAD );
        move ( RQ2, dw, HEAD );
      }
    }
    else {
      if ( ( dw.key >= tail ( RQ2 ) ) && ( up.key <= head ( RQ2 ) ) ) {
        move ( RQ2, dw, TAIL );
        move ( RQ2, up, HEAD );
      }
      else if ( ( dw.key >= tail ( RQ2 ) ) && ( up.key > head ( RQ2 ) ) ) {
        move ( RQ2, dw, TAIL );
      }
      else if ( ( dw.key < tail ( RQ2 ) ) && ( up.key <= head ( RQ2 ) ) ) {
        move ( RQ2, up, HEAD );
      }
      else {
        swap ( RQ2, dw, up );
      }
    }
    // check whether dw is close or equal to up
    if ( ( dw == up ) || ( ( dw = nextDown ( RQ2, dw ) ) == up ) ) {
      // separate top-most and bottom-most partitions
      RQ2[++bind ( __cnt ).e] = merge ( cutInfront ( RQ2, dw-old ),
                                        cutBehind ( RQ2, up-old ) );
      if ( bind ( __cnt ).e == 1 ) {
        // delegate 1st merging pass to a MultiES-thread
        evalMultiES ( RQ2[++bind ( __cnt ).e], merge, RQ2[0], RQ2[1] );
      }
      else if ( bind ( __cnt ).e > 1 ) {
        // delegate consecutive merging passes to a MultiES-thread
        evalMultiES ( RQ2[++bind ( __cnt ).e], merge,
                      RQ2[bind ( __cnt ).e - 2], RQ2[bind ( __cnt ).e - 1] );
      }
      // reinitialise dw, up, dw_old and up_old
      dw = dw_old = head ( RQ2 );
      up = up_old = tail ( RQ2 );
    }
  }
  if ( empty ( RQ2 ) == FALSE ) {                          // add last partition
    RQ2[++bind ( __cnt ).e] = RQ2;
    evalMultiES ( RQ2[++bind ( __cnt ).e], merge, RQ2[bind ( __cnt ).e - 2],
                  RQ2[bind ( __cnt ).e - 1] );
  }

  // scan ordered collections of (key, value)-pairs
  while ( ( state ( RQ2[bind ( __cnt ).e] ) != END ) ||
          ( empty ( RQ2[bind ( __cnt ).e] ) != TRUE ) ) {
    while ( state ( RQ2[bind ( __cnt ).e] ) == EMPTY ) {   // wait for results
       SYSTEMcall_nanosleep ( );
    }
    push ( RQ, RQ2[bind ( __cnt ).e].val );      // add value to results queue
    pop ( RQ2[bind ( __cnt ).e] );
  }
  release ( RQ2 );
}

// OpCode: cvORDERBYath_MERGE ≅ merging thread for the
// ascending order-by expression with multi-threading
else if ( EVALnode.code is recognised as cvORDERBYath_MERGE RQ1 RQ2 ) {
  struct { rtype ( exp2 ) key; rtype ( exp1 ) val; } *cand;

  cand = NULL;
  while ( ( state ( RQ1 ) != END ) || ( empty ( RQ1 ) == FALSE ) ||
          ( empty ( RQ2 ) == FALSE ) ) {          // browse partitions and merge
    while ( state ( RQ1 ) == EMPTY ) {     // wait until results are pipelined
       SYSTEMcall_nanosleep ( );
    }
    if ( head ( RQ1 ) == head ( RQ2 ) ) {      // if identical, add both heads
      push ( RQ, head ( RQ1 ) );
```

```
        push ( RQ, head ( RQ₂ ) );
        pop ( RQ₁ );
        pop ( RQ₂ );
      }
      else if ( head ( RQ₁ ) < head ( RQ₂ ) ) {     // if RQ₁ is smaller, add it
        push ( RQ, head ( RQ₁ ) );
        pop ( RQ₁ );
      }
      else {                                     // if RQ₂ value is smaller, add it
        push ( RQ, head ( RQ₂ ) );
        pop ( RQ₂ );
      }
    }
    // append non-empty queue to result queue
    if ( empty ( RQ₁ ) == FALSE ) {
      append ( RQ, RQ₁ );
    }
    else if ( empty ( RQ₂ ) == FALSE ) {
      append ( RQ, RQ₂ );
    }
    pop ( ES );                                        // discard __cnt counter
    release ( RQ₂[] );
    release ( RQ₂ );
    release ( RQ₁ );
  }
  ...
```

A second set of enhanced machine codes, i.e. `cvORDERBYathlc` and `cvORDERBYdthlc`, targets large collections. A new first phase divides the initial result queue of unordered collection values into $2^l$ queues that are ordered individually. The value of $l$ is based on the predicted size of the collection as known to the optimiser. In addition, a new final merging phase orders the values of all $2^l$ sorted result queues. These queues are sorted in pairs of two and pipelined to the next merging level until a final ordered result queue that contains all collection values is obtained.

A third set of operation codes, i.e. `cvORDERBYathplc` and `cvORDERBYdthplc`, targets very large collections that must be sorted portion-by-portion due to exhaustion of main memory. For such collections, unordered collection values are retrieved in blocks, which are sorted individually using one of the machine codes outlined above. Once those collection values have been sorted, the next block of unordered collection values is considered. Finally, all ordered intermediate collections are merged. This set of codes better supports the materialisation of intermediate results in order to free space in the ODBS node's main memory[8].

Furthermore, there are ordering machine instructions (e.g. `a<ORDERBYaheap`, `a<ORDERBYamerge`, `a<ORDERBYathmerge`, `cvORDERBYaextMerge` etc.) that better support arrays by implementing well known sorting algorithms such as variations of heap

[8] Consideration with respect to main memory restrictions and materialisation of intermediate results are beyond the scope of this thesis. We currently experiment with a mixture of explicit and implicit approaches that materialise intermediate results. In our second prototype (refer to Section 6.2), the heap is realised as virtual memory with the help of the page interface of a Caching Module [64]. Evaluation routines may mark result queues as being ready for materialisation. For instance, this is utilised in the implementation of operation codes `cvORDERBYathplc` and `cvORDERBYdthplc`.

sort and merge sort. Alternatively, sorting may also be based on an existing ordered index. Thus, no explicit sorting has to be performed. Corresponding operational semantics are not shown in detail but they are similar to those presented above.

iDBPQL supports a variety of join expressions. First, we consider the navigational join. It returns a collection of pairs of object references. For each pair, the second object is reachable from the first object by the specified path expression. Operational semantics for this join operation simply retrieve all qualifying starting objects and then compile pairs of references with all objects that can be reached over the specified path:

```
...
// OpCode: rvNAVJOIN ≅ navigational join
else if ( EVALnode.code is recognised as exp rvNAVJOIN pathExp ) {
  rtype ( exp ) RQ1;
  rtype ( pathExp ) RQ2;

  eval ( RQ1, exp );                          // evaluate collection expression
  while ( empty ( RQ1 ) == FALSE ) {  // for each collection value, evaluate
    push ( ES, unnest ( top ( RQ1 ) ) );                   // path expression
    eval ( RQ2, pathExpr );
    while ( empty ( RQ2 ) == FALSE ) {       // for each reachable value, add
      push ( RQ, ( top ( RQ1 ), pop ( RQ2 ) ) );  // respective result pairs
    }
    release ( RQ2 );
    pop ( ES );                            // restore the previous state of ES
    pop ( RQ1 );                 // cancel the result of the evaluation of exp
  }
  release ( RQ1 );
}
...
```

Similar to the navigational join, the natural join also has no associated join condition. Any two objects are joined if all their identically named instance variables have matching values. Thus, a collection of structured values is returned. Operational semantics are outlined for a basic loop join approach:

```
...
// OpCode: rvNATJOINloop ≅ natural join implemented as loop join
else if ( EVALnode.code is recognised as exp1 rvNATJOINloop exp2 ) {
  rtype ( exp1 ) RQ1, *r;
  rtype ( exp2 ) RQ2, *s;

  eval ( RQ1, exp1 );                   // evaluate left-hand side expression
  eval ( RQ2, exp2 );                   // evaluate left-hand side expression
  while ( ( r = next ( RQ1, r ) ) != NULL ) {    // loop through exp1 results
    while ( ( s = next ( RQ2, s ) ) != NULL ) {  // loop through exp2 results
      push ( ES, unnest ( s ) );               // unnest corresponding objects
      push ( ES, unnest ( r ) );
      if ( the top two elements on ES have the same values for all common
           fields ) {                          // match found, perform join next
        push ( RQ, pop ( ES ) & pop ( ES ) );
      }
266          }
267        }
268        release ( RQ2 );
269        release ( RQ1 );
270      }
271      ...
```

Inner and outer join expressions rely on an associated join condition to merge object variables into a structured value. There are two ways to how this join condition can be specified. On one hand, a boolean expression can be used to test which objects satisfy a particular join condition. This is the common approach as known from relational DBSs. On the other hand, a path expression can be given. Thus, the join expression is similar to a navigational join expression. However, the resulting value is of a different format. While the navigational join returns a collection of pairs of object references, a join expression with a path condition returns a collection of structured values.

We restrict ourselves to operational semantics for inner join expressions. Left outer join, right outer join and (full) outer join expressions can be formulated in a similar manner. Operational semantics for the inner join with a boolean expression as join condition can be specified as follows:

```
280      ...
281      // OpCode: rvINNJOINloop ≅ inner join implemented as loop join
282      else if ( EVALnode.code is recognised as exp1 rvINNJOINloop exp2 ON boolExp )
283      {
284        rtype ( exp1 ) RQ1, *r;
285        rtype ( exp2 ) RQ2, *s;
286        boolean RQ3;
287
288        eval ( RQ1, exp1 );                      // evaluate left-hand side expression
289        eval ( RQ2, exp2 );                      // evaluate left-hand side expression
290        while ( ( r = next ( RQ1, r ) ) != NULL ) {     // loop through exp1 results
291          while ( ( s = next ( RQ2, s ) ) != NULL ) {  // loop through exp2 results
292            push ( ES, unnest ( s ) );                // unnest corresponding objects
293            push ( ES, unnest ( r ) );
294            eval ( RQ3, boolExp );
295            if ( top ( RQ3 ) == TRUE ) {              // match found, perform join next
296              push ( RQ, pop ( ES ) & pop ( ES ) );
297            }
298            pop ( RQ3 );            // cancel the result of the evaluation of boolExp
299          }
300        }
301        release ( RQ3 );
302        release ( RQ2 );
303        release ( RQ1 );
304      }
305      ...
```

Result values are created through a concatenation operator as outlined in line 296: `pop ( ES ) & pop ( ES )`.

The inner join with a path expression is more complicated to evaluate. It is possible that the path expression (starting from exp$_1$) identifies a larger set of objects than exp$_2$.

Thus, we have to double-check, that only such objects, which are reachable from exp₁ over path expPath and also members of the collection that results from the evaluation of exp₂, are considered. Corresponding operational semantics are as follows:

```
...
// OpCode: rvINNJOINpath ≅ inner join with path expression
else if ( EVALnode.code is recognised as exp1 rvINNJOINpath exp2 ON pathExp )
{
  rtype ( exp1 ) RQ1, *r;
  rtype ( exp2 ) RQ2, RQ3 *s;
  boolean RQ4;

  eval ( RQ1, exp1 );                   // evaluate left-hand side expression exp1
  eval ( RQ2, exp2 );                  // evaluate right-hand side expression exp2
  while ( ( r = next ( RQ1, r ) ) != NULL ) {     // for each result of exp1 do
    POScall_FindEnclosure ( RQ3, r, pathExp, NULL );      // obtain all objects
                                         // that are reachable from r over pathExp
    while ( ( s = next ( RQ3, s ) ) != NULL ) {  // for each reachable object
      SYSTEMcall_isInstanceOf ( RQ4, s, RQ2 );      // test that s is a member
                     // of the collection resulting from the evaluation of exp2
      if ( top ( RQ4 ) == TRUE ) {                  // match found, perform join
        push ( ES, unnest ( r ) );
        push ( ES, unnest ( s ) );
        push ( RQ, pop ( ES ) & pop ( ES ) );     // add joined value to result
      }
      pop ( RQ4 );                   // cancel the result of the isInstanceOf test
    }
    pop ( RQ3 );             // cancel the result of following the path condition
  }
  release ( RQ4 );
  release ( RQ3 );
  release ( RQ2 );
  release ( RQ1 );
}
...
```

All join expressions have been implemented with the help of loop joins. This basic approach leaves numerous possibilities for optimisation. For instance, there are alternative machine codes that utilise pipelining when retrieving results of left-hand side and right-hand side expressions, consider blocks of collection values over the whole collection value, rely on sorting or indices etc.

**Controlling the Flow of Serial Data Processing.** Serial evaluation is directed by a number of control flow statements. First, we consider conditional and loop statements. These statements have one or more evaluation blocks associated. Each block will be evaluated in its own sub-frame. Sub-frames that correspond to loop or the SWITCH statements may be named. Having an association between blocks and sub-frames eases the implementation of statements such as the BREAK statement.

Operational semantics for the conditional IF ... THEN ... ELSE statement can be outlined as follows:

```
...
// OpCode: xxIFTHEN ≅ conditional statement
else if ( EVALnode.code is recognised as
          IF boolExp THEN blockStmt ELSE stmt ) {
  boolean RQ$_1$;

  eval ( RQ$_1$, boolExp );                        // evaluate boolean expression
  if ( top ( RQ$_1$ ) == TRUE ) {   // if boolExp evaluates to TRUE, follow the
    eval ( NULL, blockStmt );                                      THEN branch
  else {                           // if boolExp evaluates to FALSE, follow the
    eval ( NULL, stmt );                                           ELSE branch
  }
  release ( RQ$_1$ );
}
...
```

The evaluation of IF ... THEN ... ELSE IF ... ELSE statements is supported by allowing a statement to appear after the ELSE keyword. This statement may either be another conditional IF statement or an evaluation block that corresponds to the ELSE branch.

iDBPQL supports a number of loop statements. The simple, non-terminal loop statement LOOP can be described as follows:

```
...
// OpCode: xxLOOP ≅ non-terminal loop statement
else if ( EVALnode.code is recognised as LOOP DO stmt ENDDO ) {
  openNewSubScope ( __labelAnnotation, ES.transFlag );     // named sub-frame
  while ( TRUE ) {
    eval ( NULL, stmt );                               // execute loop statements
  }
  closeSubScope ( );                                  // remove the loop sub-frame
}
...
```

The evaluation of terminal loop statements is slightly more complex. The serial WHILE loop statement has the following operational semantics:

```
...
// OpCode: xxWHILEDO ≅ while loop statement
else if ( EVALnode.code is recognised as WHILE boolExp DO stmt ENDDO ) {
  boolean RQ$_1$;

  openNewSubScope ( __labelAnnotation, ES.transFlag );     // named sub-frame
  eval ( RQ$_1$, boolExp );                           // evaluate the loop condition
  while ( pop ( RQ$_1$ ) == TRUE ) {         // loop while RQ$_1$ holds a TRUE value
    eval ( NULL, stmt );                               // execute loop statements
    eval ( RQ$_1$, boolExp );                      // re-evaluate the loop condition
  }
  release ( RQ$_1$ );
  closeSubScope ( );                                  // remove the loop sub-frame
}
...
```

Non-serial versions of loop statements are discussed further below in Section 5.3.9.

Similarly to the WHILE loop statement, operational semantics for DO ... WHILE loops can be formulated as follows:

```
...
// OpCode: xxDOWHILE ≅ do-while loop statement
else if ( EVALnode.code is recognised as DO stmt ENDDO WHILE boolExp ) {
  boolean RQ1;

  openNewSubScope ( __labelAnnotation, ES.transFlag );     // named sub-frame
  push ( RQ1, TRUE );                                        // initialise RQ1
  while ( pop ( RQ1 ) == TRUE ) {         // loop while RQ1 holds a TRUE value
    eval ( NULL, stmt );                            // execute loop statements
    eval ( RQ1, boolExp );                     // re-evaluate the loop condition
  }
  release ( RQ1 );
  closeSubScope ( );                                 // remove the loop sub-frame
}
...
```

In contrast to the loop statements discussed above, the FOR EACH loop determines its point of termination on whether or not all members of a particular collection have been processed. As already indicated, access to individual collection members is possible through the AS renaming expression.

```
...
// OpCode: xxFOREACH ≅ for each loop statement
else if ( EVALnode.code is recognised as FOR EACH exp DO smts ENDDO ) {
  rtype ( exp ) RQ1;

  openNewSubScope ( __labelAnnotation, ES.transFlag );     // named sub-frame
  eval ( RQ1, exp );        // obtain the collection on which loop is executed
  while ( empty ( RQ1 ) == FALSE ) {              // loop while RQ1 is not empty
    push ( ES, unnest ( top ( RQ1 ) ) ); // move collection member into scope
    eval ( NULL, stmt );                            // execute loop statements
    pop ( ES );                                 // restore previous state of ES
    pop ( RQ1 );         // discard value at the heap of the loop collection
  }
  release ( RQ1 );
  closeSubScope ( );                                 // remove the loop sub-frame
}
...
```

Besides the IF ... THEN ... ELSE statement, there is a second conditional statement. Such a SWITCH statement may have a default statement block associated. Operational semantics for SWITCH statements with and without a default statement block are as follows:

```
...
// OpCode: xxSWITCH ≅ conditional SWITCH statement
else if ( EVALnode.code is recognised as SWITCH exp { CASE exp1 : blockStmt1
          ... CASE expn : blockStmtn } ) {
  rtype ( exp ) RQ1, RQ2;

  openNewSubScope ( __labelAnnotation, ES.transFlag );        // named sub-frame
  push ( ES, (__binder) ( __cnt, INT, 1 ) );                   // CASE-block counter
  eval ( $RQ_1$, exp );                                  // evaluate the SWITCH expression
  while ( bind ( __cnt ).e <= n ) {                          // for each CASE-block do
    eval ( $RQ_2$, $exp_{bind(\_cnt).e}$ );            // evaluate the current CASE expression
    if ( top ( $RQ_0$ ) == top ( $RQ_2$ ) ) {                  // test whether the CASE
                                               // expression fulfils the SWITCH expression
      eval ( NULL, $stmtBlock_{bind(\_cnt).e}$ );                // evaluate statement block
    }
    else {
      pop ( $RQ_2$ );              // discard result of the current CASE expression
      bind ( __cnt ).e++;                                  // increment CASE-block counter
    }
  }
  pop ( ES );                                        // remove binder for counter variable
  release ( $RQ_2$ );
  release ( $RQ_1$ );
  closeSubScope ( );                                        // remove the switch sub-frame
}

