ARNE-JØRGEN BERRE

AN OBJECT-ORIENTED FRAMEWORK FOR SYSTEMS INTEGRATION AND INTEROPERABILITY
An Object-Oriented Framework
for Systems Integration and Interoperability

Arne-Jørgen Berre
Division of Computer Systems and Telematics
Faculty of Computer Science and Electrical Engineering
Norwegian Institute of Technology
University of Trondheim

July, 1993
"Tower of Babel" Painting by F. Brentel (1580-1651)

"The Holy Bible - GENESIS 11:1-9"

1) Now the whole world had one language and a common speech.
4) Then they said, "Come, let us build ourselves a city, with a tower that reaches to the heavens, so that we may make a name for ourselves and not be scattered over the face of the whole earth".
5) But the LORD came down to see the tower that the men were building.
6) The LORD said, "If as one people speaking the same language they have begun to do this, then nothing they plan will be impossible for them."
7) "Come let us go down and confuse their language so that they will not understand each other."
8) So the LORD scattered them from there over all the earth, and they stopped building the city.
9) That is why it was called Babel, because there the LORD confused the language of the whole world. From there the LORD scattered them over the face of the whole earth.
Abstract

Today's world of information systems is becoming more heterogeneous and complex, and there is an imperative need to create methods and tools that make it easier to achieve cooperation and integration between old and new systems. The goal of this thesis is to provide a framework for systems integration and interoperability for heterogeneous systems, in order to make the development of cooperative systems an easier task.

A taxonomy for problems and solutions for systems integration and interoperability has been developed, based on a refinement of the ECMA/NIST Toaster model with concepts from the Eureka Software Factory architectural model. This has resulted in a reference model that distinguishes between request-oriented and notification-oriented control integration for the interaction between tools, single model and multi-model data integration for the sharing of data between tools, display-oriented and model-oriented presentation integration for the creation of user interfaces, and interaction-oriented and interworking-oriented process integration for the support of work processes.

The hypothesis that object-oriented technology is a suitable tool for facilitating systems integration has been validated through individual experiments in each integration-area. Object-Oriented control integration has been shown through a "Software Bus", a communication oriented interaction mechanism. Object-Oriented data integration has been shown through the development and experiences from the object-oriented model SOOM, and the development of the "HyperModel" benchmark for database-system-evaluation and corresponding evaluation of object-oriented database-systems. Object-Oriented presentation integration has been shown through a separation of the user-interface-part and the functional part of an application, and by the use of object-oriented class-libraries for user-interfaces.

The COOP Integration framework is proposed as a new solution, unifying the different integration-areas through the use of COOM, the COOP Object Model. COOM is a structurally object-oriented model above a behaviourally object-oriented model that supports control integration through encapsulation and event-notification, supports data integration through structural abstractions for attributes and relations, and supports presentation integration through explicit separation of user-interface objects and functional objects. Transparent access to underlying existing systems is supported by the explicit separation between interface and implementation for objects. Multiple implementations can be handled simultaneously both for objects and for relations, and provides a foundation for multi-model data integration. Multiple implementations are supported by query processing in the OQL query language. The COORASS methodology supports integrated behavioural and structural modeling. The COOP architecture support the use of COOM in a distributed environment, with tools for mappings to existing object-oriented programming languages, such as C++ and Smalltalk.

These results contribute to a better understanding of different aspects of systems integration, and can form a basis for more efficient development of integrated and cooperative systems in the future, and a platform for future research.
Preface

The work presented in this report is a thesis submitted to the Norwegian Institute of Technology for the doctor degree "doktor ingeniør". The work has been carried out at the Norwegian Institute of Technology and SINTEF SI\(^1\), from fall 1988 to summer 1993.

Acknowledgements

There are many people to thank because of their contribution to this work. Section 1.6 describes the history and context of this work, and gives the names on a number of persons that are acknowledged for their support to this work.

My adviser Professor Reidar Conradi at Division for Computer Systems and Telematics at NTH\(^2\) has provided inspiration and valuable comments and feedback throughout the years, and has also created an inspiring work-environment at IDT/NTH, with his never-ending technical and social efforts in establishing and running the EPOS and the REBOOT projects.

Professor Trygve Reenskaug at the University of Oslo and Taskon A/S, previously Chief Scientist at SI, initially brought me into the world of object-oriented thinking and technology. Trygve provided the inspiration for starting up this thesis-work, and a good guidance for me, and others, in his leadership in the EKI-project from 1984 to 1989, in the technical foundations of the ESF-project in 1987 to 1990, and through the industrial appliance of object-oriented technology by the Taskon-company from its startup in 1989. He has always been a source of good ideas and a challenging discussion-partner.

My stay at Oregon Graduate Institute from August 1987 to August 1988 and the stay at NTH from September 1990 to May 1992 was made possible through grants from NTNF\(^3\) and support from SINTEF SI. SINTEF SI, through research director Øystein Strandli, has provided me with time to finalize this work. The EKI-project, ESF-project and EIDER- and IDIS- pre-projects in which I have been working have all received grant-support from NTNF.

The students that I have guided through the years have given me the opportunity to discuss ideas, and to get experiences from the development and experimentation with different prototype-tools. My colleagues at SINTEF SI and SINTEF DELAB, and fellow dr.ing-students at NTH have given me valuable feedback on ideas, and provided an inspiring environment for this work. In particular Bjørn Skjellaug, Bo Kähler and Svein-Olaif Hvasshovd have given valuable comments on the final drafts of this thesis. Lucia Berre have provided professional assistance with English grammar and proofreading.

All the people mentioned above, and in section 1.6, are thanked for their valuable contributions.

Finally, my deepest thanks goes to my wife, Marianne. Without her support, encouragement and understanding, this work would not have been possible.

Oslo, July 1993

Arne-Jørgen Berre

---

\(^1\)Center for Industrial Research, (SI), until 1. January 1993
\(^2\)Norwegian Institute of Technology
\(^3\)The Royal Norwegian Council for Scientific Research
Contents

1 Introduction ................................................................................................................................. 1
  1.1 The Problems of Systems Integration and Interoperability .................................................. 1
  1.2 The Motivation for this Thesis ............................................................................................... 2
  1.3 The Vision for this Thesis ..................................................................................................... 3
  1.4 The Structure of this Thesis ................................................................................................. 5
    1.4.1 Part I: Systems Integration - A Taxonomy of Problems and Solutions ......................... 5
    1.4.2 Part II: Object-Oriented Approaches to Systems Integration ....................................... 5
    1.4.3 Part III: A Unified Object-Oriented Approach to Systems Integration ....................... 6
    1.4.4 Conclusion ...................................................................................................................... 6
  1.5 The Contributions of this Thesis .......................................................................................... 6
  1.6 History and Context of this Thesis ....................................................................................... 9
  1.7 How to Read this Thesis ? .................................................................................................... 12

I Systems Integration - A Taxonomy of Problems and Solutions .................................................. 13

2 Systems Integration and Interoperability in Cooperative Work Environments ................................ 15
  2.1 Cooperative Information Systems and Cooperative Work Environments ......................... 15
  2.2 Systems Integration and Interoperability .......................................................................... 17
  2.3 The Environment User Perspective ...................................................................................... 18
    2.3.1 User Task Support .......................................................................................................... 18
    2.3.2 Support for Cooperative Work ....................................................................................... 20
    2.3.3 Work Process Support .................................................................................................... 22
  2.4 The Environment Builder Perspective ................................................................................. 24
    2.4.1 The ECMA Toaster Reference Model for Integrated Environments ....................... 24
    2.4.2 The ESF Reference Models for Integrated Environments ........................................ 26
    2.4.3 The Refined Double Toaster Model ............................................................................. 30
  2.5 The Problem Focus of this Thesis ....................................................................................... 32

3 Integration Requirement Areas .................................................................................................... 33
  3.1 Conflicting Requirement ...................................................................................................... 33
  3.2 Requirement Areas for Integration Facilities ....................................................................... 34
  3.3 Requirement Areas for Control Integration Facilities .......................................................... 34
    3.3.1 Request-oriented Control Integration ............................................................................... 34
    3.3.2 Requirement Areas for Request-oriented Control Integration (RCR) ......................... 35
    3.3.3 Notification-oriented Control Integration ..................................................................... 38
    3.3.4 Requirement Areas for Notification-oriented Control Integration (RCN) .................. 38
  3.4 Requirement Areas for Data Integration Facilities ............................................................... 39
    3.4.1 Single Model Data Integration ...................................................................................... 39
    3.4.2 Requirement Areas for Single Model Data Integration (RDS) .................................. 40
    3.4.3 Multi-model Data Integration ....................................................................................... 44
3.4.4 Requirement Areas for Multi model Data Integration (RDM) ........ 45
3.5 Requirement Areas for Presentation Integration Facilities ............. 49
  3.5.1 Requirement Areas for Presentation integration .................. 50
3.6 Architectural Support for Process Integration Facilities ............. 52
3.7 Conclusion on Integration Requirement Areas .......................... 53

4 Evaluation of Systems Integration Frameworks .......................... 55
  4.1 A Taxonomy of Solution-approaches for Systems Integration .......... 55
    4.1.1 Taxonomy ................................................. 55
    4.1.2 Distributed Computing approach to Control Integration .......... 55
    4.1.3 Database approach to Data Integration ........................ 57
    4.1.4 Various tool-approaches to Presentation Integration ........... 59
    4.1.5 Various Approaches to Process Integration .................... 60
  4.2 Evaluation of Systems Integration Frameworks ........................ 61
  4.3 The Object Management Group .................................... 62
    4.3.1 The OMG Activity ....................................... 62
    4.3.2 The OMG Architecture ................................... 62
    4.3.3 OMG Support for Control Integration .......................... 63
    4.3.4 OMG Support for Data Integration ........................... 66
    4.3.5 OMG Support for Presentation Integration ....................... 67
    4.3.6 OMG Support for Process Integration .......................... 67
    4.3.7 OMG and Integration - Conclusion ................................ 67
  4.4 HP/Softbench - Field ............................................ 67
    4.4.1 The HP/Softbench - Field System ................................ 67
    4.4.2 The HP/Softbench - Field Architecture ........................ 68
    4.4.3 HP/Softbench - Field Support for Control Integration .......... 68
    4.4.4 HP/Softbench - Field Support for Data Integration ............. 70
    4.4.5 HP/Softbench - Field Support for Presentation Integration ...... 70
    4.4.6 HP/Softbench - Field Support for Process Integration .......... 70
    4.4.7 HP/Softbench - Field and Integration - Conclusion .............. 71
  4.5 PCTE .......................................................... 71
    4.5.1 The PCTE Project .......................................... 71
    4.5.2 The PCTE Architecture ..................................... 71
    4.5.3 PCTE Support for Control Integration .......................... 73
    4.5.4 PCTE Support for Data Integration ........................... 73
    4.5.5 PCTE Support for Presentation Integration ....................... 75
    4.5.6 PCTE Support for Process Integration .......................... 75
    4.5.7 PCTE and Integration - Conclusion ................................ 75
  4.6 Pegasus ......................................................... 75
    4.6.1 The Pegasus System ........................................... 75
    4.6.2 The Pegasus Architecture .................................... 75
    4.6.3 Pegasus Support for Control Integration ........................ 75
    4.6.4 Pegasus Support for Data Integration .......................... 76
    4.6.5 Pegasus Support for Presentation Integration .................... 77
    4.6.6 Pegasus Support for Process Integration ........................ 77
    4.6.7 Pegasus and Integration - Conclusion ............................ 77
  4.7 ESF ............................................................. 78
    4.7.1 ESF Project ................................................ 78
    4.7.2 The ESF Architecture ....................................... 78
    4.7.3 ESF Support for Control Integration ............................ 79
    4.7.4 ESF Support for Data Integration ................................ 80
    4.7.5 ESF Support for Presentation Integration ........................ 81
    4.7.6 ESF Support for Process Integration ............................ 81
CONTENTS

7.2.2 The HyperModel Application ........................................ 122
7.2.3 Benchmark-operations ............................................... 122
7.2.4 Experiences and Results from the HyperModel Benchmark ........ 124
7.2.5 Object-Oriented Databases and Single Model Data Integration ... 125
7.2.6 Client- and Server-oriented Architectures ......................... 128
7.2.7 Object Oriented Database-Systems - Conclusion ...................... 130
7.3 Contribution and Conclusion .......................................... 131

8 Experiments with Object-Oriented Presentation Integration .......... 133
8.1 Object-Oriented Presentation Integration .............................. 133
8.2 The Object-Oriented Model-View-Controller Architecture ............ 134
8.3 The ESF Separation of User Interaction and Service Components ...... 135
8.4 SITE Presentation Integration Experiences ............................. 135
8.5 CODEDISC Presentation Integration Experiences ....................... 136
8.6 PIONA Presentation Integration Experiences ............................ 138
8.7 Contribution and Conclusion ........................................... 138

III COOP: A Unified Object-Oriented Approach to Systems Integration 141

9 COOP - An Object-Oriented Framework for Systems Integration 143
9.1 Background and Motivation for the COOP Integration Framework .... 143
9.2 COOP Goals and Solution Approach .................................... 145
9.3 COOP Architecture ..................................................... 148
9.4 COOM - Principles and Concepts ....................................... 150
9.4.1 COOM - Basic Principles ........................................... 150
9.4.2 A Model with Support for both Behavioural and Structural Objects 150
9.4.3 Encapsulated Objects in a Distributed Object Space ............... 150
9.4.4 Separation between Interface, Implementation, Factory and Extent 151
9.4.5 The use of Three Languages: ODL - OML - OQL ..................... 152
9.4.6 ODL - Object Definition Language .................................. 152
9.4.7 OML - Object Manipulation Language ............................... 153
9.4.8 OQL - Object Query Language ..................................... 154
9.5 COORASS Methodology .................................................. 154
9.6 Control and Data Integration for the Component Integrator .......... 155
9.7 Presentation Integration for the Tool Builder .......................... 155
9.8 A Scenario for the use of COOP and COOM ............................ 155
9.9 Structure of Part III .................................................. 156

10 COOM Foundation Model ................................................. 157
10.1 The COOM-model ....................................................... 157
10.1.1 Values .......................................................... 159
10.2 ODL - Schema Definitions ............................................ 160
10.3 Interface Definition and Syntax ....................................... 162
10.3.1 Operation Definition - Syntax .................................... 163
10.3.2 Conformance and Specialization of Interfaces ...................... 165
10.3.3 ODL Interface Preprocessing Facilities ............................ 167
10.4 Implementation and Object Space ..................................... 169
10.5 Contribution and Conclusion ......................................... 171

11 COOM Behavioural Model and Control Integration ..................... 173
11.1 Object Interfaces and Manager Objects ............................... 174
CONTENTS

11.1.1 OML - Object Interface Operations ........................................... 178
11.2 Object Implementation and Extent Management ................................. 179
11.2.1 OML - Object Manager Operations ............................................. 182
11.2.2 Interface Management Operations.............................................. 183
11.2.3 Object Manager Creation Operations - Interface and Implementation 184
11.2.4 Object Manager Retrieval Operations - Interface and Implementation 186
11.3 OML for Object Space Management and Transactions ........................ 187
11.3.1 Object Space Management ....................................................... 187
11.3.2 Transactions ............................................................................. 187
11.3.3 Alternative Transaction Models .................................................. 189
11.4 Mapping from COOMS to Object-Oriented Languages .......................... 190
11.4.1 Mapping from COOMS to C++ .................................................... 190
11.4.2 Mapping from COOMS to Smalltalk ........................................... 193
11.4.3 COOP as an Application Framework for Systems Integration .......... 194
11.5 Notifications ............................................................................... 195
11.5.1 Notification Definition and Example ........................................... 195
11.5.2 Notification Declaration ........................................................... 196
11.5.3 OML - Notification Management ................................................ 197
11.6 Exceptions .................................................................................... 199
11.6.1 Exception Definition and Example .............................................. 199
11.6.2 Exception Declaration ............................................................... 200
11.6.3 OML - Exception Management ................................................... 201
11.7 Context ....................................................................................... 202
11.7.1 Context Definition and Example ................................................ 202
11.7.2 Context Declaration .................................................................. 203
11.7.3 OML - Context Management ..................................................... 203
11.8 COOP Architecture ..................................................................... 204
11.8.1 COOP Logical Architecture ....................................................... 204
11.8.2 COOP Physical Architectures ..................................................... 206
11.9 Contribution and Conclusion ......................................................... 207

12 COOM Structural Model and Data Integration .................................. 209
12.1 The COOMS structural model ......................................................... 209
12.2 Abstract Attributes ..................................................................... 210
12.2.1 Motivation for Abstract Attribute Support .................................. 210
12.2.2 Abstract Attribute Definition - Syntax ........................................ 211
12.2.3 Example and OML for Abstract attributes .................................. 212
12.2.4 Unique Attribute-combinations ................................................ 213
12.3 Relations ..................................................................................... 214
12.3.1 Definition and Motivation for Relation-support ............................. 214
12.3.2 Relation Example ...................................................................... 215
12.3.3 Relation Interface Syntax specification ....................................... 217
12.3.4 Relation Interface and Implementation Manager ............................ 217
12.3.5 OML - Relationship Operations ................................................ 221
12.3.6 Relation Interface Functionality .................................................. 225
12.3.7 Containment Relations ............................................................... 226
12.3.8 Sub- and Super- relations ............................................................ 227
12.3.9 OML - Relation Management operations .................................... 228
12.4 Mapping from COOMS to Object-Oriented Languages .................... 230
12.4.1 Mapping from COOMS to C++ ................................................... 230
12.4.2 Mapping from COOMS to Smalltalk ........................................... 231
12.4.3 Relations and System Autonomy ................................................ 231
12.4.4 Relations and Notifications ....................................................... 231
CONTENTS

12.5 Possible extensions to COOM structural model 231
12.6 COOM and Data Integration 232
  12.6.1 Single model data integration by structural abstractions 232
  12.6.2 COOM and Multi-model Data Integration 232
  12.6.3 Schema Integration and semantic interoperability 233
12.7 Contribution and conclusion 238

13 COOM QML - Query Processing 241
  13.1 Motivation for QML 241
  13.2 QML Principles 242
  13.3 QML Constructs and Syntax 244
    13.3.1 Comparisons 245
    13.3.2 Comparison of Objects 245
    13.3.3 Dot Notation 246
    13.3.4 Relations 246
    13.3.5 Multi-valued Relations - Exists and ForAll Semantics 246
    13.3.6 Relation Attributes 247
    13.3.7 Nested Queries 247
    13.3.8 Query Arguments 248
    13.3.9 Other Query Operations 248
  13.4 QML Query Processing 249
  13.5 Issues and Possible Extensions for QML 250
  13.6 Contribution and Conclusion 252

14 COOP Development Methodology and Support Environment 255
  14.1 COORASS - COOP Methodology 255
    14.1.1 Different User-roles 256
    14.1.2 OORASS - COOM_B Modeling Support 256
    14.1.3 From OORASS Role Models to COOM_B ODL 259
    14.1.4 COORASS - OORASS Extended to COOM_S Modeling Support 259
    14.1.5 Basic OORASS Methodology 261
  14.2 COORASS Methodology for the Component-integrator 262
  14.3 COORASS Methodology for the Tool-builder 263
  14.4 CODE - COOP Development Environment 264
  14.5 CODE - Support for the Component-integrator 265
    14.5.1 Schema Reconstructor and Mapper 265
    14.5.2 Client-Generator 266
    14.5.3 Schema Integrator 266
    14.5.4 Component Installer 266
    14.5.5 Component-integrator Support in the different COORASS Steps 266
  14.6 CODE: Support for the Tool-builder 267
    14.6.1 User-Interface Development Tools 267
    14.6.2 Schema Browser Editor 267
    14.6.3 Schema Tester 267
    14.6.4 Integrated Programming Environments (C++ and Smalltalk) 267
    14.6.5 Tool-builder Support in the Different COORASS Steps 268
  14.7 Contribution and Conclusion 268

15 Component Integration and Presentation Integration 269
  15.1 A Scenario for the Use of COOP and COOM 269
  15.2 Component Integration 270
    15.2.1 Manual Schema Reconstruction and Mapping 271
    15.2.2 (Semi)-Automatic Schema Reconstruction and Mapping 271
CONTENTS

15.2.3 Generation of New Implementations .................................. 271
15.2.4 Mapping Principles from and to Different Systems ............... 272
15.2.5 Object Interface Management ....................................... 272
15.2.6 Relation Interface Management ................................... 273
15.3 Object Identity and Autonomy ......................................... 273
15.4 Component Integration for Relational DBMSs .......................... 275
  15.4.1 Experiences with Relational DBMS Integration ..................... 275
  15.4.2 Relational DBMSs and Object Identity ............................ 275
  15.4.3 Schema Reconstruction and Mapping for Relational DBMSs ...... 276
  15.4.4 Object Implementation Management for Relational DBMSs ....... 277
  15.4.5 Object Implementation for Relational DBMSs ..................... 279
  15.4.6 Relation Implementation Management for Relational DBMSs ...... 280
15.5 Component Integration for Object-Oriented DBMSs .................... 281
  15.5.1 Experiences with Object-Oriented DBMS Integration .............. 281
  15.5.2 Object-Oriented DBMSs and Object Identity ..................... 282
  15.5.3 Schema Reconstruction and Mapping for Object-Oriented DBMSs . 282
  15.5.4 Object Implementation Management for Object-Oriented DBMSs . 282
  15.5.5 Object Implementation for Object-Oriented DBMSs ................ 284
  15.5.6 Relation Implementation Management for Object-Oriented DBMSs . 284
15.6 Component Integration for Large-grain Services ....................... 285
  15.6.1 Experiences with Integration of Large-grain Services .......... 286
  15.6.2 Large-grain Services and Object Identity ....................... 286
  15.6.3 Schema Reconstruction and Mapping for Large-grain Services .... 286
  15.6.4 Object Implementation Management for Large-grain Services .... 287
  15.6.5 Object Implementation for Large-grain Services ................ 288
  15.6.6 Relation Implementation Management for Large-grain Services .... 288
  15.6.7 Conclusion and Future Work ..................................... 288
15.7 Component Integration for Fine-grain Services ........................ 288
  15.7.1 Experiences with Integration of Fine-grain Services ............ 289
  15.7.2 Fine-grain Services and Object Identity ....................... 289
  15.7.3 Schema Reconstruction and Mapping for Fine-grain Services ..... 289
  15.7.4 Object Implementation Management for Fine-grain Services ...... 289
  15.7.5 Object Implementation for Fine-grain Services .................. 290
  15.7.6 Relation Implementation Management for Fine-grain Services ...... 290
15.8 Schema Integration and Enhancement in the IntegrationSpace .......... 290
15.9 Tool-builder Support and Presentation Integration .................... 290
  15.9.1 User-interface Specification ................................... 291
  15.9.2 Model and Interface Specification ................................ 292
  15.9.3 Schema Integration and Enhancement ................................ 292
  15.9.4 User-interface Implementation ................................... 292
  15.9.5 Model Implementation ........................................... 293
  15.9.6 Configuration and Run time support ............................. 293
  15.9.7 Contribution and Conclusion .................................... 294
16 Evaluation, Future work and Conclusion ................................ 295
  16.1 Contribution and Evaluation of Part I .............................. 295
  16.2 Contribution and Evaluation of Part II ............................. 296
  16.3 Contribution and Evaluation of Part III ............................ 297
    16.3.1 COOP/COOM and Control Integration ............................. 297
    16.3.2 COOP/COOM and Data Integration ................................ 299
    16.3.3 COOP/COOM and Process Integration ............................. 302
    16.3.4 COOP Architecture ............................................. 302
    16.3.5 COORASS methodology ......................................... 302
16.3.6 CODE specification ........................................ 303
16.3.7 Mapping and Systems Integration ............................ 303
16.3.8 COOP/COOM and Presentation integration ................. 303
16.3.9 COOP and COOM Summary .................................. 304
16.4 Experiences with COOP-related Prototypes .............. 306
16.5 Comparison with other related work ....................... 308
16.5.1 Eureka Software Factory ................................ 309
16.5.2 COOM related to the REBOOT Datamodel ............... 309
16.5.3 OMG .................................................. 309
16.5.4 Other work ............................................ 310
16.6 Problem-areas and issues for future work ............. 311
16.7 Conclusion ................................................. 312