// OpCode: xxSWITCHDEF ≅ conditional SWITCH statement with DEFAULT-block
else if ( EVALnode.code is recognised as SWITCH exp { CASE $exp_1$ : $blockStmt_1$
          ... CASE $exp_n$ : $blockStmt_n$ DEFAULT : $blockStmt_{n+1}$ } ) {
  rtype ( exp ) $RQ_1$, $RQ_2$;
  openNewSubScope ( __labelAnnotation, ES.transFlag );        // named sub-frame
  push ( ES, ( __cnt, INT, 1 ) );                      // counter to navigate CASE blocks
  push ( ES, ( __match, BOOL, FALSE ) );  // TRUE if a matching case is found
  eval ( $RQ_1$, exp );                                  // evaluate the SWITCH expression
  while ( bind ( __cnt ).e <= n ) {                          // for each CASE-block do
    eval ( $RQ_2$, $exp_{bind(\_cnt).e}$ );            // evaluate the current CASE expression
    if ( top ( $RQ_0$ ) == top ( $RQ_2$ ) ) {                  // test whether the CASE
                                               // expression fulfils the SWITCH expression
      bind ( __match ).e = TRUE;                          // matching CASE-block exists
      eval ( NULL, $stmtBlock_{bind(\_cnt).e}$ );                // evaluate statement block
    }
    else {
      pop ( $RQ_2$ );              // discard result of the current CASE expression
      bind ( __cnt ).e++;                                  // increment CASE-block counter
    }
  }
  if ( bind ( __match ).e == FALSE ) {            // if no CASE-block matched then
    eval ( NULL, $blockStmt_{n+1}$ );                             // execute the default one
  }
  pop ( ES );                                  // remove binder for the __match variable
  pop ( ES );                                        // remove binder for counter variable
  release ( $RQ_2$ );
  release ( $RQ_1$ );
  closeSubScope ( );                                        // remove the switch sub-frame
}
...
```

Besides conditional and loop statements, there are also control flow statements that interrupt the serial flow of execution. The RETURN statement terminates the processing of the current evaluation plan. As a result, all sub-frames that form a part of the top-most frame, its associated result queues and its result stack are released. Subsequently, processing continues with the evaluation plan that previously invoked the one that has just been terminated. After the `RETURN` statement has been evaluated, the return result queue is the only structure that remains accessible. Of course, object constructors and behaviour invocations that have the `VOID` type as return type do not have such a return result queue associated.

Alternatively, the continuous execution flow may be interrupted by the `BREAK` statement. This statement often appears together with the `LABEL` statement. However, a `LABEL` statement is not encountered explicitly during the evaluation procedure. As discussed earlier, labels are transformed into annotations. Such label annotations appear in the form of named sub-frames as already indicated above when operational semantics for loop and the `SWITCH` statements have been presented. The evaluation of a `BREAK;` statement terminates the processing of the current loop or `SWITCH` statement. As a result, the current sub-frame together with its associated result queues are discarded first. This current sub-frame corresponds to a `DO ... ENDDO` evaluation block. The next sub-frame, which is also discarded, either corresponds to a loop sub-frame or a switch sub-frame. Thus, all information that is local to the loop or `SWITCH` statement is removed. The only data, which remains accessible, must have been previously associated with binders (or queues) that are located in sub-frames outside the most local loop or `SWITCH` statement.

In addition to the `BREAK;` statement, there is the `BREAK` *`labelId`*`;` statement. If such a statement is encountered, the procedure, which has been described for the `BREAK` statement, is applied until the first loop or `SWITCH` statement that has a corresponding sub-frame with name *labelId* is encountered.

Finally, there is the `WAIT [ labelId ];` statement. Its semantics will be discussed in greater detail in Section 5.3.9.

**Invocation of Behaviours.** iDBPQL supports three types of behaviours, i.e. type operations, method calls and object constructors. Such behaviours are implemented through evaluation plans. For instance, we might have an evaluation plan that implements a simple type operation, the union over a collection, a static method, an instance method, an object constructor etc. The general semantics of a behaviour invocation are similar for all types of behaviours: A new scope (i.e. frame), which holds the behaviour's arguments and its local environment, is opened on the environment stack. In addition, the run-time type or class of the value or object, respectively, on which the behaviour was invoked together with the arguments that are provided during the invocation, are considered to decide which particular evaluation plan is chosen for execution. In particular, method invocation follows the single dispatch approach as discussed before. To facilitate the binding of an evaluation plan to a type operation, method invocation or object constructor the `bindTypeOpEvalPlan` and `bindMethodEvalPlan` routines have been introduced. In contrast to type operation invocations, invocations of instance methods and object constructors result in the update of the `THIS` pointer that is associated with every frame. The `THIS` pointer is set to point to the object on which the method or constructor was invoked.

The treatment of static class methods is different to that of instance methods. The evaluation plan to be processed can already be determined at compile time. Thus, invoking a class method is based upon a binding annotation.

Operational semantics for all types of behaviour invocations are as follows:

```
...
// OpCode: rvMETHCALLdyn ≅ dynamic invocation of an instance method
else if ( EVALnode.code is recognised as rvMETHCALL methodName ( arg1, ...,
          argn ) on object obj {
  openNewScope ( obj );       // create new frame; obj becomes the THIS object
  push ( ES, unnest ( [ arg1, ..., argn ] ) );  // create argument name binders
  eval ( RQ, bindMethodEvalPlan ( methodName ) );             // evaluate method
  closeScope ( );                                        // discard current scope
}

// OpCode: rvMETHCALLsta ≅ static invocation of a class method
else if ( EVALnode.code is recognised as rvMETHCALL methodName ( arg1, ...,
          argn ) {
  openNewScope ( NULL );                                     // create new frame
  push ( ES, unnest ( [ arg1, ..., argn ] ) );  // create argument name binders
  eval ( RQ, __evalPlanAnnotation );                  // evaluate static method
  closeScope ( );                                        // discard current scope
}

// OpCode: rvCONSTRCALL ≅ object constructor invocation
else if ( EVALnode.code is recognised as rvCONSTRCALL constrName ( arg1, ...,
          argn ) on object obj {
  openNewScope ( obj );       // create new frame; obj becomes the THIS object
  push ( ES, unnest ( [ arg1, ..., argn ] ) );  // create argument name binders
  eval ( NULL, bindMethodEvalPlan ( constrName ) );    // evaluate constructor
  closeScope ( );                                        // discard current scope
}

// OpCode: nrTYPOPCALL ≅ type operation invocation
else if ( EVALnode.code is recognised as nrTYPOPCALL typOpName ( arg1, ...,
          argn ) {
  openNewScope ( NULL );                                     // create new frame
  push ( ES, unnest ( [ arg1, ..., argn ] ) );  // create argument name binders
  eval ( RQ, bindTypeOpEvalPlan ( typOpName ) );    // evaluate type operation
  closeScope ( );                                        // discard current scope
}
...
```

**Object Creation and Assignments.** Object creation and assignments have side effects. Creating a new object of a class that resides in the run-time metadata catalogue will:

– Create a new object in the heap's main memory object store;
– Verify that all associated constraints are met;
– Add a reference to this object (i.e. a main memory pointer) to the shallow and deep extent of its class; and
– Add a reference to this object to the deep extent of all super-classes of its class that also reside in the run-time metadata catalogue.

However, a transient class may also sub-class one or more persistent classes. The inheritance relation (i.e. the `__dag` associated with the main evaluation plan's run-time metadata entry) bridges this gap between transient and persistent classes. Access by class name, e.g. to the persistent class `PersonC` with its persistent sub-classes `StudentC` and `AcademicC`$_{CC}$, and its transient sub-class `StudentAcademicC`$_{CC}$, will result in:

1. Retrieving references from classes `PersonC`, `StudentC` and `AcademicC`$_{CC}$ to all objects through the `POScall_FindFromCollection` primitive;
2. Retrieving references from the deep extent of class `StudentAcademicC`$_{CC}$ to all its instances; and
3. Returning the union of the collections resulting from steps 1 and 2.

Due to the fact that persistent classes never sub-class transient classes, the creation of a new object of a class that resides in the DBS metadata catalogue is less complex but it involves the persistent object store. Steps to be performed are as follows:

– Create a new object in the shared memory area of POS. This is achieved by invoking the `POScall_AddNewObject` primitive;
– Verify that all associated constraints are met; and
– Add a reference to this object (i.e. its OID) to the collection-class that is maintained by POS. This is achieved by invoking the `POScall_InsertObject` primitive.

Example 5.5 (6) outlines corresponding POS-level calls that are executed when creating a new (persistent) object.

Assignment operations may also affect the persistent object store. In the event that the value of an object, which resides in POS's shared memory area, is updated, the `POScall_UpdateObject` primitive must be invoked.

`POScall` primitives are always encapsulated into a transaction block. Corresponding details are discussed further below.

**Cast Expressions, `SUPER` and `THIS`.** The evaluation of cast expressions, invocations that contain the `SUPER` keyword and the `THIS` keyword remain to be discussed in greater detail.

As already mentioned, the evaluation of the `THIS` keyword takes advantage of a special `THIS` pointer that is maintained with every frame. It refers to the name binder on ES that corresponds to the current object on which evaluation takes place.

The evaluation of a cast expression affects the run-time type of all values that are returned as the result of the processing of the associated expression. Updating an value's associated type may trigger type conversion. For instance, when discussing expressions with binary operators, we have seen the application of such a type conversion from a Natural value into an Integer value. If a cast expression is associated with a reference value, it is verified that the value refers to an object whose class is compatible with the specified cast.

Finally, we turn our attention to the evaluation of invocation statements that contain the `SUPER` keyword. On one hand, the evaluation plan that implements an object constructor may begin with a `SUPER ( );` call or a sequence of `SUPER ( cName );`

or `SUPER ( cName, arg`$_1$`, ..., arg`$_n$` );` calls. Operational semantics are based on semantics of behaviour invocations. The `SUPER ( );` call will result in a sequence of object constructor invocations for all direct super-classes in the same order as they are specified in the current object's inheritance clause. This information is obtained by invoking the `SYSTEMcall_getSuperClasses ( obj )` primitive, where `obj` is the current object. If an argument is provided, the first argument always identifies the class (e.g. class `cName`) from which an object constructor should be selected for invocation. The remaining arguments `arg`$_1$`, ..., arg`$_n$ determine which of the available constructors must be invoked. Analogous to the invocation of methods and object constructors, the `bindMethodEvalPlan` primitive is utilised for this purpose.

On the other hand, an evaluation plan that implements a method may contain a `SUPER ( ).methodCall ( ... );` or `SUPER ( cName ).methodCall ( ... );` behaviour invocation statement. Again, operational semantics are based on those for behaviour invocation. The evaluation routine selects the evaluation plan that corresponds to the specified method of the named class or, if no class name is specified, the matching method with the highest priority (refer to Section 4.2.5 on page 82). The method `methodCall` is invoked on the same object on which the current method was invoked. Hence, the implementation of a super-class's method is re-used.

### 5.3.7 Evaluating Statements and Blocks of Statements

The processing of a whole statement can be regarded as the evaluation of a sequence of expressions, which consist of other simpler expressions, operators, keywords and literals. We may execute one expression after the other, utilise pipelining, multi-threading, distribution or a mixture of those. If intermediate results are returned at once, they may be materialised to free main memory space. These processes are similar to those known from relational DBSs [37, 43, 50, 103, 118].

Statements themselves do not return any values. In fact, they are executed for their side effects. Statements are executed one after the other, multi-threaded or distributed. First, we will turn our attention to issues arising when processing blocks of statements that are encapsulated in an evaluation block. We restrict ourselves to regular `DO` ... `ENDDO` blocks, `DO ATOMIC` ... `ENDDO` blocks and `DO TRANSACTION` *tid* ... `ENDDO` blocks. The remaining types of blocks that utilise simultaneous processing are discussed in Section 5.3.9. Details with respect to distributed evaluation are outlined in Section 5.3.10.

**Blocks of Statements.** As outlined in Section 5.1.5, metadata references are associated with evaluation blocks. The main two tasks that have to be performed when such a block is encountered consist of:

1. Creating a new sub-frame on the environment stack that is used to hold the block's local environment; and
2. Add a name binder for each attached metadata reference.

Corresponding operational semantics can be formulated as follows:

```
...
// OpCode: xxDO ≅ simple statement block
else if ( EVALnode.code is recognised as xxDO stmt ENDDO; with an attached
        metadata annotation (__symbInfo *) symbols[] ) {
  openNewSubScope ( __labelAnnotation, ES.transFlag );      // named sub-frame
  while ( there exists another metadata reference __cnt in the symbols
        array ) {
    if ( __symbols[__cnt].__symbolDescriptor corresponds to a
         non-reference-type ) {
      // add a binder for a variable of a non-reference-type; its value is
      // initialised using the default type initialiser
      push ( ES, (__binder) ( __symbols[__cnt].name,
             typeOf ( __symbols[__cnt] ), bindTypeOpEvalPlan ( INIT ) ) );
    }
    else {
      // add a binder for a variable of a reference-type; its value is
      // initialised to NULL; a subsequent invocation of an object
      // constructor must occur before this variable is accessed
      push ( ES, (__binder) ( __symbols[__cnt].name,
             typeOf ( __symbols[__cnt] ), NULL ) );
    }
  }
  eval ( NULL, stmt );      // execute the statement(s) enclosed in this block
  while ( there exists another metadata reference in the symbols array ) {
    pop ( ES );                                 // restore the state of ES
  }
  closeSubScope ( );                         // remove the top-most sub-frame
}
...
```

The evaluation of all other types of evaluation blocks will have to include these steps as well.

**Simple Transactions.** Blocks are used to model transactions. First, we concentrate on simple transactions that are executed on the local ODBS node within the same execution thread. The processing of statements, expressions and operators within such transaction blocks must be monitored. The transaction management system will ensure that serialisability and recoverability properties are not violated when accessing shared data.

Access to shared data is possible through the service interface of the persistent object store. Thus, the `unnest` operator, the `NEW` keyword and `POScall` primitives are the only operations that can directly access / create shared data. Once a reference to a shared object has been obtained, this reference is placed on either the environment stack or in a result queue on a result stack. Subsequent operations on such binders or intermediate results must also be brought to the attention of the TMS.

EXAMPLE 5.7. Let us revisit the extended university application as discussed in Example 4.16 (on page 98). For instance, consider the following four statements:

```
...
(SET < StudentC >) x = StudentC WHERE ( name.lastName == "Kirchberg" );
```

```
03    (SET < StudentAcademicC_CC >) y = StudentAcademicC_CC WHERE
04      ( name.lastName == "Kirchberg" );
05    (SET < StudentC >) z = x.UNION ( y );
06    RETURN ( z.COUNT ( ) );
07  }
```

The first statement in line 02 results in a collection of references to persistent objects being placed on the environment stack. The second statement outlined in lines 03 and 04 leaves a collection of references to transient objects on the environment stack. The third statement creates a union of the two collections that have been generated in the previous two steps. This unified collection consists of references to both objects held in the heap and objects maintained by POS. Finally, the number of elements in the unified collection is returned. With respect to the transaction property, the first, third and fourth statements affect shared data. The second statement only accesses a locally maintained collection. □

The state of an object or the value of any object's instance variables may only be modified if a reference to the respective object is available. This is naturally true when an instance method is invoked. However, iDBPQL also permits direct access to structural class members. Modifying the value of an instance variable, say `name.lastName`, of object `obj` is permitted only if:

- `name.lastName` succeeds an object reference `ref` that refers to `obj`. Thus, `ref.name.lastName` is the corresponding code segment; or
- `name.lastName` is specified in an environment in which the current `THIS` pointer refers to object `obj`.

Thus, modifying the value of a variable that is no longer attached to an object (i.e. as a result of a projection operation) will have no effect on the state of its former object.

Modifications are always performed on the environment stack. If an update affects an object, some additional evaluation tasks have to be performed:

1. The update is executed;
2. Constraints associated with any modified instance variable and all class constraints are verified; and
3. Depending on whether the object is transient or persistent, the update is reflected on the main memory object store or POS's shared memory store, respectively.

Similar to object access, modifications of shared objects or collections referencing shared objects must be synchronised with the transaction management system. Before we introduce how this synchronisation is achieved, it is outlined how the evaluation component supports the transaction concept.

A transaction flag is assigned to each sub-frame on ES. The initial value of this flag is `__none`. The `__none`-value indicates that no transaction block has been encountered since the beginning of the processing of the current request or since the execution of the last transaction was completed. As the first or next `DO TRANSACTION` *tid* `stmt ENDDO;` block is encountered, the following steps are performed in addition to those steps that are executed for every evaluation block: The value of the new sub-frame's transaction flag is set to the identifier of the transaction. This indicates that every subsequent evaluation, which involves shared data, must be monitored by the TMS. Such evaluations are those that create and update persistent objects.

Operational semantics of the **xxDOTRANS** machine instruction are similar to the semantics of the xxDO operator code. However, additional tasks must be performed at the beginning and end of the block's evaluation:

```
...
// OpCode: xxDOTRANS ≅ transaction statement block
else if ( EVALnode.code is recognised as xxDOTRANS tid stmt ENDDO; with an
          attached metadata annotation (__symbInfo *) symbols[] ) {
  INT RQ1;

  if ( ES.transFlag == "__none" ) {          // true if no transaction is active
    TMScall_openTrans ( NULL, tid );           // notify TMS of new transaction
    openNewSubScope ( __labelAnnotation, tid ); // create a new transactional
  }                                                                // sub-frame
  else if ( ES.transFlag == "__approved" ) {  // true if a sub-transaction is
  // encountered; ignore, its parent transaction has been approved by the TMS
    openNewSubScope ( __labelAnnotation, ES.transFlag ); // same as for xxDO
  }
  else {    // true only if another transaction is active that was explicitly
      // interleaved in the user request; interrupt; will be continued later
    TMScall_openTrans ( NULL, tid );           // notify TMS of new transaction
    openNewSubScope ( __labelAnnotation, tid ); // create a new transactional
  }                                                                // sub-frame

  while ( there exists another metadata reference __cnt in the symbols
          array ) {
    if ( __symbols[__cnt].__symbolDescriptor corresponds to a
         non-reference-type ) {
      // add a binder for a variable of a non-reference-type; its value is
      // initialised using the default type initialiser
      push ( ES, (__binder) ( __symbols[__cnt].name,
             typeOf ( __symbols[__cnt] ), bindTypeOpEvalPlan ( INIT ) ) );
    }
    else {
      // add a binder for a variable of a reference-type; its value is
      // initialised to NULL; a subsequent invocation of an object
      // constructor must occur before this variable is accessed
      push ( ES, (__binder) ( __symbols[__cnt].name,
             typeOf ( __symbols[__cnt] ), NULL ) );
    }
  }
  eval ( NULL, stmt );      // execute the statement(s) enclosed in this block
  while ( there exists another metadata reference in the symbols array ) {
    pop ( ES );                                      // restore the state of ES
  }

  if ( ES.transFlag == "__none" ) {          // true if no transaction is active
    closeSubScope ( );                   // same as the xxDO machine instruction
  }
  else if ( ES.transFlag == "__approved" ) {  // true if a sub-transaction is
```

```
                        // encountered; ignore since parent transaction is serialised
        closeSubScope ( );                   // same as the xxDO machine instruction
      }
      else {                             // true if transaction is still active; commit
        TMScall_commitTrans ( RQ1, tid );
        if ( top (RQ1 ) == RESTART ) {       // test whether commit was successful
          restart transaction or process from savepoint[9];
        }
        closeSubScope ( );                          // remove the top-most sub-frame
      }
    }
    ...
```

Operational semantics show that three different transactional states of a sub-frame may be encountered. If no transaction is active (i.e. transFlag carries the __none-value), a new transactional sub-frame is created. Every evaluation that is performed in the sub-frame must be monitored by the TMS. As outlined in greater detail below, this monitoring process may set the value of the transaction flag of another sub-evaluation to the __approved-value. This can only occur when another behaviour is invoked and the TMS has sufficient knowledge to determine serialisability and recoverability for the evaluation of the entire behaviour implementation (e.g. compatibility information for this behaviour has been provided explicitly by the behaviour's programmer or derived automatically by an internal mechanism[10]). Every sub-evaluation performed in a sub-frame, which has such an __approved flag, does not have to be monitored by the TMS. Monitoring continues once the approved behaviour has been evaluated successfully. The third state relates to situations as discussed in Section 4.4.2. Two or more transactions may be interleaved explicitly by the programmer. In order to support such a functionality, the TMScall primitive filters out repeated openTrans requests that result from multiple manual interleavings of two or more transactions. This is easy to do since this primitive already maps user-level transaction identifiers to internal transaction identifiers.

Accordingly, during the evaluation process, the following additional measures have to be taken if the top-most sub-frame carries a transaction identifier:

- For each encountered eval ( rq, eNode ), evalThreaded ( rq, eNode ) or evalDistributed ( rq, eNode ) routine, we execute the following additional operational semantics prior to the respective evaluation routine:

```
{
  INT RQ1;

  TMScall_execute ( RQ, ES.transFlag, eNode );
  if ( top ( RQ1 ) == APPROVED ) {          // true if permission is granted
    if ( eNode describes the invocation of a method or object constructor )
```

[9] At this point, we omit any discussion of how the recovery manager will interact with the evaluation procedure to correctly abort one or more transactions. Instead, we simply assume that the evaluation plan is altered appropriately at run-time.

[10] Means of providing or deriving compatibility information for user-defined behaviours are beyond the scope of this thesis. However, it should be mentioned that iDBPQL's transaction mechanism functions with and without the ability of adding such compatibility information.

```
      {
        invoke the respective evaluation but enforce a change of the
          transFlag-value (set to "__approved") in that behaviour's first
          sub-frame;
      }
      else {
        invoke the respective evaluation;
      }
    }
    else if ( top ( RQ1 ) == RESTART ) {     // true if an abort was performed
      restart transaction or process from savepoint;
    }
    else if ( top ( RQ1 ) == MONITOR ) {    // true if the body of a behaviour
        // invocation must also be monitored; thus, forward current transFlag
      invoke the respective evaluation;
    }
  }
```

Finally, an evaluation plan may contain one of the method invocations as outlined in the `Transaction` class (refer to page 92). Considering the `commit ( );` and `abort ( );` call, a transaction is terminated early, i.e. before the end of the respective transaction block is reached. Operational semantics for the `commit` method are as follows:

```
  {
    INT RQ1;

    TMScall_commitTrans ( RQ1, tid );
    if ( top ( RQ1 ) == RESTART ) {      // test whether commit was unsuccessful
      restart transaction or process from savepoint;
    }
    set the transaction flag of the top-most sub-frame to __none;
  }
```

Analogously, the processing of the `abort` method can be formulated as follows:

```
  {
    TMScall_abortTrans ( NULL, tid );
    set the transaction flag of the top-most sub-frame to __none;
  }
```

**Atomic Statement Blocks.** A further type of block statements are atomic blocks. They facilitate a grouping construct for a number of statements, which are meant to be executed at once. As a result, an update operation that affects an object is now performed as follows:

1. Exclusive access to the affected object is obtained;
2. The update is executed;
3. Constraints associated with any modified instance variable and all class constraints are deferred to the end of the atomic block; and
4. Depending on whether the object is transient or persistent, the update is reflected on the main memory object store or POS's shared memory store, respectively. However, exclusive access to the object is retained until the end of the atomic block is reached.

This implies that a new object is not immediately added to a transient class's shallow and deep extents nor is it added to a persistent class's class-collection. These updates are also deferred to the end of the respective atomic block.

### 5.3.8 Processing Evaluation Plans

An evaluation plan consists of an optional initialisation block and an evaluation block. Both blocks are evaluated in the same manner as discussed in Section 5.3.7. The only difference concerns the treatment of the main evaluation plan. During this first step of the evaluation of a user request, the environment stack must be initialised and its global environment has to be generated. Metadata references that describe this global environment are attached to the evaluation graph's root node of the initialisation block. This block is evaluated as any other `xxDO` evaluation block.

The main evaluation plan is processed in its own REE stack area. At first, an empty environment stack is initialised. Subsequently, the evaluation may commence. Operational semantics are as follows:

```
...
// OpCode: xxMAIN ≅ main evaluation plan
else if ( EVALnode.code is recognised as xxMAIN initBlock evalBlock ) {
  openNewScope ( NULL );                          // create the first frame on ES
  eval ( NULL, initBlock );                        // initialise evaluation stack
  eval ( RQ, evalBlock );                // commence evaluation of the user request
  closeScope ( );                                  // discard the last frame on ES
}
...
```

Discarding the last frame on ES will result in the destruction of the corresponding REE stack area. Only the return result queue `RQ` remains present in the heap. This queue is only removed from the heap when the higher-level component that initiated the execution of this request terminates or releases its hold on the request queue. The heap's garbage collector will oversee this process.

### 5.3.9 Simultaneous Evaluation of Statements and Expressions

Previously, we have already seen how concurrent processing may be utilised to enhance the processing of internal operation codes. Recall the usage of the `evalMultiES` routine. However, iDBPQL also allows simultaneous processing to be requested explicitly. In particular, semantics of `INDEPENDENT DO` ... `ENDDO` blocks, `FOR EACH exp CONCURRENT DO` ... `ENDDO` loop statements, and `WAIT` statements remain to be discussed. Corresponding operation semantics are based on the `evalThreaded` routine. It splits the current execution stream into two separate streams. The new stream will be assigned to a new REE stack area and operates on its own environment stack. This stack, however, is not empty at the beginning of the processing. Instead, it will be initialised by cloning the main stream's ES. Thus, both streams continue processing on identical environments but execute independently on a large scale from then on. The `WAIT` statement corresponds to the only means of synchronising processing between two or more execution threads that originate from a common main execution stream.

Operational semantics of those blocks and statements can be summarised as follows:

```
...
// OpCode: xxDOINDEP ≅ multi-threaded statement block
else if ( EVALnode.code is recognised as xxDOINDEP stmt ENDDO; ) {
  evalThreaded ( NULL, xxDO stmt ENDDO; );
}

// OpCode: xxFOREACHCONC ≅ multi-threaded for each loop statement
else if ( EVALnode.code is recognised as FOR EACH exp CONCURRENT DO stmt THEN
          blockStmt ENDDO; ) {
  rtype ( exp ) RQ1, RQ2;

  evalThreaded ( RQ1, exp );
  evalThreaded ( RQ2, xxFOREACH RQ1 xxDO stmt ENDDO; );
  evalThreaded ( NULL, xxFOREACH RQ2 blockStmt );
}

// OpCode: xxWAIT ≅ wait statement
else if ( EVALnode.code is recognised as WAIT; ) {
  while ( wait until a spawned threads have terminated ) {
    SYSTEMcall_nanosleep ( );
  }
}

// OpCode: xxWAITid ≅ wait statement with label
else if ( EVALnode.code is recognised as WAIT labelId; ) {
  while ( wait until the spawned thread with identifier labelId has
          terminated ) {
    SYSTEMcall_nanosleep ( );
  }
}
...
```

A different means of simultaneous execution is discussed in Section 5.3.10. Instead of utilising multi-threading, sub-evaluations are distributed to other ODBS nodes.

### 5.3.10 Distributed Processing of Evaluation Plans

iDBPQL allows evaluation processes to be distributed and data to be transfered for distributed processing. Figure 5.5 (on page 170) has already indicated that a second shared memory may be embedded into the local heap. This area is maintained by the remote communication module, which facilitates access to remote objects and also supports the distribution of (sub-)evaluations. Object migration is initialised by the `unnest` operator as mentioned in Section 5.3. In order to distribute processing to a remote node, a separate evaluation routine `void evalDistributed ( RQ * rq, EVALnode * eNode );` is defined. This routine is similar to the `evalThreaded` routine but processing continues on a remote ODBS node instead of another local thread.

**Migrating Objects.** The support of object migration improves the accessibility of objects and enables load balancing [22]. However, it also requires a more sophisticated transaction management system. The iDBPQL run-time system supports the temporary migration of individual objects. Permanent migration is only possible by changing the allocation of classes.

Temporary object migration from a remote node to the local node is executed in four steps:

1. Ensure that the object is not actively involved in another invocation (otherwise, wait until the object become available);
2. Notify the remote transaction management system of a temporary transfer of ownership of the object (i.e. issue a `TMScall_transferOwner` call);
3. Move the requested object (together with its metadata information) to the local node; and
4. Leave a forwarding reference on the remote node[11].

Once an object has been migrated, the local transaction management system takes over the responsibility of serialising object access. A migrated object is returned as soon as it is no longer needed on the local node.

**Processing Evaluation Plans on a Remote Node.** Instead of moving objects to a remote location, we can also distribute the processing to the location where the object (or the majority of objects) resides. This occurs whenever a location annotation is encountered during the execution of an evaluation plan. Such location annotations are attached to an evaluation graph's *ctrlFLOWedge*. When encountering such an annotation, the following operational semantics are invoked:

```
...
else if ( EVALnode.code is recognised as expression exp with a location
          annotation node ) {
  rtype ( exp ) RQ1;

  notify the TMS of the impending distributed evaluation (i.e. issue a
    TMScall_join) if the top-most frame has a transFlag with value other
    than __none;
  evalDistributed ( RQ1, exp );
}
...
```

The `evalDistributed` routine returns a queue of values or (remote) object references.

Note: Remote method invocation is a special case of this evaluation step.

**Distributed Transactions.** Supporting distribution and object migration has an impact on the processing of transactions. Distributed transactions require the support of more sophisticated commit protocols such as the two-phase commit protocol [108]. Object migration requires extensions to the recovery mechanism. These include, among others, the necessity that crash recovery procedures may have to involve non-crash nodes to return a local database to its most recent consistent state. Corresponding extensions to the popular ARIES and the ARIES/ML recovery mechanisms are proposed in [122, 123].

[11] A forwarding reference is left in the form of a proxy object [35].

Beyond those TMS-specific issues, there are no further actions that have to be taken during the evaluation process. Let us briefly demonstrate this by considering the processing of a distributed transaction.

EXAMPLE 5.8. We revisit Example 3.4, which contains the following distributed transaction:

```
10     ...
11     DO TRANSACTION tr1               // a transaction object is created implicitly
12       rslt1 = AcademicC$_{CC}$ WHERE ( specialisation == "Database Systems" );
13       rslt2 = AcademicC$_{LS}$ WHERE ( specialisation == "Database Systems" );
14       rslt3 = AcademicC$_{TO}$ WHERE ( specialisation == "Database Systems" );
15       rslt = ( rslt1.union ( rslt2 ) ).union ( rslt3 );
16       tr1.commit ( );                                  // explicit transaction commit
17     ENDDO;
18     ...
```

Corresponding annotations are outlined in Example 3.4 (on page 54).
While the evaluation of lines 13 and 14 will be distributed to remote ODBS nodes, processing may be enhanced further. In accordance with optimisation procedures as discussed in Section 5.4, the optimiser is likely to rewrite the code segment shown above as follows:

```
20     ...
21     DO TRANSACTION tr1               // a transaction object is created implicitly
22       INDEPENDENT DO
23         rslt2 = AcademicC$_{LS}$ WHERE ( specialisation == "Database Systems" );
24       ENDDO;
25
26       INDEPENDENT DO
27         rslt3 = AcademicC$_{TO}$ WHERE ( specialisation == "Database Systems" );
28       ENDDO;
29
30       INDEPENDENT DO
31         rslt1 = AcademicC$_{CC}$ WHERE ( specialisation == "Database Systems" );
32       ENDDO;
33
34       rslt = ( rslt1.union ( rslt2 ) ).union ( rslt3 );
35       tr1.commit ( );                                  // explicit transaction commit
36     ENDDO;
37     ...
```

Now, distribution and multi-threading are mixed. Thus, the four statements outlined in lines 23, 27, 31 and 34 are executed simultaneously: Lines 23 and 27 utilise both multi-threading and distribution (i.e. asynchronous distribution); line 31 only utilises multi-threading. The statement in line 34 commences while the three threads are active but only terminates after all threads have been evaluated. The two consecutive union operations synchronise processing implicitly due to the fact that these operations cannot be completed without having received all input values. Internally, the evaluation of line 34 is further optimised by multi-threading the two union operations. Intermediate results are pipelined.

When commencing the evaluation of line 21, name binders for the local variables **rslt1**, **rslt2**, **rslt3** and **rslt** already reside in the top-most frame on ES. Upon encountering the **DO TRANSACTION tr1** block, a new sub-frame is pushed onto ES (with **transFlag** set to **tr1**) and a new transaction object is created. Thus, the TMS is informed that the evaluation of a new transaction begins.
The evaluation of lines 22 to 24 and 26 to 28 is almost identical:

1. Processing is multi-threaded;
2. The evaluation of the assignment operator is initialised;
3. The evaluation of the right-hand side assignment expression is being distributed but only after the local TMS is being notified of this step;
4. The respective remote node evaluates the expression. Serialisability and recoverability are ensured by its local TMS;
5. The passing of result values from the remote node to the thread and then from the thread to the to the main execution stream are implemented through pipelining.
6. The evaluation procedure on the remote node terminates once all result values have been returned; and
7. The multi-threaded evaluation procedure terminates once the distributed procedure has terminated.

Similarly, the evaluation described in lines 30 to 32 is executed. However, no distribution occurs. The local TMS monitors processing.
Once line 34 has been evaluated, all three threads have been terminated. Subsequently, an explicit `commit` call is encountered. Thus, the local TMS invokes the commit process (e.g. using a two-phase commit protocol). The success or failure of the evaluation of transaction `tr1` is determined by all three involved nodes. □

## 5.4 Notes on the Optimisation of the Evaluation Process

In this chapter, we mainly focused on the evaluation process of user requests. Only a few enhanced machine codes have been discussed for a selected number of iDBPQL statements, expressions and operators in greater detail. We like to conclude this chapter by drawing attention to the potential of how the evaluation process may be optimised. The majority of query optimisation techniques known from relational DBSs can be carried over to the stack-based approach to request evaluation. For instance, Section 5.3 has already outlined how selections and sorting may benefit from the used of indices, how reordering of conjuncts improves the performance of selections, how different evaluation plans for the same iDBPQL primitive can be utilised to better process a small, medium-sized or large collection etc. In addition, the following optimisation techniques are also applicable [37, 103, 118]:

- Reordering of query expressions. Commutativity properties of selections, joins and various collection operators (e.g. unions and intersections) and associativity properties of join operators can be utilised. Selections may be pushed down the evaluation path etc.

- Pre-evaluate query expressions that are executed repeatedly without having any of its parameters modified. The performance of evaluations of quantifier expressions and all types of loops may benefit from this type of optimisation.
- Perform a cost-model based optimisation of evaluation plans.

Furthermore, simultaneous and distributed processing and the pipelining of intermediate results have great influence on performance characteristics. Sections 5.3 and 5.3.10 have already indicated how these techniques apply to the execution of evaluation plans formulated in iDBPQL.

In a similar manner, code enhancement approaches as known from OOPLs can be applied. Corresponding applicable techniques include [116, 137]:

- Eliminate redundant value access;
- Perform simple arithmetic calculations at compile time;
- Propagate constant values at compile time;
- Eliminate common sub-expressions;
- Copy propagation;
- Strength reduction;
- Eliminate useless instructions;
- Eliminate redundancies in basic blocks;
- Improve loops through pre-evaluation of loop invariants, application of pipelining and loop reordering;
- Re-use intermediate results; and more.

# Chapter 6

# Proof of Concept

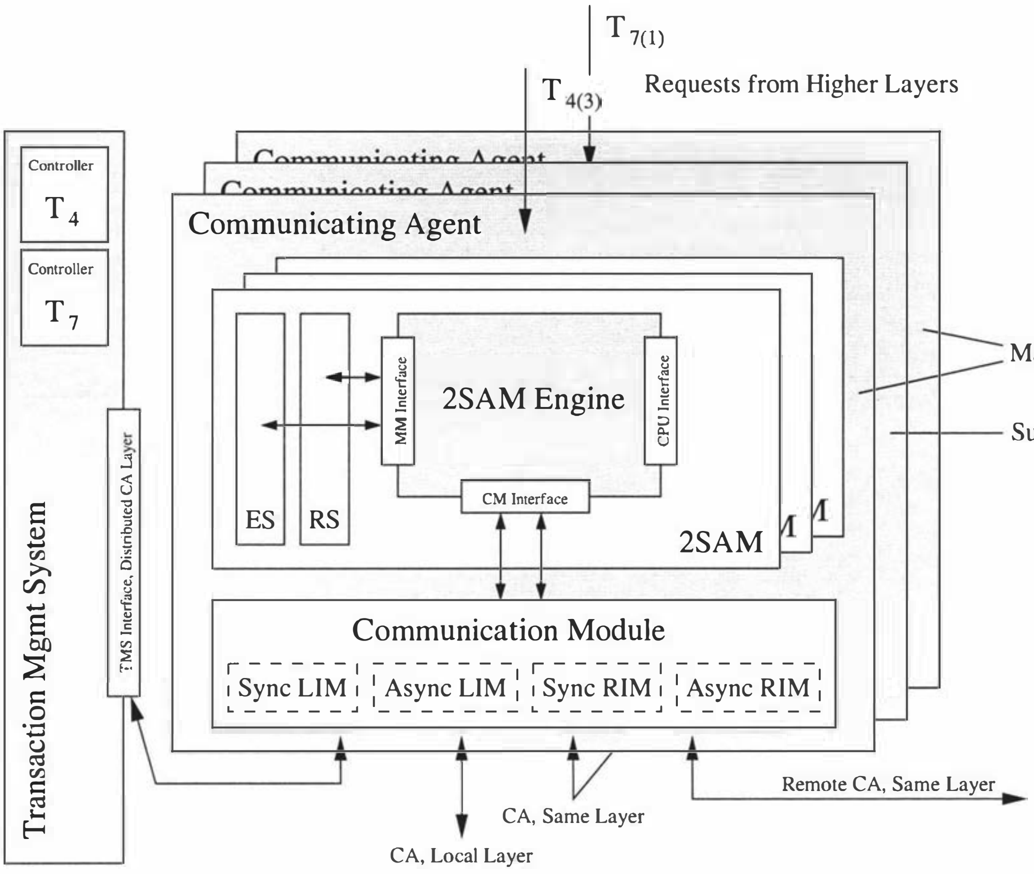
During the development of the proposed iDBPQL language and its run-time environment, two prototype systems have been implemented.

The first prototype corresponds to a feasibility test that has been realised at an early stage of our research. This prototype can be regarded as an extension of the stack-based approach [129, 131] with capabilities to support multi-threading, distribution and parallelism. Section 6.1 will provide an overview of this system.

Based on our experiences with the first prototype system, the iDBPQL language and run-time environment has been developed. While support for parallelism has been dropped, the second prototype has a much more sophisticated run-time system, is built on top of the persistent object store and processes evaluation plans that are formulated in the integrated database programming and querying language iDBPQL. An overview of this proof of concept prototype implementation is presented in Section 6.2.

## 6.1 A Prototype of Stack-Based Abstract Machines that Support Multi-Threading, Distribution and Parallelism

The first prototype can be regarded as an attempt to adapt the SBA approach [129, 130, 131] to a computing environment that supports multi-threading, parallel processing and remote object invocation. To cope with the added complexity, two levels of abstract machines have been defined on each node. These machines are referred to as *Communication Agents (CAs)*. Agents on the higher level (refer Figure 6.1) are aware of data distribution. They serve higher-level requests, model transactions, distribute requests to remote communication agents, utilise parallelism by creating new agents on the same level, forward the evaluation of sub-requests to the lower level of communicating agents, compose results as necessary etc. Agents on the lower level (refer Figure 6.2) deal with requests on local objects only. They further utilise parallelism but are not aware of distribution. Every agent consists of one or more two-stack abstract machines and a communication module. The stack-based machine implements a simple version of the integrated query and programming language iDBPQL on top of an environment stack and a result stack. The communication module provides local and remote communication capabilities. It offers synchronous and asynchronous calls for both, local and remote requests. If two or more abstract machines exist within the same communicating agent, multi-threading is utilised. These threads may access the



| | | | |
|---|---|---|---|
| 2SAM | Two–Stack Abstract Machine | MM | Main Memory |
| Async | Asynchronous | RIM | Remote Invocation Mechanism |
| CA | Communicating Agent | RS | Result Stack |
| CM | Communication Module | Sync | Synchronous |
| ES | Environment Stack | T | Transaction |
| LIM | Local Invocation Mechanism | TMS | Transaction Management System |

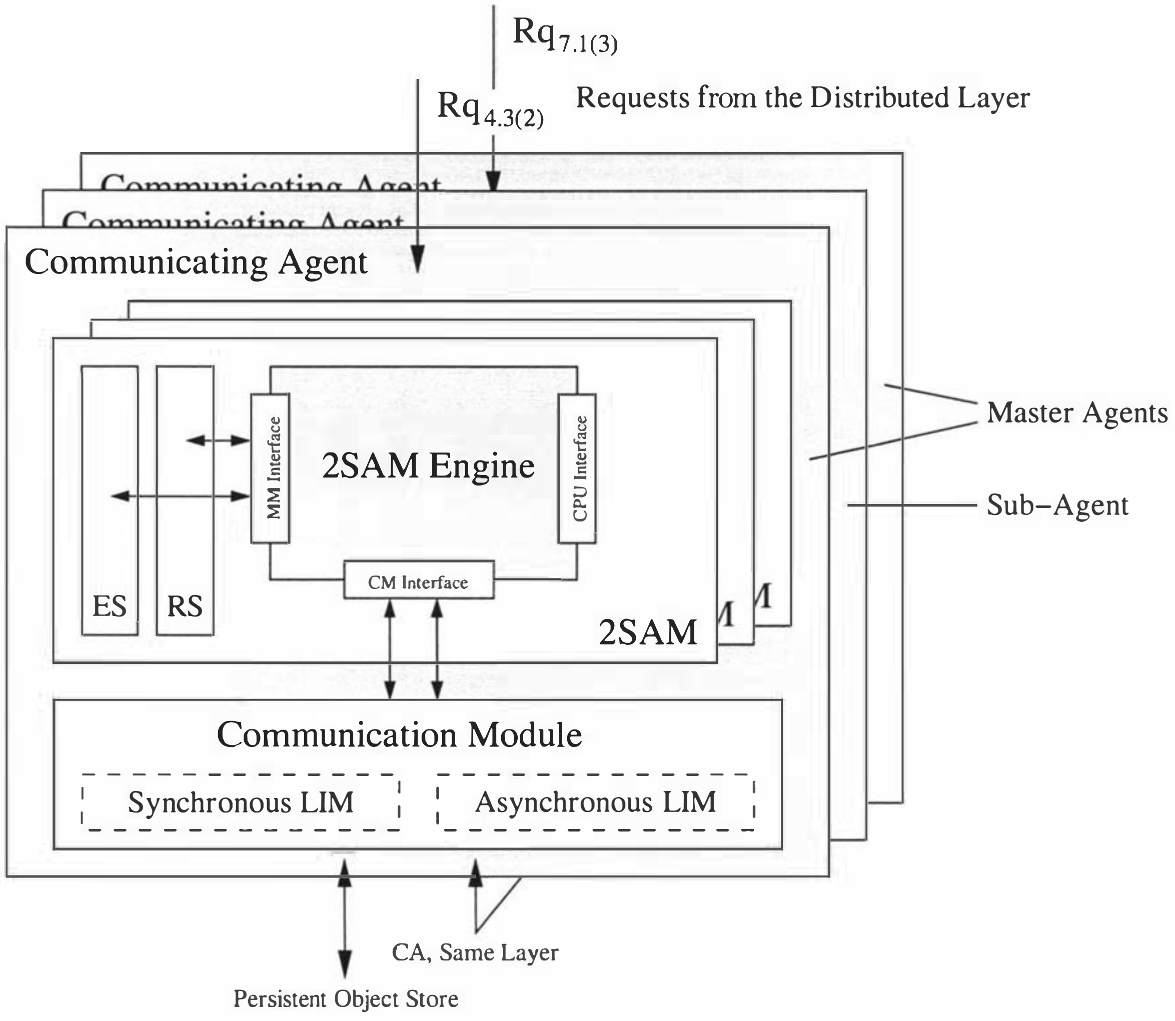
**Fig. 6.1.** Overview of a Stack-Based Abstract Machine with Distribution Capabilities.

same result queue in a shared memory area and synchronise their actions through the use of semaphores.

Due to the unavailability of a true parallel machine, virtual parallelism is utilised with the help of the PVM-library [44].

In addition to communicating agents, there exists a *Master Agent (MA)* on each participating processing unit. This agent acts as the first point of contact and creates the first communicating agent that is responsible for overseeing the evaluation of an incoming (sub-)request.

The first prototype did not support the evaluation of a completely integrated database programming and querying language. Only a pre-selected number of object-oriented concepts, control flow statements, programming language expressions and



| | | | |
|---|---|---|---|
| 2SAM | Two–Stack Abstract Machine | LIM | Local Invocation Mechanism |
| CA | Communicating Agent | MM | Main Memory |
| CM | Communication Module | Rq | Request |
| ES | Environment Stack | RS | Result Stack |

**Fig. 6.2.** Overview of a Local Stack-Based Abstract Machine.

query language expressions have been supported. These concepts include: atomic types, bag, set, list and record type constructors, the notion of (complex) objects, multiple inheritance, `WHILE` and `FOR EACH` loops, the conditional `IF` ... `THEN` ... `ELSE` statement, selection, parallel selection, projection, sorting, parallel sorting, and navigational join.

## 6.2 The iDBPQL Prototype System

The second prototype is a proof of concept implementation of the iDBPQL language and run-time environment as proposed in this thesis. While corresponding concepts have already been introduced in Chapters 4 and 5, we will only summarise restrictions of the current prototype, refer to libraries and other prototypes that have been utilised, and outline how the prototype system may be programmed.

In contrast to the first prototype system, support for parallel processing has been dropped. Potential performance gains stand in contrast to challenges with respect to serialising access to and updates of shared data, and ensuring recoverability when using a multi-level transaction management system. Adding support for parallelism is one of the items on the list for future work.

The current iDBPQL prototype system contains evaluation routines for all different types of language constructs. While support for pipelining is generally available, not all constructs have additional optimised machine codes associated. However, all those operational semantics as outlined in Section 5.3 are already available in the prototype system. The iDBPQL library only contains a minimal set of built-in features. In particular, primitive types and types that are adopted in a straightforward manner from the object-oriented programming domain are constrained. The rationale behind those restrictions can be explained by the fact that adding those features is not a challenging but only a time-consuming task.

During the processing of evaluation plans, services from a number of additional ODBS components are utilised. Prototypes for these components have the following restrictions:

- The persistent object store offers the service interface as outlined in Section 5.2.1. However, associative and navigational access structures only rely on look-up tables. Thus, affecting the performance of evaluation processes significantly. Furthermore, data persistence is not supported beyond an invocation cycle of the iDBPQL run-time system. In particular, all modifications to a given database are discarded when the iDBPQL run-time system terminates. The rationale behind this restriction is due to the fact that data consistency cannot be guaranteed without having an exception mechanism and a fully operational recovery mechanism (refer below). As a result, sample databases must be generated either manually or using serial execution followed by manual verification.
- The transaction management system is based on a prototype, which was developed at an earlier stage [65, 71]. Extensions have been applied to support weak and strong operation ordering as proposed by Alonso et. al. [6]. However, the prototype does not yet support multi-level recovery. While the non-distributed ARIES/ML recovery mechanism [104, 111] has been prototyped, necessary extensions to distributed systems [122, 123] have yet to be implemented. Similarly, support for weak and strong operation orderings affect the recovery process. In particular, cascading aborts may occur without refining the weak operation ordering property further. This, however, is still an open research issue.

All prototype systems have been implemented in the programming language C [59] or C++ [128]. The following additional libraries have been utilised:

- The POSIX threads library [23];

– The OOSP Shared Memory Allocation library [38] as a utility that simplifies the usage of shared memory between strongly related processes;
– The OSSP Universally Unique Identifier (UUID) library [39] as a utility to generate globally unique object identifiers; and
– The Library for Efficient Data types and Algorithms (LEDA) [88].

Finally, we want to outline how this second prototype can be programmed. Chapters 4 and 5 presented two low-level representations of evaluation plans. While iDBPQL-based evaluation plans are more easy to read, an internal representation is required for the evaluation process to execute efficiently. The latter representation will also be the one that is used by a higher-level ODBS module. The former, more human friendly means of programming the prototype is supported only to assist with readability and testing.

Figure 6.3 provides an overview of how the prototype system can be programmed. From a human being's perspective, the best way to use the prototype system is as follows:

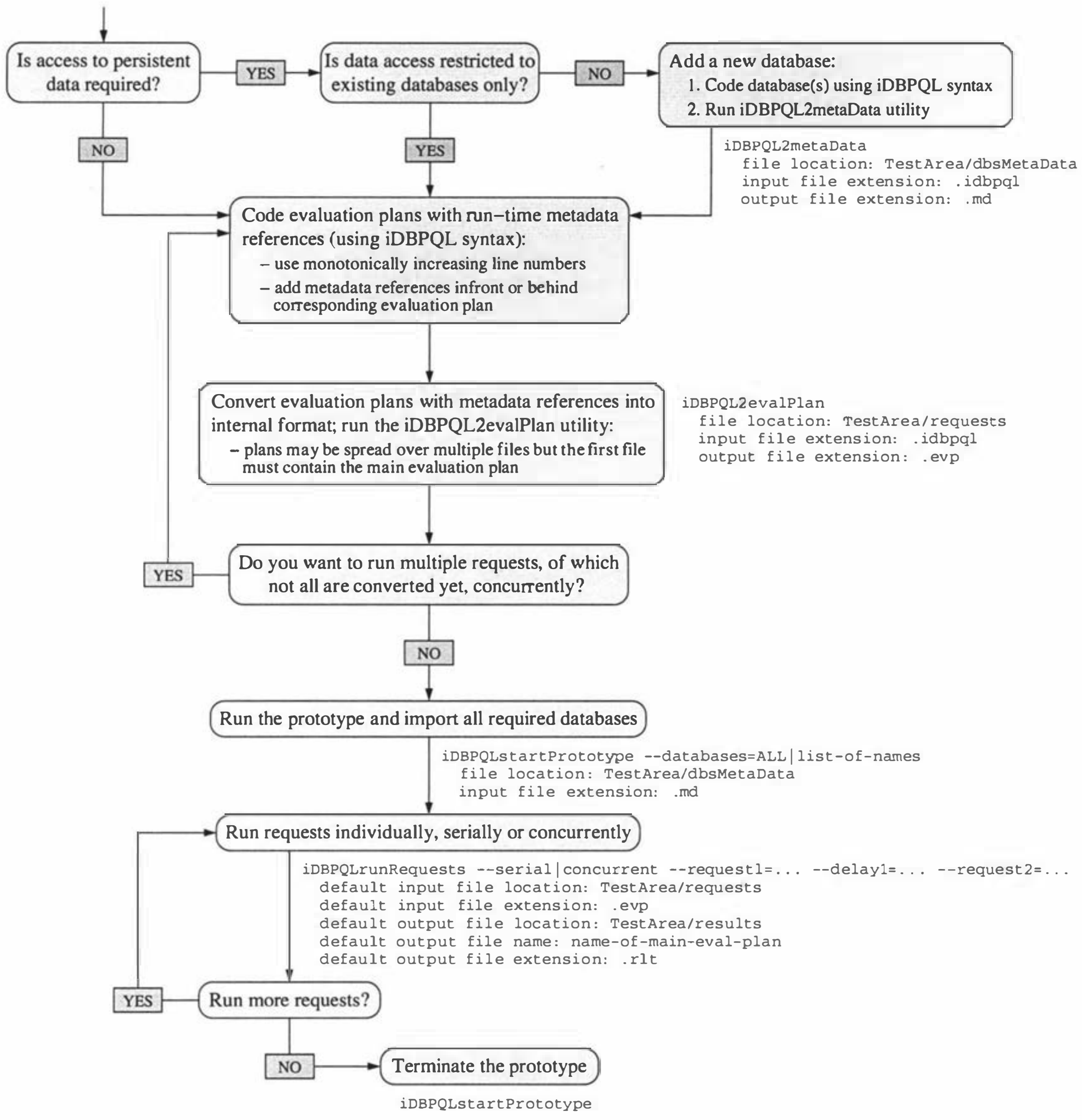
1. Program the user request in terms of evaluation plans according to the iDBPQL syntax as presented in Chapter 4. Each line must start with a unique, monotonically increasing line number. The main evaluation plan must be the first evaluation plan that is specified. For instance, the popular '*Hello world!*' example can be specified as follows:

```
1  EVALPLAN HelloWorld ( VOID ) : STRING {
2    myString = "Hello world!";
3    RETURN ( myString );
4  }
```

2. Add metadata references in-front the corresponding evaluation plan. Metadata references must be associated with an evaluation plan's line that opens an evaluation block, i.e. contains a '{' or the DO keyword. Let us consider the '*Hello world!*' example again:

```
1  STRING myString;

1  EVALPLAN HelloWorld ( VOID ) : STRING {
2    myString = "Hello world!";
3    RETURN ( myString );
4  }
```

   Here, the local variable declaration `STRING myString;` is associated with the first line of the `HelloWorld` (main) evaluation plan.
   If a metadata definition spans multiple lines, the second to last lines of the specification do not require line numbers. Multiple metadata references may be associated with the same evaluation plan line.
3. Use the `iDBPQL2evalPlan` utility that is supplied with the prototype system. Arguments are all file names that make up the user request. Note: It is not necessary that all evaluation plans of a request are placed into the same file. However, metadata references must be in the same file as their associated evaluation plans and


Is access to persistent data required?
YES
Is data access restricted to existing databases only?
NO
Add a new database:
1. Code database(s) using iDBPQL syntax
2. Run iDBPQL2metaData utility
NO
YES
Code evaluation plans with run–time metadata references (using iDBPQL syntax):
– use monotonically increasing line numbers
– add metadata references infront or behind corresponding evaluation plan
Convert evaluation plans with metadata references into internal format; run the iDBPQL2evalPlan utility:
– plans may be spread over multiple files but the first file must contain the main evaluation plan
YES
Do you want to run multiple requests, of which not all are converted yet, concurrently?
NO
Run the prototype and import all required databases
Run requests individually, serially or concurrently
YES
Run more requests?
NO
Terminate the prototype

```
iDBPQL2metaData
  file location: TestArea/dbsMetaData
  input file extension: .idbpql
  output file extension: .md
```

```
iDBPQL2evalPlan
  file location: TestArea/requests
  input file extension: .idbpql
  output file extension: .evp
```

```
iDBPQLstartPrototype --databases=ALL|list-of-names
  file location: TestArea/dbsMetaData
  input file extension: .md
```

```
iDBPQLrunRequests --serial|concurrent --request1=... --delay1=... --request2=...
  default input file location: TestArea/requests
  default input file extension: .evp
  default output file location: TestArea/results
  default output file name: name-of-main-eval-plan
  default output file extension: .rlt
```

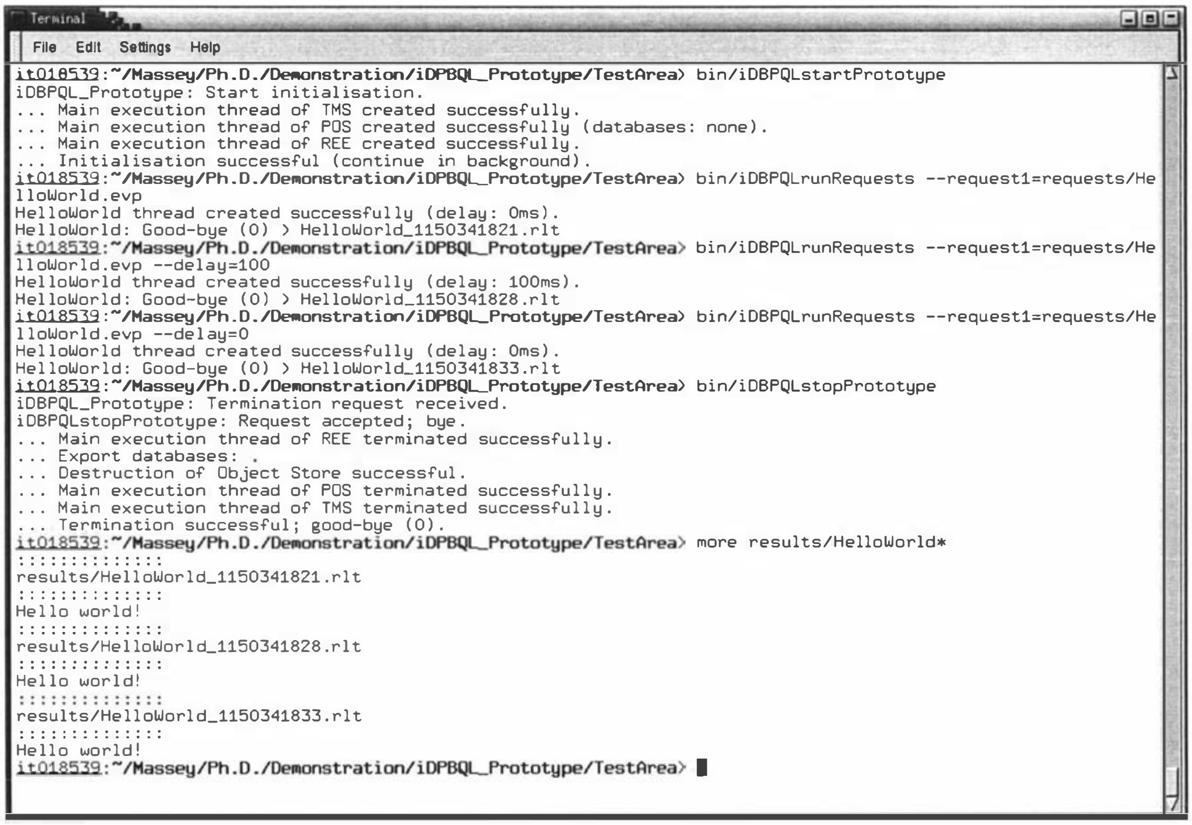
```
iDBPQLstartPrototype
```

**Fig. 6.3.** Usage Diagram for the iDBPQL Prototype.

the main evaluation plan must have the same name as and be located in the file provided as first argument.
Two invocations of the `iDBPQL2evalPlan` utility with the same input files may produce different internal representations. Due to the absence of a compiler and optimiser, a certain degree of randomness has been added to this utility.

4. The resulting file may be handed over to the run-time system or can be refined manually. However, refinements require care since the run-time system expects well formed code without any syntax errors.

Let us consider some examples. First, Figure 6.4 demonstrates how a centralised version of the iDBPQL prototype system is invoked and the previously described