IV Appendices 315

A COOM ODL 317
A.1 The Object Definition Language .................. 317
A.2 Lexical Conventions ..................................... 317
  A.2.1 Tokens ............................................ 317
  A.2.2 Comments ......................................... 317
  A.2.3 Identifiers ........................................ 318
  A.2.4 Keywords ......................................... 318
  A.2.5 Literals ........................................... 318
  A.2.6 Preprocessing ...................................... 320
A.3 Basic Types .............................................. 320
  A.3.1 Simple Types ....................................... 320
  A.3.2 Constructed Types ................................ 321
  A.3.3 Type Names ......................................... 321
  A.3.4 Constant Declaration .............................. 324
A.4 ODL Grammar ............................................. 325
A.5 ODL Example .............................................. 332

B COOM OML - Object Manipulation Language 335
B.1 Object Interfaces ...................................... 335
  B.1.1 ObjInt ............................................... 335
  B.1.2 METAObjInt ........................................ 336
  B.1.3 M(OBJINTERFACE T) ................................ 336
  B.1.4 M1(Type T) .......................................... 337
B.2 Relations ................................................ 337
  B.2.1 Basic definitions ................................... 337
  B.2.2 ROI .................................................. 337
  B.2.3 RI(F, T, A) .......................................... 338
  B.2.4 R(F, T, A) ........................................... 338
  B.2.5 RIT(F, T, A) ......................................... 339
B.3 Interfaces for Support services ...................... 340
  B.3.1 Set(Type T) .......................................... 340
  B.3.2 List(Type T) ......................................... 340
  B.3.3 Object Space ......................................... 340
  B.3.4 Notification Management Services ........... 341
B.4 OML C++ Example ...................................... 341
  B.4.1 Interface Classes - Framework predefined ... 342
CONTENTS

B.4.2 Interface Classes - Generated by ODL Compiler .......... 342
B.4.3 Implementation Classes - Generated by a Component Integrator .. 344
B.4.4 Tool/Application - Created by a Tool Builder ............. 345

C OQL Grammar ......... 349
C.1 OQL Grammar .......... 349
C.2 OQL Examples .......... 350
# List of Figures

1.1 Unified Integration through Distributed Persistent Objects .......................... 3
1.2 Thesis structure ...................................................................................... 4
2.1 Cooperative Work Environment for "Upstream" Offshore ......................... 16
2.2 Environment User Perspective ............................................................... 18
2.3 User Task Support - Uniform access to Tools and Information Resources ... 19
2.4 Cooperative Work Support .................................................................. 21
2.5 Work Process Support .......................................................................... 22
2.6 The "Toaster" Model ............................................................................ 24
2.7 Environment Layers - "The ESF Ladder" ................................................. 26
2.8 ESF Architectural Reference Model ....................................................... 28
2.9 From ESF Conceptual Model to ESF Architectural Model ..................... 29
2.10 The "Double" Toaster-model ................................................................. 30
2.11 Double Toaster-model refined from ESF architectural model ................. 31
3.1 Control integration - Request-oriented ................................................... 35
3.2 Control integration - Notification-oriented .............................................. 38
3.3 Single-model data integration ............................................................... 40
3.4 Multi-model data integration .................................................................. 44
3.5 Presentation Integration ........................................................................ 49
4.1 Evolution of Integration Framework Architectures .................................... 56
4.2 Taxonomy of solution-approaches for multi-model data integration .......... 58
4.3 Object Management Architecture Reference Model ................................ 63
4.4 CORBA - Common Object Request Broker Architecture ..................... 64
4.5 HP/Softbench and Field Architecture .................................................... 68
4.6 PCTE Architecture .............................................................................. 71
4.7 Pegasus Architecture .......................................................................... 76
4.8 ESF Reference Architecture ................................................................. 78
5.1 Interface(Type) - Implementation(Class) and Subset(Category) ............... 99
6.1 SITE Architecture .................................................................................. 106
6.2 The structure of the SITE update mechanism ......................................... 107
6.3 CODEDISC components - using the architecture on itself ..................... 110
6.4 Generation of C++-class from NIDL-based component description .......... 111
7.1 Multi-layered model .............................................................................. 117
7.2 Application-Working Store-Data Store Interface ................................... 119
7.3 The HyperModel schema in COORASS ............................................... 122
7.4 OODBMS vs RELDBMS performance on a Closure Traversal-operation ..... 124
7.5 The programmer's responsibility and the database system's approaches .... 128
7.6 Server-oriented (Extended DB) or Client-oriented (Extended PL) OODBMSs 129
8.1 Project Manager's and Programmer's User Interaction Components .......... 136
8.2 CODEDISC Component Browser and Tester .......................................... 137
9.1 Object-Oriented Approaches to Integration ............................................. 144
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Principal COOP Architecture</td>
<td>145</td>
</tr>
<tr>
<td>9.3</td>
<td>COOP related to the Double Toaster model</td>
<td>146</td>
</tr>
<tr>
<td>9.4</td>
<td>COOP Architecture related to ESF Architecture</td>
<td>147</td>
</tr>
<tr>
<td>9.5</td>
<td>COOP Architecture Development-time / Run-time</td>
<td>149</td>
</tr>
<tr>
<td>9.6</td>
<td>Type-Hierarchies for Interface - Implementation and Extents</td>
<td>151</td>
</tr>
<tr>
<td>9.7</td>
<td>Object-hierarchy with manager objects</td>
<td>152</td>
</tr>
<tr>
<td>10.1</td>
<td>COOM Layers</td>
<td>157</td>
</tr>
<tr>
<td>10.2</td>
<td>Meta-model for $COOM_F$</td>
<td>158</td>
</tr>
<tr>
<td>10.3</td>
<td>COOM Basic Types</td>
<td>159</td>
</tr>
<tr>
<td>10.4</td>
<td>Person and Employee interfaces</td>
<td>162</td>
</tr>
<tr>
<td>10.5</td>
<td>Multiple inheritance and Conformance</td>
<td>166</td>
</tr>
<tr>
<td>10.6</td>
<td>Implementations conform to interfaces</td>
<td>170</td>
</tr>
<tr>
<td>10.7</td>
<td>Different Object Spaces</td>
<td>171</td>
</tr>
<tr>
<td>11.1</td>
<td>COOM Layers</td>
<td>173</td>
</tr>
<tr>
<td>11.2</td>
<td>Type-and MetaType-hierarchy with operations for Person and Employee interfaces, and corresponding manager objects</td>
<td>175</td>
</tr>
<tr>
<td>11.3</td>
<td>Type-hierarchy for Interfaces and Implementations</td>
<td>180</td>
</tr>
<tr>
<td>11.4</td>
<td>Object-hierarchy with types and objects for different Object spaces</td>
<td>181</td>
</tr>
<tr>
<td>11.5</td>
<td>Object-hierarchy with named objects</td>
<td>182</td>
</tr>
<tr>
<td>11.6</td>
<td>Enhanced view of the metaPerson interface manager</td>
<td>183</td>
</tr>
<tr>
<td>11.7</td>
<td>Enhanced view of the metaEmployeeIngres implementation manager</td>
<td>185</td>
</tr>
<tr>
<td>11.8</td>
<td>Local and Global Notification Managers</td>
<td>198</td>
</tr>
<tr>
<td>11.9</td>
<td>Exception Management with embedded SQL</td>
<td>201</td>
</tr>
<tr>
<td>11.10</td>
<td>Context Management</td>
<td>203</td>
</tr>
<tr>
<td>11.11</td>
<td>COOP Architecture</td>
<td>204</td>
</tr>
<tr>
<td>11.12</td>
<td>Different Physical COOP Architectures</td>
<td>206</td>
</tr>
<tr>
<td>12.1</td>
<td>COOM for Data Integration</td>
<td>209</td>
</tr>
<tr>
<td>12.2</td>
<td>COORASS Person-Department example</td>
<td>215</td>
</tr>
<tr>
<td>12.3</td>
<td>Type-hierarchy with operations for Relation interfaces</td>
<td>216</td>
</tr>
<tr>
<td>12.4</td>
<td>Type-hierarchy for Relation Interfaces and Relation Implementations</td>
<td>219</td>
</tr>
<tr>
<td>12.5</td>
<td>Relation interface managers and Relation implementation managers</td>
<td>220</td>
</tr>
<tr>
<td>12.6</td>
<td>Relation Iterator - Interface and Implementations</td>
<td>221</td>
</tr>
<tr>
<td>13.1</td>
<td>Query Processing - OQL Compiler and Interpreter</td>
<td>244</td>
</tr>
<tr>
<td>13.2</td>
<td>Query Processing - Different steps</td>
<td>249</td>
</tr>
<tr>
<td>13.3</td>
<td>Query Processing - Interface and Implementation Managers (Objects and Relations)</td>
<td>250</td>
</tr>
<tr>
<td>14.1</td>
<td>OORASS Role Diagrams</td>
<td>257</td>
</tr>
<tr>
<td>14.2</td>
<td>Synthesis of OORASS Role Diagrams</td>
<td>258</td>
</tr>
<tr>
<td>14.3</td>
<td>COORASS diagram symbols</td>
<td>259</td>
</tr>
<tr>
<td>14.4</td>
<td>COORASS</td>
<td>260</td>
</tr>
<tr>
<td>14.5</td>
<td>The basic OORASS methodology</td>
<td>261</td>
</tr>
<tr>
<td>14.6</td>
<td>COORASS methodology for the Component-integrator</td>
<td>262</td>
</tr>
<tr>
<td>14.7</td>
<td>COORASS methodology for the Tool-builder</td>
<td>263</td>
</tr>
<tr>
<td>14.8</td>
<td>COOP Development Environment</td>
<td>265</td>
</tr>
<tr>
<td>15.1</td>
<td>Schema Reconstruction and Mapping</td>
<td>270</td>
</tr>
<tr>
<td>15.2</td>
<td>Object Dictionary and local object spaces</td>
<td>272</td>
</tr>
<tr>
<td>15.3</td>
<td>Integration of a Relational DBMS</td>
<td>276</td>
</tr>
<tr>
<td>15.4</td>
<td>Integration of an Object-Oriented DBMS</td>
<td>283</td>
</tr>
<tr>
<td>15.5</td>
<td>Integration of a functional service component</td>
<td>287</td>
</tr>
<tr>
<td>15.6</td>
<td>Integration of a fine-grained functional service component</td>
<td>289</td>
</tr>
</tbody>
</table>
15.7 The user interface of the example tool ........................................... 291
15.8 User Interface - Model interaction ......................................................... 292
16.1 COOP Prototypes ................................................................. 307
# List of Tables

3.1 Requirement Areas for Request-oriented Control Integration ............... 35  
3.2 Requirement Areas for Notification-oriented Control integration .......... 39  
3.3 Requirement areas for Data Integration - Single Model ..................... 40  
3.4 Requirement Areas for Data Integration - Multi Model ...................... 45  
3.5 Requirement areas for Presentation integration ............................. 50  
4.1 User-interface tools in different areas ...................................... 59  
4.2 Integration-areas supported by the systems ................................. 86  
4.3 Evaluation for Request-oriented control-integration ....................... 86  
4.4 Evaluation for Notification-oriented Control-integration ................. 87  
4.5 Evaluation for Single-model Data-integration ............................. 87  
4.6 Evaluation for Multi model Data-integration .............................. 87  
4.7 Evaluation for Presentation integration ..................................... 88  
6.1 Evaluation for Request-oriented control-integration ...................... 113  
6.2 Evaluation for Notification-oriented Control-integration ............... 113  
7.1 Evaluation for Single-model Data integration ............................. 125  
8.1 Evaluation for Presentation Integration .................................... 138  
13.1 OQL - Object-Preserving/Generating at Compile/Run-time ............... 243  
16.1 Evaluation of COOP/COOM for Request-oriented Control-integration .... 298  
16.2 Evaluation of COOP/COOM for Notification-oriented Control-integration 299  
16.3 Evaluation of COOP/COOM for Single-model Data-integration .......... 300  
16.4 Evaluation of COOP/COOM for Multi-model Data integration .......... 301  
16.5 Evaluation of COOP/COOM for Presentation integration ................ 303  
A.1 Keywords in ODL ................................................................. 318  
A.2 Predefined escape sequence .................................................. 319  
A.3 Integer size ........................................................................... 321  
A.4 Case label matching ............................................................... 324  
A.5 Constant value ........................................................................ 325
Chapter 1

Introduction

This thesis presents an object-oriented framework for systems integration and interoperability. It contains three major parts. Part I presents a taxonomy of problem areas and solution approaches for systems integration and interoperability. It contains a description of requirement areas and solution approaches for the three focused integration areas: control integration, data integration and presentation integration, and a discussion of the requirements for architectural support for process integration. Part II presents experiments with object-oriented approaches to the three integration areas. Part III presents the object-oriented COOP Integration Framework and the COOM object model, which takes a unified object-oriented approach to the three integration areas.

This chapter introduces the thesis by an introduction to the problems of systems integration and interoperability, the motivation and vision for this thesis, the structure and contribution of this thesis, and the history and context for this thesis. The last section in this chapter is a guide for how to read this thesis.

1.1 The Problems of Systems Integration and Interoperability

Computer usage is expanding into all functions of large organizations. Many future information-systems will support cooperative and individual task-oriented work in organizations. Such environments will consist of a large network of heterogeneous, autonomous and distributed computing and information resources, on a wide variety of different computers, and a wide variety of different software systems.

As more and more functions in an organization are supported by computer systems, there is an increasing need to integrate these systems with each other. Systems Integration and interoperability is thus becoming a major issue for the next generation of cooperative work environments. It is possible to handcraft integration solutions for particular situations, but the challenge is to create methods, formalisms and tools that makes the creation of integrated systems an easier task. The goal of this work is to provide a basis for systems integration in cooperative work environments, with particular emphasis on tool integration and interoperation.

Systems integration can be viewed both from an environment user perspective and from an environment builder perspective. The environment user is concerned with the perceived integration at the environment's interface, while the environment builder is concerned with the feasibility and effort needed to achieve this perceived integration. The environment user perspective is further described in section 2.3. The environment builder perspective
1.2 The Motivation for this Thesis

is further described in section 2.4.

A taxonomy for problems and solutions for systems integration and interoperability has been developed, based on a refinement of the ECMA/NIST Toaster model [NS91] with concepts from the Eureka Software Factory architectural model [Eur89, GHH+92]. This has resulted in the Double Toaster model that distinguishes between request-oriented and notification-oriented control integration for the interaction between tools, single model and multi-model data integration for the sharing of data between tools, display-oriented and model-oriented presentation integration for the creation of user interfaces, and interaction-oriented and interworking-oriented process integration for the support of work processes. The Double Toaster model is described in section 2.4.3.

Systems integration and interoperability is a complex problem-area where solution approaches often need to make compromises for conflicting requirements. A solution approach will often have to decide on a trade off between conflicting requirements such as heterogeneity versus homogeneity, autonomy versus common policy, distribution versus centralization, fine-grain versus large-grain components, efficiency versus flexibility and extensibility, and so on. These conflict areas are presented in section 3.1, and are discussed for the different solution approaches. The conflicting requirements above will be discussed with respect to the solution approaches in chapter 4, and with respect to COOP and COOM in chapter 9 and chapter 16.

Part I of this thesis presents requirement areas and existing solution approaches in more detail. Chapter 2 sets systems integration and interoperability in a context, by description of reference models for integration, while chapter 3 presents more detailed definitions and related requirement areas. Chapter 4 discusses various solution approaches.

1.2 The Motivation for this Thesis

The motivation for doing this work came from experiences with work on systems integration in the ESF\textsuperscript{1} project, in the years 1988 to 1990. Involvement in three different sub-projects that each worked independently on one integration-area lead to the belief that it was possible to take a unified approach to these integration-areas. The three projects were ESF Software Bus (SWB) on control integration, ESF Object Storage Services (OSS) on data integration and ESF User Interaction Services (AVIS/BUIC) on presentation integration.

None of these projects were using an object-oriented approach, but earlier and parallel work with object-oriented technology in different projects at SI\textsuperscript{2} and OGP\textsuperscript{3}, suggested that object-oriented technology was a good candidate for a unified approach.

The belief that object-oriented technology could be used as the unifying technology for different integration areas, has been the motivation for carrying out the research for this thesis.

Part II of this thesis presents experiments with object-oriented approaches to systems integration that shows that an object-oriented approach is better than a traditional approach in the integration areas control integration, data integration and presentation integration.

The goal of this thesis is to supply technology that make it easier to cope with the problems of integration and interoperability in heterogeneous systems, by providing an object-oriented framework for systems integration and interoperability.

\textsuperscript{1}Eureka Software Factory  
\textsuperscript{2}Center for Industrial Research, SINTEF SI from 1. January 1993  
\textsuperscript{3}Oregon Graduate Institute
1.3 The Vision for this Thesis

Part III presents the COOP Integration Framework, an object-oriented framework for systems integration and interoperability. The COOP Vision is to unify control integration, data integration and presentation integration through the notion of "Distributed Persistent Objects".

In the ESF-project [Fer91] all these areas were addressed in separate sub-projects, with quite different approaches. COOP shows that it is possible to take an object-oriented approach in each of these areas, and gain a conceptual simplicity and higher productivity as a result.

![Figure 1.1: Unified Integration through Distributed Persistent Objects](image.png)

The conceptual model for the application-programmer is to have a world of distributed persistent objects. It should be transparent that objects might represent information and functionality in underlying connected systems. Figure 1.1 shows the pool of distributed persistent objects which represents the totality of functionality and data available as objects. This gives a uniform support for both large-grained and fine-grained objects. The figure illustrates the view that both components and items within the components might be viewed as objects.

COOP can be viewed as a distributed heterogeneous object management system with support for heterogeneous implementations for objects. The notion of distributed persistent objects represents an integration of concepts from three different technology-areas: Distributed Systems, Database Systems and Object-oriented Systems. The integration of these three technologies is done through COOM, COOP Object-Oriented Model, in a way that gives unified support to the three integration areas: control integration, data integration and presentation integration.
Figure 1.2: Thesis structure
1.4 The Structure of this Thesis

The goals of the research have been:

1. To identify different systems integration areas and their requirements for solution.

2. To verify that an object-oriented approach is promising, through experiments in each individual integration area.

3. To develop an integration framework which is able to meet the requirements for solution from each integration-area, by taking a unified object-oriented approach.

The contents of this thesis is thus divided into the following three parts:

- Part I: Systems Integration - A Taxonomy of Problems and Solutions
- Part II: Experiments with Object-Oriented Approaches to Systems Integration
- Part III: COOP: A Unified Object-Oriented Approach to Systems Integration

An introduction-chapter is given before part I and a conclusion-chapter is given after part III. The structure of the thesis is illustrated in figure 1.2.

1.4.1 Part I: Systems Integration - A Taxonomy of Problems and Solutions

Part I gives an introduction to the problems of systems integration, and identifies three integration areas with sub areas and corresponding requirements. Different systems integration architectures are evaluated with respect to the requirements.

Chapter 2 gives an introduction to the problem of systems integration seen both from an environment user perspective, and from an environment builder perspective. Chapter 3 presents requirements for the three identified integration-areas: data integration (single and multi model), control integration (request-oriented and notification-oriented) and presentation integration (display-oriented and model-oriented). Chapter 4 discusses a number of solution approaches with respect to support for the requirement areas.

1.4.2 Part II: Object-Oriented Approaches to Systems Integration

Part II presents experiments from using an object-oriented approach in the three identified areas of integration. The hypothesis that object-oriented technology is a better approach for systems integration has been validated through these individual experiments.

Chapter 5 gives a general introduction to object-oriented concepts and terminology. Chapter 6 reports on experiences from experiments with an object-oriented approach for control integration. This is based on a prototype of an object-oriented "Software Bus" interaction-mechanism that facilitates both request-oriented interaction and notification-oriented interaction. Chapter 7 presents experiences from experiments with object-oriented data integration. Object-Oriented database integration is shown through the development and experiences from the object-oriented model SOOM, and the "HyperModel" benchmark for database-system-evaluation. Chapter 8 reports on experiences from experiments with Object-Oriented presentation integration, based on the use of object-oriented user-interface application-frameworks.
1.5 The Contributions of this Thesis

1.4.3 Part III: A Unified Object-Oriented Approach to Systems Integration

Part III describes the COOP Integration framework. COOP unifies the three different integration-areas: control integration (request-oriented and notification-oriented), data integration (single model and multi-model), and presentation integration (display-oriented and model-oriented) through COOM, the COOP Object Model.

Chapter 9 shows how the results from Part I and Part II are used as a basis for the COOP Integration Framework, presents the goals and high-level architecture for COOP, and describes the basic principles and concepts for COOM. Chapter 10 presents the COOM foundation model, $COOM_F$, which is used as a basis for the COOM behavioural model, $COOM_B$, and the COOM structural model, $COOM_S$. Chapter 11 presents $COOM_B$ and its relation to control integration. Each of the basic constructs are defined, with its corresponding ODL (Object Definition Language) syntax and semantics and mapping to OML (Object Manipulation Language). Chapter 12 presents $COOM_S$ and its relation to data integration. Attributes and relations are defined as a layer on top of $COOM_F$ and $COOM_B$. It is shown how COOM provides a foundation for further work on multi-model data integration. Chapter 13 describes query processing through OQL (Object Query Language). Chapter 14 presents the COOP Development methodology, COORASS, and gives a description of a possible COOP-compliant development environment. Chapter 15 illustrates the possible use of COOP and COOM in component integration and in tool building. Tool building also illustrates the support for presentation integration.

The appendices A, B and C present definitions for the COOM sub-languages: the ODL grammar (A), the OML interface-definitions (B) and the OQL grammer(C).

1.4.4 Conclusion

Chapter 16 describes the contributions of the three parts. The results from part III are evaluated with respect to the requirements from part I, and then compared to related work. Finally areas for future work are identified.

1.5 The Contributions of this Thesis

The main contribution of this thesis is the COOP Integration framework presented in part III. The following presents more specific the contributions in part III.

1. An Object-Oriented Integration Framework that unifies control, data and presentation integration

The COOP integration framework unifies control integration, data integration and presentation integration through the COOP Object Model, COOM. COOM is a fully object-oriented model that supports control integration through encapsulation and event-notification, supports data integration through structural abstractions for attributes and relations, and supports presentation integration through explicit separation of user-interface objects and functional objects. Transparent access to underlying existing systems is supported by the explicit separation between interface and implementation for objects. Multiple implementations can be handled simultaneously. The contribution of the COOP architecture is the notion of a distributed persistent object space, handled through the integration framework. The architecture provides a basis for a transparent handling of both large-grained and fine-grained
1.5 The Contributions of this Thesis

objects. The COOP architecture support the use of COOM in a distributed environment, with tools for mappings to existing object-oriented programming languages, such as C++ and Smalltalk. COOP and COOM are supported by the COORASS methodology.

2. An Abstract Structural model realized through a Behavioural model
An important feature in COOM and COORASS is the extension of a pure object-oriented behavioural model with abstract structures, based on ER-oriented constructs. This is different from the commonly found approach of extending a structural model with object-oriented concepts. The motivation for extending from an object-oriented model is that an object-oriented model with operations provide an encapsulation that facilitates for multiple implementations, and hiding of underlying mappings and caching-strategies. Approaches that uses an ER-oriented basis normally put less emphasis on encapsulation, and tend to think about physical attributes and relationships, instead of abstract structures. Approaches that uses a pure object-oriented basis normally gives only limited support for relations and attributes, and often then breaks encapsulation. Note that the important point with respect to systems integration is full encapsulation-support. This goal can also be achieved by layering a behavioural model on top of a structural model, if all attributes and relations are encapsulated and reflected in the operational interface.

3. Support for both large-grained and fine-grained objects, through a distributed object space
The distributed object space provide for fine-grained objects which also can be used as surrogates for large-grain components. The contribution of COOP/COOM with respect to mappings and systems integration is to show that it is possible to map to and from different underlying systems with different granularity. The discussion on object identity shows that the use of separate object implementation managers facilitates for different local identity-schemes in different underlying systems.

4. The introduction of separate interface and implementation managers
The use of object-space specific interface and implementation managers contributes to an integration framework where it is easy to add new implementations. COOM introduced the novel use of object-spaces for the management of the mapping from interfaces to implementations.

5. Support for multiple implementations of a relation interface
The separation of relation interface and implementation with support for multiple implementations of a relation interface is novel in COOM. The dual support for both role-name access and relation-access is new in COOM. Role-name access provides the same encapsulation of actual implementation as a relation interface manager.

6. Notifications as part of the schema description.
A novel contribution in COOM is the specification of notification-types as a part of the schema, and the use of inheritance between notification-types.

7. Attribute specifications with notifications, exceptions and unique-specification
The possibility for the association of exceptions and notifications with attributespecifications is novel. The possibility for the specification of unique attribute-combinations in an interface is also new.