```
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLstartPrototype
iDBPQL_Prototype: Start initialisation.
... Main execution thread of TMS created successfully.
... Main execution thread of POS created successfully (databases: none).
... Main execution thread of REE created successfully.
... Initialisation successful (continue in background).
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLrunRequests --request1=requests/He
lloWorld.evp
HelloWorld thread created successfully (delay: 0ms).
HelloWorld: Good-bye (0) > HelloWorld_1150341821.rlt
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLrunRequests --request1=requests/He
lloWorld.evp --delay=100
HelloWorld thread created successfully (delay: 100ms).
HelloWorld: Good-bye (0) > HelloWorld_1150341828.rlt
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLrunRequests --request1=requests/He
lloWorld.evp --delay=0
HelloWorld thread created successfully (delay: 0ms).
HelloWorld: Good-bye (0) > HelloWorld_1150341833.rlt
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLstopPrototype
iDBPQL_Prototype: Termination request received.
iDBPQLstopPrototype: Request accepted; bye.
... Main execution thread of REE terminated successfully.
... Export databases: .
... Destruction of Object Store successful.
... Main execution thread of POS terminated successfully.
... Main execution thread of TMS terminated successfully.
... Termination successful; good-bye (0).
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> more results/HelloWorld*
::::::::::::::
results/HelloWorld_1150341821.rlt
::::::::::::::
Hello world!
::::::::::::::
results/HelloWorld_1150341828.rlt
::::::::::::::
Hello world!
::::::::::::::
results/HelloWorld_1150341833.rlt
::::::::::::::
Hello world!
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> █
```

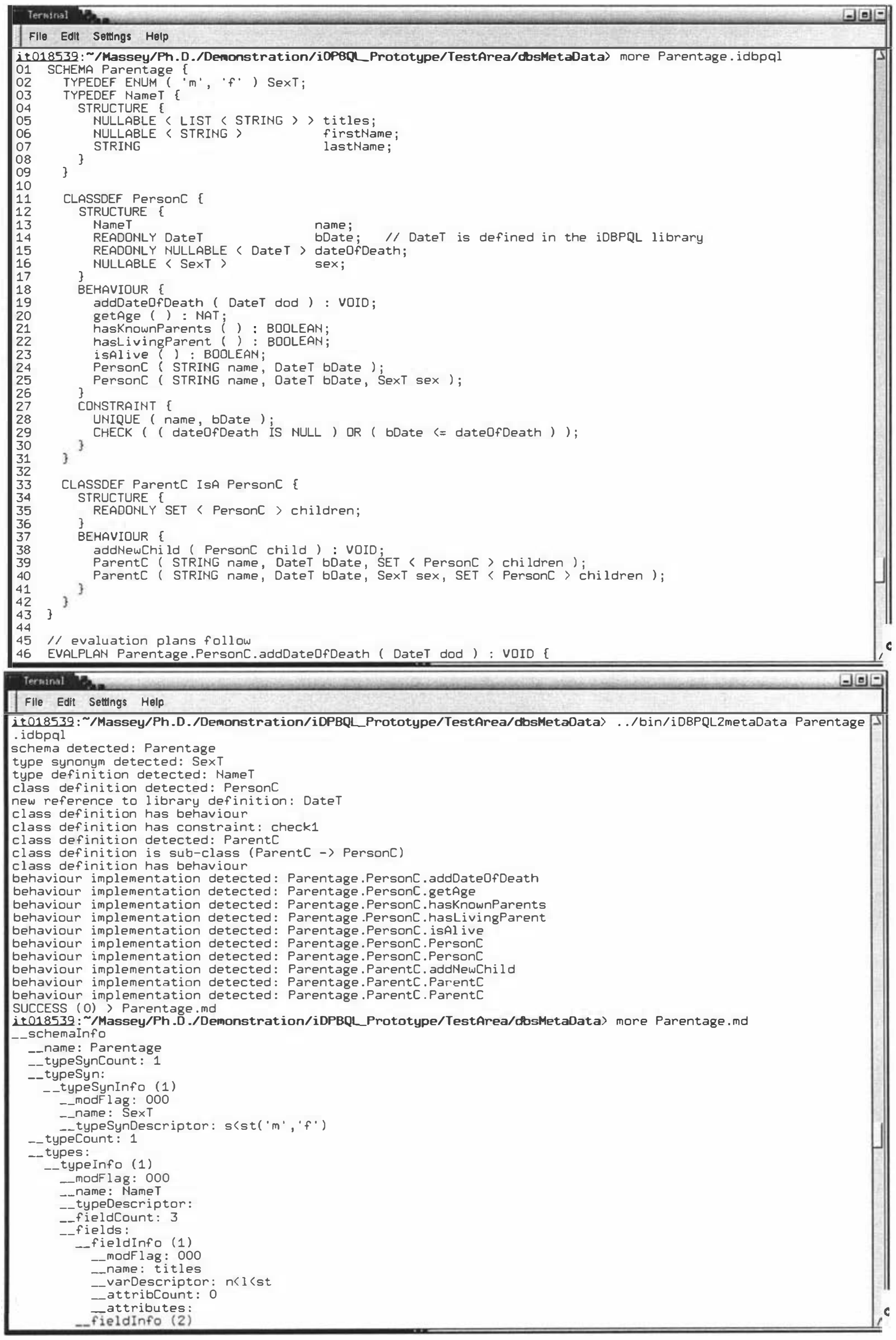
**Fig. 6.4.** Evaluation of the *HelloWorld* Example.

`HelloWorld` evaluation plan is run.

As a second example, let us consider a simple *Parentage* database. Figure 6.5 outlines snapshots of the iDBPQL representation as well as the internal representation of this database schema. A more complete version of the internal representation of the *Parentage* database schema can be found in Appendix B. The internal schema is represented by a `__schemaInfo` structure and each type synonym, type definition and class definition has an associated `__typeSynInfo` structure, `__typeInfo` structure or `__classInfo` structure, respectively. Inherited properties are stored with the sub-class. While the uniqueness constraint is fully represented in the data definition portion, the check constraint is transformed into an evaluation plan such as:

```
EVALPLAN check1 ( ) : BOOLEAN
{
  if ( ( dateOfDeath IS NULL ) OR ( bDate <= dateOfDeath ) ) {
    RETURN ( TRUE );
  }
  RETURN ( FALSE );
}
```

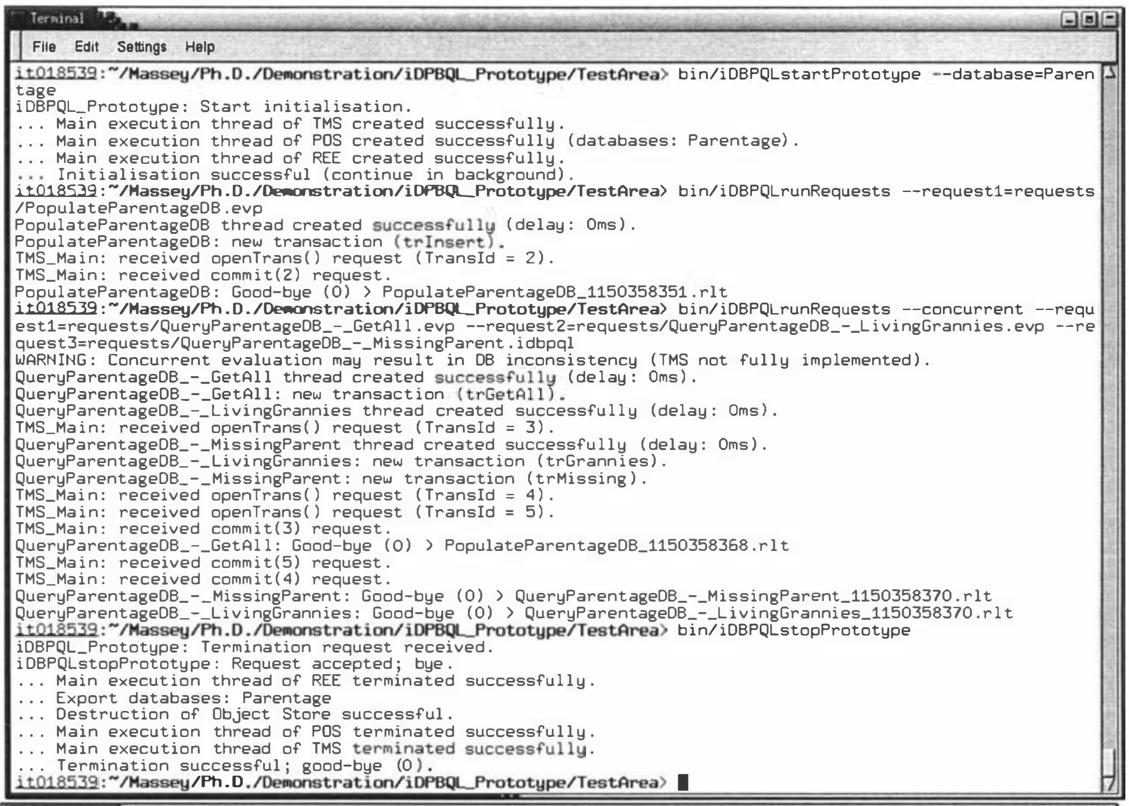
Using this *Parentage* database schema, we may now execute some initial requests. The top-most snapshot of Figure 6.6 demonstrates the execution of four requests that:



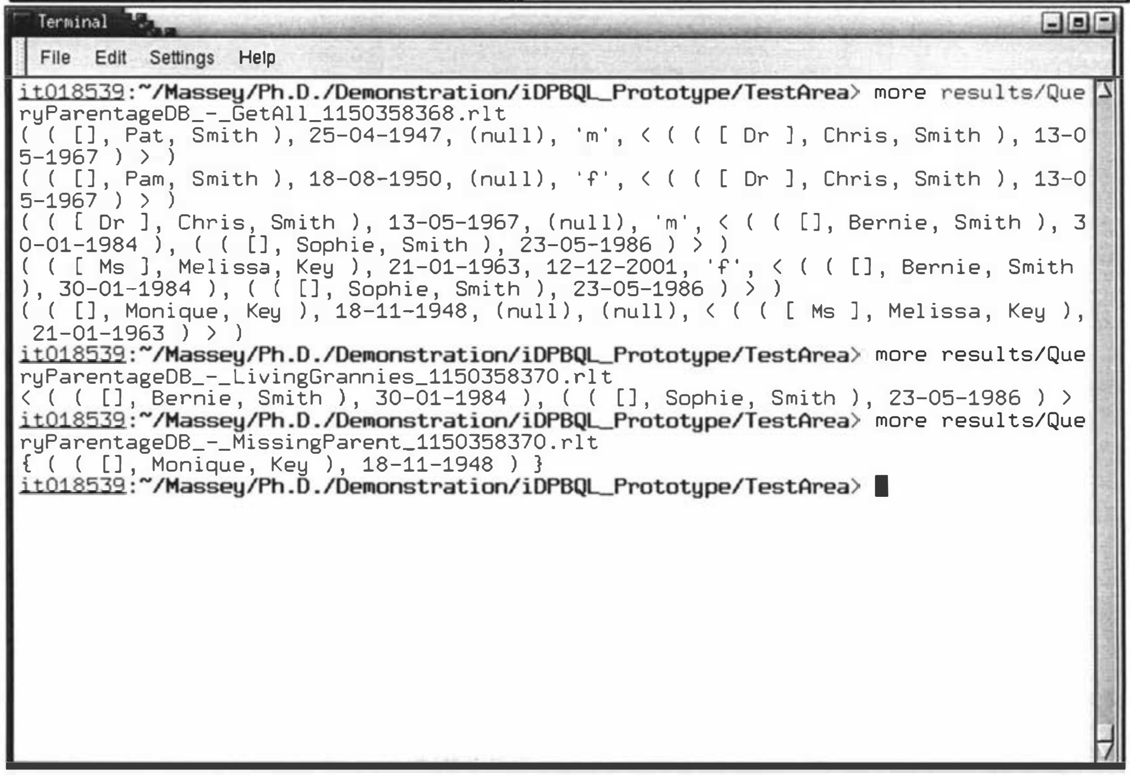
```
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea/dbsMetaData> more Parentage.idbpql
SCHEMA Parentage {
  TYPEDEF ENUM ( 'm', 'f' ) SexT;
  TYPEDEF NameT {
    STRUCTURE {
      NULLABLE < LIST < STRING > > titles;
      NULLABLE < STRING >          firstName;
      STRING                       lastName;
    }
  }

  CLASSDEF PersonC {
    STRUCTURE {
      NameT                        name;
      READONLY DateT               bDate;   // DateT is defined in the iDBPQL library
      READONLY NULLABLE < DateT > dateOfDeath;
      NULLABLE < SexT >            sex;
    }
    BEHAVIOUR {
      addDateOfDeath ( DateT dod ) : VOID;
      getAge ( ) : NAT;
      hasKnownParents ( ) : BOOLEAN;
      hasLivingParent ( ) : BOOLEAN;
      isAlive ( ) : BOOLEAN;
      PersonC ( STRING name, DateT bDate );
      PersonC ( STRING name, DateT bDate, SexT sex );
    }
    CONSTRAINT {
      UNIQUE ( name, bDate );
      CHECK ( ( dateOfDeath IS NULL ) OR ( bDate <= dateOfDeath ) );
    }
  }

  CLASSDEF ParentC IsA PersonC {
    STRUCTURE {
      READONLY SET < PersonC > children;
    }
    BEHAVIOUR {
      addNewChild ( PersonC child ) : VOID;
      ParentC ( STRING name, DateT bDate, SET < PersonC > children );
      ParentC ( STRING name, DateT bDate, SexT sex, SET < PersonC > children );
    }
  }
}

// evaluation plans follow
EVALPLAN Parentage.PersonC.addDateOfDeath ( DateT dod ) : VOID {
```

```
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea/dbsMetaData> ../bin/iDBPQL2metaData Parentage
.idbpql
schema detected: Parentage
type synonym detected: SexT
type definition detected: NameT
class definition detected: PersonC
new reference to library definition: DateT
class definition has behaviour
class definition has constraint: check1
class definition detected: ParentC
class definition is sub-class (ParentC -> PersonC)
class definition has behaviour
behaviour implementation detected: Parentage.PersonC.addDateOfDeath
behaviour implementation detected: Parentage.PersonC.getAge
behaviour implementation detected: Parentage.PersonC.hasKnownParents
behaviour implementation detected: Parentage.PersonC.hasLivingParent
behaviour implementation detected: Parentage.PersonC.isAlive
behaviour implementation detected: Parentage.PersonC.PersonC
behaviour implementation detected: Parentage.PersonC.PersonC
behaviour implementation detected: Parentage.ParentC.addNewChild
behaviour implementation detected: Parentage.ParentC.ParentC
behaviour implementation detected: Parentage.ParentC.ParentC
SUCCESS (0) > Parentage.md
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea/dbsMetaData> more Parentage.md
__schemaInfo
  __name: Parentage
  __typeSynCount: 1
  __typeSyn:
    __typeSynInfo (1)
      __modFlag: 000
      __name: SexT
      __typeSynDescriptor: s<st('m','f')
  __typeCount: 1
  __types:
    __typeInfo (1)
      __modFlag: 000
      __name: NameT
      __typeDescriptor:
      __fieldCount: 3
      __fields:
        __fieldInfo (1)
          __modFlag: 000
          __name: titles
          __varDescriptor: n<l<st
          __attribCount: 0
          __attributes:
        __fieldInfo (2)
```

**Fig. 6.5.** iDBPQL and Internal Representations of the *Parentage* Database.



```
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLstartPrototype --database=Paren
tage
iDBPQL_Prototype: Start initialisation.
... Main execution thread of TMS created successfully.
... Main execution thread of POS created successfully (databases: Parentage).
... Main execution thread of REE created successfully.
... Initialisation successful (continue in background).
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLrunRequests --request1=requests
/PopulateParentageDB.evp
PopulateParentageDB thread created successfully (delay: 0ms).
PopulateParentageDB: new transaction (trInsert).
TMS_Main: received openTrans() request (TransId = 2).
TMS_Main: received commit(2) request.
PopulateParentageDB: Good-bye (0) > PopulateParentageDB_1150358351.rlt
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLrunRequests --concurrent --requ
est1=requests/QueryParentageDB_-_GetAll.evp --request2=requests/QueryParentageDB_-_LivingGrannies.evp --re
quest3=requests/QueryParentageDB_-_MissingParent.idbpql
WARNING: Concurrent evaluation may result in DB inconsistency (TMS not fully implemented).
QueryParentageDB_-_GetAll thread created successfully (delay: 0ms).
QueryParentageDB_-_GetAll: new transaction (trGetAll).
QueryParentageDB_-_LivingGrannies thread created successfully (delay: 0ms).
TMS_Main: received openTrans() request (TransId = 3).
QueryParentageDB_-_MissingParent thread created successfully (delay: 0ms).
QueryParentageDB_-_LivingGrannies: new transaction (trGrannies).
QueryParentageDB_-_MissingParent: new transaction (trMissing).
TMS_Main: received openTrans() request (TransId = 4).
TMS_Main: received openTrans() request (TransId = 5).
TMS_Main: received commit(3) request.
QueryParentageDB_-_GetAll: Good-bye (0) > PopulateParentageDB_1150358368.rlt
TMS_Main: received commit(5) request.
TMS_Main: received commit(4) request.
QueryParentageDB_-_MissingParent: Good-bye (0) > QueryParentageDB_-_MissingParent_1150358370.rlt
QueryParentageDB_-_LivingGrannies: Good-bye (0) > QueryParentageDB_-_LivingGrannies_1150358370.rlt
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> bin/iDBPQLstopPrototype
iDBPQL_Prototype: Termination request received.
iDBPQLstopPrototype: Request accepted; bye.
... Main execution thread of REE terminated successfully.
... Export databases: Parentage
... Destruction of Object Store successful.
... Main execution thread of POS terminated successfully.
... Main execution thread of TMS terminated successfully.
... Termination successful; good-bye (0).
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> █
```



```
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> more results/Que
ryParentageDB_-_GetAll_1150358368.rlt
( ( [], Pat, Smith ), 25-04-1947, (null), 'm', < ( ( [ Dr ], Chris, Smith ), 13-0
5-1967 ) > )
( ( [], Pam, Smith ), 18-08-1950, (null), 'f', < ( ( [ Dr ], Chris, Smith ), 13-0
5-1967 ) > )
( ( [ Dr ], Chris, Smith ), 13-05-1967, (null), 'm', < ( ( [], Bernie, Smith ), 3
0-01-1984 ), ( ( [], Sophie, Smith ), 23-05-1986 ) > )
( ( [ Ms ], Melissa, Key ), 21-01-1963, 12-12-2001, 'f', < ( ( [], Bernie, Smith
), 30-01-1984 ), ( ( [], Sophie, Smith ), 23-05-1986 ) > )
( ( [], Monique, Key ), 18-11-1948, (null), (null), < ( ( [ Ms ], Melissa, Key ),
 21-01-1963 ) > )
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> more results/Que
ryParentageDB_-_LivingGrannies_1150358370.rlt
< ( ( [], Bernie, Smith ), 30-01-1984 ), ( ( [], Sophie, Smith ), 23-05-1986 ) >
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> more results/Que
ryParentageDB_-_MissingParent_1150358370.rlt
{ ( ( [], Monique, Key ), 18-11-1948 ) }
it018539:~/Massey/Ph.D./Demonstration/iDPBQL_Prototype/TestArea> █
```

**Fig. 6.6.** Executing Requests on the *Parentage* Database.

1. Populate the *Parentage* database;
2a. Retrieve all `ParentC` entries from the *Parentage* database;
2b. Retrieve a list of `( name, dateOfBirth )` entries that identify all persons, which have living grandparents associated.
2c. Retrieve a set of `( name, dateOfBirth )` entries that identify all persons, which have only one parent entered in the *Parentage* database.

The first data population step utilises object constructors to create new person and parent objects. The three subsequent requests are evaluated concurrently and generate results as shown in the bottom-most snapshot of Figure 6.6.

More examples, demonstration slides, demonstration software, technical reports, reports on our research progress, relevant research publications, project information etc. are available on the following Web-site:

`http://dbpql.thekirchbergs.info/`

# Chapter 7

# Summary

In this thesis, we have proposed the design and implementation of a intermediate-level, object-oriented database programming and querying language iDBPQL. This language continues the research path leading towards a fully integrated database language that seamlessly unites the domains of object-oriented programming, database query languages and traditional database management systems.

The remainder of this final chapter is organised as follows: First, we summarise the main contribution of this thesis and subsequently, comments on future research plans are discussed.

## 7.1 Main Contribution

The research contribution of this thesis consists of four inter-related achievements. The first major contribution consists of the proposed object database architecture and a demonstration that this architecture is capable to process typical database requests. In contrast to corresponding research and development efforts of the 1980s and 1990s, our proposal cannot be considered as a direct extension of relational database technologies. Instead, we have proposed a novel architecture that recognises research advances from domains such as programming languages, database programming languages and compiler construction.

The second major contribution is the proposal of the integrated database programming and querying language iDBPQL. This language has been designed to support the evaluation of user requests on an internal DBS layer that is commonly referred to as (request) evaluation engine. iDBPQL is Turing-complete[1] [21] and separates the specification of data definitions from the implementation of behaviours.

Data definitions are associated with metadata catalogues. Internally, a distinction is made between specifications that correspond to shared, persistent data and specifications of private, transient run-time entities. iDBPQL further distinguishes between

[1] A language is said to be Turing-complete if it fulfils the following property: For each function that can be calculated with a Turing machine, there exists a program in this language that performs the same function [21]. Turing-completeness can be verified by providing a mapping from each possible Turing machine to a program in the language, by demonstrating that there is a program in the language that emulates a universal Turing machine or by proving that the language is a super-set of a language that is known to be Turing-complete. iDBPQL can be proven to be a super-set of the Turing-complete Brainfuck language [94].

values and objects, where values are (mutable) language entities that are identified by their value and objects are entities that have an immutable object identifier independent of associated values. While types structure values, the concept of classes is utilised to group objects. Besides commonly supported types such as `BOOL`, `CHAR`, `INT`, `NAT`, `REAL`, the record type, and `ARRAY`-type, iDBPQL also supports (parameterised) user types, collection types including `BAG`, `SET` and `LIST` and the `NULL`able type, which extends value types with the `NULL` value. Furthermore, reference-types and a `UNION`-type that supports the unification of identical or similar objects are provided. Classes can be regarded as templates for creating objects, expose structural properties, allow for the definition of (reverse) references, may have associated behaviour, support multiple inheritance, and may have associated, system-maintained collections through which access to all objects of a class and its sub-classes is possible.

Behaviour associated with objects and types, and the specification of a user request's main execution stream is provided in the form of evaluation plans. Evaluation plans are comprised of control flow statements, assignments, expressions, method calls, and evaluation blocks. Common programming abstractions and query language constructs have been included. The integration of both concepts evolves around collections. Evaluation blocks are used to group statements together, form atomic execution units, model local and distributed transactions and support independent or multi-threaded processing. Data persistence is treated as an orthogonal language concept. iDBPQL does not distinguish between persistent and transient values, types, objects or classes. Persistence is supported simply by adding a class definition to a schema or by creating a new object on a persistent class.

The third major contribution consists of three proposals. First, an internal representation of data definitions and evaluation plans is detailed. Metadata entries are described by modular pseudo-structures, which can (in full or in parts) effectively be mapped to persistent storage or transfered to a remote ODBS instance. Evaluation plans, in turn, are represented as graphs with various annotations. Annotations link behaviour specifications with metadata entries, capture information that is later utilised to enhance the evaluation or distribute sub-evaluations to remote ODBS instances. Secondly, general service interfaces of additional ODBS modules, which are required during the evaluation process, are defined. The definition of these service interfaces is partly more general than required for the evaluation of requests that are formulated in iDBPQL. For instance, the persistent object store also supports common access patterns that are utilised by query languages, which are designed for XML database systems. Thirdly, a run-time environment that enables the evaluation of iDBPQL requests is presented. During the processing of user requests, the environment for a particular evaluation stream is kept in a separate stack area. In addition to the environment stack, result stacks are maintained for each behaviour invocation. Individual results are held in result queues, which reside on result stacks. Result queues are supported to facilitate the pipelining of intermediate results between two or more evaluation units. The evaluation of user requests is performed by four evaluation routines, which recursively traverse respective evaluation graphs following a depth-first approach. These evaluation routines determine whether processing takes place in serial, multi-threaded or distributed manner. This decision is either based on annotations or explicitly encoded in the operational semantics of internal machine instructions. For each iDBPQL statement, expression, operator or keyword there exists one or more internal machine instruction. While basic operational semantics have been presented for the majority of language constructs, only a selected number of enhanced semantics have been presented in greater detail. These more complex semantics have been selected to demonstrate how different styles of processing can be utilised in the proposed run-time environment.

The final contribution relates to two prototype systems that have been implemented as proof of concepts. While the early prototype only corresponds to a feasibility study, the second prototype closely follows the proposals as presented in this thesis.

Research results presented in this thesis mainly address the database engine that processes evaluation plans formulated in the proposed integrated database programming and querying language iDBPQL. In order to achieve our particular research objectives, we have made a number of assumptions, which include:

- Concepts that relate to user interaction, code compilation, fragmentation and allocation, code optimisation, code rewriting, execution plan generation etc. are beyond the scope of this thesis. Instead, we adopt a black box approach and assume that user requests arrive in a form that is suitable for request evaluation.
- Typical high-level programming language concepts such as packages or modules, interfaces, type and class definitions, classes (as code structuring primitives) etc. are omitted. Proposing suitable high-level language interfaces is considered as a future research step.
- A number of services from supporting database components are utilised. While we describe the general functionality and service interfaces of corresponding database components, there are still a number of open research problems that need to be investigated before those components can be implemented in an effective and efficient manner. Section 7.2 includes a brief discussion of those open research problems.

## 7.2 Future Plans and Open Problems

Future research opportunities can be divided into four categories: Developing demonstration tools that assist with the promotion of the research results presented in this thesis, improving the proposed iDBPQL language and run-time system, addressing issues related to the development of the proposed distributed object-oriented database system [72] and adapting the iDBPQL language to other database environments such as XML database systems.

One of the biggest shortcomings of the proposed iDBPQL language is the lack of an exception mechanism. However, respective approaches as found in current object-oriented PLs can be adopted to our proposal. In particular, the internal representation of metadata entities has been designed in a way that support for exceptions can be added easily. For instance, exceptions that may be associated with a particular behaviour signature will be captured in the behaviour's `__attribute` array. This is an approach similar to the one implemented in the Java programming language [80]. Considering the run-time system, refining the memory allocation of the heap is one area of interest. As briefly indicated in Section 5.3, it is our intention to utilise the nature of stack-based accesses (i.e. generally from the top only). In order to free valuable main memory space, certain objects may be cached out to larger but slower memory devices. Such objects include those that are only referenced from bottom sections of environment and result stacks as well as objects that reside in the middle of large result queues. These are less likely to be accessed in the near future when compared to objects at the top of a stack or at the head and tail of pipelined result queues. With respect to the prototype systems, adding more enhanced machine codes and improving the usability are the first shortcomings that have to be addressed. Other areas of interest include support for true parallelism, batch updates, views etc.

As it has been outlined at the beginning of this thesis, research work presented in this thesis is related to the development of a distributed object-oriented database system. As such, it is only natural that future research will focus on the completion of this project. Currently, there are other PhD research projects active that are concerned with:

- Support for generic update operations [105], query compilation and value-representability [106]; and
- The investigation of fragmentation, allocation and optimisation techniques for higher-order data models.

Furthermore, there are a number of open research issues. These include:

- Efficient storage structures and clustering techniques for persistent objects. The POS prototype mentioned above is a very naive implementation. Research into clustering techniques for storage objects and compression techniques for indices are still required.
- Merging of navigational and associative access structures. Uniting these two types of access structures has the potential to further increase ODBS performance [96].
- Definition and development of a variety of ways for users to access the ODBS. This includes a high-level version of the DBPQL language.
- DBPQL compilation and code optimisation techniques.
- Definition and development of Database Administrator interfaces. These are required to enable administrators to fine-tune the system and monitor its performance.

XML database systems have moved into the centre of database research in recent years. Current proposals and implementations either propose an extension of relational, object-relational or object-oriented systems to also support the storage, querying and processing of XML documents or advocate the use of a native XML database system. Among others, IBM has extended their DB2 Universal Database to include native XML support [95]. Mapping XML documents and queries to non-native XML databases suffers from similar problems as those outline in Section 1.1.1. As a result, the tendency goes towards native XML database systems. When developing native XML databases, new challenges such as effective internal representations of tree-structured XML documents, efficient processing of mainly navigation-oriented access and fine-granularity locking have to be faced . Among others, [49] provides an overview of corresponding challenges and suggests new research directions that could lead to better XML DBS architectures.

The proposed iDBPQL system together with the multi-level transaction management system and the persistent object store have the potential to support the storage and processing of XML documents quite naturally. Concepts such as fine granularity locking, navigational data access, support for bulk data types and user-defined data types are readily incorporated. For instance, the proposed persistent object store may model indices on XML documents using both embedded references and explicit references. Values of types `IDREF` and `IDREFS` map to embedded references while parent-child relationships would correspond to explicit references.

# Bibliography

1. Agrawal, R., Dar, S., and Gehani, N. H. The O++ database programming language: Implementation and experience. In *Proceedings of the 9th International Conference on Data Engineering* (Washington, DC, USA, 1993), IEEE Computer Society, pp. 61–70.
2. Agrawal, R., and Gehani, N. H. ODE (object database and environment): the language and the data model. In *Proceedings of the international conference on Management of data* (New York, NY, USA, 1989), ACM Press, pp. 36–45.
3. Aït-Kaci, H. An overview of LIFE. In *Proc. Next Generation Information Systems Technology* (1991), J. W. Schmidt and A. A. Stognij, Eds., vol. 504 of *LNCS*, Springer-Verlag, pp. 42–58.
4. Albano, A., Cardelli, L., and Orsini, R. GALILEO: a strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems (TODS) 10*, 2 (1985), 230–260.
5. Albano, A., Ghelli, G., and Orsini, R. Fibonacci: a programming language for object databases. *The VLDB Journal 4*, 3 (1995), 403–444.
6. Alonso, G., Blott, S., Fessler, A., and Schek, H.-J. Correctness and parallelism in composite systems. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (1997), ACM Press, pp. 197–208.
7. American National Standards Institute. Coded character set – 7-bit american national standard code for information interchange, 1986.
8. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan, 1989), pp. 223–240.
9. Atkinson, M., Chisholm, K., and Cockshott, P. PS-algol: an algol with a persistent heap. *ACM SIGPLAN Notices 17*, 7 (1982), 24–31.
10. Atkinson, M., and Morrison, R. Orthogonally persistent object systems. *The VLDB Journal 4*, 3 (1995), 319–402.
11. Atkinson, M. P., and Buneman, P. Types and persistence in database programming languages. *ACM Computing Surveys (CSUR) 19*, 2 (1987), 105–170.
12. Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., and Spence, S. An orthogonally persistent Java. *ACM SIGMOD Record 25*, 4 (1996), 68–75.
13. Bancilhon, F., Briggs, T., Khoshafian, S., and Valduriez, P. FAD, a powerful and simple database language. In *Proceedings of 13th International*

*Conference on Very Large Data Bases* (1987), P. M. Stocker, W. Kent, and P. Hammersley, Eds., Morgan Kaufmann, pp. 97–105.

14. BANCILHON, F., DELOBEL, C., AND KANELLAKIS, P. *Building an object-oriented database system: the story of $O_2$*. Morgan Kaufmann Publishers Inc., 1992.
15. BEERI, C. A formal approach to object-oriented databases. *Data & Knowledge Engineering 5*, 4 (1990), 353–382.
16. BEERI, C., BERNSTEIN, P. A., AND GOODMAN, N. A model for concurrency in nested transactions systems. *Journal of the ACM (JACM) 36*, 2 (1989), 230–269.
17. BERNSTEIN, P. A., AND GOODMAN, N. Serializability theory for replicated databases. *Journal of Computer and System Sciences 31*, 3 (1985), 355–374.
18. BJØRNERSTEDT, A., AND BRITTS, S. AVANCE - an object management system. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), San Diego, California, USA* (September 1988), N. K. Meyrowitz, Ed., pp. 206–221.
19. BLOOM, T., AND ZDONIK, S. B. Issues in the design of object-oriented database programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), ACM Press, pp. 441–451.
20. BOBROW, D. G., AND STEFIK, M. The Loops manual. Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13, Xerox Corp., January 1983.
21. BRAINERD, W. S., AND LANDWEBER, L. H. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, USA, 1974.
22. BRIOT, J.-P., GUERRAOUI, R., AND LOHR, K.-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys (CSUR) 30*, 3 (1998), 291–329.
23. BUTENHOF, D. R. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., 1997.
24. BUTTERWORTH, P., OTIS, A., AND STEIN, J. The GemStone object database management system. *Communications of the ACM 34*, 10 (1991), 64–77.
25. CARDELLI, L. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds. Springer-Verlag, Berlin, 1991, pp. 431–507.
26. CARDELLI, L. Type systems. *ACM Computing Surveys 28*, 1 (1996), 263–264.
27. CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR) 17*, 4 (1985), 471–523.
28. CATELL, R. G. G., BARRY, D. K., BERLER, M., EASTMAN, J., JORDAN, D., RUSSELL, C., SCHADOW, O., STANIENDA, T., AND VELEZ, F. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers Inc., 2000.
29. COOK, W. R., GREENE, R., LINSKEY, P., MEIJER, E., RUGG, K., RUSSELL, C., WALKER, B., AND WITTIG, C. Objects and databases: State of the union in 2006. Panel at the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2006.
30. COOK, W. R., HILL, W., AND CANNING, P. S. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1990), ACM Press, pp. 125–135.

31. Cook, W. R., and Ibrahim, A. H. Integrating programming languages & databases: What's the problem? Draft Conference Submission, October 2005.
32. Cook, W. R., and Rosenberger, C. Native queries for persistent objects, a design white paper. *Dr. Doob's Journal (DDJ)* (February 2006).
33. Courcelle, B. Fundamental properties of infinite trees. *Theoretical Computer Science 25* (1983), 95–169.
34. Darwen, H., and Date, C. J. The third manifesto. *SIGMOD Rec. 24*, 1 (1995), 39–49.
35. Dickman, P. W. The Bellerophon project: A scalable object-support architecture suitable for a large OODBMS? In *Proceedings of the International Workshop on Distributed Object Management*, M. T. Özsu, U. Dayal, and P. Valduriez, Eds. Morgan Kaufmann, 1992, pp. 287–299.
36. Eisenberg, A., and Melton, J. SQL: 1999, formerly known as SQL3. *SIGMOD Rec. 28*, 1 (1999), 131–138.
37. Elmasri, R., and Navathe, S. B. *Fundamentals of database systems.* Addison Wesley, Pearson Education, Inc., 2004.
38. Engelschall, R. S. OSSP shared memory allocation, 2006. [Online; accessed 02-Aug-2006], `http://www.ossp.org/pkg/lib/mm/`.
39. Engelschall, R. S. OSSP universally unique identifier (UUID), 2006. [Online; accessed 02-May-2006], `http://www.ossp.org/pkg/lib/uuid/`.
40. Feuerstein, S., and Pribyl, B. *Oracle PL/SQL Programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
41. Finin, T., Fritzson, R., McKay, D., and McEntire, R. KQML as an agent communication language. In *Proceedings of the 3rd international conference on Information and knowledge management* (New York, NY, USA, 1994), ACM Press, pp. 456–463.
42. Fishman, D. H., Annevelink, J., Chow, E., Conners, T., Davis, J. W., Hasan, W., Hoch, C. G., Kent, W., Leichner, S., Lyngbaek, P., Mahbod, B., Neimat, M. A., Risch, T., Shan, M. C., and Wilkinson, W. K. Overview of the IRIS dbms. *Object-oriented concepts, databases, and applications* (1989), 219–250.
43. Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database System Implementation.* Prentice-Hall, 2000.
44. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, USA, 1994.
45. Goldberg, A., and Robson, D. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
46. Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java$^{TM}$ Language Specification, Third Edition: The Java Series.* Addison-Wesley Professional, 2005.
47. Gould, L., Zanevsky, A., and Kline, K. *Transact-SQL Programming.* O'Reilly Media, Inc., 1999.
48. Grehan, R. Introduction to odbms. ODBMS.ORG: Object Database Management Systems - The Resource Portal for Educa-

tion and Research, 2006. [Online; accessed 30-September-2006], http://www.odbms.org/introduction_rdbms2odbms.html.

49. HÄRDER, T. XML databases and beyond - plenty of architectural challenges ahead. In *Proceedings of the 9th East European Conference on Advances in Databases and Information Systems* (2005), J. Eder, H.-M. Haav, A. Kalja, and J. Penjam, Eds., vol. 3631 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.
50. HÄRDER, T., AND RAHM, E. *Datenbanksysteme – Konzepte und Techniken der Implementierung.* Springer, 1999.
51. HART, B. E., DANFORTH, S., AND VALDURIEZ, P. Parallelizing a database programming language. In *Proceedings of the 1st international symposium on Databases in parallel and distributed systems* (1988), IEEE Computer Society Press, pp. 72–79.
52. HRYNIOW, R., LENTNER, M., STENCEL, K., AND SUBIETA, K. Types and type checking in stack-based query languages. Tech. Rep. 984, Polish-Japanese Institute of Information Technology, Warsaw, Poland, March 2005.
53. IGARASHI, A., AND NAGIRA, H. Union types for object-oriented programming. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC)* (New York, NY, USA, 2006), ACM Press, pp. 1435–1441.
54. JODLOWSKI, A., HABELA, P., PLODZIEN, J., AND SUBIETA, K. Objects and roles in the stack-based approach. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications* (2002), R. Cicchetti, A. Hameurlain, and R. Traunmüller, Eds., vol. 2453 of *Lecture Notes in Computer Science*, Springer, pp. 514–523.
55. JORDAN, D., AND RUSSELL, C. *Java Data Objects.* O'Reilly Media, Inc., 2003.
56. KEENE, S. E. *A programmer's guide to object-oriented programming in Common LISP.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
57. KEMPER, A., AND KOSSMANN, D. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal 4*, 3 (1995), 519–567.
58. KENT, W. The evolving role of database in object systems. In *Proceedings of the 8th Bristish National Conference on Databases (BNCOD)* (1990), A. W. Brown and P. Hitchcock, Eds., Pitman Publishing, London, pp. 1–9.
59. KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, Inc., 1978.
60. KIM, W. Research directions in object-oriented database systems. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1990), ACM Press, pp. 1–15.
61. KIM, W. Object-oriented database systems: Promises, reality, and future. In *Proceedings of the 19th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1993), Morgan Kaufmann Publishers Inc., pp. 676–687.
62. KIM, W., BALLOU, N., CHOU, H.-T., GARZA, J. F., AND WOELK, D. Features of the ORION object-oriented database system. *Object-oriented concepts, databases, and applications* (1989), 251–282.

63. KIM, W., BALLOU, N., GARZA, J. F., AND WOELK, D. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems 9*, 1 (1991), 31–51.
64. KIRCHBERG, M. Seiten-, systempuffer- und satzverwaltung als grundlage eines datenbanksystems. Research Report, Clausthal University of Technology Germany. In German, December 1999.
65. KIRCHBERG, M. Ein experimenteller vergleich von transaktionsschedulern für mehrschichten-transaktionen. Master's thesis, Clausthal University of Technology, Germany, May 2000. In German.
66. KIRCHBERG, M. Exploiting multi-level transactions in distributed database systems. In *Distributed Data & Structures 4: Records of the 4th International Meeting*, W. Litwin and G. Lévy, Eds., vol. 14 of *Proceedings in Informatics*. Carleton Scientific, 2002, pp. 37–58.
67. KIRCHBERG, M. DBAA/ACL - a database agent architecture and communication language. In *Proceedings of the IADIS International Conference e-Society 2006* (July 2006), P. Isaías, M. McPherson, and F. Bannister, Eds., IADIS Press, pp. 244–249.
68. KIRCHBERG, M. An integrated database programming and querying language with support for simultaneous processing. In *Proceedings of the 2nd International Conference on Software Engineering Advances (ICSEA)* (2007), IEEE Computer Society Press.
69. KIRCHBERG, M. An overview of the object-oriented database programming language DBPQL. In *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS)* (2007), J. Cardoso, J. Cordeiro, and J. Filipe, Eds., vol. 1, INSTICC Press.
70. KIRCHBERG, M., KUCKELBERG, A., SCHEWE, K.-D., AND TRETIAKOV, A. On coding navigation paths for in-memory navigation in persistent object stores. In *Proceedings of the 19th Brazilian Symposium on Databases (SBBD)* (2004), S. Lifschitz, Ed., UnB, pp. 259–268.
71. KIRCHBERG, M., AND SCHEWE, K.-D. A comparison of multi-level concurrency control protocols. In *Proceedings of the 12th Australasian Database Conference (ADC)* (2001), M. E. Orlowska and J. F. Roddick, Eds., vol. 23 of *Australian Computer Science Communications: Database Technologies*, IEEE Computer Society Press, pp. 153–160.
72. KIRCHBERG, M., SCHEWE, K.-D., TRETIAKOV, A., AND WANG, B. R. A multi-level architecture for distributed object bases. *Data & Knowledge Engineering 60*, 1 (January 2007), 150–184.
73. KIRCHBERG, M., AND TRETIAKOV, A. A persistent object store as platform for integrated database programming and querying languages. Internal Report, Information Science Research Centre, Massey University, New Zealand, October 2006.
74. KOZANKIEWICZ, H., AND SUBIETA, K. SBQL views - prototype of updateable views. In *Local Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems* (2004).
75. LABROU, Y., FININ, T., AND PENG, Y. Agent communication languages: The current landscape. *IEEE Intelligent Systems 14*, 2 (1999), 45–52.

76. LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. The ObjectStore database system. *Communications of the ACM 34*, 10 (1991), 50–63.
77. LENTNER, M., STENCEL, K., AND SUBIETA, K. Semi-strong static type checking of object-oriented query languages. In *Proceedings of the 32nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)* (2006), J. Wiedermann, G. Tel, J. Pokorný, M. Bieliková, and J. Štuller, Eds., vol. 3831 of *Lecture Notes in Computer Science*, Springer, pp. 399–408.
78. LEONTIEV, Y. *Type system for an object-oriented database programming language.* PhD thesis, 1999. Adviser-M. Tamer Özsu and Adviser-Duane Szafron.
79. LEONTIEV, Y., ÖZSU, M. T., AND SZAFRON, D. On type systems for object-oriented database programming languages. *ACM Computing Surveys (CSUR) 34*, 4 (2002), 409–449.
80. LINDHOLM, T., AND YELLIN, F. *The Java$^{TM}$ Virtual Machine Specification, Second Edition.* Addison-Wesley Longman, Inc., 1999.
81. LIPKA, A. The design and implementation of TIGUKAT user languages. Tech. Rep. TR93-11, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, July 1993.
82. LISKOV, B., CURTIS, D., DAY, M., GHEMAWAT, S., GRUBER, R., JOHNSON, P., AND MYERS, A. C. *Theta Reference Manual - Preliminary Version*, February 2005. [Online], `http://www.pmg.lcs.mit.edu/papers/thetaref/`.
83. LISKOV, B., DAY, M., AND SHRIRA, L. Distributed object management in Thor. *Distributed Object Management* (1993), 79–91.
84. LOHMAN, G. M., LINDSAY, B., PIRAHESH, H., AND SCHIEFER, K. B. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM 34*, 10 (1991), 94–109.
85. LOMET, D. B. MLR: a recovery method for multi-level systems. In *Proceedings of the ACM SIGMOD international conference on Management of data* (1992), ACM Press, pp. 185–194.
86. MA, H., AND SCHEWE, K.-D. A heuristic approach to horizontal fragmentation in object oriented databases. In *Databases and Information Systems - Selected Papers from the 6th International Baltic Conference DB&IS'2004*, J. Barzdins and A. Caplinskas, Eds., vol. 118 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2005, pp. 20–33.
87. MA, H., AND SCHEWE, K.-D. Query optimisation as part of distribution design for complex value databases. In *Proceedings of the 15th European - Japanese Conference on Information Modelling and Knowledge Bases (EJC)* (2005), Y. Kiyoki, H. Kangassalo, H. Jaakkola, and J. Henno, Eds., IOS Press, pp. 269–276.
88. MEHLHORN, K., NÄHER, S., SEEL, M., AND UHRIG, C. *The LEDA User Manual (Version 4.2)*, 2000.
89. MEYER, B. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
90. MILLSTEIN, R. E. Control structures in Illiac IV Fortran. *Communications of the ACM 16*, 10 (1973), 621–627.
91. MITCHELL, J. C. *Type systems for programming languages*, vol. B of *Handbook of theoretical computer science.* MIT Press, 1990, ch. 8, pp. 365–458.

92. Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., and Schwarz, P. M. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (1992), 94–162.
93. Morrison, R., Connor, R. C. H., Kirby, G. N. C., Munro, D. S., Atkinson, M. P., Cutts, Q. I., Brown, A. L., and Dearle, A. The Napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, M. P. Atkinson and R. Welland, Eds. Springer, 1999, pp. 98–154.
94. Müller, U. The Brainfuck programming language. As cited by Brian Raiter: *Brainfuck – An Eight-Instruction Turing-Complete Programming Language.* [Online; accessed 02-Dec-2006], `http://www.muppetlabs.com/~breadbox/bf/`, 1993.
95. Nicola, M., and van der Linden, B. Native XML support in DB2 universal database. In *Proceedings of the 31st international conference on Very Large Data Bases* (2005), VLDB Endowment, pp. 1164–1174.
96. Nusdin, W. Associative access in persistent object stores. Master's thesis, Department of Information Systems, Massey University, New Zealand, 2004.
97. Nygaard, K., and Dahl, O.-J. The development of the SIMULA languages. In *Proceedings of the 1st ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 1978), ACM Press, pp. 245–272.
98. ODBMS.ORG Panel of Experts. ODBMS.ORG: Object database management systems - the resource portal for education and research, 2006. [Online; accessed 30-September-2006], `http://www.odbms.org/`.
99. Ontologic, Inc. ONTOS developer's guide. Burlington, MA, 1991.
100. Özsu, M. T., Peters, R. J., Szafron, D., Irani, B., Lipka, A., and Muñoz, A. TIGUKAT: A uniform behavioral objectbase management system. *VLDB Journal 4*, 3 (1995), 445–492.
101. Özsu, M. T., and Valduriez, P. *Principles of distributed database systems (2nd ed.).* Prentice-Hall, 1999.
102. Paterson, J., Edlich, S., Hörning, H., and Hörning, R. *The Definitive Guide to db4o.* Apress, Berkely, CA, USA, 2006.
103. Ramakrishnan, R., and Gehrke, J. *Database Management Systems.* McGraw-Hill Higher Education, 2003.
104. Riaz-ud-din, F. An implementation of the ARIES/ML recovery manager. Master's thesis, Department of Information Systems, Massey University, New Zealand, February 2002.
105. Riaz-ud-Din, F., and Schewe, K.-D. A query and update language for rational tree-type data structures. Internal Report, Information Science Research Centre, Massey University, New Zealand, December 2006.
106. Riaz-ud-Din, F., and Schewe, K.-D. A query compiler architecture for achieving value-representability in object-oriented databases. In *Electronic Proceedings of the International Conference on Innovations in Information Technologies (IIT)* (2006).
107. Rothermel, K., and Mohan, C. ARIES/NT: a recovery method based on write-ahead logging for nested transactions. In *Proceedings of the 15th interna-*

*tional conference on Very large data bases* (1989), Morgan Kaufmann Publishers Inc., pp. 337–346.

108. SAMARAS, G., BRITTON, K., CITRON, A., AND MOHAN, C. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases 3*, 4 (1995), 325–360.
109. SCHEWE, K.-D. On the unification of query algebras and their extension to rational tree structures. In *Proceedings of the 12th Australasian conference on Database technologies* (2001), IEEE Computer Society Press, pp. 52–59.
110. SCHEWE, K.-D. Fragmentation of object oriented and semistructured data. In *Proceedings of the Baltic Conference, BalticDB&IS 2002* (2002), Institute of Cybernetics at Tallinn Technical University, pp. 253–266.
111. SCHEWE, K.-D., RIPKE, T., AND DRECHSLER, S. Hybrid concurrency control and recovery for multi-level transactions. *Acta Cybernetica 14*, 3 (2000), 419–453.
112. SCHEWE, K.-D., AND SCHEWE, B. Integrating database and dialogue design. *Knowledge and Information Systems 2*, 1 (2000), 1–32.
113. SCHEWE, K.-D., STEMPLE, D. W., AND THALHEIM, B. Higher-level genericity in object-oriented databases. In *Conference on Management of Data* (1994).
114. SCHEWE, K.-D., AND THALHEIM, B. Fundamental concepts of object oriented databases. *Acta Cybernetica 11*, 1-2 (1993), 49–84.
115. SCHMIDT, J. W., AND MATTHES, F. The database programming language DBPL - rationale and report. Tech. rep., Hamburg, Germany, Germany, 1992.
116. SCOTT, M. L. *Programming language pragmatics.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
117. SESTOFT, P., AND HANSEN, H. I. *C# precisely.* MIT Press, Cambridge, MA, USA, 2004.
118. SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. *Database Systems Concepts.* McGraw-Hill Higher Education, 2002.
119. SIMONS, A. J. H. The theory of classification, part 17: Multiple inheritance and the resolution of inheritance conflicts. *Journal of Object Technology 4*, 2 (March - April 2005), 15–26.
120. SINGH, M. P. Agent communication languages: Rethinking the principles. *Computer 31*, 12 (1998), 40–47.
121. SKARRA, A. H., ZDONIK, S. B., AND REISS, S. P. ObServer: An object server for an object-oriented database system. In *On Object-Oriented Database System*, K. R. Dittrich, U. Dayal, and A. P. Buchmann, Eds., Topics in Information Systems. Springer, 1991, pp. 275–290.
122. SPEER, J. Database recovery: Expanding the ARIES constellation. Master's thesis, Department of Information Systems, Massey University, New Zealand, May 2005.
123. SPEER, J., AND KIRCHBERG, M. D-ARIES: A distributed version of the ARIES recovery algorithm. In *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems* (2005), J. Eder, H.-M. Haav, A. Kalja, and J. Penjam, Eds., Tallinn University of Technlogy Press, pp. 13–30.
124. STEMPLE, D., FEGARAS, L., SHEARD, T., AND SOCORRO, A. Exceeding the limits of polymorphism in database programming languages. In *Proceedings of*

*the international conference on extending database technology on Advances in database technology* (1990), Springer-Verlag New York, Inc., pp. 269–285.

125. Stemple, D., and Sheard, T. A recursive base for database programming primitives. *Lecture Notes in Computer Science 504* (1991), 311–332.

126. Stemple, D., Sheard, T., and Fegaras, L. Reflection: A bridge from programming to database languages. In *Proc of the Hawaii Conf on System Sciences* (1992).

127. Straube, D. D., and Özsu, M. T. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Engineering 7*, 2 (1995), 210–227.

128. Stroustrup, B. *The C++ programming language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

129. Subieta, K. LOQIS: The object-oriented database programming system. *Lecture Notes in Computer Science 504* (1991), 403–421.

130. Subieta, K. *Theory and Construction of Object-Oriented Query Languages.* Editors of the Polish-Japanese Institute of Information Technology, Warsaw, 2004.

131. Subieta, K., Beeri, C., Matthes, F., and Schmidt, J. W. A stack-based approach to query languages. Tech. Rep. 738, Institute of Computer Science Polish Academy of Sciences, Warszawa, Poland, dec 1993.

132. Szyperski, C. A. Import is not inheritance - why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming* (London, UK, 1992), Springer-Verlag, pp. 19–32.

133. Thalheim, B. *Entity-Relationship Modeling - Foundations of Database Technology.* Springer, 2000.

134. The Committee for Advanced DBMS Function Corporate. Third-generation database system manifesto. *SIGMOD Rec. 19*, 3 (1990), 31–44.

135. The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison Wesley Professional, 2003.

136. Torgersen, M. Inheritance is specialisation. In *Proceedings of the Inheritance Workshop at ECOOP 2002* (2002), G. Arévalo, A. Black, Y. Crespo, M. Dao, E. Ernst, P. Grogono, M. Huchard, and M. Sakkinen, Eds., vol. 2548 of *Lecture Notes in Computer Science*, Springer, pp. 117–134.

137. Tucker, A. B., and Noonan, R. E. *Programming Languages: Principles and Paradigms.* McGraw-Hill Higher Education, 2001.

138. Velez, F., Bernard, G., and Darnis, V. The $O_2$ object manager: An overview. In *Proceedings of the 15th International Conference on Very Large Data Bases* (1989), Morgan Kaufmann Publishers Inc., pp. 357–366.

139. Versant Object Technology, Inc. Versant dbms, 1992.

140. Wang, R. B., Kirchberg, M., and Schewe, K.-D. OORPC: A communication mechanism for distributed object bases. In *Proceedings of the 3rd International Conference on Electronic Commerce Engineering (ICeCE)* (October 2003), Z. Chen, X. Gu, G. Qi, and S. Fang, Eds., International Academic Publisher/World Publishing Corporation, pp. 676–680.

141. Weikum, G. A theoretical foundation of multi-level concurrency control. In *Proceedings of the 5th ACM SIGACT-SIGMOD symposium on Principles of database systems* (1986), ACM Press, pp. 31–43.

142. WEIKUM, G. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS) 16*, 1 (1991), 132–180.
143. WEIKUM, G., HASSE, C., BROESSLER, P., AND MUTH, P. Multi-level recovery. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (1990), ACM Press, pp. 109–123.
144. WIRTH, N. *Programming in MODULA-2 (3rd corrected ed.).* Springer-Verlag, New York, NY, USA, 1985.
145. WOLF, A. L. An initial look at abstraction mechanisms and persistence. In *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the 4th International Workshop on Persistent Objects* (1990), A. Dearle, G. M. Shaw, and S. B. Zdonik, Eds., Morgan Kaufmann, pp. 360–368.
146. ZEZULA, P., AND RABITTI, F. Object store with navigation acceleration. *Information Systems 18*, 7 (1993), 429–459.

# Chapter A

# The Syntax of iDBPQL

## A.1 The Lexical Syntax of iDBPQL