8. Query Processing with multiple implementations
The main contribution of OQL and OQL query processing, is to show how it is possible to realize a query processing facility as part of the integration framework. The contribution of OQL is to show a basis for object-based query processing in
1.5 The Contributions of this Thesis

an environment supporting multiple parallel implementations of object and relation interfaces. The novel contribution in this work is a framework with multiple parallel implementations for both object and relation interfaces, and a query processing strategy that utilizes this.

9. Foundation for multi-model data integration
The main focus has been to provide a foundation-layer for later work on multi-model data integration. The COOM model is a suitable canonical data model, and the separation of interface and multiple implementations provides a good basis for schema mapping. The current work has not aimed to resolve all problems related to multi-model data integration, but instead to develop a framework where such problems can be investigated further.

10. Methodology for Integrated Behavioural and Structural modeling
The contribution of the COORASS methodology is modeling-concepts for the basic behavioural-oriented model $COOM_B$ in the basic OORASS layer, and enhancements for the structural-oriented model $COOM_S$ in the additional COORASS-layer. COORASS follows COOM in extending a pure object-oriented behavioural model with structural ER-oriented constructs.

Part I and Part II provide the basis for part III, and the following describes the contributions from these parts:

1. Double Toaster Model with requirement areas
The "Double" Toaster reference model developed in part I gave a basis for understanding of the requirements, in the three focused integration-areas, control-, data- and presentation-integration. The "Double" Toaster model is a good basis for evaluation and understanding of different proposals for integration frameworks and systems integration architectures.

2. Experiences with an Object-Oriented Software Bus
The contribution of these results is a validation of the hypothesis that object-oriented technology is a suitable tool for facilitating systems integration, and a summary of experiences related to the different areas of integration. Object-Oriented control integration was shown through an object-oriented "Software Bus", in the SITE and CODEDISC prototypes.

3. Combined Structural and Behavioural model in SOOM
Object-Oriented data integration was shown through the development and experiences from the object-oriented model SOOM. In particular it was shown that a structural model with relationships can be combined with a behavioural model with operations.

4. HyperModel Benchmark for OODBMS evaluation
The "HyperModel" benchmark for was developed to check if object-oriented database-systems did provide the performance and functionality required for data integration in integrated environments. The benchmarking showed that the first generation of object-oriented database-systems did not provide the required performance for data integration, while the second generation improved on this and was then selected as a basis for further work.

5. Object-Oriented Presentation Integration for Distributed Objects
Object-Oriented presentation integration was shown through a separation of the user-interface-part and the functional part of an application, and by the use of object-oriented class-libraries for user-interfaces. It was shown that this provides for an
extra flexibility in distributed systems, not automatically provided by client/server separation as in X-Windows.

The following papers have been published as results of this work: [Ber88c, B+89d, ABM+90, BA91, Ber91c, Ber92c, Ber92b, Ber92a]

The contributions are discussed in more detail in chapter 16.

1.6 History and Context of this Thesis

This section presents parts of the history and context for the work done in this thesis. The work has been done as a continuation to work on object-oriented cooperative work environments in the EKI and ESF projects.

My initial work in Object-Oriented technology and Object-Oriented databases was done in the EKI-project (Effectivity and Quality in Engineering) in the years 1984 to 1987 at SI. From August 1987 to August 1988 I was a visiting scientist during the "Database Year" at Oregon Graduate Institute. From 1988 to 1990 I worked again at SI as a research scientist, mainly on the ESF project. From September 1990 to May 1992 I was at NTH as a research fellow, doing the main research part for this thesis. From May 1992 and onwards, I have been back at SI as senior research scientist and head of section for the department for Cooperative Information Systems. The final writing of this thesis has been done part-time in this period.

The EKI-project was a stimulating and creative forum for both research and development. The leadership of Eirik Næss-Ulseth and Trygve Reenskaug was instrumental in creating a good environment, and the project members Else Nordhagen, Pål Stenslet, Anton Landmark, Gro Oftedal, Anne Hurlen, Anne Lise Skaar, Tor Bothner, Henrik Lund Hansen, Carl Petter Swensson, Magnus Rygh, Anders Bremer and Odd Arild Lehne all contributed to making this to memorable and valuable years of my working life.

The EKI-group later became department for Cooperative Information Systems, and gained new members with knowledge in systems integration and databases, such as Erik G. Nilsson, Svein Johnsen and Bjørn Skjellang. The working group on Systems Integration for Heterogeneous Systems, with among others Svein Arne Solbakk, Gisle Stokke and Bjørn Skjellang gave an opportunity to discuss various aspects of systems integration. The work in this area later done by Erik G. Nilsson, Else Nordhagen and Gro Oftedal has also provided valuable input.

Professor David Maier at OGI\(^4\), presented me with the opportunity to participate in the arrangements and activities in the "Database Year" at OGI, Oregon, USA, during the academic year 1987-1988. We met first at the "First International Workshop on Object-Oriented Databases" in Monterey in September 1986, then at OOPSLA'86 in Portland in October 1986. At that time Portland was viewed as the object-oriented "capitol" of the world, because of the large Smalltalk-activity at Tektronix, and other object-oriented activity such as the Gemstone OODB at Servio Logic. I also had the opportunity to work with evaluation of Object-Oriented databases at Tektronix Computer Research Laboratory with Lougie Anderson and others in the database group, in particular with Harry Porter with whom I also shared an office at OGI. Goetz Graefe provided insight into relational based query processing and query optimization. Alan Purdy gave me the possibility to spend some time at Xerox PARC/NW, and to discuss issues related to object-oriented modeling, cooperative work and transactions. Jacob Stein at Servio Logic and Timothy

\(^4\)Oregon Graduate Institute
Andrews and Joshua Duhi at Ontologic, the two object-oriented database vendors at that
time, supported me in evaluating their products Gemstone and Vbase, and in discussing
object-oriented databases in general. Rick Cattell from Sun contributed with thoughts and
ideas on database-system benchmarking, based on his work on the "Engineering database
benchmark" which we extended in the "HyperModel benchmark". The "Database year"
had a number of short-time visitors and I have later enjoyed contact with some of them,
in particular Tom Atwood, the founder of Object Design, and Francois Banchilhon, the
founder of O2 Technologies.

The European research and development project, Eureka Software Factory (ESF) has been
an international setting for discussing ideas related to integrated environments and systems
In particular the technical design group of ESF with among others, Christer Fernström,
Lennart Ohlsson, Bernhard Lang, Herbert Weber, Wilhelm Schäfer and Trygve Reenskaug,
provided a good foundation for further work. Through participation in the early ESF
architecture definition, and as technical authority in the two subprojects ESF Software
Bus and ESF Object Storage Services, I have had an opportunity to work with a number
of people eager to share experiences and ideas. The Software Bus team: Anton Landmark,
Thierry Bingen, Lee Morgan, Malcolm Verral with Ray Foulkes as an excellent leader. The
OSS subproject with Gebhard Greiter, Helmut Abbenhardt and some short-term members,
such as Klaus Dittrich.

The students I have been adviser for through the years have been valuable discussion-
partners. A number of projects and master theses at NTH have provided a context for
trying out and getting experiences with ideas and concepts developed in the research for
this thesis. The project students in the CODEDISC-project at NTH in fall of 1990, and the
SEROO-project at NTH in fall of 1991, provided an ideal setting for discussing and imple-
menting some of the COO and COOM ideas. Roy Tingstad developed a mapping from
the REBOOT-compliant part of COOM to the ObjectStore OODBMS in MSc-work in the
fall of 1991, based on my initial COOM-compiler. Petter Lowzow and Per Solberg worked
further on the implementation of a COOM ODL compiler in the CoCom-project. Viger
Östensen worked on a mapping from COOM to Smalltalk-80 in the STONE-project. Tor
Stenvaag did work on the OQL Object Query Language in the project "Object-Oriented
Distributed Query Processing". All these three projects took place at NTH in the spring

In continuing MSc-thesis work in the fall of 1992 Per Solberg developed a generic and
a framework specific C++ backend for the CoCom ODL compiler, and extended the
COOM model and framework to support N-ary Relations. Petter Lowzow developed a
SQL-inspired version of the Object Query Language, together with investigations into
mappings between COOM ODL and relational database systems. Viger Østensen de-
developed two Smalltalk backends for the CoCom ODL compiler, one generic Smalltalk backend
and one OMG Object Request Broker backend, together with investigations into COOM
Smalltalk mappings to relational and object oriented database systems. This resulted in
the following MSc-theses: "Object-Oriented Interface for a Relational Database System",
by Petter Lowzow, NTH 1992, "C++ Framework-support and N-ary Relations for the
COOM Data Model", by Per Solberg, NTH 1992, and "Smalltalk Interface for COOM",

Other master-thesis works I have been adviser for have included "Abstraction in Databases"
by Tor Øystein Molnes, NTH 1987, "Hypertext and Object Oriented Databases" by
Shadana Shenoy, OGI 1988, "FOOD - Fully Object Oriented Database" by Erik Odberg
and Svein Erik Bratsberg, NTH 1989 and 1990, and "Evaluation-procedure for OODBMSs"
by Arne Bodin Larsen, UIO 1992.
1.6 History and Context of this Thesis

Erik Odborg and Svein Erik Bratsberg have later also become fellow dr.ing-students at NTH, together with other dr.ing-students Øystein Nytra, Even A. Karlsson, Bjørn Gulla, Per Harald Westby, Jens Otto Larsen, Håvard Tveite, Øystein Thorbjørnsen and Svein Olaf Hvasshovd. We have had many stimulating discussions, and other activities such as "cake"-eating, dinners, orienteering and the yearly meeting in "Munkholmens lægeforening".

In fall of 1991 and spring of 1992 we had a team of people evaluating different aspects of object-oriented database systems: Ivar Mølnvik "OODBSs vs RELDBMs for a Document Management System", Carl Fredrik Sørensen and Margareta Neumann "OODBSs and Geographical Information Systems", Frode Stortiseth "Object-Oriented Database Systems and Product Models". These have been guided together with Guttorm Sindre and Stein Inge Dale.

The participants at my courses "CSE509 Object Oriented Programming" at OGI in spring 1988, and the EEU-course: "Object Oriented System Development" at NTH in spring 1991, have been active and provoking groups for stimulating discussions and experiments on the use of object oriented technology in practice.

The FOORUM discussion group at UiO\(^5\), has been an interesting forum for discussion of ideas on object-oriented technology. Ragnar Normann has been the leader of this group in Oslo, and together with Geir Skytolstad I chaired a similar Trondheim-based group in fall of 1990 and spring of 1991.

Frank Eliassen, University of Tromsø, has shared thoughts and ideas on integration frameworks, with basis in his work on a functional object-oriented model for this. Frank presented the COOM datamodel at the International Workshop on Distributed Object Management, in Edmonton in 1992, when I was unable to attend.

The Object Management Group, OMG, started in 1990 an effort to create an object oriented architecture and framework for systems integration, following some of the same visions as we had for the ESF architecture. My contacts with technical director dr. Richard Mark Soley at OOPSLA'90 led to further participation in the Object Request Broker, Object Model and Object Services Task Forces, where I again met "old" object-oriented database people like Tom Atwood, Bill Kent, Peter Lyngbaek and Rick Cattell.

Professor David Maier visited Norway in December of 1990, and together we gave a seminar on Object-Oriented databases. Professor Stanley Zdonik visited Norway in October of 1991 and gave talks on Object-Oriented databases. He also used time to discuss various issues related to systems integration and object-oriented technology.

The EIDER and ORBIT Esprit III proposal work in the area of "Interoperable Object Oriented Database Systems" done in fall 1991 and spring 1993, provided a good forum for discussions on issues related to integration of heterogeneous systems. Even if these proposals were not successful we had an interesting time creating the proposal. Yaron Shavit, CIG-Paris, was the leader of this activity. Some of the other participants I worked with were Jane Grimson, TCD, Chris Horn, IONA, Frode Aschim and Erik Amundrud, Sydco, Mads Nygård and Oddvar Risnes, SINTEF DELAB, and Bjørn Skjellaug, SINTEF SI.

The work with the IDIS (Integrated Distributed Information System) pre-project from April 1992 to December 1992 gave a good opportunity to be in contact with end-users, and understand their requirements for tools and methods for integrated distributed systems. This work was done together with Øystein Strandli and Bjørn Skjellaug from SI, Bo Kähler and Mads Nygård from SINTEF DELAB, and professor Arne Solvberg from NTH.

\(^5\) UiO - University of Oslo
The research for this thesis has benefited from industrial interest from companies in the NSR Consortium\(^6\), in the context of the REBOOT Esprit-project. Valuable comments and feedback have been given from Frank Lilleshagen (Metis), Odd Arild Lehne and Pål Stenslet (all Taskon), and Jon Skandsen, Lars Ivar Næss, Erik Amundrud and Frode Aschim (all Sysdeco).

Many of the problem-areas and issues raised in this thesis need further investigation and research. The COOP integration framework also serves as a conceptual framework for the coordination of further research. In particular this is valid for multi-model data integration, process integration and object-oriented query processing. I am adviser and co-adviser for a number of students who started up research work in these areas in the fall of 1992: Espen Frimann Koren and Steinar Kindingstad at UIO have started MSc work on "Schema Integration and Semantic Interoperability". Geir Kjaerstad at UIO has started MSc work on "Data Model Mappings and Schema Reconstruction". Ketil Andenæs at UIO has started MSc work on "Extensions to COOM and COOP for Process Integration and Workflow support". Ole Jørgen Anfjønsen at UIO has started PhD research on "Object Oriented Query Processing in Multidatabase Systems".

1.7 How to Read this Thesis?

The way to read this thesis depends on the ambition you have for understanding the contents of the thesis.

The following ambitions can be reached stepwise or directly:

1. **To get an overview of the thesis**
   You hopefully should have it by now, after having read chapter 1.

2. **To get an overview of COOP and COOM**
   Read in addition about the COOP framework and COOM model in chapter 9, and the evaluation and conclusion in chapter 16. This means you should read the chapters 1, 9 and 16.

3. **To get a basic understanding of COOP and COOM**
   Read in addition chapter 2 on integration reference models, chapter 10, 11 and 12 on COOM, and the scenario-examples in chapter 15. This means you should read the chapters 1, 2, 9, 10, 11, 12, 13, 14, 15 and 16.

4. **To get an in-depth understanding of COOP and COOM**
   Read in addition about integration requirement areas in chapter 3, the evaluation of existing solution approaches in chapter 4, COOM OQL and query processing in chapter 13, and the COORASS methodology in chapter 14. This means you should read the chapters 1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15 and 16.

5. **To get a full understanding of the thesis, and COOP and COOM**
   Read in addition about the experiments with object-oriented approaches to control integration, data integration and presentation integration in chapter 6, 7 and 8 in part II. This means you should read the whole thesis.

Chapter 5 in part II should be read if you would like to have an introduction to the object-oriented concepts and terminology used in this thesis. Appendices A, B and C can be consulted for details on COOM ODL, OML and OQL.

\(^6\)NSR - Norsk System Rammeverk
Part I

Systems Integration - A Taxonomy of Problems and Solutions
Chapter 2

Systems Integration and Interoperability in Cooperative Work Environments

This chapter sets the context for this thesis. The chapter describes integration and interoperability in cooperative work environments from the perspectives of an environment user and an environment builder. The environment builder perspective is described based on the ECMA/NIST Toaster model [ECM90], and the ESF architectural and conceptual reference models. The Double Toaster reference model presented in the end of this chapter is a refinement to the ECMA/NIST Toaster model, based on ESF concepts. The Double Toaster model is used further in this thesis as the reference model for integration.

Chapter 3 gives a more detailed presentation of different requirement areas related to the integration areas in the Double Toaster model.

Chapter 4 describes and classifies different solution approaches, with respect to the the Double Toaster model and the requirement-areas from chapter 3.

2.1 Cooperative Information Systems and Cooperative Work Environments

The problem focus in this thesis is on support for integration and interoperability in cooperative work environments. Cooperative work environments is a subset of cooperative information systems, where the goal is to give support for individual work environments with integrated tools, and also to extend this with support for cooperation between people working in teams.

Productive work environments for different users lead to a requirement for better utilization of the data and functionality in existing systems, and better support for persons working in teams. The goal is to provide the right information at the right place at the right time, and to provide the right tools for the tasks assigned to a person. Task-assignment is typically related to which role or roles a person has in the organization.

There will be different work-environments for different kinds of work. Software Engineering Environments, Concurrent Engineering Environments, Geographical Information Environments, Office-Work Environments, etc. With respect to the need for integration it seems, however, that a majority of the requirements is the same. Most of the work in the area of integration frameworks has so far been done in the areas of Software Engineer-
ing Environments (SEE) and Computer Integrated Manufacturing (CIM), but this can be extended to other areas.

The baseline for this thesis is work on software engineering environments, done in the Eureka Software Factory project. The focus in this thesis is on support for integration and interoperability, and the belief is that solutions for this will be applicable to other cooperative work environments.

One example of another environment is an environment for a field-evaluation project in an oil exploration and production company, where several people in different roles work together, as shown in figure 2.1. Different roles are: geologist, geophysicist, reservoir engineer, project-manager, secretary etc. These roles are associated with the domain of "Upstream" offshore, which is concerned with bringing the oil up to production. The different roles need tools that access different aspects of underlying systems and databases, such as a well database, a seismic database, a reservoir database, a risk-analysis system, a project management system or a financial system. The environment should provide different tools for the different roles, to provide the functionality needed for the tasks of that particular role. It should provide support for cooperative work on the production of the field-evaluation-report, and it should guide the workflow so that it follows the procedures and guidelines for a field-evaluation project.

Another environment example could be a bank, with roles such as: customer-sales-person, market-analyst or loan-evaluator. Their tools will use different aspects of underlying systems and databases, such as account-information, customer-information, company-information, and bank-product information. The environment should provide the different users with the data and functionality they need in order to fulfill their different tasks. It should give work-process support for different standard procedures such as investment-analysis and support cooperative work in areas such as loan-evaluation.
2.2 Systems Integration and Interoperability

This section defines integration and interoperability in general, and shows how it can be viewed differently from the perspectives of an environment user and an environment builder, respectively.

The dictionary-definition of integration from [TM80] is:

**Definition 2.1 Integration:**
Complete (imperfect thing) by addition of parts; combine (parts) into a whole. (lat. Integrare: make whole) The focus is on making the whole by combining the parts.

Integration from an environment user perspective, means that the system looks and feels like a complete whole, seamlessly providing the tools the user needs. The word "integration" will, in this context, be used as a term for an aggregate that from the outside looks like a complete whole. The environment user is concerned with the perceived integration at the environment's interface. The user desires a seamless tool collection that supports the tasks of the roles of the user.

The goal of an environment builder is to provide these tools, through an efficient utilization of available functionality and data in other tools. The environment builder will then be concerned with the feasibility and effort needed to achieve this perceived integration.

In this chapter the following roles will be covered through the term environment builder:

- A **tool builder**
  that creates a new tool for the environment user.

- A **component integrator**
  that brings an existing tool or tool-component into the environment.

- A **framework builder**
  that provides the basic technology and mechanisms in the environment.

The environment builder challenge is to facilitate the environment user view of an integrated environment by seamlessly "gluing" different underlying system parts together. "Gluing" implies a need for interconnectivity between the systems. Two or more systems are interconnected if they can exchange messages. This, however, only guarantees communication. In order to achieve integration it is also necessary with interoperability between the systems. Interoperability means mutual accessibility and usability of information representation, information equipment, information systems and users.

A definition of interoperability is given in [BC92]:

**Definition 2.2 Interoperability:**
Two components X and Y can interoperate (are interoperable) if X can send requests Ri for services to Y, based on a mutual understanding of Ri by X and Y, and if Y can similarly return responses Si to X.

This means that two systems can interact jointly to execute tasks. What is required is a mechanism that facilitates interoperability between system components, by providing support for high level interaction among components.

The following two sections will describe integration and interoperability in cooperative work environments in more detail. First from the perspective of an environment user and then from the perspective of an environment builder.
2.3 The Environment User Perspective

The environment user is concerned with the perceived integration at the environment's interface. Figure 2.2 shows three support-areas related to how a user perceives an environment:

- **User Task Support**: How the system gives an integrated view on the tools a user needs to perform his tasks.

- **Cooperative Work Support**: How the system gives support to cooperative work for a group of people, or a group of organizations.

- **Work Process Support**: How the system gives support to a sequential use of tools in order to complete a total working process, through different phases.

This is related to integration as follows: User Task Support provides an integrated and seamless view of functionality and data needed for the different roles of a user. Cooperative Work Support provides an integrated and seamless view onto the interworking of different people. Work Process Support provides an integrated and seamless view on how different tools are used in sequence or parallel in a total work-process.

These support-areas will be described in more detail in the following. This will then be used as a basis for deriving some of the requirements for integration and interoperability for the environment builder, which will be the focus in the next section.

2.3.1 User Task Support

**Goal of User Task Support**
2.3.1 User Task Support

User task support means that the system provides a set of tools to support the tasks that a user is responsible for. This set is what the user meets at the interaction interface between a user and the system. The goal is to have task-oriented environments customized to each of the different roles a user has. Figure 2.3 illustrates how the tools on a workstation can provide an integrated view on an underlying set of heterogeneous systems.

Task-oriented integration means that the user perceives an integrated system when he performs a particular task. The system should support the task of the user in such a way that the user feels that he is working with an integrated and uniform system.

In the oil-company-example this can be illustrated by how the project manager's role can have a tool which supports project planning and resource- allocation, by having an activity- network view onto the project management system and a connection to the resource- allocation database to plan for the use of persons. A project member might have his private view into this, showing only the information for activities which are relevant to him. For unified access to information it should be possible to find related information in the well database, seismic database, and reservoir database.

In the bank-example this can be illustrated by how a customer-sales person needs to have tools to access all relevant information about a customer. In addition he needs access to product-information such as interest-rates, investment-options etc. A market-analyst needs to have tools for identifying groups of customers based on different criterias and tools for doing different kinds of "what-if" analysis.

Solutions for User Task Support

In order to realize user task support one needs underlying system facilities that provide [Pen92]:

- The ability to invoke tools from tools - Functional integration
  The ability to invoke tools from tools means that it should be transparent for the user, which tools that actually support a task. Traditional tools have been application-oriented, with one tool performing one particular function. To do a more complex task a user might need to explicitly invoke a number of tools. For instance, to write a monthly report the user might need to invoke one or two different database- applications to get some data, to invoke a spreadsheet to do some calculations, and to invoke a text-editor to write the report. A goal is to achieve a higher level of
function-integration. It should be possible to use only one tool for a particular task, and from within that tool be able to invoke the necessary functionality. Examples of such tools are hypertext/hypermedia-editors that support the notion of live-links between the different elements of a node-link structure. In [NNO90] this has been contrasted as task-oriented versus application-oriented user interaction.

- **A uniform way of invoking tools - Location integration**
  A uniform way for invoking tools means that all tools are started in the same way, independent of their physical location (i.e. they may reside on different possibly heterogeneous machines). That is, users are not aware of the hardware configuration and possibly operating system heterogeneity.

- **Sharing of information between tools - Data integration**
  When one or more users apply a set of tools, the different tools should be able to work with the same set of current and updated information. For design environment like SEE's or CIM environments, different tools should also be able to work on a common product-model.

- **Unified access to information in multiple databases - Multi-model Data integration**
  Very often the user needs to access information in already existing systems or databases. It should be possible to access multiple systems and databases and present information in a unified way.

- **An integrated Look-and-Feel - Presentation integration**
  Integrated Look and Feel means that the user will interact with the system through a uniform look and feel user interface. It does not mean that all users will have the same look-and-feel, but that each individual user shall be able to interact with the system in a uniform way. The window-based user-interfaces that we have seen lately, provide typical examples for this. Different systems have somewhat different looks and feels, e.g. OSF/Motif, Open Look, OS/2 Presentation Manager, New Wave, Windows, Apple Macintosh etc. The requirement is to allow different users to select their preferred look and feel-style, and to have the tools they are using conform to that.

### 2.3.2 Support for Cooperative Work

Cooperative work support is support for team-working for a group of people, and a group of organizations. Figure 2.4 illustrates cooperation between people in different roles, at both the same and different geographical locations.

#### Goal of Cooperative Work Support

Typically people work in groups and need to coordinate their work with the work of other people. CSCW (Computer Supported Cooperative Work) is a new research area that focuses on how to create tools and environments for the support of group- and team-work [Gre89]. The basic requirement is to be able to transfer work results between people, and to be able to coordinate concurrent work on common underlying information.

Different organizations interacting with the purpose of achieving a common business task might also be viewed as a kind of cooperative activity. Useful computer support can be given for standard interactions, such as interchange of standard documents and forms.

Support for cooperative work can be given in three different areas, according to [EGR91].
• **Communication:** A separation exists between asynchronous, text-based communication such as electronic mail and bulletin boards, and synchronous communication such as telephone and face-to-face conversations. Applications such as voice mail and talk programs are located somewhere between these two.

• **Collaboration:** This demands that people share information. The goal of traditional information systems has been to insulate users from each other. Working on a common product-model requires a shared environment that offers up-to-date information for groups working on the same part of the model, and an ability to give explicit notifications for changes that other users need to be aware of.

• **Coordination:** The coordination of a group’s activities enhances the effectiveness of communication and collaboration. Without coordination, e.g., a team of programmers, will often engage in conflicting or repetitive actions. Coordination can be viewed as an activity in itself, as a necessary overhead when several parties are performing a task.

In the oil-company example this can be illustrated by how a project manager communicates task-allocation to the project-members, and how the project-members need to coordinate their work on the collaboration on the development of a field-evaluation report.

In the bank-example this can be illustrated by how a market-segment sales-team can be set up between the main office and different branch offices in order to sell a particular bank-product. There is a need for communication and sharing of information between the different members of a team, and for coordination of the activities.

**Solutions for Cooperative Work Support**

A time-space taxonomy has been described in [Joh91]. The taxonomy distinguishes between cooperative work at the same time (synchronous) at different places (e.g. Teleconferencing, Shared screen) or at the same time at the same place (e.g. Meeting), at different times (asynchronous) at different places (e.g. Email) or at different times at the same place (e.g. log-book information in a laboratory).
A number of different approaches exists, and much work is currently ongoing, in particular related to the combined use of different media like video, voice and computer screens [IM91].

GroupWare is a growing commercial area addressing support for cooperative work. Typical groupware tools are conferencing tools, electronic mail, group calendar and multi-user editors. In particular, various forms of advanced electronic mail are getting attention. This might involve sending a mail which references information in an existing database, and makes it possible to reflect changes in the underlying data at a later point in time.

Cooperative work among different organizations might involve the exchange of data in different structured formats, e.g. for documents or product-models. Typically using standard data-exchange formats such as ODA/ODIF, SGML, EDIFACT.

### 2.3.3 Work Process Support

Work process support includes support for a sequential use of tools to complete a total working process through different phases [HP90]. Here the emphasis is on the environment user perspective of support for a work process. The high-level goals for work process support is to be able to model both static and dynamic aspects of a work process, and to be able to plan, reason and assess the plan during execution.

#### Goal of Work Process Support

![Diagram](image)

**Figure 2.5: Work Process Support**

A number of process-sensitive or process-centered environments, which provides work process support are currently being experimented with [EJP+91, Gru91, RJ89, H+88, Con89, PR92], in particular in the area of software engineering environments.

Work process support is the environment user perspective on the area which later will be called process integration from the environment builder perspective. Process integration from an environment user perspective is the ability to interact with environment capabilities based on a process-oriented user interface, e.g. a user interface with process guidance or enforcement [Pen92].

Work process support can enhance both user task support and cooperative work support. Support for cooperative work can be given without work process support, while work process support almost always is given in connection with various forms for cooperative work support [KMW91]. User task support can be given without work process support,
2.3.3 Work Process Support

while work process support almost always includes a user task specific component, to guide the individual user tasks.

In the master-thesis-report "Dynamic Modeling of Projects" [BL85] the author has defined an object-oriented model for process-modeling of large offshore-engineering projects. A Work process-model was then defined to model the static and dynamic relationships between the following three main-classes of objects:

- Product
- Resources: Tools / Roles-Persons
- Activities

Figure 2.5 adapted from the ESF-project [Fer91] shows a similar relationship between roles, activities and tools/product-model.

In addition to have an integrated system for performing one task, the user would like to have a system which guides him through a sequence of tasks, as natural steps in a working-process. The requirement to process-oriented integration is to be able to use results from previous tasks in a new task, and to use the results from this new task in later tasks.

A typical bank-example is the process for approval of a loan which a customer has applied for. The approval-process consists of a number of steps, and typically the involvement of different people in different roles. It might also include interaction between a branch-office and the main-office.

In the oil-company example, work process support can be illustrated by the need to go through a predefined set of activities in a field-evaluation project. Different people in different roles have different tasks to do in this process.

Work process support should include facilities for the following:

- **Work process-definition:** Make an initial specification of the process for a particular area of work.
- **Work process-planning:** Do planning and replanning for a particular piece of work.
- **Work process-enactment:** Let the process-model be an enactable process that may enact with tools and humans.
- **Work process-monitoring:** The process-model should monitor the state of activities and tools in the environment.
- **Work process-resource-management:** The process-model should be able to manage the different resources in the environment.

Work process models also be used utilized in InterWorking support, in particular as an underlying model for coordination.

Solutions for Work Process Support

Process modeling is currently a highly active research field. Most research and development so far, have been done in the area of software process modeling. Current research issues are how to support flexible modeling, evolution and distribution. There is also a growing interest for extending this to other areas as well.

Different solutions for work process support are discussed in section 4.1.5 and 4.7.6.
2.4 The Environment Builder Perspective

The environment builder is concerned with the feasibility and effort needed to achieve the integration to be perceived by the environment user.

The software engineering community has created different reference models for integration from an environment builder perspective. This section first presents the ECMA/NIST Toaster Reference model, and then the ESF Conceptual and Architectural Reference models. The refined Double Toaster model to be used in the rest of this work is based on a refinement of the ECMA/NIST Toaster model with concepts from the ESF Architectural reference model.

2.4.1 The ECMA Toaster Reference Model for Integrated Environments

![Diagram of the Toaster model]

Figure 2.6: The "Toaster" Model

Figure 2.6 shows the ECMA¹/NIST² Toaster Reference Model for Integrated Environments [ECM90].

The Toaster model is based on a grouping of integration-services in four areas: process integration, control integration, data integration and presentation integration, as described in [Was89]. This model has been called the Toaster model, because of the way tools are sliced in between presentation services and data-storage services.

Initially some services for work-process support and sequencing of tools was put into the control integration-area. Later process integration was added [NIS91] as a front-layer behind the presentation integration layer. The areas are then called service areas instead of integration areas.

Process, control-, data- and presentation-integration is here defined as follows:

- **Process integration**: The degree to which the user's working process and use of tools

---

¹ European Computer Manufacturers Association  
² U.S. National Institute of Standards
can be guided by a model of the work process and the methodology to be followed, possibly in cooperation with other users.

Process integration is concerned with the relationship of functions between two tools. Two tools are well integrated with respect to process if the function of one tool directly precedes or follows the function of the other as part of a defined work process for the user [X3H93].

- **Control integration**: The degree to which tools are able to interact directly with each other, by requesting and providing functional services.

  Control integration is concerned with the relationship of actions between two tools. Two tools are well integrated with respect to control if one tool provides some of its functions as services to the other tool [X3H93]. This also implies mechanisms for how tools can be executed (i.e. registered, activated, de-activated, suspended) and combined.

- **Data integration**: The degree to which tools are able to share common data and information.

  Data integration is concerned with sharing data between two tools. Two tools are well integrated with respect to data if the data produced by one tool may be easily used by the other [X3H93].

- **Presentation integration**: The degree to which a user-interface program might provide the access to the functionality needed by the user through a uniform look and feel.

  Presentation integration is concerned with the relationship between the appearance and behaviour of two tools. Two tools are well integrated with respect to presentation if their appearance is similar and their behaviour is consistent [X3H93].

An important aspect of this model is that these different kinds of integration might be viewed independently of each other.

In addition to these four integration areas it is possible to distinguish different forms of integration with respect to how a tool is brought into an environment [Pen92]:

- **Platform integration**: The degree to which tools run and communicate independent of the underlying hardware/software platform (e.g. virtual operating system, network services, etc.).

- **Tool Building integration**: The degree to which tools rely on common services/automated mechanisms in support of tool building. This addresses integration support for the development of new tools.

- **Tool Insertion integration**: The degree to which tools can be integrated (with respect to control, data, presentation, process) with existing capabilities. This addresses integration support for tools which exist and need to be inserted into the environment.

These integration areas can be used to provide the support areas described from the environment user perspective as follows:

- **User Task Support**: is provided through tools integrated by control integration, data integration and presentation integration.

- **Cooperative Work Support**: is provided through special tools that support interaction among people.

- **Work Process Support**: is provided through process integration, with a user interface provided through presentation integration.
2.4 The Environment Builder Perspective

2.4.2 The ESF Reference Models for Integrated Environments

The ESF\textsuperscript{3} project [Fer91, FkNO92] has developed two reference models, The ESF Conceptual Reference Model [GHH+92] and the ESF Architectural Reference Model [Eur89]. These models will be used as a basis for a refined Toaster-model in the next section. Section 4.7 gives a more detailed description of the ESF project.

The ESF Conceptual Reference Model (CoRe)

![Diagram](image)

Figure 2.7: Environment Layers - "The ESF Ladder"

The goal of the ESF Conceptual Reference Model (ESF CoRe) [GHH+92] is to provide a framework for understanding of issues related to integrated software factory environments. The reference model is based on the view that the focus of such an environment should be to support the communication between people, between people and tools and between tools. An environment should support mechanisms to help coordinating such communication in different layers. The concept of communication layers is based on an analogy to the ISO/OSI Reference Model for data communication[Zim80].

The ESF CoRe is specialized for software factories, but the concepts are applicable to all kinds of cooperative work environments, and are thus generalized in this context. Figure 2.7 shows a principal architecture for the communication between the different entities involved in a cooperative work environment. The "Ladder" shows three layers (Person, Tool and Platform respectively) with their protocols (InterWorking, InterOperation and InterConnection) and interfaces (InterAction and InterFunction). The protocols and interfaces are defined as follows:

- **Interworking** is communication between people. The goal of support for Interworking is to improve group performance through cooperation based on defined methods, shared experience and the current status of work. An example of this is the communication between two or more people about the tasks they are responsible for, this might be done through a process model with information about these tasks, by direct synchronous communication in a desktop video-conference or asynchronous communication through electronic mail.

- **Interaction** is communication between people and tools. The goal of support for

\textsuperscript{3}Eureka Software Factory
Interaction is to improve individual effectiveness through work contexts\(^4\) offering the right information and the right tools at the right time. An example of this is communication between a person and his working environment, where the user interacts with a set of tools supporting the tasks he has to do.

- **Interoperation** is communication between tools. The goal of support for interoperation is to improve tool integration through mechanisms for interaction between tools. An example of this is communication between two tools, either done through data in a shared database, through direct requests or through notifications.

- **Interfunction** is communication between tools and platforms. The goal of support for interfunction is to make it easy to support tools on multiple platforms. In this context platform services will be viewed as specialized tools and components which can be utilized through the interoperation mechanisms. The interfunction interface is then effectively the mechanism for interoperation-support. An example of this is communication between a tool and underlying services, where this services are available on multiple platforms.

The move towards "open" systems shows the user-community requirements for not being locked into the use of hardware and software from one vendor. This has so far, however, been taken to mean the use of one single operating system interface, i.e. Posix/Unix. From an integration and interoperability perspective it is however more important to have mechanisms for efficient interaction between systems. "Middleware" has lately been used as the name for such mechanisms, which are realized as a layer above the native operating system. Future standards for "Middleware" will make it possible to integrate systems running on different underlying operating systems, such as Unix, VMS, DOS, Windows NT etc.

- **Interconnection** is communication between platforms. The goal of support for interconnection is to make it easy for different platforms to communicate with each other. Interconnection provide for the more basic integration of machines as defined in the ISO/OSI reference model [Zim80]. In this context, interconnection will be hidden by the interfunction interface of the interoperation mechanism. An example of this is communication between two tools using a request-oriented interoperation mechanism, where the transport of the data is done through the TCP/IP protocol.

The concepts of ESF CoRe can be mapped to the environment user perspective on integration and interoperability as follows: Interworking is mapped to tools that support cooperative work, and to the part of work process support that is concerned with coordination of multiple persons. Work process support lies between Interworking and Interaction, because it will give support for a process-oriented interworking between people, but it will also give each individual user support for the work-process associated with his role. Interaction is mapped to user task support, and to the part of work process support that is concerned with work process support for one individual person. Interoperation, Interfunction and Interconnection is hidden from the environment user.

The concepts of ESF CoRe can be mapped to the ECMA/NIST Toaster model as follows: Interworking is mapped to process integration and tools that support cooperative work. Interaction is mapped to presentation integration and partly to the user interaction part of a tool. Interoperation is mapped to control integration and data integration. The focus on interaction means a focus on control- and data-integration in the following. Presentation integration is also included, because user interaction components is an important part of a tool. Interfunction and Interconnection is hidden by the control integration and data integration services.

---

\(^4\)Work context is described further in section 5.2
The ESF Architectural Reference Model

Figure 2.8: ESF Architectural Reference Model

The ESF Technical Reference Guide [Eur89] defines a communication-oriented architectural reference model, where an environment consists of communicating components, as shown in figure 2.8. This is an architectural model that provides a basis for creation of solutions for the areas described in the conceptual reference model.

The architecture consists of the following parts:

- **User Interaction Environment**
  Each user has a user interaction environment which is his personal work environment. Facilities for cooperation with other users make this into a cooperative work environment.

- **User Interaction Component**
  Tools are divided into two parts, a user interaction component and one or more service components. User interaction components provides the user interfaces of the tools.

- **Service Component**
  Service components provide functional services.

- **Software Bus**
  The Software Bus is the mechanism for interaction between components, it includes an interaction-mechanism for request-oriented interaction and an event multicaster service for notification-oriented interaction.

- **Database / OMS**
  The environment allows for the use of multiple databases.

- **Process Interaction Engine**
  A Process interaction Engine exists for each environment user, and provides guidance for the tasks of an individual user. The Process Interaction engines interacts with the Process interworking engines.
- **Process Interworking Engine**
  One or more Process Interworking Engines are used to maintain the common process models for one or more projects.

![ESF Conceptual Reference Model](image)

![ESF Architectural Reference Model](image)

**Figure 2.9: From ESF Conceptual Model to ESF Architectural Model**

The relationship between the ESF Conceptual Reference Model (CoRe) and the ESF Architectural reference model is shown in figure 2.9.

- **Interworking** is supported through a Process Interworking Engine interacting with the Process Interaction Engines of different users.

- **Interaction** is supported through user interaction components for the different tools, made available in a user interaction environment for each user.

- **Interoperation** is supported through component interaction, either request-oriented or notification-oriented through the software bus, or through data in a shared database.

- **Interfunction** is supported by the availability of the software bus and database interoperation services on multiple platforms.

- **Interconnection** is supported through the use of common transport-protocols for the implementation of the software bus on multiple platforms. The protocols are hidden by the software bus.

The ESF architectural reference model can be used to provide support for the areas described from the environment user perspective as follows:

- **User Task Support**: is provided through a User Interaction Environment, with a process interaction engine and a number of user interaction components for the different tools.
Cooperative Work Support: is provided through special tools that support interaction among people, and by the cooperation support in the process interworking engines.

Work Process Support: is provided through process interaction engines in cooperation with process interworking engines.

2.4.3 The Refined Double Toaster Model

Figure 2.10: The "Double" Toaster-model

The ESF architectural reference model can be used as a basis for refining the integration areas in the Toaster model into useful sub-areas. In the ESF architectural reference model the integration areas in the ECMA/NIST Toaster model is facilitated as follows:

Control integration
is facilitated through a software bus for request-oriented interaction, and through an event-multicaster service for notification-oriented interaction. This is further described in section 4.7.3.

Data integration
is facilitated through one or multiple database systems. These might be accessible through the software bus, or used only locally by one or more components. This is further described in section 4.7.4.

Presentation integration
is facilitated through user interaction components that interacts with one or more service components. The display-oriented part of these components might be portable to different windowing systems, the model-oriented part of these components will
provide a unified view on one or more service components. This is further described in section 4.7.5

- **Process integration**
  
is facilitated through process interaction engines for the work support of individual users, and through process interworking engines for the coordination of work between multiple users. This is further described in section 4.7.6

![Diagram](image)

**Figure 2.11: Double Toaster-model refined from ESF architectural model**

A refinement of these areas in the Toaster model leads to the Double Toaster model as shown in figure 2.10. The mapping from the ESF model is shown in figure 2.11.

- **Splitting of tools** by a separation into a user-interaction tool component and one or more functional service components. This separation is an ideal goal. Many tools do not expose their functionality to other tools, but only offer a user interface. These are mapped to user-interaction components.

- **Presentation integration** is divided into two sub-areas, display-oriented and model-oriented. Display-oriented is the previous front-layer, while model-oriented is the user-interface part of a tool.

- **Process integration** is divided into two sub-areas, interaction process-support and interworking process-support. The interaction support is the previous process layer, extended to also include a user-interface-component. These might interact with process interworking services, which are shown as a functional service.

- **Control integration** is divided into two sub-areas, request-oriented and notification-oriented

- **Data integration** is divided into two sub-areas, single-model and multi-model
This refinement is called the Double Toaster model, because the tools are split in two parts, a user interaction part and a service part, and each integration area is split into two sub-areas.

The Double Toaster model has a more architectural flavour than the basic Toaster model. It should, however, not be interpreted as an architectural reference model, but as a conceptual model for description of environments. Existing environments can be described by the degree to which they provide support for the 8 integration sub-areas. This will be shown in chapter 4.

2.5 The Problem Focus of this Thesis

This thesis will focus on systems integration and interoperability in cooperative work environments, from an environment builder perspective. The Double Toaster model in figure 2.10 will be used as the reference model for integration areas to be supported.

The goal is to provide mechanisms for interoperation, in particular for control integration and data integration. Presentation integration will be facilitated through support for interoperation between user interaction components and service components. Process integration will be facilitated through support for interoperation between process interaction engines and process interworking engines.

The next chapter will provide definitions for the integration sub-areas shown in the Double Toaster model, and give a more detailed presentation of different requirement areas related to these.
Chapter 3

Integration Requirement Areas

The goal of this thesis was stated in section 2.5 as: "to provide mechanisms for interoperation, in particular for control integration and data integration. Presentation integration will be facilitated through support for interoperation between user interaction components and service components. Process integration will be facilitated through support for interoperation between process interaction engines and process interworking engines." This chapter gives a detailed discussion of requirement areas related to the three focused kinds of integration: control, data and presentation integration. It also provides a discussion on conflicting requirements, and on requirements for architectural support for process integration facilities.

3.1 Conflicting Requirements

Systems integration and interoperability is a complex problem-area where solution approaches often need to make compromises for conflicting requirements. Some of these conflicting requirements are described in the following:

- **Heterogeneity versus Homogeneity**
  To which degree is it possible to provide a homogeneous view on underlying heterogeneous components?

- **Autonomy versus Common policy**
  To which degree is it possible to support autonomy for components in a federation where some common policies are required?

- **Distribution versus Centralization**
  To which degree is it possible to provide a centralized view on distributed components?

- **Fine-grain versus Large-grain**
  To which degree is it possible to provide simultaneously support for fine-grained and large-grained components?

- **Efficiency versus Flexibility and Extensibility**
  To which degree is it possible to have an efficient system and still support dynamic changes and future extensions?

- **Simplicity versus Coverage**
  To which degree is it possible to have a simple model and also support a large variety of requirements?
3.3 Requirement Areas for Control Integration Facilities

- *New language versus Existing languages*
  To which degree is it possible to introduce new concepts and still support existing languages?

- *Variety of Components versus Functional Usability*
  To which degree is it possible to integrate a large variety of different components and also have a comprehensive functional usability for the integrated components?

- *Syntactical basis versus Semantical integration*
  To which degree does the syntactical support provide a basis for semantical integration?

Different existing solution-approaches and the proposed COOP integration framework in part III will be discussed with respect to these conflicting requirements.

3.2 Requirement Areas for Integration Facilities

The Double Toaster reference model in figure 2.10 in section 2.4.3 separated the different kinds of integration from the ECMA/NIST Toaster model into sub-areas as follows:

Control integration has been separated into two sub-areas, one for request-oriented control integration and another for notification-oriented control integration. Data integration has been separated into two sub-areas, one for single-model data integration and another layer for multi-model data integration. Presentation integration has been split into two sub-areas, the display-system and the user-interface part of a tool. The user-interface-part of a tool has two interfaces, display-oriented towards the display-system and model-oriented towards the functional parts of a tool.

The following sections will describe requirement areas for each of these in more detail.

3.3 Requirement Areas for Control Integration Facilities

Control integration has been separated into the two sub-areas: *Request-oriented* and *Notification-oriented* control integration. This separation was based on the ESF architectural reference model. The initial ESF software bus focused only on request-oriented control integration, while the ESF Event Multicaster focused on notification-oriented control integration. Both mechanisms can be used to emulate the other.

Requirements for the two areas are described separately in the following sections.

3.3.1 Request-oriented Control Integration

**Definition 3.1 Request-oriented control integration:**
Request-oriented control integration is the extent to which tools are able to interact directly with each other, by requesting and providing named functional services. □

Request-oriented control integration is needed by user task support for the ability of tools to interact with each other, work process support for the ability to start up new tools.

Figure 3.1 illustrates interaction between system-components, by showing components that interact through the matching of requested and provided services.
3.3.2 Requirement Areas for Request-oriented Control Integration (RCR)

Table 3.1: Requirement Areas for Request-oriented Control Integration

<table>
<thead>
<tr>
<th>Requirement-areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCR-1: Interface-description language</td>
</tr>
<tr>
<td>RCR-2: Interaction model</td>
</tr>
<tr>
<td>RCR-3: Run-time</td>
</tr>
<tr>
<td>RCR-4: Robustness and error-handling</td>
</tr>
<tr>
<td>RCR-5: Multi-user</td>
</tr>
<tr>
<td>RCR-6: Version handling</td>
</tr>
<tr>
<td>RCR-7: Security aspects</td>
</tr>
<tr>
<td>RCR-8: Administration</td>
</tr>
<tr>
<td>RCR-9: Heterogeneity</td>
</tr>
</tbody>
</table>

Table 3.1 shows 9 requirement-areas which solutions for request-oriented control integration should take into consideration. RCR means "Requirement-area for Control integration - Request-oriented".

The high-level goal of request-oriented control integration is to support pluggability of system-components conforming to the same interface, so that it is easy to plug out one component, and plug in another compatible component conforming to the same interface, or an extended interface. The interoperability among components will be supported if they request and provide services through an interaction-mechanism. Existing applications can be attached to an adoptor in order to be able to use the interaction-mechanism.

These requirements are built on the requirements to the ESF Software Bus and the OMG Object Request Broker, which were specified while the author was working in the ESF Software Bus subproject from October 1988 to January 1990 [B+89c, tea89], and in the OMG Object Request Broker task force in 1991 [Obj91b].

Each sub-area is described individually in the following:

**Interface-description Requirements (RCR-1)**

- Interface Definition Language
- Complex structures (graphs/trees) as parameters
3.3 Requirement Areas for Control Integration Facilities

- Semantic specifications of operations and arguments
- Component Description Language
- Component Interaction Description (Interaction Description Language)

In order to have a request-oriented model, there is a need for a way of specifying and describing services. A service is an interface providing several operations, and should be precisely described through an Interface Definition Language. Each operation may have several input and output arguments, and these should possibly be both simple and complex data-types like lists, trees and networks. References to other services should also be possible as arguments. An interface should be described as semantically complete as possible, ideally giving a full specification of the meaning of the operations and their arguments. A component description for a piece of software might include both provided and requested services, as well as other information such as licensing. A component description language could use the interface definition language for the specification of requested and provided interfaces. A component interaction description language with the goal of specifying mappings between differences between requested and provided services could be provided. This would be useful if there

Interaction Model Requirements (RCR-2)

- Support for Request Dispatch and Binding
- Support for Delivery and Activation
- Support for Reply Synchronization

A model of components interacting through the offering and request of services, needs an interaction-model that provides facilities for the dispatch, delivery and reply of requests. The interaction-model might provide both synchronous (procedure-based) and asynchronous (message-based) interactions. If a requested service is not active, it should be automatically activated.

Run-time (RCR-3)

- Support for Service Naming and Locating
- Queries based on object names, attributes, and relationships
- Support for Parameter Representation and Encoding
- Performance for local and remote server object requests
- Identify service requester
- Provide a naming system architecture

At run-time, there should be a facility for handling the interaction among different tools. This should give support for identifying available services and to make it possible to do interactions.
Robustness and Error-handling (RCR-4)

- Error-handling
- Support for Exception Handling
- Provide robust operation and a high level of availability

It should exist a facility for error-management and exception-handling, to make a system as robust as possible.

Multi-user (RCR-5)

- Multi-user components (multi client/servers)
- Provide concurrency control
- Transactions - critical sequence

It should be possible for one component providing a service, to support multiple clients at the same time. This requires facilities for concurrency and transaction-control.