```
Boolean       = "TRUE" | "FALSE";
Character     = ? Characters of the ASCII Standard ?;
Letter        = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
                'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
                'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
                'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';
ZeroDigit     = '0';
PosDigit      = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit         = ZeroDigit | PosDigit;
HexDigit      = Digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' |
                'E' | 'F';
HexIndicator = 'x' | 'X';
PosNatural    = PosDigit, { Digit };
Natural       = ZeroDigit | PosNatural;
Integer       = ZeroDigit | ( [ '-' ], PosNatural );
Hexal         = ZeroDigit, HexIndicator, HexDigit, { HexDigit };
Fraction      = ZeroDigit | ( { ZeroDigit }, PosNatural );
Real          = ( Integer, '.', Fraction ) |
                ( [ Integer ], '.', Fraction, ( 'e' | 'E' ), Integer );
Null          = "NULL";

Literal       = Boolean | Character | Integer | Hexal | Natural | Null | Real;
Identifier    = Letter, { Letter | Digit | '_' };

Newline       = ? a newline character, e.g. '\n' ?;
WhiteSpace    = ? all white space characters such as newline, tabulator etc. ?;
NonNewlineWS = WhiteSpace - Newline;
Comment       = "//", { Character - Newline }, Newline;

Keyword       = "ABSTRACT" | "AND" | "ARRAY" | "AS" | "ASC" | "ATOMIC" | "BAG" |
                "BEHAVIOUR" | "BOOL" | "BOOLEAN" | "BREAK" | "CASE" | "CHAR" |
                "CHARACTER" | "CHECK" | "CLASSDEF" | "COLLECTION" | "COMBINES" |
                "CONCRETE" | "CONCURRENT" | "CONST" | "CONSTRAINT" | "DEFAULT" |
                "DESC" | "DISTINCT" | "DO" | "ELSE" | "ELSEIF" | "ENDDO" | "ENUM" |
                "EVALPLAN" | "EXISTS" | "FALSE" | "FINAL" | "FOR ANY" | "FOR EACH" |
                 "FROM" | "GROUP" | "IF" | "IMPORTS" | "IN" | "INDEPENDENT" |
                 "INIT" | "INNER" | "INT" | "INTEGER" | "IS" | "IS ACCEPTED FROM" |
                 "ISINSTANCEOF" | "ISSUBCLASSOF" | "ISSUBTYPEOF" | "IsA" | "JOIN" |
                 "LABEL" | "LEFT" | "LIKE" | "LIST" | "LOOP" | "NAT" | "NATURAL" |
                 "NAVIGATIONAL" | "NEW" | "NOT" | "NULL" | "ON" | "ORDER BY" |
                 "OUTER" | "PRIVATE" | "PUBLIC" | "READONLY" | "REAL" | "RETURN" |
                 "REVERSE" | "RIGHT" | "SCHEMA" | "SET" | "STATIC" | "STRING" |
                 "STRUCT" | "STRUCTURE" | "SUBRANGE" | "SUBTYPE" | "SWITCH" |
                 "THEN" | "TO" | "TRANSACTION" | "TRUE" | "TYPEDEF" | "UNIONDEF" |
                 "UNIQUE" | "WAIT" | "WHERE" | "WHILE" | "WITH";
Separator      = '(' | ')' | ':' | ';' | '<' | '>' | '{' | '}' | WhiteSpace;
Operator       = '!' | "!=" | '%' | "%=" | "&&" | '&' | "&=" | '*' | "*=" | '+' |
                 "++" | "+=" | '-' | "--" | "-=" | '.' | '/' | "/=" | '<' | "<<" |
                 "<<=" | "<=" | '=' | "==" | '>' | ">=" | ">>" | ">>=" | '^' | "^=" |
                 "|=" | "||" | '~';

Token          = Identifier | Keyword | Literal | Operator | Separator;
Lexeme         = Comment | Token | WhiteSpace;
Syntax         = Lexeme, { Lexeme };
```

## A.2 The Syntax of MetaData Catalogue Entries

### A.2.1 Syntax of DBS MetaData Units

```
DBSMetaDataUnit   = Schema;
Schema            = "SCHEMA", Id, '{', SchemaBlock, '}';
SchemaBlock       = { ImportDeclaration }, { SchemaDefinition };
ImportDeclaration = "IMPORTS", [ "SCHEMA" ], Id, [ '.', Id ], [ "AS", Id ];
SchemaDefinition  = ClassDefinition | ConstantDeclaration | TypeDeclaration;
```

### A.2.2 Syntax of Run-Time MetaData Units

```
RunTimeMetaDataUnit = EvalPlanAnnotation | EvalAnnotation;
EvalPlanAnnotation  = ClassDefinition | ConstantDeclaration | TypeDeclaration;
EvalAnnotation      = LocalDeclaration;
LocalDeclaration    = ConstantDeclaration | TypeDeclaration | VariableDecl;
```

### A.2.3 Common Syntax of Type-System-Related Definitions

```
ClassDefinition       = ClassModifierDecl, "CLASSDEF", Id,
                          [ '<', ClassParameter-List, '>' ], [ "IsA", ClassId-List ],
                          [ "WITH", '{', { ( ClassParaConstrClause |
                            PrecedenceClause | RenamingExpr ), ';' }, '}' ], '{',
                          [ StructuredType ],
                          [ "BEHAVIOUR", '{', { MethodSignature }, '}' ],
                          [ ConstraintDeclaration ], '}';
ClassModifierDecl     = [ ScopeModifierDecl ], [ StaticModifierDecl ],
                          [ ClassCatModifierDecl ], [ FinalModifierDecl ];
ScopeModifierDecl     = [ "PRIVATE" | "PUBLIC" ];
StaticModifierDecl    = "STATIC";
ClassCatModifierDecl  = AbstractModifierDecl | "COLLECTION" | "CONCRETE";
AbstractModifierDecl  = "ABSTRACT";
```

```
FinalModifierDecl      = "FINAL";
ClassParaConstrClause = ClassParameter, "IsA", ClassId-List;
PrecedenceClause       = ( Id, "IS ACCEPTED FROM", Id ) |
                         ( Id, "COMBINES", Id, { "AND", Id } );
MethodSignature        = MethodModifierDecl, Id, '(', [ Parameter-List ], ')', ':',
                         ResultType;
MethodModifierDecl     = [ ScopeModifierDecl ], [ StaticModifierDecl ],
                         [ AbstractModifierDecl | FinalModifierDecl ];
ConstraintDeclaration = "CONSTRAINT", [ Id ], '{'
                         { DomainConstraint | EntityConstraint }, '}';
DomainConstraint       = CheckConstraintDecl | NotNullConstraintDecl;
CheckConstraintDecl    = "CHECK", '(', Expression, ')';
NotNullConstraintDecl = "NOT NULL", '(', Id-List, ')';
EntityConstraint       = UniqueConstraintDecl;
UniqueConstraintDecl   = "UNIQUE", '(', Id-List, ')';

ConstantDeclaration    = ScopeModifierDecl, "CONST", Type, Id, [ '=', Expression ];

TypeDeclaration        = TypeDefinition | SubTypeDecl;
TypeDefinition         = ScopeModifierDecl, ( UserTypeDecl | TypeSynonymDecl );
UserTypeDecl           = "TYPEDEF", Id, [ '<', TypeParameter-List, '>' ],
                         [ "WITH", '{', { TypeParaConstrClause }, '}' ], '{',
                         StructuredType, [ "BEHAVIOUR", { TypeOpSignature } ], '}';
TypeParaConstrClause   = "SUBTYPE", '(', TypeParameter, ',', TypeId-List, ')', ';';
TypeOpSignature        = ScopeModifierDecl, ( Id | "INIT" ),
                         '(', [ Parameter-List ], ')', ':', ResultType;
TypeSynonymDecl        = "TYPEDEF", NoneVoidType, Id;
SubTypeDecl            = "SUBTYPE", '(', TypeId, ',', TypeId-List, ')';