Version Handling (RCR-6)

- Several versions of same component

The management-facilities for components need to take component-evolution into account. This means a possibility for the handling of several versions of one component.

Security Aspects (RCR-7)

- Access rights to components
- Support for Security Mechanisms
- Support discretionary access control of objects
- Encryption on/off

Security is important in a multi-user environment. It should be support for a mechanism for restricting access to different components, based on user or user-group. Encryption should be possible for communicated information.

Administration and Installation (RCR-8)

- Environment Administration
- Accounting
- Licensing
- Invoicing for use of components
- Debugging

A total environment needs support for the administration of all installed systems, in particular with respect to general installation-management, accounting and licensing.
3.3 Requirement Areas for Control Integration Facilities

Heterogeneity Transparency (RCR-9)

- Transparency with respect to: Distribution, Concurrency, Location, Migration, Replication, Access-mechanism, Implementation-code
- Interface-mappings for multiple programming languages
- Adapter building
- Multiple communication-mechanisms

It is important to have mechanisms which make underlying heterogeneity and differences as transparent as possible. It is particularly important to provide transparency for location and physical access-mechanisms for component-interaction.

3.3.3 Notification-oriented Control Integration

![Diagram](image)

Figure 3.2: Control integration - Notification-oriented

**Definition 3.2 Notification-oriented control integration:**
Notification-oriented control integration is the extent to which tools are able to interact by sending out notification about certain events. Other tools might register interest for being notified about certain events. □

Notification-oriented control integration is needed by user task support for the possibility to keep user-interfaces updated with respect to changes in underlying work process support for the ability to be notified about events that are relevant for the process-model.

Figure 3.2 illustrates notification-management through an event-notification service.

3.3.4 Requirement Areas for Notification-oriented Control Integration (RCN)

Table 3.2 shows 4 requirement-areas which solutions for notification-oriented control integration, should take into consideration. RCN means "Requirement-area for Control integration - Notification-oriented".

The main requirement is to be able to let tools "publish" information about events that have happened. Other tools might then "subscribe" to information about these events.
Table 3.2: Requirement Areas for Notification-oriented Control integration

<table>
<thead>
<tr>
<th>Requirement-areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCN-1: Event Publishing</td>
</tr>
<tr>
<td>RCN-2: Event Subscribing</td>
</tr>
<tr>
<td>RCN-3: Heterogeneity</td>
</tr>
</tbody>
</table>

If tools are able to register their interest for certain events, they can let such events be a basis for their control-flow. This is for instance a useful facility for a process-modeling service, which can use notifications to monitor the ongoing activities.

Each sub-area is described individually in the following:

**Event Publishing (RCN-1)**

It should be possible to identify when certain events occur, so that tools interested in the event can be notified.

**Event Subscribing (RCN-2)**

It is necessary that tools might request to be notified when specific events occur.

**Heterogeneity (RCN-3)**

Mechanisms for activation and identifying events might be different in different environment. It should be possible to use a uniform approach for this, even if different implementations is used in different systems.

### 3.4 Requirement Areas for Data Integration Facilities

Data integration can be separated into the two sub-areas: *Single Model* and *Multi Model* data integration. This separation is based on experiences from the ESF project, where both of these areas were focused on by different partners in the OSS sub-project.

Requirements for the two areas are described separately in the following sections.

#### 3.4.1 Single Model Data Integration

**Definition 3.3 Single-model data integration:**

Single-model data integration is the extent to which tools are able to share common data and information that are stored and manipulated through one single data model and a corresponding storage service.

Single-model data integration is needed by user task support for tools to share common data, by cooperative work support to possibly do collaborative work on shared data, by workprocess support to represent the process-model in association with a product-model and resource-model.
3.4 Requirement Areas for Data Integration Facilities

Figure 3.3: Single-model data integration

Figure 3.3 illustrates two applications that are integrated through the schema of a common data storage service.

3.4.2 Requirement Areas for Single Model Data Integration (RDS)

<table>
<thead>
<tr>
<th>RDS-1: Basic ERA-model</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDS-2: Complex objects</td>
</tr>
<tr>
<td>RDS-3: User-defined data types</td>
</tr>
<tr>
<td>RDS-4: Prog. language support</td>
</tr>
<tr>
<td>RDS-5: Versions/variants</td>
</tr>
<tr>
<td>RDS-6: Metadata/Schema</td>
</tr>
<tr>
<td>RDS-7: Views</td>
</tr>
<tr>
<td>RDS-8: Rules / Triggers</td>
</tr>
<tr>
<td>RDS-9: Query language</td>
</tr>
<tr>
<td>RDS-10: Security</td>
</tr>
<tr>
<td>RDS-11: Distributed Architecture</td>
</tr>
<tr>
<td>RDS-12: High Performance</td>
</tr>
<tr>
<td>RDS-13: Concurrency</td>
</tr>
<tr>
<td>RDS-14: Cooperation</td>
</tr>
<tr>
<td>RDS-15: Logging - recovery</td>
</tr>
<tr>
<td>RDS-16: Data Interchange</td>
</tr>
</tbody>
</table>

Table 3.3: Requirement areas for Data Integration - Single Model

The main requirements are from tools that need to share information. Different kind of integrated environments will have different requirements for data model and DBMS.

Most data-models in our bank-example are fairly simple, while data models for different kinds of engineering design are more complex. We have used the more complex models as a basis for the requirements.

The main focus for requirements have been “new” application areas for databases, like CAD/CAM, SEE (Software Engineering Environments), GIS (Geographical Information Systems) and Office Systems/Multi Media. Some typical modeling-situations for these areas are:
3.4.2 Requirement Areas for Single Model Data Integration (RDS)

- CAD/CAM: Product-model structures (directed graphs).
- GIS: Sequences and structures of lines, points (splines), areas, ...

Common for these application areas is that they are concerned with complex structures of data of different types. They are often typical "Design"-applications with multiple users working together on a common set of complex information. The first 10 requirements are requirement areas for a data-model which can support typical engineering-applications found in design-environments. Since this is the kind of data typically found in integrated environments, there needs to be database-support for this.

The 16 requirements are partly deduced from "The HyperModel Benchmark", [ABM+90, Ber88b, B+89d, BA91]. This benchmark was specified based on requirements from a generic engineering data model. In [Lar92] a revised and smaller version of this benchmark has been presented. Chapter 7 will present how the HyperModel Benchmark has been used as a basis for getting experiences from the use of object-oriented databases.

The 16 requirements has also partly been extracted from the "OSS Requirement Specification" [Ber89], which was made as part of the ESF OSS (Object Storage Services) subproject.

Basic ERA-model (RDS-1)

The basic data model could in principle be of different kinds, e.g. a functional model, an object-oriented model, or an entity-relationship model or a relational model.

We have, however, taken the stand that we want the underlying data model to be an extended Entity-Relationship model [Ber89], because this has been a successful approach in a number of environments, [GMT86].

The lowest requirement is to have a structurally object-oriented datamodel, which is an entity-relationship model with support for complex entities (or objects) it would be preferred to extend this to a fully object-oriented datamodel, as described in [Dit86].

The model should support:

- **Entities (or Objects)**: These should be defined by object-identity as opposed to identification by value or path. Both identification by value and by path should be supported on top of object-identity. References between objects should be done through object-identity [KC86].

- **Relationships**: Relationships connect objects and can have attributes. Binary relationships of the form 1:1, 1:M and M:N should be supported. It should be possible to specify ordered 1:N and M:N relationships (e.g. to model the sequence of chapters in a document).

- **Attributes**: are literal values which can be textually represented. Attributes might me associated with both entities/objects and relationships.

- **Generalization/Specialization relationships**: should be supported between entity-types, in order to allow for inheritance of attributes and relationships.
3.4 Requirement Areas for Data Integration Facilities

- **Aggregation**: It should be possible to construct complex objects by using a special "is-part-of" relationship, with predefined semantics for copying, locking and deletion (see requirement RDS-2).

**Complex Objects (RDS-2)**

The basis of the data-model should be complex objects, composed by more elementary objects. In summary, we require the data model to be structurally object-oriented, that is the data model shall allow representation of every application entity by exactly one object and include operators that deal with complex objects in their entirety.

It should be possible to describe aggregate-structures in terms of complex objects, and arbitrary relationships between components. Complex objects might have shared sub-parts. Operations like transitive closure, should be supported on the hierarchical structures.

**User-defined Data Types (RDS-3)**

The datamodel should be extensible to capture data types like text and bitmap, and the definition of user-defined types like drawing, picture, voice etc.

It should be possible to make user-defined operations for both attribute-types and object-types.

**Programming Language Support (RDS-4)**

It should be possible to access the data from multiple programming languages. Ideally there should be no impedance mismatch [MP84] between the programming language and the data model. In particular it would be useful with language-oriented mechanisms for handling shared and persistent data, for both singular entities/objects and sets of such.

**Versions/Variants (RDS-5)**

The system should give support for history maintenance by an ability to store versions and variants of entities and relationships. This can be used as a basis for a configuration management system.

**Metadata/Schema (RDS-6)**

The schema should be handled similar to other information, i.e. as objects. The database should be self-describing in the sense that the database schema is represented as a set of object with special semantics, but not with operators different from those necessary for creating and/or modifying other objects. This includes version/variant support similar to for data on the instance level.

**Views (RDS-7)**

Different applications will require different views on the shared and persistent data. These views should allow for data independence, so the same views might be used even if the underlying representation changes.
3.4.2 Requirement Areas for Single Model Data Integration (RDS)

Rules and Triggers (RDS-8)

It should be possible to activate different database-operations at certain events.

Query Language (RDS-9)

There should be a high-level query language with both a programming and an end-user interface. It should preferably be an integrated part of the programming interface.

Security (RDS-10)

There should be a mechanism to impose different kinds of access-control policies, for access to shared and persistent data. The mechanisms for access-control and authorization should be integrated in the data model.

Authorization shall be performed by granting a specific access right to a subject (discretionary access control), i.e. an access right shall be defined by a subject, an entity of protection and an access mode.

Distributed Architecture (RDS-11)

The system should take advantage of the possibilities for distribution in a workstation-based environment, either by a client/server architecture or as a fully distributed system. As small part of the system as possible should be centralized.

High Performance (RDS-12)

High performance is the most important requirement to a DBMS in an integrated environment [Mai87]. Cached objects should have an access-time similar to the access-time for normal in-memory objects. It should be possible to load some thousands related objects into the cache with a performance that cold be tolerated by a waiting user.

Concurrency (RDS-13)

Short operations on the database should be administrated by serializable transactions. A concurrency control mechanism should give support for "short" transactions, with ACID properties (Atomicity, Consistency, Integrity, Durability).

Cooperation (RDS-14)

The users will do collaborative work on shared structures, and the system should give support for their cooperation and coordination. Support for private and shared workspaces with a check-out/check-in mechanism is desirable.

Logging, Backup and Recovery (RDS-15)

There should be facilitates for doing recovery after crashes on both server machines and client workstations. Garbage collection of non-referenced objects should also be supported.
Data Interchange (RDS-16)

A facility for sequential streaming of data to a file is useful for data interchange by communication or physical media. This includes the need to transport schema and data. This might also include a transformation-facility for the mapping to different data interchange standards.

3.4.3 Multi-model Data Integration

![Diagram of Multi-model Data Integration]

Figure 3.4: Multi-model data integration

Definition 3.4 Multi-model data integration:
Multi-model data integration is the extent to which tools are able to access and share common data and information that are stored and manipulated through multiple data models with corresponding multiple storage services. □

Multi-model data integration is needed by user task support to transparently read and modify information from multiple sources, and by workprocess to possibly interface to existing process-planning systems and databases.

Figure 3.4 is a reference model for multibase architectures derived from [SL90]. To cope with heterogeneous systems, the traditional 3-schema architecture has been extended to a 5-schema architecture. Figure 3.4 shows how each component DBMS have a local schema.
in its local datamodel, which will be transformed to a component schema in a canonical datamodel. Different export schemas might be derived from the component schemas and be imported into federated (integrated) schemas. Based on a federated schema, different external schemas might be made for different users/user-groups. The task of schema translation involves transforming a schema describing data in one data model into an equivalent schema describing data in another data model. The task of command translation uses the result of schema translation to translate commands involving the schema objects of one schema.

The logical system architecture describes some necessary parts of a total system architecture. Note that their separation is logical, and that in practice some of the parts might be combined.

The purpose of a Filtering processor is to constrain the commands and associated data that can be passed to another processor. This can be done by syntax-checking of commands, by semantic integrity checking, or by access-control. A filtering processor can also be used to handle the way a view-update is to be done.

The purpose of a Constructing processor is to merge data produced by several processors into a single set, and to partition and/or replicate an operation submitted by a single processor into operations that are accepted by two or more processors. A constructing processor may support location, distribution and replication transparency for a processor submitting a command. Other tasks that might be handled include: schema integration, negotiation of protocol for interaction among owners of various schemas, query decomposition and optimization, and global transaction management.

The purpose of a Transforming processor is to translate schemas and commands from one language to another language. They provide a data model transparency by hiding differences in data formats and command language. To perform these transformations, a transforming processor need mappings between the local and canonical data models.

The task of schema translation involves transforming a schema describing data in one data model into an equivalent schema describing data in another data model. The task of command translation uses the result of schema translation to translate commands involving the schema objects of one schema.

### 3.4.4 Requirement Areas for Multi model Data Integration (RDM)

<table>
<thead>
<tr>
<th>Requirement-areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDM-1: Canonical Data Model</td>
</tr>
<tr>
<td>RDM-2: Schema Mapping</td>
</tr>
<tr>
<td>RDM-3: Schema Integration</td>
</tr>
<tr>
<td>RDM-4: Query Processing</td>
</tr>
<tr>
<td>RDM-5: Transaction Management</td>
</tr>
<tr>
<td>RDM-6: Security</td>
</tr>
<tr>
<td>RDM-7: Autonomy and Evolution</td>
</tr>
<tr>
<td>RDM-8: Functional Components</td>
</tr>
</tbody>
</table>

Table 3.4: Requirement Areas for Data Integration - Multi Model

A goal of an integration framework is to permit the incorporation of existing software products and the incorporation of new components as technology advances. In particular, this is true for storage systems: The integration of existing storage tools, traditional file
3.4 Requirement Areas for Data Integration Facilities

systems, relational database systems, object management systems, and future data storage systems has to be supported. Some different storage system categories are:

- File/Record-based model
- Network/Hierarchical model
- Relational model
- Entity-Relation-Attribute model
- Structurally Object-Oriented model
- Behaviourally Object-Oriented model
- Fully Object-Oriented model
- Extended Relational model
- Functional Data model
- Logic, rule-based or Deductive Database

Table 3.4 shows 8 requirement-areas which have been derived from general requirements for multi-model data integration and the reference model in figure 3.4.

A more detailed description of each requirement-area is given in the following:

Canonical Data Model (RDM-1)

In the article "Suitability of data models as canonical models for federated databases" [SCG91] it is argued that a canonical data model should have the following properties:

- **Expressiveness:** A Canonical Data Model should have a semantic expressiveness equal or greater than any of the native models of the component DBs that are going to interoperate.
  - *Ea:* Classification/instantiation: Classes and Instances should be supported.
  - *Eb:* Generalization/specialization: Superclasses - Subclasses with multiple inheritance should be supported.
  - *Ec:* Aggregation/decomposition: Cartesian aggregation and Cover aggregation (grouping) to form complex objects should be supported.
  - *Ed:* Definition of encapsulated behaviour: This helps in the schema integration process, because it allows for hiding of source(s) of data and operations, their local implementation(s) and the solutions given to semantic conflicts.

- **Semantic Relativism:** The semantic relativism is the degree to which a model can accommodate different conceptualizations of the same real world. The power of a model to derive external schema from its database schema is therefore the measure of semantic relativism of the model. We have seen in the reference model in figure 3.4 that there is a need for constructing a federated schema, and a need for deriving external schemas.
3.4.4 Requirement Areas for Multi model Data Integration (RDM)

- **Sa**: Integration operators: There should be integration operators with similar power to the operators meet, join, add subtype and upward inheritance (generalization). This is useful for making external schemas or views.

- **Sb**: External schemas generated with power similar to relational algebra: There should be a view mechanism that can produce any kind of structures available in the model.

- **Sc**: One basic structure: It is simpler to integrate with only one basic structure, such as functions in the functional model, or objects in the object-oriented model. Two constructs, such as entities and relationships in the ER-model augments the already difficult process of schema integration.

- **Sd**: Support for multiple semantics: This can be supported by having multiple federated schemas. Different users might need different conceptualizations of the world.

An analysis of different data models such as the relational model, the Entity-Relationship model, the functional data model and object-oriented models based on these requirements [SCG91] concludes that the most suitable data model are functional models and object-oriented models supporting views.

**Schema Mapping and Transformation (RDM-2)**

There is a two-way process for schema-mapping and schema-transformation. The first direction is mapping upwards from a schema in the local data model to a schema in the canonical data model. The canonical model will typically have more semantic expressiveness than the underlying local models. This leads to the need of enriching the canonical schema through a knowledge acquisition process. Methods for doing this have been explored in [CS91]. Such knowledge makes the later schema integration process less difficult.

The other direction is the transformation of operations on the canonical data model to equivalent operations on the local datamodel(s).

**Schema Integration (RDM-3)**

Schema Integration is necessary when two individually developed databases need to be integrated.

All the different architectures have a common problem when it comes to actual integration. There has to be a common UOD (Universe of Discourse) among the tools which shall be integrated 1.

The idea of standard schemas/class descriptions might as well be used in databases as for object-models. However, the integrity of an object model with respect to the separation of an external interface and an internal implementation gives more than a separation between conceptual schemas and physical schema in a database. Views can be used to achieve some of the same effect, but the integrity-protection is higher with a service-specific interface, than with a generic data-model interface. The same effect can be achieved in a database by stored procedures, and the two models are getting close if the database is viewed as a set of functional application-specific stored procedures.

1Note that it here is an interesting duality between data integration and control integration. In both cases different tools need to understand the UOD for the area where they will interoperate. From this perspective it makes no difference whether that UOD is described through the schemas of a database or through the semantic description of object interfaces.
3.4 Requirement Areas for Data Integration Facilities

Many techniques for schema integration have been reported in the literature. The survey paper in [BLN86] discusses and compares 12 different methods. It divides schema (view) integration activities into 5 steps: preintegration, comparison, conformation, merging and restructuring.

Reverse engineering technology is applicable particularly for the mapping from local schemas to a schema in the canonical object model. E.g. mapping from relational schemas to an ER-model can be automatically done when knowing about primary and secondary keys. The main problem of schema integration is to resolve and cope with semantic heterogeneity.

Query Processing (RDM-4)

Query processing involves converting a query against a federated schema into several queries against the export schemas, and executing these queries. Query optimization should consider various cost-factors when making a query evaluation plan.

Transaction Management (RDM-5)

Supporting global transaction management in a traditional way in an environment with multiple heterogeneous and autonomous component database-systems is difficult. Due to the existence of local transactions, it is very difficult to recognize when the execution order differs from the serialization order at any site. Proposed solutions are implying certain restrictions, such as weaker consistency criterias.

In recent research transaction-mechanisms like: S-transactions (semantic transactions), Quasi-serializability, Sagas [MS87], open nested transactions [Wei91], and Compensating Transactions has been developed.

Security (RDM-6)

Access-control can be done at various levels, e.g. both by a filtering processor for the federated schema, and directly in the component databases. The Mermaid system [T+87] support four levels of access-control.

Autonomy and Evolution (RDM-7)

Different kinds of autonomy need to be supported:

- **Local operation autonomy**: Local applications should continue to coexist with new applications on the federated level.

- **Communication autonomy**: means that a system freely decides when to communicate with another system, and when not to communicate.

- **Execution autonomy**: means that a system freely can decide whether and when it acts upon a request sent by another system, or whether it even ignores it.

- **Schema autonomy**: means that the underlying system can decide to change its local schemas.

Full autonomy for underlying systems comes in conflict with the requirement for consistency, so a trade-off approach needs to be taken.
In the MSQL multi-database-language [Lit85] there is some support for allowing schema-changes at the local system level, without changing access-programs at the federated level. A set of databases participating in a multi-database program can be determined dynamically, and referred to by one name. A schema-entity name can be stated as a pattern that is matched against local schema-names, such that some variations can be catered for.

Functional Services as Local Components (RDM-8)

In a large organization it might not always be desirable to export the total power of a data-manipulation language from a database-system. For integrity and security reasons it might be useful to instead offer a functional interface of operations which directly offer some services, and which itself takes care of integrity of data. For instance, a service which transfers money from one bank-account to another will include a debit-transaction on one account and a credit-transaction in another account. Instead of allowing external clients to do this by a sequence of operations, this is already encapsulated into a provided service.

This might be easily supported by "stored procedures" as they exist in some relational database-systems. This also have the extra advantage of better performance, through the use of precompiled queries. We do think that this will become more important in future systems.

If this encapsulation-view is taken, it becomes similar to request-oriented control integration, described in section 3.3.1.

3.5 Requirement Areas for Presentation Integration Facilities

![Diagram of Presentation Integration]

Presentation integration can be divided into the two areas, display-oriented and model-oriented presentation integration. This separation is based on experiences from the ESF
project, where this separation was applied on the component level, through a separation of User Interaction Components (UICs) and functional Service Components (SCs). This separation was again motivated by the separation of model and view/controllers in the Smalltalk Model-View-Controller architecture, that successfully had been applied in the earlier EKI-project at SINTEF SI.

**Definition 3.5 Display-oriented Presentation Integration:**
Display-oriented Presentation Integration is the degree to which a common look-and-feel is provided by the tools which are used. □

Display-oriented Presentation Integration is needed by user task support to present all tools in a uniform way, by work process support to visualize process-models and to support interactive enactment.

**Definition 3.6 Model-oriented Presentation Integration:**
Model-oriented Presentation Integration is the degree to which the functionality presented through the display is accessed and combined from one or more underlying functional models. A model is here defined as a functional part of a system, without any knowledge of input and output devices. □

Model-oriented Presentation Integration allows for multiple user-interfaces for one functional part of a tool, and for one user-interface to interact with multiple functional tool-parts. This is needed by user task support to facilitate multiple role-dependent views on the same functional model.

Figure 3.5 illustrates a logical architecture for presentation integration. A display- or windowing-system is separated from the actual user-interface of a tool. The user-interface is itself separated from the functional model-part of a tool. The functional model-part is separated from a underlying data-storage system.

### 3.5.1 Requirement Areas for Presentation integration

<table>
<thead>
<tr>
<th>Display-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPD-1: Look-and-Feel and Portability</td>
</tr>
<tr>
<td>RPD-2: Dialogue-support</td>
</tr>
<tr>
<td>RPD-3: Graphics-support</td>
</tr>
<tr>
<td>RPD-4: Interactive Tools</td>
</tr>
<tr>
<td>RPD-5: Copy/Cut and Paste</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPM-1: Model-interaction</td>
</tr>
<tr>
<td>RPM-2: Multi-Model-interaction</td>
</tr>
<tr>
<td>RPM-3: Change/Update</td>
</tr>
</tbody>
</table>

Table 3.5: Requirement areas for Presentation integration

Table 3.5 shows 5 requirement-areas for dialogue-oriented presentation integration, and 3 requirement-areas for model-oriented presentation integration.

The high-level goal of presentation integration is to provide a uniform user-interface for the user. This puts a set of requirements on the system-designer and tool-builder. A particular goal is to take advantage of a split between functional services and presentation services,
3.5.1 Requirement Areas for Presentation integration

so that modern user interface development tools might be used. The major requirement is thus to have an infrastructure where it is easy to support a logical separation between the user-interface part and functional part of a tool.

In the following we describe requirement-areas for both display-oriented (RPD) and model-oriented presentation integration (RPM).

Look-and-Feel and Portability (RPD-1)

We see an increasing use of graphical user interfaces. Different windowing environments have slightly different look-and-feel guidelines, but a typical requirement is that one user would like to use a set of tools that support the look-and-feel style he has decided to use.

This puts a requirement on tool-builders to develop user-interface software that is portable across different window systems. This means that one should write the user-interface once, and it will be transparent for the code which underlying window-system it is attached to.

Dialogue Support (RPD-2)

The largest part of modern window-based applications is support for the interaction-dialogue between the user and the system. Dialogue support includes mechanisms for creating windows, menus, dialogue-boxes etc. to support such an interaction-dialogue.

Graphics Support (RPD-3)

Many user-interface building tools have little support for graphics modeling and presentation, though this might be quite useful in many applications.