VariableDecl           = VarModifierDecl, Type, Id, [ SimpleConstraint ],
                         [ "REVERSE", Id ];
VarModifierDecl        = [ ScopeModifierDecl | "READONLY" ], [ StaticModifierDecl ];
SimpleConstraint       = "NOT NULL";
```

### A.2.4 Common Syntax of iDBPQL Types

```
Type             = NoneVoidType | VoidType;
NoneVoidType     = BasicType | RefType;
BasicType        = AtomType | CollectionType | NULLableType | StructuredType |
                   TypeId | TypeParameter;
AtomType         = "BOOL" | "BOOLEAN" | "CHAR" | "CHARACTER" | NumericType;
NumericType      = "INT" | "INTEGER" | "NAT" | "NATURAL" | "REAL";
CollectionType   = ( "BAG", '<', NoneVoidType, '>' ) |
                   ( "SET", '<', NoneVoidType, '>' ) |
                   ( "LIST", '<', NoneVoidType, '>' ) | ( "STRING" ) |
                   ( "ARRAY", '<', NoneVoidType, '>', '[', [ NaturalValue ], ']' ) |
                   ( "ENUM", '(', StringValue, { ',', StringValue }, ')' ) |
                   ( "SUBRANGE", '<', NumericType, '>', "FROM", NumericValue,
                     "TO", NumericValue );
NULLableType     = "NULLABLE", '<', NoneVoidType, '>';
StructuredType   = ( "STRUCT" | "STRUCTURE" ), [ Id ],
                   ( ( '{', { StructMemberDecl }, '}' ) | StructuredType ),
                   [ '&', StructuredType, [ "WITH", '{', { RenamingExpr }, '}' ] ];
StructMemberDecl = StructuredType | VariableDecl;
```

```
RefType          = ClassId | ClassParameter | UnionType;
UnionType        = ScopeModifierDecl, "UNIONDEF", [ Id ] ,
                   '<', RefType, ',', RefType, '>';
VoidType         = "VOID";

ResultType       = VoidType | Type;
```

## A.3 The Remaining Syntax of iDBPQL

### A.3.1 Syntax of Evaluation Units

```
EvaluationUnit = iDBPQLProgram;
iDBPQLProgram  = "EVALPLAN", Id, '(', Argument-List, ')', [ ':', ReturnType ],
                 EvalPlanBlock;
EvalPlanBlock  = [ EvalPlanInit ], EvalBlock;
EvalPlanInit   = "INIT", DoBlock;
EvalBlock      = DoBlock;
```

### A.3.2 Syntax of Evaluation Blocks

```
DoBlock           = ( ( '{' | ( [ "INDEPENDENT" ], "DO", [ "ATOMIC" ],
                      [ "TRANSACTION", Tid ] ) ), Statements, ( '}' | "ENDDO" ) ) |
                    DoThenBlock;
DoThenBlock       = "DO", Statements, [ "THEN", DoBlock ], "ENDDO";

ConcurrentDoBlock = "CONCURRENT", "DO", Statements,
                    { "THEN", "DO", Statements, "ENDDO", ';' }, "ENDDO", ';';
```

### A.3.3 Syntax of Statements

```
Statements      = [ Statement, { ';', Statement } ];
Statement       = ControlFlowStmt | DoBlock | ExpressionStmt;
ControlFlowStmt = BreakStmt | ConditionStmt | LabelStmt | LoopStmt |
                  ReturnStmt | SwitchStmt | WaitStmt;
BreakStmt       = "BREAK", [ LabelId ], ';';
ConditionStmt   = "IF", '(', Expression, ')', DoBlock,
                  { "ELSEIF", '(', Expression, ')', DoBlock }, [ "ELSE", DoBlock ];
LabelStmt       = "LABEL", LabelId, ':', Statement;
LoopStmt        = DoWhileLoop | ForEach | LoopLoop | WhileLoop;
DoWhileLoop     = DoBlock, "WHILE", '(', BooleanExpr, ')', ';';
ForEach         = "FOR EACH", Expression, ( DoBlock | ConcurrentDoBlock );
LoopLoop        = "LOOP", DoBlock;
WhileLoop       = "WHILE", '(', BooleanExpr, ')', DoBlock;
ReturnStmt      = "RETURN", '(', [ Expression ], ')', ';';
SwitchStmt      = "SWITCH", '(', Expression, ')', SwitchBlock;
SwitchBlock     = '{', { CaseBlock }, [ "DEFAULT", ':', DoBlock ], '}';
CaseBlock       = "CASE", Expression, ':', DoBlock;
WaitStmt        = "WAIT", [ LabelId ], ';';
ExpressionStmt  = ( AssignmentExpr | CreationExpr | MethodCallExpr |
                    TypeOpCallExpr), ';';
```

### A.3.4 Syntax of Expressions

```
Expression        = AssignmentExpr | BinaryTypeOpExpr | BooleanExpr | CastExpr |
                    CreationExpr | Identifier | Literal | MethodCallExpr |
                    QueryExpr | RenamingExpr | TypeOpCallExpr | UnaryTypeOpExpr |
                    ( '(', Expression, ')' );
AssignmentExpr    = ( Expression, '=', Expression ) |
                    ( Expression, CompoundAssignOp, Expression ) |
                    ( Expression, InDeCrementOp ) | ( InDeCrementOp, Expression );
CompoundAssignOp = "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=";
InDeCrementOp     = "++" | "--";
BinaryExpr        = Expression, BinaryOperator, Expression;
BinaryOperator    = '%' | "&&" | '&' | '*' | '+' | '-' | '.' | '/' | "<<" | ">>" |
                    '^' | "||" | '~';
BooleanExpr       = EqualityExpr | InCollectionExpr | InheritanceExpr |
                    InstanceOfExpr | LogicalExpr | NULLExpr | QuantifierExpr |
                    RelationalExpr;
EqualityExpr      = Expression, ( "==" | "!=" ), Expression;
InCollectionExpr = Expression, "IN", Expression;
InheritanceExpr   = Expression, ( "ISSUBTYPEOF" | "ISSUBCLASSOF" ), Expression;
InstanceOfExpr    = Expression, "ISINSTANCEOF", Expression;
LogicalExpr       = ( BooleanExpr, ( "&&" | "||" ), BooleanExpr ) |
                    ( '!', BooleanExpr );
NULLExpr          = Expression, "IS", [ "NOT" ], "NULL";
QuantifierExpr    = ( "EXISTS" | "FOR ANY" ), Expression, '(', BooleanExpr, ')';
RelationalExpr    = ( Expression, ( '<' | "<=" | ">=" | '>' | "LIKE" ), Expression );
CastExpr          = '(', ( TypeId | ClassId ), ')', Expression;
CreationExpr      = "NEW", ClassId, '(', [ Argument-List ], ')';
MethodCallExpr    = MethodId, '(', [ Argument-List ], ')';
QueryExpr         = JoinExpr | OrderByExpr | ProjectionExpr | SelectionExpr |
                    UniquenessExpr;
JoinExpr          = Expression, [ "NAVIGATIONAL" | "INNER" | ( [ "NATURAL" ],
                      [ "LEFT" | "RIGHT" ], [ "OUTER" ] ) ], "JOIN", Expression,
                    [ "ON", ( PathExpr | BooleanExpr ) ];
PathExpr          = Id, PathComponent, { PathComponent };
PathComponent     = '.', { Id };
OrderByExpr       = Expression, "ORDER BY", Expression, [ "ASC" | "DESC" ];
ProjectionExpr    = Expression, '.', ProjectComponent;
ProjectComponent = Expression |
                    ( '(', ProjectComponent, { ',', ProjectComponent }, ')' );
SelectionExpr     = Expression, "WHERE", BooleanExpr;
UniquenessExpr    = ( "DISTINCT" | "UNIQUE" ), Expression;
RenamingExpr      = Expression, [ "GROUP" ], "AS", Id;
TypeOpCallExpr    = TypeOpId, '(', [ Argument-List ], ')';
UnaryExpr         = ( '+' | '-' ), Expression;
```

### A.3.5 Identifiers, Labels, Values and More

```
Argument            = Id;
Argument-List       = Argument, { ',', Argument };
AtomicValue         = Literal;
ClassId             = Id;
ClassId-List        = ClassId, [ "AS", Id ], { ',', ClassId, [ "AS", Id ] };
ClassParameter      = Id;
```

```
ClassParameter-List = ClassParameter, { ',', ClassParameter };
Id                  = Identifier;
Id-List             = Id, { ',', Id };
LabelId             = Id;
MethodId            = Id;
NaturalValue        = Natural;
NumericValue        = Integer | Natural | Real;
Parameter           = Id;
Parameter-List      = Parameter, { ',', Parameter };
StringValue         = { Character };
Tid                 = Id;
TypeId              = Id;
TypeId-List         = TypeId, { ',', TypeId };
TypeOpId            = Id;
TypeParameter       = Id;
TypeParameter-List  = TypeParameter, { ',', TypeParameter };
Value               = AtomicValue | StringValue;
```

# Chapter B

# The *Parentage* Database Example

Section 6.2 (refer to Figure 6.5 on Page 236) briefly introduces a *Parentage* database. The example's corresponding schema definition is as follows:

```
SCHEMA Parentage {
  TYPEDEF ENUM ( 'm', 'f' ) SexT;
  TYPEDEF NameT {
    STRUCTURE {
      NULLABLE < LIST < STRING > > titles;
      NULLABLE < STRING >         firstName;
      STRING                      lastName;
    }
  }

  CLASSDEF PersonC {
    STRUCTURE {
      NameT                       name;
      READONLY DateT              bDate;   // DateT defined in iDBPQL library
      READONLY NULLABLE < DateT > dateOfDeath;
      NULLABLE < SexT >           sex;
    }
    BEHAVIOUR {
      addDateOfDeath ( DateT dod ) : VOID;
      getAge ( ) : NAT;
      hasKnownParents ( ) : BOOLEAN;
      hasLivingParent ( ) : BOOLEAN;
      isAlive ( ) : BOOLEAN;
      PersonC ( STRING name, DateT bDate );
      PersonC ( STRING name, DateT bDate, SexT sex );
    }
    CONSTRAINT {
      UNIQUE ( name, bDate );
      CHECK ( ( dateOfDeath IS NULL ) OR ( bDate <= dateOfDeath ) );
    }
  }

  CLASSDEF ParentC IsA PersonC {
    STRUCTURE {
      READONLY SET < PersonC > children;
    }
    BEHAVIOUR {
      addNewChild ( PersonC child ) : VOID;
      ParentC ( STRING name, DateT bDate, SET < PersonC > children );
      ParentC ( STRING name, DateT bDate, SexT sex, SET < PersonC > children );
    }
  }
}
```

Due to its complexity, the internal representation of this schema has not been detailed in Section 6.2. A more complete version of the *Parentage* database's internal representation is as follows:

```
// file name: Parentage.md
__schemaInfo
  __name: Parentage
  __typeSynCount: 1
  __typeSyn:
    __typeSynInfo (1)
      __modFlag: 000
      __name: SexT
      __typeSynDescriptor: s<st('m','f')
  __typeCount: 1
  __types:
    __typeInfo (1)
      __modFlag: 000
      __name: NameT
      __typeDescriptor:
      __fieldCount: 3
      __fields:
        __fieldInfo (1)
          __modFlag: 000
          __name: titles
          __varDescriptor: n<l<st
          __attribCount: 0
          __attributes:
        __fieldInfo (2)
          __modFlag: 000
          __name: firstName
          __varDescriptor: n<st
          __attribCount: 0
          __attributes:
        __fieldInfo (3)
          __modFlag: 000
          __name: lastName
          __varDescriptor: st
          __attribCount: 0
          __attributes:
      __typeOpCount: 0
      __typeOps:
  __classCount: 2
  __classes:
    __classInfo (1)
      __modFlag: 040
      __name: PersonC
```

```
__classDescriptor:
__supClassCount: 0
__supClasses:
__fieldCount: 4
__fields:
  __fieldInfo (1)
    __modFlag: 000
    __name: name
    __varDescriptor: t<NameT
    __attribCount: 0
    __attributes:
  __fieldInfo (2)
    __modFlag: 001
    __name: bDate
    __varDescriptor: t<DateT
    __attribCount: 0
    __attributes:
  __fieldInfo (3)
    __modFlag: 001
    __name: dateOfDeath
    __varDescriptor: n<t<DateT
    __attribCount: 0
    __attributes:
  __fieldInfo (4)
    __modFlag: 000
    __name: sex
    __varDescriptor: t<SexT
    __attribCount: 0
    __attributes:
__methodCount: 7
__methods:
  __methodInfo (1)
    __modFlag: 000
    __name: addDateOfDeath
    __methodDescriptor: (t<DateT dod)vd
    __attribCount: 1
    __attributes:
  __methodInfo (2)
    __modFlag: 000
    __name: getAge
    __methodDescriptor: ()na
    __attribCount: 1
    __attributes:
      __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: 5a3ee33c-88b2-44cb-b1c5-3f24202a6fda
  __methodInfo (3)
    __modFlag: 000
    __name: hasKnownParents
    __methodDescriptor: ()bo
    __attribCount: 1
```

```
      __attributes:
       __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: 65a002e3-c34f-450d-9b31-df7938df6174
     __methodInfo (4)
      __modFlag: 000
      __name: hasLivingParent
      __methodDescriptor: ()bo
      __attribCount: 1
      __attributes:
       __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: 9f5027a8-f2e1-468e-b5cb-802a0b4436c8
     __methodInfo (5)
      __modFlag: 000
      __name: isAlive
      __methodDescriptor: ()bo
      __attribCount: 1
      __attributes:
       __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: a06cb694-e38d-4b14-9998-2ca292470fee
     __methodInfo (6)
      __modFlag: 000
      __name: PersonC
      __methodDescriptor: (st name,t<DateT bDate)
      __attribCount: 1
      __attributes:
       __attribInfo (1)
        __attribType: CONSTRUCTOR
        __name:
        __value:
        __code: 04193a34-1010-422d-91f5-a611cec75120
     __methodInfo (7)
      __modFlag: 000
      __name: PersonC
      __methodDescriptor: (st name,t<DateT bDate,t<SexT sex)
      __attribCount: 1
      __attributes:
       __attribInfo (1)
        __attribType: CONSTRUCTOR
        __name:
        __value:
        __code: 37d848dd-a79c-40f9-b57e-d02f0c05a7a9
    __constrCount: 1
    __classConstrs:
     __classConstrInfo (1)
```

```
          __constrCount: 2
          __constraints:
            __constrInfo (1)
              __type: 1
              __fieldCount: 2
              __fields:
                __fieldInfo (1) *name
                __fieldInfo (2) *bDate
              __attribCount: 0
              __attributes:
            __constrInfo (2)
              __type: 2
              __fieldCount: 0
              __fields:
              __attribCount: 1
              __attributes:
                __attribInfo (1)
                  __attribType: CONSTRAINT
                  __name: check1
                  __value:
                  __code: 7afcfa51-1910-4ff5-b383-571687c98c8a
    __classInfo (2)
      __modFlag: 040
      __name: ParentC
      __classDescriptor:
      __supClassCount: 1
      __supClasses:
        __classInfo (1) *PersonC
      __fieldCount: 5
      __fields:
        __fieldInfo (1)
          __modFlag: 000
          __name: name
          __varDescriptor: t<NameT
          __attribCount: 0
          __attributes:
        __fieldInfo (2)
          __modFlag: 001
          __name: bDate
          __varDescriptor: t<DateT
          __attribCount: 0
          __attributes:
        __fieldInfo (3)
          __modFlag: 001
          __name: dateOfDeath
          __varDescriptor: n<t<DateT
          __attribCount: 0
          __attributes:
        __fieldInfo (4)
          __modFlag: 000
          __name: sex
          __varDescriptor: t<SexT
          __attribCount: 0
          __attributes:
        __fieldInfo (5)
          __modFlag: 001
          __name: children
          __varDescriptor: s<c<PersonC
          __attribCount: 0
          __attributes:
      __methodCount: 8
      __methods:
        __methodInfo (1)
          __modFlag: 000
          __name: addDateOfDeath
          __methodDescriptor: (t<DateT dod)vd
          __attribCount: 1
          __attributes:
        __methodInfo (2)
          __modFlag: 000
          __name: getAge
          __methodDescriptor: ()na
          __attribCount: 1
          __attributes:
            __attribInfo (1)
              __attribType: EVALPLAN
              __name:
              __value:
              __code: 5a3ee33c-88b2-44cb-b1c5-3f24202a6fda
        __methodInfo (3)
          __modFlag: 000
          __name: hasKnownParents
          __methodDescriptor: ()bo
          __attribCount: 1
          __attributes:
            __attribInfo (1)
              __attribType: EVALPLAN
              __name:
              __value:
              __code: 65a002e3-c34f-450d-9b31-df7938df6174
        __methodInfo (4)
          __modFlag: 000
          __name: hasLivingParent
          __methodDescriptor: ()bo
          __attribCount: 1
          __attributes:
            __attribInfo (1)
              __attribType: EVALPLAN
              __name:
              __value:
              __code: 9f5027a8-f2e1-468e-b5cb-802a0b4436c8
        __methodInfo (5)
          __modFlag: 000
          __name: isAlive
          __methodDescriptor: ()bo
          __attribCount: 1
```

```
    __attributes:
      __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: a06cb694-e38d-4b14-9998-2ca292470fee
  __methodInfo (6)
    __modFlag: 000
    __name: addNewChild
    __methodDescriptor: (c<PersonC child)vd
    __attribCount: 1
    __attributes:
      __attribInfo (1)
        __attribType: EVALPLAN
        __name:
        __value:
        __code: a54d3d13-d276-4059-a954-6f5bfdc3cecc
  __methodInfo (7)
    __modFlag: 000
    __name: ParentC
    __methodDescriptor: (st name,t<DateT bDate,s<c<PersonC children)
    __attribCount: 1
    __attributes:
      __attribInfo (1)
        __attribType: CONSTRUCTOR
        __name:
        __value:
        __code: b8760f92-5af7-4009-8651-6b669386e891
  __methodInfo (8)
    __modFlag: 000
    __name: ParentC
    __methodDescriptor: (st name,t<DateT bDate,t<SexT sex,s<c<PersonC
                         children)
    __attribCount: 1
    __attributes:
      __attribInfo (1)
        __attribType: CONSTRUCTOR
        __name:
        __value:
        __code: d63cc78b-8e3a-4d63-a3f2-c550dc5d1bff
__constrCount: 1
__classConstrs:
  __classConstrInfo (1)
    __constrCount: 2
    __constraints:
      __constrInfo (1)
       __type: 1
        __fieldCount: 2
        __fields:
          __fieldInfo (1) *name
          __fieldInfo (2) *bDate
        __attribCount: 0
        __attributes:
          __constrInfo (2)
            __type: 2
            __fieldCount: 0
            __fields:
            __attribCount: 1
            __attributes:
              __attribInfo (1)
                __attribType: CONSTRAINT
                __name: check1
                __value:
                __code: 7afcfa51-1910-4ff5-b383-571687c98c8a
  __isaRelation:
    ParentC IsA PersonC
```

The internal schema is represented by a `__schemaInfo` structure and each type synonym, type definition and class definition has an associated `__typeSynInfo` structure, `__typeInfo` structure or `__classInfo` structure, respectively. Inherited properties are stored with the sub-class. While the uniqueness constraint is fully represented in the data definition portion, the check constraint is transformed into an evaluation plan such as:

```
EVALPLAN check1 ( ) : BOOLEAN {
  if ( ( dateOfDeath IS NULL ) OR ( bDate <= dateOfDeath ) ) {
    RETURN ( TRUE );
  }
  RETURN ( FALSE );
}
```