Interactive User-interface Development-tools (RPD-4)

It seems quite obvious that a user-interface-dialogue should be constructed through the use of interactive tools. Lately such tools have come into more common use, and been more open for customization.

Copy/Cut and Paste (RPD-5)

When different tools are using different windows it is useful to support cut/copy and paste between the windows.

Model-interaction (RPM-1)

Modern application development style is to split the presentation-part and the functional part of an application from each other. This makes it possible for two different developers to work independently of each part. It also makes it easier to reuse the functional part for different user interfaces, and for one user-interface to utilize different functional services in the same user interface.

We therefore advocate that good support for presentation integration should facilitate and support a split between the user-interface-part and the functional part of a tool, as
3.6 Architectural Support for Process Integration Facilities

illustrated in figure 3.5, it should then also give good support for interaction between the two parts. One way of achieving this is to view this as two interacting programs.

Multi-Model-interaction (RPM-2)

It is often quite useful to access multiple functional models from one user-interface. It is then a requirement to be able to support simultaneous interaction with multiple models. A possible way of doing that is also to create a new model which takes on that responsibility.

Change/Update Support (RPM-3)

The split between the user-interface-part and functional part of an application, creates a possibility to have multiple different user-interfaces for one functional part. This creates the problem of keeping different independent presentation-parts in synchronization with changes in one common functional part.

3.6 Architectural Support for Process Integration Facilities

This thesis do not aim at providing solutions for work process support or process integration. According to the ESF Architectural Reference model in section 2.4.2 such solutions can be provided by process interaction engines in interaction with process interworking engines. It is a goal, however, that the solutions for interoperability and integration provided here can provide the architectural support needed for process integration facilities.

According to discussions at the 7th International Software Process Workshop [CLJ91], the following is a list of some properties and architectural characteristics that were cited as being supportive of Process Sensitive Software Engineering Environments (PSEE).

- Independence of communication and tool functionality
- Message broadcasting as a means of component communication
- Separation of user interface and tool functionality
- Concurrent execution and control (multi-user and multi-process)
- Possibility of intercepting a tool's interaction with other PSEE components
- Rich data model to represent process and product
- Data and process distribution
- Transaction support
- Security
- Multiplicity of heterogeneous process engines
- Distribution of process engines
All of these can to some extend be found in the requirement areas for control integration, data integration and presentation integration.

A recent workshop on "Architecture for Process Sensitive Software Engineering Environments" [PR92] identified three different architectural approaches. These are further discussed in section 4.1.5.

- Architecture 1: Logically centralized database
- Architecture 2: Direct Agent to Agent connections
- Architecture 3: Bus-oriented communication medium and protocol

Architectural Support for these architectures can be provided by data integration facilities for architecture 1, request-oriented control integration for architecture 2, and notification-oriented control integration for architecture 3.

The ESF architectural approach described in section 2.4.2 shows the need for architectural support for the interoperation between process interaction engines and process interworking engines.

This interoperation will be quite similar to the interoperation between user interaction components and service components and can be supported by facilities for control integration and data integration. In particular notification-oriented control integration is assumed to be a useful service for the notification about events.

It is important to note that process integration facilities can offer support for advanced transaction-mechanisms needed by control integration and data integration.

### 3.7 Conclusion on Integration Requirement Areas

This chapter have described requirement-areas for control, data and presentation-integration in more detail. It has also given a short description of requirements for architectural support for process integration.

Chapter 4 will look at different solution-approaches and how they support the requirement areas presented here. These requirement areas will be used as a basis for the requirements for the development of the COOP integration framework in part III.
Chapter 4

Evaluation of Systems Integration Frameworks

This chapter will present and evaluate different solution-approaches with respect to the requirement areas given in chapter 3.

4.1 A Taxonomy of Solution-approaches for Systems Integration

The architecture of integration frameworks has gone through a historical development from independent tools interacting through files, through database-centered environments, and towards more open communication-oriented frameworks. Figure 4.1 shows the evolution-path for integration framework architectures.

4.1.1 Taxonomy

The framework-oriented architecture in figure 4.1 shows that a framework contains services for distributed computing, database support, and user interfaces. These three solution-areas are respectively addressing control integration, data integration and presentation integration. A process engine might be included as a component supporting process integration.

There is a current trend towards Process centered or Process sensitive Environments, providing work process support as described in chapter 2. It is, however, unclear if these environments require a different architectural support than what might be provided by framework-oriented environments [PR92].

The next four subsections will describe individual solution-approaches for control integration, data integration, presentation integration and process integration. The rest of this chapter will present some more comprehensive systems integration frameworks, as representative examples of different approaches.

4.1.2 Distributed Computing approach to Control Integration

The introduction of the remote procedure call paradigm (RPC) in 1981 at Xerox PARC [BN84] was a significant step towards well-structured distributed systems.
A remote-procedure call looks like a local one. The fact that execution actually takes place in another address space, is hidden from the programmer through a stub-routine. A stub-routine makes the connection with a remote server that shall execute the procedure, and packs/unpacks parameters to/from a transportable format. A stub on the server-side unpacks the parameters, executes the routine and returns the result. A stub-compiler can automatically generate necessary stubs for both client-side and server-side, possibly in different languages, based on a description of the signatures of a set of procedures (Interface Definition).

RPC-systems have been commercially available from different vendors since around 1986. (Apollo NCS and Sun RPC/ONC). A problem so far has been that these systems have been tightly connected to specific platforms, and not been available for interaction among heterogeneous systems. This has been changed by the recent introduction of the OSF/DCE Distributed Computing Environment and the Sun RPC/ONC being available on more platforms.

Open Software Foundation (OSF) has decided on the technology to use in its Distributed Computing Environment (DCE), [osf90]. The main component is a RPC service based on NCS from Apollo/HP with extensions from Digital. The technology was released in
1992. This RPC-mechanism has also been adopted by Microsoft as Microsoft RPC in the Windows NT operating system.

ATLAS is the competing alternative to OSF/DCE from Unix International [Hub91]. Although it somewhat complements OSF/DCE and includes interoperability access to DCE, it has for instance its own RPC-mechanism and a file-system based on Sun RPC and NFS.

Management-environments with higher-level services for distributed systems are being built in a number of European research-projects, like ANSA [ans89]. Work on Open Distributed Processing in ISO (ISO/ODP) has adapted a number of concepts from ANSA, in particular the notion of trading for getting a connection to a component that provides a service that matches with a requested service.

Network Management is another area which might provide solutions for control-integration. In particular with regard to the management of system components. The OSI Management model [IS91] has defined a framework for "Managed objects", based on an object-oriented model. OSF/DME, which is the Distributed Management Environment to be provided by OSF, follows up on this model.

Distributed system facilities, like RPC, are useful for realizing request-oriented control-integration. Notification-oriented control-integration might be realized by an event notification service component. RPC-systems are synchronous and will be waiting for a reply for a call. It is possible to achieve a higher degree of concurrency by supporting a message-facility that does not wait for replies. Replies are instead sent as reply-messages back to the requestor [Agh86].

What is lacking in the current systems are powerful development-tools to make the use of such mechanisms more transparent. In particular there are possibilities to use the mechanisms as a basis for an object-oriented interaction mechanism. The recent article "Object Orientation in Heterogeneous Distributed Computing Systems" [NWM93] gives a good overview of this area.

4.1.3 Database approach to Data Integration

There are a number of different database systems that might be used for single-model data integration. The PCTE OMS is a possible candidate, which is described more closely in section 4.5.2. The potential use of object-oriented database-systems is described in chapter 7.

This section will take a closer look at solution-approaches for multi-model data integration. Solutions for multi-model data integration are often called "Multi-databases" or "Federated Databases". This is currently a very active research-area [KRS91, LA96]. Figure 4.2 provides a taxonomy for different solution-approaches, that is derived from [SL90].

Autonomous means that the existing database-systems might be accessed locally. Heterogeneous means that the different database-systems to be integrated might have different data-models (e.g., relational, network, or object-oriented). Loosely coupled means that the users can be involved in creating views, and can integrate from different sources, while in a tightly coupled system integration is only done by the database-administrator. Single federation means that there is one total global schema, while Multi-federation means that different users might do integration differently.

See also figure 3.4 in section 3.4.3.

The focus for integration frameworks is on loosely coupled, autonomous and heterogeneous database systems. Typical of a loosely coupled federated database is a multi-database-
4.1 A Taxonomy of Solution-approaches for Systems Integration

Figure 4.2: Taxonomy of solution-approaches for multi-model data integration

language which provides explicit mechanisms for the management of data location, distribution and replication in multiple databases.

Historically research in this area has made a distinction between multidatabases and federated databases.

Definition 4.1 Multi-database:
A Multi-database is a collection of databases provided with a multi-database manipulation language (MML), and eventually a language for defining interdatabase dependencies. The MML enables the formulation of operations involving jointly several databases. The idea in multi-databases is that the databases to be managed together will have no global schema.

The term federated database system was introduced in [HM85a] to mean "collection of components to unite loosely coupled federation in order to share and exchange information" and "an organization model based on equal, autonomous databases, with sharing controlled by explicit interfaces".

Definition 4.2 Federated Database:
A federated database is a collection of databases with export schemas and with no global schema. Different users might define a view by importing different export schemas. The federated database approach is not supporting a global schema either. The Multidatabase approach has an MML as a basic concept, while a Federated database rather considers only multidatabase views, called import schemas.

There are currently no commercial multidatabase systems, but a number of research-prototypes have been developed: Most of them are tightly coupled, and are using the relational data model as the common datamodel (Mermaid, ADDS).
### Various tool-approaches to Presentation Integration

<table>
<thead>
<tr>
<th>Interface Aspect</th>
<th>Unix</th>
<th>DOS/OS-2</th>
<th>Macintosh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look and Feel</td>
<td>OSF/Motif, Open Look X, Desktop</td>
<td>Windows/PM-CUA New Wave, OS2-Drag</td>
<td>Mac Multfinder</td>
</tr>
<tr>
<td>Interactive Tools</td>
<td>UIMX, TeleUSE ST-80, CommonV</td>
<td>CASE:W/PM ST-80, CommonV</td>
<td>Prototyper ST-80, CommonV</td>
</tr>
<tr>
<td>OO Frameworks</td>
<td>XT, XView XGKS, PEX</td>
<td>Windows-C, PM-C</td>
<td>Mac Toolbox</td>
</tr>
<tr>
<td>Toolkits</td>
<td>X Window</td>
<td>Windows, PM</td>
<td>Mac</td>
</tr>
<tr>
<td>Graphics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Window System</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: User-interface tools in different areas

- **Multibase [LR82]:** was one of the first multidatabase-systems. It is a tightly coupled heterogeneous systems with multi-federations. The canonical data-model is based on the functional Daplex-model [Shi81]. The functional model facilitates the integration of heterogeneous data models. The system is aimed at queries only.

- **MRDSM [Lit55]:** is the most well-known loosely-coupled system, with a relational multidatabase language (MSQL) as the most important feature. The canonical data-model is relational, and it has been aimed at the integration of relational databases, thus it is categorized as a homogeneous system.

- **Mermaid [T+87]:** is a tightly coupled system based on a relational canonical model.

- **Pegasus [KLK91]:** is a loosely coupled system based on a functional object-oriented canonical model. The object-oriented model facilitates the integration of heterogeneous data models.

The object-oriented approach taken in Pegasus seems promising, and has also been advocated by others [BNPS89, KDN90, HMZ90, Man89, CT91, LM91].

Section 4.6 we will describe Pegasus in some more detail.

### Various tool-approaches to Presentation Integration

A number of alternatives exist for the realization of user-interfaces.

Some tools in the different categories are shown in table 4.1. They will typically form parts of the Presentation integration services of a Systems Integration Framework. In particular note that the Object-Oriented frameworks are portable across the different platforms.

- **Look and Feel:** This is the user-interface style, where different “standards” exist. There are multiple choices, such as OSF/Motif or Open Look on the different platforms.

- **Desktop:** This is the metaphor for how the user interacts with the system. Alternative metaphors such as “information-room” might also be used.

- **Interactive Tools:** These are tools for interactively creating a user interface, and typically saves a lot of alternative coding-time.

- **Object Oriented Frameworks:** These are object-oriented application frameworks.
• **Toolkits:** These are the normal procedural toolkits for programming a window system.

• **Graphics:** These are packages which support 2D and 3D graphics. Tools for modern window systems typically support dialogue-design and window-interaction, but are not very sophisticated for "old-fashioned" graphic modeling and presentation.

• **Window System:** These are different window-systems which exist on different or on the same platform, e.g. both X-Window and NeWS are alternative window-systems running under Unix.

Requirements for display-oriented presentation integration are met by the toolkits, object-oriented frameworks, and interactive tools.

Requirements for model-oriented presentation integration might be met by these tools, if the underlying system design supports the split between the user-interface part and functional part of tools.

### 4.1.5 Various Approaches to Process Integration

Process integration is typically supported through a set of interoperating process engines, where some of these are supporting individual users while others are responsible for the coordination between users. This has been exemplified by the ESF process interaction engines and the ESF process interworking engines described in section 2.4.2, and further described in section 4.7.6.

The process engines might be based on different modeling and enactment approaches. Some different approaches are [CLJ91]:

- **The Active DB approach (Triggers/Notifiers):** The basic concept here is Event-Condition-Action (ECA) rules. When an event occurs, an action is done if the condition is true. An example of such as system is Adele-2 [BE86, BE87].

- **The AI or rule-based approach:** The basic concept here is to use forward and/or backward chaining of rules, which might be organized around a common blackboard. Some example systems are Marvel [KFP88], Oikos [ACM90] and Alf [B+89a].

- **The graph/net-based approach:** The basic concept here is to model the process as a horizontal task network, with synchronization constructs and concurrency. A Petri-net approach has been used by many [BG90, EJP+91]. Some existing systems are SPECIMEN [Sch90], MELMAC [DG90, DGS89].

- **The process programming approach:** The basic concept here is to model the process as an abstract process program. The program has concurrent threads with synchronization facilities. Some existing systems are Arcadia [WWRT91] and IPSE 2.5 [War89].

- **The hybrid approaches:** The hybrid approaches combines different aspects from the approaches described above [Sch91], and are the most common, even if they might be categorized after their main approach. Some examples are SPECIMEN [Con89] (merges Petri-Nets with a rule-based language), MELMAC [DG90], EPOS [CWM+89] (a task-network with rules and object-oriented constructs), and SPELL [CJM+92, Maz92] (a further development of EPOS in an object-oriented direction).
A recent workshop on "Architecture for Process Sensitive Software Engineering Environments" [PR92] identified three different architectural approaches:

- **Architecture 1: Logically centralized database**
  - All agents are integrated through a database: single logical point of control for communication/coordination.
  - Very often data is also stored in a central repository as well
  - Examples are: Marvel [KF87], PREIS, EAST [Bou90]

- **Architecture 2: Direct Agent to Agent connections**
  - Decentralized communication and coordination
  - Each agent is aware of whom it is communicating with
  - Possibly different protocols for different pairs of agents
  - Examples are: ARCADIA [WWRT91], ESF Kernel-1 [Fer93]

- **Architecture 3: Bus-oriented communication medium and protocol**
  - Used in a multicast/broadcast setup
  - Agent initiating communication does not know destination
  - All agents use the same medium and protocol
  - Examples are: HP Synervision and ESF Kernel-2R [Gru91]

These three different architectural approaches might all support process engines with different modeling and enactment approaches.

Process modeling is a highly active research field. Most research and development so far, have been done in the area of software process modeling. Current research issues are how to support flexible modeling, evolution and distribution. There is also a growing interest for extending this to other areas as well.

There is an interesting future in merging workflow support products from office information systems, and tools for computer-supported cooperative work (CSCW) with process modeling tools for integrated environments. A number of products for workflow- and process-support have recently appeared on the market. Some examples on these are ProcessWeaver [Fer93] from CAP, ProcessIT from NCR, ProcessWise from ICL, Action for Lotus Notes and others.

### 4.2 Evaluation of Systems Integration Frameworks

Systems Integration Frameworks are normally addressing the integration areas to various degrees. Even if most frameworks are addressing all areas, they often have chosen either control-integration or data-integration as the major integration area.

A number of different approaches has been described in detail in [BA91, Ber90a, H+89b]. Here some selected frameworks are presented, in order to illustrate the different approaches:

The following frameworks have been selected, because of their approaches and basis:

- **OMG** (basis in request-oriented control integration)
• HP/Softbench (basis in notification-oriented control integration)
• PCTE (basis in single-model data integration)
• Pegasus (basis in multi-model data integration)
• ESF (basis in request-oriented control integration, presentation and process integration)

4.3 The Object Management Group

4.3.1 The OMG Activity

A more detailed description of OMG can be found in [Ber91f]. OMG was formed in 1989 by 8 members: Hewlett-Packard/Apollo, Unisys, Data General, American Airlines, 3Com, and Canon. By the mid of 1993 there are more than 300 members of OMG, including DEC, Apple, ATT, Borland, Bull, ICL, NCR, Nixdorf, Olivetti, Prime, Wang, Xerox and the majority of companies delivering object-oriented technology. NTH and SINTEF are among the Norwegian members. Microsoft joined as member in March 1991, and IBM in June 1991.

The goal of OMG is to establish industry guidelines and object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop heterogeneous application environments across all major hardware and operating systems. The OMG defines object management as a basis for software development that models facts or phenomena through representations of "objects"¹. These objects are the encapsulation of attributes, relationships and operations of software identifiable program components.

To accomplish this goal the OMG is focusing its efforts in the following areas:

• Definition of terms
• Specification of an Object Request Broker
• Specification of an Object Model
• Specification of Object Services
• Special Interest Groups on various OO themes

The specification work is done by a RFP (Request For Product)-process where vendors answer a request by providing specifications of products that satisfy the given requirements. The specifications will, if selected, be publically available as an OMG-specification. A task force is responsible for the administration and selection-process for each RFP. Currently there are three active task forces: Object Services, Revised Object Request Broker, and Object Model.

4.3.2 The OMG Architecture

The OMG Architecture shown in figure 4.3 is called the "Object Management Architecture - OMA". The initial version is described in [omg90], a later version in [Obj91a].

¹Chapter 5 gives an introduction to object-oriented concepts and terminology
application which is "OMA"-compliant consists of a set of interworking objects that interact via the Object Request Broker. The Object Request Broker component of the Object Management Architecture is the kernel of the standard. It provides an infrastructure allowing objects to communicate, independent of the specific platforms and techniques used to implement the addressed objects. The Object Request Broker component guarantees interoperability of objects over a network of heterogeneous systems.

![Object Management Architecture Reference Model](image)

Figure 4.3: Object Management Architecture Reference Model

The Object Management Architecture shown in figure 4.3 has four major parts:

- **The Object Request Broker (ORB)** provides the mechanisms by which objects transparently send and receive requests and responses. In so doing, the ORB is providing interoperability between applications on possibly different machines in heterogeneous distributed environments and seamlessly interconnecting multiple object systems. See section 4.3.3.

- **Object Services (OS)** are a set of services which will extend the functionality of the Object Request Broker.

- **Common Facilities (CF)** comprise facilities that are useful in many application domains and which are made available through OMA-compliant object interfaces. Unlike Object Services which will be supported on all platforms, Common Facilities are optional.

- **Application Objects (AO)** correspond to the traditional notion of application. Generally developed by VARs (Value-Added-Resellers) and ISVs (Independent Software Vendors), they represent individual related sets of functionality (e.g., word processing, ...). The ability to integrate application classes also extends to the integration of traditional, existing, non-object-oriented applications within the same framework.

### 4.3.3 OMG Support for Control Integration

Request-oriented control integration is supported by the Object Request Broker and associated Object Services. There are plans for a future inclusion of an event-notification-service as an object service, in order to support notification-oriented control integration.
The selected Object Request Broker is called CORBA - Common Object Request Broker Architecture and Specification. CORBA is a merged technology from the two submitter-consortiums (HP-SUN-NCR-ODI and DEC/HyperDesk). Implementations of the merged specifications are expected from Digital, HyperDesk, Sun, HP, NCR and ODI by the end of 1993. CORBA is announced to be a bundled part of the Solaris 2.3 operating system from Sun.

The CORBA Architecture

![CORBA Architecture Diagram]

Figure 4.4: CORBA - Common Object Request Broker Architecture

The CORBA Object Model defines an object request and its associated result (response) as the fundamental interaction mechanism. An object-request names an operation and includes zero or more parameter values, any of which may be object references. The ORB arranges for the request to be processed. This implies identifying and causing a procedure to be invoked that performs the operation using the parameters. After the operation terminates, the ORB conveys the results back to the initiator of the request.

The basic elements in CORBA is the ORB Core (the communication infrastructure), with a set of Object Adapters for different object implementations. A client will access services through object references. This can be done either through stubs generated from an IDL (Interface Definition Language) specification, or through a dynamic API interface. The static IDL interface and the dynamic API interface are two alternative approaches for object interactions, respectively from the HP/Sun ORB-proposal and the Digital/HyperDesk proposal. Both alternatives are supported in CORBA.

A request will be delivered to the Object Adapter which is responsible for the associated object-reference. The Object Adapter will activate the implementation associated with this object-reference (if it is not already active) and perform the request through server-stubs from an IDL-generated skeleton. Information on interface-types might be retrieved from an Interface repository. An Implementation repository (with information about the actual
implementation) will also be available, but this interface will be ORB-implementation specific. It is possible to have multiple implementations of the ORB. (Typically different implementations will be available from the different vendors. They will have the same basic interface and will be able to interoperate through a gateway, but the different vendors will also supply their own extensions).

An Object Adapter is responsible for the following functions:

- generation and interpretation of object references
- operation invocation
- security of interactions
- object and implementation activation/deactivation
- registration of implementations

The CORBA IDL language is quite similar to RPC-based interface definition languages like OSF/DCE-NIDL (NCS-NIDL) [ncs89] and ANSA IDL [ans89]. The IDL-grammar is a subset of C++. Although only a C-stub interface is currently defined, a C++ interface is requested in a RFP.

CORBA 2.0 will be specified as a result of a new RFP issued in the fall of 1993. The functionality of CORBA will also be extended by the addition of object-services. There is a plan to have multiple RFPs for object services. The first RFP was issued in the fall of 1992, with selection in September 1993. This RFP included request for the following 4 services:

- Object Lifecycle Service
- Object Persistence
- Object Naming Service
- Object Events Service

A proposal called JOSS, Joint Object-Oriented Services has been delivered for this, as a joint effort by 15 different companies, including Sun, HP, DEC, NCR, Bull and a number of object-oriented database-companies

A second RFP for Object Services will be issued in the fall of 1993 with request for the following 5 services:

- Object Concurrency Service
- Object Transactions Service
- Object Relationships Service
- Object Time Service
- Object Externalization Service

Other services, such as the following, will be requested later.
• Object Recovery
• Object Versioning and Configuration
• Object Data Interchange
• Object Query
• Object Security
• Object Licensing
• Object Trading
• Object Replication
• Object Management relocation

As can be seen from this list, many of the requirement areas for control-integration from chapter 3 may be supported by future object services, e.g. RCR-5, RCR-6, RCR-7, RCR-8 and RCN-1/2. Also support for data integration might be included by object relationships and object query (RDS-1, RDS-9).

4.3.4 OMG Support for Data Integration

In parallel with the development of CORBA and the basic CORBA object model, there has been a task force working on a higher level object model.

The goals of this object model are to facilitate portability of applications, reusability of type libraries, and interoperability of software components in a distributed heterogeneous environment.

Some non-goals are: The Object Model is not a meta model for describing other object models. The Object Model has not been constrained to be either a strict superset or a least common denominator of the features of existing OPLs or OODBMSs. The Object Model is not intended to be the 'ultimate' object model. It is acknowledged that it might be difficult to reach consensus on one object model for all purposes, - therefore it is defined as a core and a set of components. Different areas might select different components and form a profile.

The semantic description of the CORE-model was published in July 1992 [Obj92b]. It is a subset of the CORBA object-model, containing the following:

• OIDs (Object Identifiers)
• non-objects - basic types
• operations
• parameter passing

Different areas like user-interfaces, class-libraries, CORBA and object-oriented databases will need different extensions to this basic model. Forming a profile for a certain area is done by selecting additional components.

The CORBA Object model can be described as an object model that is compliant with the CORE Model, and in addition has selected the following components: Non-object types,
4.3.5 OMG Support for Presentation Integration


The extended object model will not have a direct impact before it is casted into a concrete syntax, so for the initial work in the OMG architecture the object model will be the one implicit in CORBA.

Single model data integration will eventually be supported by object model profiles which select components like attributes and relationships. CORBA already supports the notion of attributes. An attribute-definition leads to the definition of two operations, - get- and set- attribute.

Multi model data integration is not yet considered, but might be supported by the use of an extended object model as a canonical model.

4.3.5 OMG Support for Presentation Integration

Presentation Integration is not directly supported through any specification-work done by OMG. There is, however, a special interest group for Object-oriented class-libraries. Class-libraries for user-interfaces and graphics are obviously in the interest-domain for this group, and later work here might lead on to specifications in this area.

4.3.6 OMG Support for Process Integration

OMG has not announced any planned support for process integration. The CORBA architecture, however, gives architectural support for architectures of type 2 (Direct Agent to Agent connections), as described in section 4.1.5. A future event-notification service might give support for architectures of type 3 (Bus-oriented communication medium).

4.3.7 OMG and Integration - Conclusion

The focus of OMG is on request-oriented control-integration, by support for interaction between objects. Later work on object-services might lead to the inclusion of a service for event-notification, in order to support notification-oriented control-integration. Later work on the object-model might extend the current CORBA-model to also include single-model data-integration, (it already does so to some extent, by supporting the notion of abstract attributes), and possibly also multi-model data-integration. Presentation integration might also be addressed in the future.

Many of the major hardware-vendors have announced support for the ORB in future "middleware" products. The COSE consortium, "Common Open Software Environment", with IBM, HP, Sun among others, have announced planned releases of CORBA-based technology.

4.4 HP/Softbench - Field

4.4.1 The HP/Softbench - Field System

The HP/Softbench environment [HP 89] is based on the notification-oriented broadcast-based tool-integration philosophy first developed in the Field programming environment from Brown University [Rei90a, Rei90b]. The architecture of these environments is based
Figure 4.5: HP/Softbench and Field Architecture

on tools interacting through a Broadcast Message Server (BMS). A similar BMS approach has been taken by the Sun ToolTalk [Sun91a, Sun91b] mechanism. Both Softbench and ToolTalk supporters have created vendor forums for agreement on standard message-sets for tool integration, CASE Communique and CASE Interoperability Alliance, respectively. These groups are working together in ANSI X3H6 to standardize on a common messaging architecture [X3H93]. HP/Softbench-Field is here described as a typical example of different BMS architectures.

The difference between HP/Softbench and Field is mainly that HP/Softbench uses HP-developed programming-tools and the OSF/Motif user interface, while Field uses ordinary Unix-tools and a user interface-system developed at Brown. The following description is mainly based on Field-documentation [Rei90a], [Rei90b], but is generally valid for both environments.

4.4.2 The HP/Softbench - Field Architecture

Figure 4.5 shows the HP/Softbench - Field architecture with its central broadcast message server.

The requirements for integration were the following:

- The tools must be able to interact directly with each other.
- The programmer must access the source code through a common editor.
- The tools must share dynamic information.
- The environment must make static specialized information available to all the tools that need it.

4.4.3 HP/Softbench - Field Support for Control Integration

In selective broadcasting, all tools talk to a central message server. Each tool registers a set of message patterns with the server. Tools communicate by sending messages to the server and receiving those messages that match their announced and registered patterns.
The following characterizes the Field architecture:

- Field passes all messages and patterns as text strings of arbitrary length
- Msg incorporates and extracts string and numeric arguments according to pattern.
- Messages can be sent either asynchronously or synchronously.
- The full functionality of the debugger is available through message-based commands which reflects the normal user-commands.
- Each tool defines a set of events of possible interest to other tools and sends messages about these events as they occur.

The event-message facility is used by different tools. The Program Builder sends out messages for each error or warning the compiler detects, so the Program Editor can associate the errors with the program source code. A cross-referencer and flow-graph viewer send out messages when the user clicks on an output reference so the Program Editor can shift its focus to the referenced location. The Program Editor sends out a message, whenever a file is opened or closed.

The actual Msg server runs as a separate Unix process and communicates via sockets. Each tool includes a client interface that talks to the server and distributes the message from the server to the tool. When a tool sends a message, the client interface passes it to the Msg server. The server passes the message to the client interface for each process which have requested to be notified on a pattern matching the message. The client interface decodes the pattern and calls a routine associated with the pattern with the decoded arguments. The call to define a pattern can also set default values for arguments which the pattern does not define directly, so that a common routine can handle many messages.

A difficulty in putting together an environment based on message passing is standardizing the messages. Messages added by new clients must not conflict with existing messages, and their messages must be clearly defined. The set of messages used by Field can be divided into two categories: command messages and informational messages. Command messages are directed toward a specific tool and are generally sent synchronously. Informational messages are broadcasted by a tool to all other tools that might be interested in them. These are generally sent asynchronously.

The underlying message facility provides a simple means whereby a broad collection of tools can be easily combined. Developing an external interface for an existing tool involves two steps. First, one develops a window-based user interface front-end. This must be consistent with other tools in the environment. Secondly, one defines and provides a message interface. This involves registering patterns with the message server, corresponding to those commands that will be offered as a service to other tools. Next step is to generate whatever messages might be relevant to other tools. These messages generally reflect output gathered from the existing tool or state changes in the interface. The experience from Field is that one can build an interface to an existing tool in about a week.

The managing of events is to some extent hardcoded into the explicit registration by the different tools. This architecture can be extended for more flexibility by associating a rule-system with the event-manager (or the broadcast message server). Each event will invoke a rule, which can check conditions and do alternative actions. This rule-base can be independently programmed and used for customizations of the environment. The Forest-system specifies a way of easily extending the flexibility and programmability of Field by the use of a rule-system.
HP/Softbench - Field Support for Request-oriented Control Integration

The support for command messages provides the functionality for request-oriented control-integration. There is no explicit model for interface description. Knowledge about message-interfaces and patterns needs to be handled manually.

HP/Softbench - Field Support for Notification-oriented Control Integration

The support for informational messages provides the functionality for notification-oriented control-integration. This is the major integration-mechanism used by the HP/Softbench - Field system.

4.4.4 HP/Softbench - Field Support for Data Integration

There is no direct support for data integration by HP/Softbench - Field.

To facilitate sharing of static information, a central database could have been used, but Field uses active servers. Active servers are Field components that receive information requests through the message server and fulfill them either by dynamically computing the information, or by caching it in a local database. This message interface lets the various tools get necessary information without needing to know how it is stored.

This means that data-integration is viewed as a special case of request-oriented control-integration, where the data of concern is transported as message-arguments. Multi-model data integration is not considered.

4.4.5 HP/Softbench - Field Support for Presentation Integration

HP/Softbench - Field Support for Display-oriented Presentation Integration

A key ingredient in HP/Softbench - Field is the use of a high-level user interface toolset to make different user-interfaces on top of the functional tools of the environment, separating the user interface and the functional tools. HP/Softbench also provides an interpreted user interface language in order to interface an otherwise textual tool into a window-oriented environment. This is provided through the HP Encapsulator facility.

HP/Softbench - Field Support for Model-oriented Presentation Integration

HP/Softbench - Field make a strict separation between the user interface and functionality of a tool. This is a key ingredient to facilitate model-based presentation integration. It is easy to develop multiple user-interfaces for one functional tool.

4.4.6 HP/Softbench - Field Support for Process Integration

HP/Softbench - Field does not support process integration directly. The architecture, however, gives architectural support for architectures of type 3 (Bus-oriented communication medium), as described in section 4.1.5. The BMS facility has by many been viewed as a good architectural support for process integration, because of its event handling capability, and loose coupling of tools. The HP SynergyVision environment support process integration, with architectural support from the BMS facility.
4.4.7 HP/Softbench - Field and Integration - Conclusion

The focus of HP/Softbench - Field is on notification-oriented control integration, but support is also given for request-oriented control integration and presentation integration. No support is given for data integration, while the BMS facility provides good architectural support for process integration.

4.5 PCTE

4.5.1 The PCTE Project

The major solution for tool-integration put forward in Europe, especially for software-engineering, has been the Esprit-founded PCTE - Portable Common Tool Environment - around its OMS - Object Management System [GMT86, HSE89].

PCTE has been selected as the basis in a number of Software Engineering Environment projects, such as PACT, EAST, SFIX, Enterprise-2 and others. The basic development took place between 1984 and 1987. A number of extensions were proposed by the German PCTE Initiative from 1988 to 1990, and some of these extensions are included into the ECMA-standardization of PCTE.

4.5.2 The PCTE Architecture

![PCTE Architecture Diagram]

Figure 4.6: PCTE Architecture

Figure 4.6 shows the different parts of the PCTE architecture. The heart of the architecture is the Object Management System - OMS. The goal of PCTE is to integrate around the OMS: The OMS is an ER-oriented database for large-grained objects, like source-files, programs, documents; based on an extension of the Unix-file-system.

The other parts of PCTE: Notification/Activities, Distribution and Security are closely attached to the OMS. The UI is to support a uniform user-interface, while EXE/IPC are interprocess communication, modeled after the Unix-mechanisms.
PCTE OMS

The datamodel of PCTE/OMS is a node-link Entity-Relationship oriented model. Nodes have a position in a directory-like hierarchical structure, in addition to the arbitrary links between nodes. Different aspects of the OMS are described in more detail in the evaluation of PCTE/OMS for single model data integration in section 4.5.4.

PCTE ACT/NOT

ACT - Activities: provides concurrency and integrity control facilities. This is a transaction mechanism that provides nested and long atomic transactions, and also protected non-atomic activities and unprotected activities. NOT - Notification: provides a facility for being notified about changes.

PCTE SEC

SEC - Security: Is both discretionary and mandatory access control.

PCTE DIS/COM

DIS - Distribution: provides distribution-facilities for workstations and servers.

COM - Communication: is a mechanism to deal with "unstructured" long-field data objects. It is similar to a UNIX-file with operations for file I/O (open, close, read, write, seek).

PCTE EXE/IPC

EXE - Execution: is a facility that manages the execution of a program in a heterogeneous distributed environment.

IPC - Inter Process Communication: provides message-queues and UNIX-like pipes and signals. A message-queue is modeled as an OMS-object.

PCTE UI

UI - User Interface: is a display-independent user-interface library, with routines for handling of windows, dialogues and menus. It is supported for different underlying windowing systems, X-window in particular.

The main way of doing integration in PCTE is by letting different tools share a common sub-schema in the database.

Compared to the reference-model, PCTE offers a solution for data integration based on a single data storage service (OMS). It offers tools for presentation integration through the UI-dialogue toolkit. The support for Control-integration is weaker, and closely associated with data in the OMS.
4.5.3 PCTE Support for Control Integration

The PCTE IPC-mechanisms provide some basic facilities for doing request-oriented control-integration, but the facilities are on a rather low level.

4.5.4 PCTE Support for Data Integration

The strong point of PCTE is single model data integration. The OMS has a datamodel meeting most of the requirements RDS-1 to RDS-16.

- **RDS-1: Basic ERA-model:**
  PCTE/OMS is an ER-oriented data-model with Objects, Links, Relationships and attributes. Objects can have a contents-part. Attributes can be defined on their own, and be applied to both objects and links.

  The OMS data model offers the following modeling facilities:
  - Persistent objects with attributes, links and relationships.
  - Binary links, representing unidirectional associations between a given set of origin and destination objects.
  - Relationships are pairs of binary links. According to the cardinality of the links in a relationship it is termed one-one, one-many or many-many.
  - Named attributes on objects and relationships. The attribute values are boolean, integer, string and date.
  - Both objects and relationships are typed. Subtypes may be created of parent types, and inheritance of parent properties is supported.
  - Definitions are grouped in self-contained Schema Definition Sets, (SDSs). Common definitions can be imported from one SDS to another.
  - Object-creation is done by "anchoring" the new object with a link to an existing "origin" object.
  - Intended granularity for objects is on the file level
  - PCTE+ extensions to PCTE OMS provides support for complex objects

  The object-types are structured in a type-hierarchy and sub-types inherits attributes and links from its parent-type.

  Navigation uses the path-structure defined by objects and links. For link-types of cardinality "many", its key-attributes are used to find the right link. When the cardinality is "many", the key can be sufficient to name a link, if the corresponding link type has been declared as preferred link-type. Even if the datamodel in itself is a general ER-oriented model, the DDL/DML is strongly reflecting that the stored objects are supposed to be similar to Unix-files.

- **RDS-2: Complex objects:** There is a similarity between the "composition"-relationship and the "is-part-of"-relationship to build complex objects. There is, however, no special operations on composite objects or any support for variants. PCTE/OMS supports only a very restricted version concept, namely automatic numbering of elementary concepts. The meaning of the number must be defined by the user. Older version can be made stable, i.e. further modification are inhibited. Configuration (or equivalent representations) are not supported.

- **RDS-3: User-defined data types:** There is no support for creating use-defined data types, or abstract data types.
• **RDS-4: Prog. language support:** The programming-language interface is through library-calls in C.

• **RDS-5: Versions/variants:** There is no inherent support for versions or variants.

• **RDS-6: Metadata/Schema and RDS-7: Views:** The schema-concepts is handled by the definition of SDS, Schema Definition Sets and WS, Working Schemas. By importing definitions from other SDSs, a SDS can be one view on shared information.

A SDS is a complete, consistent and self-contained subset of the OMS type definition space. SDSs are OMS objects and can thus be accessed according to OMS naming conventions. In addition each SDS has a system unique identifier. A running process has a Working Schema, which is a set of SDSs.

A SDS contains a set of definitions for object-types, links and attributes. Links/relationships are said to be applied to object-types, and attributes applied to object-types and links. A SDS can be extended, and links and attributes can be disappeared.

• **RDS-8: Rules / Triggers:** There is no support for rules or triggers.

• **RDS-9: Query language:** There is no described query-language for PCTE/OMS, this facility is however developed in the PACT-project as an added layer.

• **RDS-10: Security:** The access-control mechanisms are based on concepts from the Unix file-system applied to each object, but not to each link.

• **RDS-11: Distributed Architecture:** PCTE/OMS is aimed at a distributed architecture of networked workstations.

• **RDS-12: High Performance:** The OMS storage philosophy is based on an extension of the Unix file-system, with extension for distribution. No segmentation or clustering besides this is supported. The rather large granularity of objects has led to poor performance on initial PCTE/OMS-implementations.

• **RDS-13: Concurrency:** Transactions and concurrency are handled by the Activity-concept, defined in a separate part in the PCTE documentation. An Activity is the framework in which a set of related operations take place. Each operations is always carried out on behalf of just one activity. An activity always has a set of acquired resources and associated locks which are in all cases held on these resources. One distinguishes three classes of activities: Unprotected, Protected (non-atomic) and Transaction(atomic) activities. A resource is either an OMS objects with its contents and attributes or a link and its attributes.

A lock is a liaison between an activity and a resource and it is characterized by an external mode, an internal mode and a duration. The Lock-mode is a combination of Read/Write-access and Unprotection (concurrently write), Protection(No concurrently write) and Transaction. Nested activities are allowed, but for rollback purposes only the highest level activity is handled. Deadlocks are neither prevented nor automatically detected.

• **RDS-14: Cooperation:** There is no support for long transactions, or for private databases.

• **RDS-15: Logging - recovery:** Archiving and restoration will be handled by separate tools, like in the Unix file-system.

• **RDS-16: Data Interchange:** There are no described solutions for handling external objects or conversions to external representations.

PCTE/OMS offer no support for multi-model data integration.
4.5.5 PCTE Support for Presentation Integration

The PCTE UI-mechanisms give support for display-oriented presentation integration. No
attention is given to model-oriented presentation integration.

4.5.6 PCTE Support for Process Integration

PCTE does not support process integration directly. The architecture, however, gives
architectural support for architectures of type 1 (Logically centralized database), as de-
scribed in section 4.1.5. The EAST project [Bou90] has been using the PCTE OMS to
realize process model support.

4.5.7 PCTE and Integration - Conclusion

The focus of PCTE is on single model data-integration, in a distributed workstation-
environment. No support is given to multi-model data-integration. The PCTE OMS pro-
vides architectural support for process integration based on a logically centralized database.
Basic facilities for low-level control-integration is provided, but the intention is to do all
integration through the OMS. A user-interface library provides a basis for display-oriented
presentation integration.

4.6 Pegasus

4.6.1 The Pegasus System

Pegasus is an object-oriented multidatabase system being developed at Hewlett-Packard
Laboratories [AK+91, AR90, K1K91, R+91, Ken91].

The goal of the system is to provide facilities for multidatabase applications for accessing
and manipulating multiple autonomous heterogeneous object-oriented, relational and other
databases.

The focus of Pegasus is thus in the area of multi-model data integration.

4.6.2 The Pegasus Architecture

Figure 4.7 shows the Pegasus architecture. Expressions in "Heterogeneous Object SQL"
are parsed and split into sub-queries directed to the underlying local DBMSs.

Pegasus takes advantage of the encapsulation-mechanism of object-oriented programming
by hiding the heterogeneous aspects of various systems in the implementation-part of types.
Different external data-bases might be attached to a local "imported db" which contains
the schema of the local data-base represented in the "canonical" object-oriented model of
Pegasus.

4.6.3 Pegasus Support for Control Integration

Pegasus is not focusing directly on control integration. However, its object-oriented data-
model makes it possible to define the interface of systems as types.
4.6.4 Pegasus Support for Data Integration

Pegasus Support for Single Model Data Integration

Pegasus is based on a functional object-oriented derived from the IRIS-system

As a functional object-oriented database it can be used for single model data integration. However, there is currently no high-performance programming-language interface towards the system.

Pegasus Support for Multi Model Data Integration

In the following we comment on how Pegasus meets the requirements stated for multi model data integration:

- **RDM-1: Canonical Data Model**: The canonical data model of Pegasus is based on a functional object-oriented data-model, being an extension of the IRIS-data model from the HP product OpenODB. This model meets the expressiveness requirements stated in RDM-1.
4.6.5 Pegasus Support for Presentation Integration

- **RDM-2: Schema Mapping:** A local-translator mapper component is specified for each individual type of participating database-system that is attached. This component takes care of mapping to and from the Pegasus-model and the model of the underlying system.

- **RDM-3: Schema Integration:** Schema integration is done on the canonical data-model layer. HOSQL provides mechanisms for importing a schema from a participating database. It is possible to define a type as a generalization of two underlying types. It is then possible to give criterias for equivalence between two objects from different sub-types. Techniques for dealing with domain mismatches have been investigated.

- **RDM-4: Query Processing:** Expressions in "Heterogeneous Object SQL - HOSQL" are parsed and split into sub-queries directed to the underlying local DBMSs. A global optimizer can take advantage of knowledge from built-in local translators. Query-optimization techniques are under investigation.

- **RDM-5: Transaction Management:** A Transaction manager is defined. Techniques for global transaction management are under investigation.

- **RDM-6: Security:** Security-issues have not been particularly addressed.

- **RDM-7: Autonomy and Evolution:** There are constructs in HOSQL to allow for name-variations for underlying types, i.e. all types beginning with certain letters.

- **RDM-8: Functional Components:** It is possible to attach also non-database-systems to Pegasus, by describing the interface of the component as a type.

4.6.5 Pegasus Support for Presentation Integration

Pegasus does have an interactive graphical query-tool. It does not provide any other particular support for presentation integration. However, it is possible to interface existing user-interface tools to the system through a programming interface.

4.6.6 Pegasus Support for Process Integration

Pegasus does not have any support for process integration. The system, however, might give architectural support for architectures of type 2 (Direct Agent to Agent connections), as described in section 4.1.5.

4.6.7 Pegasus and Integration - Conclusion

The focus of Pegasus is on multi-model data integration. Pegasus is, however, not a front-end to multiple databases but has its own complete data management capabilities. Thus it also provides for single-model data integration. By having an object-oriented data model it is possible for Pegasus to also give support to request-oriented control-integration.
4.7 ESF

4.7.1 ESF Project

The mission of the ESF\textsuperscript{2} project [Fer91, FkNO92] is to make a factory-environment for creating software, which is adaptable to different organizations and application areas. The project is carried out by a consortium of 14 different companies in 5 different European countries, working within the Eureka program. It started in 1987 and had a major milestone in 1992, with a planned continuation until 1996. A total effort of more than 2000 person-years over 10 years is planned for. The kernel-projects in ESF have focused on establishing the basic architectural modules:

- Software Bus
- Process Modeling
- User Interface Building Tools
- OSS: Object Storage Services

The author participated in ESF from the planning-phase in 1986 until the end of 1990. In particular in the role as technical authority in the sub-projects Software Bus and Object Storage Services

In addition to the kernel-projects there has been a number of application-oriented projects, which have used the architectural foundation when developing specific solutions related to Software Engineering Environments and Factories.

Support for integration in ESF is concerned with three aspects of communication: human-to-human (Interworking or Cooperative Work and Work Process Support/Process integration), human-to-system (Interaction or Presentation integration/Process integration) and system-to-system (Interoperation or Control integration/Data integration), according to the ESF CoRe presented in section 2.4.2. These areas are handled by work on process-modeling, user interaction services, the software bus, and object storage services.

4.7.2 The ESF Architecture

Figure 4.8 shows the ESF reference architecture, this has been introduced earlier in section 2.4.2. The basis of this architecture is a "Software Bus" which supports interaction among

\textsuperscript{2}Eureka Software Factory
software components in a similar way to how a "Hardware Bus" supports interaction among hardware components.

The ESF reference architecture defines an architecture of components interacting through a software bus, [Eur89]. There are two different kind of components, User Interaction Components (UICs) providing presentation services and Service Components (SCs) providing functional services. From a software bus viewpoint they are equal as components being able to serve both as clients and as servers. From a system development viewpoint the distinction is important, in splitting the user-interface part of an application from the its functional service part. This makes it possible to have multiple user-interfaces on one functional service, and to let one user-interface interact with multiple services.

The same architectural principles are applied for process integration, where Process Interaction Engines interact with Process Interworking Engines.

Tools are made as UICs interacting with one or more SCs. Database-systems might be hidden by the SCs, but SCs in need for tight integration might decide to use a common underlying storage service. Both UICs and SCs might act as both clients and servers for other components.

The end-user will have his private, role-oriented user interaction environment, giving him access to those tools he needs in order to fulfill his role. The architecture makes it possible to more easily make role-oriented tools, utilizing common functional services. A tool is created as a coupled set of user-interaction components and service-components. To guide an end-user through sets of role-specific tasks a process-model is used as a guide. The process-model describes the sequence of tasks and activities that each role is supposed to carry out. From an architectural viewpoint, the process-model is handled by a special service-component.

The goal of the software bus is to be the mechanism for interaction among components. It will support the incorporation of components into the Software Factory at installation time and be a high-level communication mechanism at run-time. The ESF Architecture is thus communication-oriented as opposed to the more traditional database-oriented environment-architectures. The aim of the work has been to support integrated software engineering and CASE environments, but the architecture should also work well for integrated environments in other areas.

4.7.3 ESF Support for Control Integration

ESF Support for Request-Oriented Control Integration

Request-oriented control integration is provided by the ESF Software Bus [Eur91a]. The goal of the Software Bus is to support pluggability of components conforming to the same interface. The interoperability among components will be supported if they request and provide services through the mechanisms of the software bus. Existing applications can be attached to an adoptor in order to be plugged into the bus. This approach is object-oriented from the point of view that components might be seen as instances of types described through their interface, a component is a pluggable alternative to another component if it is an instance of the same type, or if it is an instance of a subtype that is conform with the supertype, [RTL91]. One type might have many alternative implementations, conforming to the same interface.

The purpose of the software bus is to provide an interaction mechanism between two or more factory components. A Component is a piece of software, offering and/or requiring certain function(s) called Services. Components offer each other services and request
the performing of a Service by message passing, not by memory sharing. The basis of
Component interactions is the exchange of services. A Service is a set of Service Elements,
primitives of the service which might be individually invoked. The invocation of a Service
Element is known as a Service Element Request. - this is an abstract concept which might
have many concrete realizations. Components communicate by issuing Control Exchange
Statements, which is a programming-language statement that is an implementation of a
Service Element Request.

A Component is described in terms of the Services offered and Services required, as well
as other attributes, by the Component Type Description Language (CTD-L). Services are
abstractly described through the Service Abstract Description Language (SAD-L), and
concretely mapped through the Service Representation Description Language (SRD-L).
For each service in the factory, there is a SAD-L description. There may be more than one
service described in a SA module. The Service Abstract Definition describes the service
itself, and not its implementations. Since there may be more than one implementation of
a given service, an SAD module may refer to more than one SRD module.

From the repository one can generate output for different languages: Ada, C/C++, Prolog
and Lisp

This approach is object-oriented from the point of view that components might be seen as
instances of types described through their interface. A component is a pluggable alternative
to another component if it is an instance of the same type, or if it is an instance of
a subtype that is conform with the supertype, [RTL91]. One type might have many
alternative implementations, conforming to the same interface. This separation between
interface (type) and implementation (class) has also been used in other recent object-
oriented terminology [omg90, ANS92].

ESF Support for Notification-Oriented Control Integration

Notification-oriented control integration is provided by the ESF Event Multicaster, which
is a mechanism supported separately from the Software Bus [Eur91b]. The ESF Event
Multicaster Component (EMC) is a component that supports the communication among
components with broadcasted events, similar to the HP/Softbench - Field-system described
in section 4.4.

A central server (named the Event Multicaster) receives the registration and unregistra-
tion of components. Components send messages without specifying any receiver. These
events are captured by the Event Multicaster. Components subscribe to a set of events,
they define some filters that specify what kind of events that they want to receive. The
event-mechanism defines a procedure-call interface which might be used directly from
components. This event-mechanism might eventually be merged with the SAD-L of the
software bus, in order to describe which kind of events that might be generated from a
certain service.

4.7.4 ESF Support for Data Integration

A subproject for the specification of Object Storage Services, ESF OSS, has been carried
out.

Data storage services is hidden by service-components in the ESF reference architecture. It
is, however, allowed for tightly connected components to use a common underlying storage
system. The need for integration of multiple storage services will naturally arise in large
integrated environments. Both single-model and multi-model data integration has thus
been addressed by the OSS subproject.

The OSS subproject has made a requirement-specification for single-model data integration: The ESF OMS (Object Management System), and for multi-model data integration: The ESF OSS Integration framework [Ber89].

The ESF OMS requirements has been met by different commercial data storage services and systems provided by ESF partners. The ESF OSS Integration framework has not been realized, due to funding-problems.

### 4.7.5 ESF Support for Presentation Integration

Display-oriented presentation integration is supported through various tools for user-interface building. Model-oriented presentation integration is supported through the architectural division between UICs and SCs.

### 4.7.6 ESF Support for Process Integration

The ESF-project has identified process integration as an important area [Sch92]. The ESF architectural approach to process integration has been described in section 2.4.2. The ESF approach is to support multiple interacting process engines [Hol92]. The process engines are divided into two types, process interaction engines and process interworking engines.

The basic functionalities of the interaction process engine are to notify the user what he has to do, how to do it, when to do it to make available tools to perform the work, and to make available the necessary information items to be manipulated. The interworking process engine and an interaction process engine are closely related components.

The interworking process engine controls the interworking of people working at their interaction process engine. The interplay between the interworking process engine and several interaction process engines needs communication means for the exchange of control between these components. This is supported by the software bus.

Within ESF there have been experiments with multiple different approaches to process modeling:

The FUNSOFT net approach are high level Petri nets whose semantics is defined in terms of Predicate/Transition nets extentioned by multi-sets [EIP+91, Gru91, Del92, Ado92]. Another approach is taken in the Process WEAVER tool [Fer93, Fer92], which uses a hierarchy of activity types broken down to activity steps based on a different Petri net notation. Both the MELMAC/Merlin environment with its FUNSOFT net approach and the Process WEAVER tool uses an architecture of communicating process engines. A number of WEAVER Agenda tools interacts with the WEAVER Process Engine.

Current work is investigating new process modeling paradigms and in particular mixed paradigm approaches [Sch91].

### 4.7.7 ESF and Integration - Conclusion

The focus of the ESF architecture is on request-oriented control-integration, through software-bus based interaction between User Interaction and Service Components. The separate Event Multicaster facilitates for notification-oriented control-integration. In addition much effort has been put into the area of WorkProcess support and Process-integration [Con89], see chapter 2.
ESF is different from other integration frameworks in the fact that it takes a comprehensive approach to systems integration, and addresses all four areas: control integration, data integration, presentation integration and process integration. However, the support for each area is offered through different approaches and mechanisms.

4.8 Other Integration Frameworks - State of the Art

There is currently a lot of activity in the area of integration frameworks and system architectures.

Here follows a short description of some other approaches:

**SAA - Systems Application Architecture with AD/Cycle** is the IBM framework for enabling interoperability and integration of applications [IBM91]. SAA focuses on achieving interoperability and consistency across four of their proprietary platforms: MVS, VM, OS/400 and OS/2. There is a communication connection to their AIX Unix-platform. The SAA architecture consists of three related components:

- Common User Access
- Common Programming Interfaces
- Common Communications Support

AD/Cycle is a life-cycle-oriented development environment. The integration approach is data integration-oriented around the single MVS-Repository, which is an ER-based datamodel on top of the relational DB2. An interesting point with this is the work in the Information Model area, where the different vendors are trying to agree on common conceptual schemas for the data in the repository. This takes time, and so far not many of the products are well integrated. User-interface integration is done through development-tools for the development of CUA-compliant user-interfaces, such as Digitalk (Smalltalk-V) and Enfin. IBM decided in 1992 that the database-oriented approach is too difficult to manage, and has decided to refocus the AD/Cycle environment. IBM has licensed the HP/Softbench technology from HP, and is currently experimenting with the notification-oriented control integration offered here.

**CARNOT** is under development at MCC [CHS91]. Similarly to Pegasus it is aimed at integrating information-resources from heterogeneous database-systems. The "canonical" data model used is AI-based extended first-order logic.

**Apple Interapplication communication architecture** In the System 7.0 version of the Apple operating system, released in 1991, the event-based operating system is enhanced to also support application-specific events. The basic operating-system and window-system oriented event-handling is enhanced with other kind of events through the Event Manager. The Event Manager is a basic facility for sending and receiving events. It uses the facilities of the Program-to-Program Communications (PPC) toolbox to send and receive events.

**ToolTalk; SPARCworks Tool Integration** is a suite of tools for software application developers [Sun91a, Sun91b]. This environment is inspired from the Field/HP Softbench approach. The ToolTalk service is a network spanning, interapplication communication service that allows applications to communicate with other autonomous applications. The ToolTalk service provides a multicast message-service, that is, an application sends a message that is delivered by the ToolTalk service to multiple receivers. Two types of messages are supported: Process-oriented messages are addressed to other processes, while object-oriented messages are addressed to objects managed by processes.
ATIS - Software Backplane is a hybrid database-oriented approach which has been developed by Atherton, and been adopted of Digital (DEC), is named ATIS. (Atherton Tool Integration Services) [No90]. It has also been marketed under the name CIS - CASE Integration Services, and been proposed to various standardization committees. ATIS is the interface to CDD/Repository, Digital's CASE Repository that is part of the COHESION CASE environment. The current version of ATIS is placed on top of the CDD/Repository, while the figure presents a proposal for placing it on top of PCTE+. The basis idea for ATIS is to describe the tools of an environment as instances of types in a type-hierarchy. The type-hierarchy is predefined and includes types for the most commonly found entities in a Software Engineering Environment. Based on the type-hierarchy a set of semantic models addressing the behaviour for different areas, have been defined. ATIS is supporting an object-oriented view on the tools in an environment. It does not support data-control and presentation integration services for individual tools, but instead is a kind of meta-environment for describing existing tools. It can be compared with the IRDS-system (Information Resource Dictionary System ANSI/ISO) for database-oriented environments, but describing tools in addition to data. ATIS has been proposed as a basis for the second generation IRDS.

ARISE/Museion: The communication-oriented paradigm is also used in the Museion architecture developed in the ARISE project. [Tel90] The different tools are communicating with each other and with the database by means of the POSTMASTER communication medium. The POSTMASTER is an active medium delivering messages to anyone which subscribes to the particular type of message. The ARISE architecture does presentation integration by using the same look-and-feel for all tools (Motif/X) and data integration by letting the different tools share the same data in one database. The use of the POSTMASTER is somewhat analogous to the Blackboard architecture developed for Expert-Systems. The functionality of the POSTMASTER is to put through messages between the participating tools and the database machine in Museion. A message is of a particular type and can specify that a reply message is required. Data is transferred as encoded character strings. If needed the sender waits for the reply until a time-out is received. Museion addresses the three different kinds of integration, presentation, data and control. The architecture in particular addresses notification-oriented control-integration. The tools broadcast information that they have caused some event, (e.g. the deletion of a file). Other tools which are interested in such events can react appropriately. The control-oriented approach is simple and promotes evolution and openness, it does however become incomprehensible by large number of components and therefore tends to demand an additional abstraction. The data-centered architecture on the other hand promotes integration and is more close to the human users notion of the architecture for an environment. The database-centered architecture is, on the other hand, more restrictive and prescriptive and thereby less open to foreign tools and to evolution.

4.9 Comparison of Systems Integration Frameworks

In the following each system is evaluated related to the requirements for the integration-areas that it supports. The score is + for well-covered - for medium-covered and ÷ for little or no support.

OMG is aimed at request-oriented control integration, it may in the future easily be enhanced to support notification-oriented control integration and single model data integration. No support is given for multi-model data integration or presentation integration. HIP/Softbench - Field focuses on notification-oriented control integration with a possibility to also do request-oriented control integration. PCTE provides good support for
single model data integration and presentation integration. Pegasus provides support for multi-model data integration, and also implicitly for single-model data integration. ESF is focused on request-oriented control integration through the software bus architecture. Notification-oriented control integration is supported through a separate service. Presentation integration is supported by a number of user-interface building tools. Single- and multi-model data integration is specified in a requirement-specification for object-storage services, where different solutions might be selected. Only ESF offers direct support for process integration. The others might, however, provide architectural support for the various architectural approaches described in section 4.1.5.

The following discusses how the solution approaches make compromises in the conflicting requirements described in section 3.1, in particular with respect to control integration and data integration.

- **Heterogeneity versus Homogeneity**
  OMG uses an interface-oriented object-model to hide underlying heterogeneity. HP Softbench/Field uses a loose coupling through the BMS, which allows for encapsulation of different heterogeneous components. PCTE does not support heterogeneous components. Pegasus uses an object-oriented model to hide underlying heterogeneity. ESF uses an interface-oriented software bus to hide underlying heterogeneity.

- **Autonomy versus Common policy**
  OMG requires the tool-components to use the CORBA mechanisms. Existing components can be made available through IDL-described interfaces. HP/Field includes an encapsulation facility that can be used to encapsulate existing components. PCTE requires tools to be written against the OMS and to follow the policies for this. Pegasus requires underlying components to not delete managed objects without notification. ESF requires client-tools to use the Software Bus, server-components might be encapsulated and made available without knowledge of this.

- **Distribution versus Centralization**
  OMG hides distribution through the object model. A future federated name service will be used to support multiple name spaces. HP/Field hides distribution through the centralized BMS. PCTE hides distribution through the schema definition sets. Pegasus hides distribution through its schema definition language, but can allow for explicit handling of different databases. ESF hides distribution through the software bus.

- **Fine-grain versus Large-grain**
  OMG supports objects of all granularities, from large-grain to fine-grain. It is, however, a question about how efficient the support for fine-grain objects can be, if there always is the overhead of going through CORBA. Light-weight CORBA implementations and object adapters for object-oriented databases might later provide efficient support also for fine-grain objects. HP/Field supports only large-grain objects. PCTE supports only large-grain (file-size) objects in current implementations. Pegasus supports fine-grained objects through its data integration approach. ESF supports only large-grain objects, through service components.

- **Efficiency versus Flexibility and Extensibility**
  OMG provides two alternative interfaces, an efficient, static approach based on stubs generated from definitions in the IDL interface-definition language, and a dynamic approach based on a procedural API. HP/Field focuses on flexibility and extensibility by using an interpreted string as the basis for messages. PCTE focuses on flexibility through the support for meta-data as normal data. Pegasus focuses on flexibility
and extensibility through the HOSQL-language. ESF focuses on efficiency through the component description language of the software bus.

- **Simplicity versus Coverage**
  OMG focuses on simplicity through a common object model, but lacks coverage by only focusing initially on control integration. HP/Field focuses on simplicity through one string interpretation based messaging mechanism. PCTE focuses on coverage for the area of data integration and presentation integration, looses some simplicity by the use of a procedural model with a lot of routines with many arguments. Pegasus focuses on simplicity through a uniform object model and the HOSQL language, but is initially aimed only at multi-mode data integration. ESF focuses on coverage by addressing all the integration areas, but looses simplicity by the use of different mechanisms in each area.

- **New language versus Existing languages**
  OMG supports existing languages, initially C and later C++ and others. HP/Field supports existing languages, C. PCTE supports existing languages, C. Pegasus has created a new language OSQL, but suffers from easy mapping to existing languages. ESF supports existing languages, C, C++, Ada and Prolog.

- **Variety of Components versus Functional Usability**
  OMG allows for all components that can be accessed through an IDL interface. HP/Field allows for all components that can be encapsulated with a message-based interface. PCTE does not support integration of different components, unless they use common OMS schemas. Pegasus requires the components to support the handling of set of objects. ESF allows for all components that can be accessed through descriptions in the SAD interface language.

- **Syntactical basis versus Semantical integration**
  OMG has a syntactical basis in IDL, and has plans for an interest group for tool vendors that want to define standard interfaces based on IDL. Later RFPs for Common Facilities might be a basis for standard interfaces that can be discussed and understood in order to get a basis for semantical integration. HP/Field (and ToolTalk) has only partial syntactical support for the description of messages. However, the CASE Communique group and the CASE Interoperability Alliance [X3H93], are groups which work on standardization of message-sets. PCTE has been used as basis for different integrated environments, where different vendors have tried to agree on common schema definitions. Pegasus has a syntactical object definition language. The Pegasus group has done research in various areas related to the problems of semantic interoperability for multi model databases. ESF has created a catalogue of interfaces described through the SAD interface language. Tool vendors might use this as a basis for understanding the semantics necessary for semantical integration.

The proposed integration frameworks covers to some degree the requirements for the different integration areas. OMG focuses on request-oriented control integration, HP/Field focuses on notification-oriented control integration and display-oriented presentation integration, PCTE focuses on single model data integration and display-oriented presentation integration, and Pegasus focuses on multi model data integration. ESF offers the broadest range of integration-support, by addressing all integration areas. However, support in different areas are offered through different mechanisms. This coverage is done on the expense of simplicity. With different mechanisms used in each integration area, it requires a lot of effort to understand and use all the mechanisms.

The goal of this thesis is to show that an object-oriented approach can be used as a unifying mechanism for the three integration areas control integration, data integration and
presentation integration. Process integration can be given some architectural support, through support for object-oriented interoperation between interacting process engines. Part II will present object-oriented concepts and terminology and experiences from experimenting with an object-oriented approach in one of the three integration-areas. Part III will present the unifying COOP Integration Framework.

<table>
<thead>
<tr>
<th>Supports</th>
<th>Control-R</th>
<th>Control-N</th>
<th>Data-Single</th>
<th>Data-Multi</th>
<th>Present.</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMG</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>HP/Field</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>PCTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Pegasus</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ESF</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.2: Integration-areas supported by the systems

<table>
<thead>
<tr>
<th>Requirement-area</th>
<th>OMG</th>
<th>HP/Field</th>
<th>PCTE</th>
<th>Pegasus</th>
<th>ESF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCR-1: Interface-description</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RCR-2: Interaction model</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>RCR-3: Run-time</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCR-4: Error-handling</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>RCR-5: Multi-user</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCR-6: Version handling</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RCR-7: Security aspects</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RCR-8: Administration</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCR-9: Heterogeneity</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.3: Evaluation for Request-oriented control-integration
### 4.9 Comparison of Systems Integration Frameworks

<table>
<thead>
<tr>
<th>Requirements-area</th>
<th>OMG</th>
<th>HP/Field</th>
<th>PCTE</th>
<th>Pegasus</th>
<th>ESF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCN-1: Event Notification</td>
<td>-</td>
<td>+</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RCN-2: Event Posting</td>
<td>-</td>
<td>+</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RCN-3: Activation</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RCN-4: Heterogeneity</td>
<td>-</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.4: Evaluation for Notification-oriented Control-integration

<table>
<thead>
<tr>
<th>Requirements-area</th>
<th>OMG</th>
<th>HP/Field</th>
<th>PCTE</th>
<th>Pegasus</th>
<th>ESF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDS-1: Basic ERA-model</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-2: Complex objects</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-3: User-defined data types</td>
<td>+</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-4: Prog. language support</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RDS-5: Versions/variants</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RDS-6: Metadata/Schema</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-7: Views</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-8: Rules / Triggers</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-9: Query language</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-10: Security</td>
<td>-</td>
<td>÷</td>
<td>+</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-11: Distributed Architecture</td>
<td>+</td>
<td>÷</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RDS-12: High Performance</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-13: Concurrency</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-14: Cooperation</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDS-15: Logging - recovery</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDS-16: Data Interchange</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.5: Evaluation for Single-model Data-integration

<table>
<thead>
<tr>
<th>Requirements-area</th>
<th>OMG</th>
<th>HP/Field</th>
<th>PCTE</th>
<th>Pegasus</th>
<th>ESF-OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDM-1: Canonical Data Model</td>
<td>-</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RDM-2: Schema Mapping</td>
<td>-</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RDM-3: Schema Integration</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-4: Query Processing</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-5: Transaction Management</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-6: Security</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-7: Autonomy and Evolution</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>-</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-8: Functional Components</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>+</td>
<td>÷</td>
</tr>
</tbody>
</table>

Table 4.6: Evaluation for Multi model Data-integration
### 4.9 Comparison of Systems Integration Frameworks

<table>
<thead>
<tr>
<th>Requirements-area</th>
<th>OMG</th>
<th>HP/Field</th>
<th>PCTE</th>
<th>Pegasus</th>
<th>ESF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPD-1: Look-and-Feel and Portability</td>
<td>✖</td>
<td>+</td>
<td>+</td>
<td>✖</td>
<td>+</td>
</tr>
<tr>
<td>RPD-2: Dialogue-support</td>
<td>✖</td>
<td>+</td>
<td>+</td>
<td>✖</td>
<td>+</td>
</tr>
<tr>
<td>RPD-3: Graphics-support</td>
<td>✖</td>
<td>✖</td>
<td>+</td>
<td>✖</td>
<td>+</td>
</tr>
<tr>
<td>RPD-4: Interactive Tools</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
</tr>
<tr>
<td>RPD-5: Copy/Cut and Paste</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
</tr>
<tr>
<td>RPM-1: Model-interaction</td>
<td>✖</td>
<td>+</td>
<td>✖</td>
<td>✖</td>
<td>+</td>
</tr>
<tr>
<td>RPM-2: Multi-Model-interaction</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>+</td>
</tr>
<tr>
<td>RPM-3: Change/Update</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.7: Evaluation for Presentation integration
Part II

Experiments with Object-Oriented Approaches to Systems Integration
Chapter 5

Object Oriented Concepts and Terminology

Part II, Experiments with object-oriented approaches to systems integration, presents experiences from experiments with object-oriented technology in the three main integration-areas: control-, data- and presentation-integration.

Chapter 6 presents experiments with object-oriented control integration. Chapter 7 presents experiments with object-oriented data integration, while chapter 8 presents experiments with object-oriented presentation integration. The goal of these chapters is to evaluate an object-oriented approach to the three integration areas. The conclusion is that an object-oriented approach is better than traditional approaches in these areas.

This chapter gives an introduction to object-oriented concepts and terminology, which will be used later in this work, and contains the following:

- 5.1 Object-Oriented Definitions and Terminology
- 5.2 Object-Oriented Architectures and Control Integration
- 5.3 Object-Oriented Databases and Data Integration
- 5.4 Object-Oriented User Interfaces and Presentation Integration
- 5.5 Object-Oriented Application Frameworks
- 5.6 Object-Oriented Programming Languages
- 5.7 Object-Oriented Analysis and Design
- 5.8 Current research in Object-Oriented Technology
- 5.9 Object Oriented Technology and Systems Integration

More detailed descriptions can be found in the course-notes for the course "CSE509 Object-Oriented Programming" which the author gave at Oregon Graduate Institute in spring 1988 [Ber88a] and "Object-Oriented System Development" given at the Norwegian Institute of Technology in spring 1991, [Ber91e], and in the notes for course "45091 System Programming" from spring 1992 [Con92].

More information can also be found in the separate reports: "Object-Oriented Analysis and Design - an Overview of Some Existing Methods and Techniques" [Ber90a], "Object-Oriented Operating Systems" [Ber90b], "Object-Oriented Architectures and Distributed Systems" [Ber91b], and "Object-Oriented Database Systems" [Ber91d].
5.1 Object-Oriented Definitions and Terminology

There are no common agreed upon definitions of what object-orientation really is or should be. Existing definitions differ in particular in which perspective they use.

The following presents four different perspectives:

- The Scandinavian "Modeling" perspective
- The American "Language" perspective
- The English "Abstraction" perspective
- The "Systems Integration" perspective (used in this work)

5.1.1 The Scandinavian "Modeling" Perspective

The Scandinavian "Modeling" Definition has been described in [PM88].

Definition 5.1 Object-Oriented Programming:
A program execution is regarded as a physical model, simulating the behaviour of either a real or imaginary part of the world. A physical model consists of objects, each object is characterized by attributes and a sequence of actions. Objects organize the substance aspect of phenomena, and transformations on substance are reflected by objects executing actions. Objects may have part-objects. An attribute may be a reference to a part object or to a separate object. Some attributes represent measurable properties of the object. The state of an object at a given moment is expressed by its substance, its measurable properties and the actions going on then. The state of the model is the states of the objects in the model.

The focus in this perspective is that an object-oriented program should be a set of objects, modeling the phenomena of interest. This definition do not say anything about which mechanisms that are used in order to create such a model.

5.1.2 The American "Language" Perspective

The definition of object-oriented programming found most often in textbooks and in the "commercial" trade-press, is associated with which features should be required from a programming language to call it object-oriented.

This is the definition which is closest to a definition of the terminology used for most object-oriented programming languages.

Definition 5.2 Object-Oriented Programming:
requires a language which supports the following four techniques:

- Encapsulation
- Classification/Instantiation
- Inheritance
- Dynamic Binding
These and related concepts are defined in the following:

**Definition 5.3 Encapsulation:**
An object is encapsulated, if an *operation set* and a *data set* are incorporated in a single entity (i.e. the object). Furthermore, clients should be restricted to accessing the object only through the well-defined, external, operational *interface*. □

This implicitly defines objects as follows:

**Definition 5.4 Object:**
An object consists of an encapsulated representation (state) and a set of messages (operations and procedures) that can be applied to that object. *Encapsulation* is the technical name for modularization and information hiding, and emphasizes that an object has an *outside* and an *inside*. □

Objects interacts through the sending of messages:

**Definition 5.5 Message:**
An executing procedure-call. A message is sent from one object (sender/client) to another (receiver/server), in order to invoke some service. It is similar to a *procedure call*, but the sender has no knowledge of the *actual code* executed. The message may adhere to a given interface. □

Messages results in the execution of method bodies:

**Definition 5.6 Method body:**
When a message is received, the receiver will find the appropriate code to execute as a response. The *actual code* to be executed is represented in a *method* similar to a *function* or *procedure*, with *input/output-parameters* and a *body*. □

Objects are classified through classes:

**Definition 5.7 Class:**
A class *specifies* the common *interface* for a set of objects, and describes how these objects (instances) will be *implemented* in terms of *instance variables* (state of representation) and *methods* (operations). The class is responsible for the *creation* of new objects – it acts as a factory for objects. □

Objects are created as instances of classes:

**Definition 5.8 Instance:**
An instance of a class is an *object* being produced by that class, or a subclass of that class – having both its interface and its implementation. □

**Definition 5.9 Instantiation:**
The process of creating an object belonging to a class (instance). □
Classes might inherit from other classes:

**Definition 5.10 Inheritance:**
It is possible to make a *conceptual hierarchy* of classes. A subclass of a superclass inherits both its interface (methods to respond to) and its implementation (instance variables and methods). It might *add* new methods to the interface and new methods and instance variables; or it might *override* the current definition of methods and do something differently. *Multiple inheritance* means that a subclass can inherit from several superclasses.

Polymorphism is provided through dynamic binding:

**Definition 5.11 Polymorphism:**
The ability for properties (procedures etc.) to operate on more than one type.

**Definition 5.12 Dynamic (Late) Binding:**
This is what takes place when a message is sent to an object, and the object decides which method to execute. All the sender knows, is that this object can respond to message xl, and is of class X or a subclass of X. The actual method to invoke is decided at run-time.

5.1.3 The English ”Abstraction” Perspective

The language-oriented definition above is also a definition of state-of-the-art terminology for object-oriented programming languages.

However, it has been argued that the language-oriented definition focuses too much on specific techniques. Instead one should focus on the abstract principles which should be adhered to, and have a possibility of using different techniques to support each principle.

In [B+91b] a definition which emphasize principles is given:

**Definition 5.13 Object-Oriented Programming:**
requires a language which supports the following four principles, which might be realized by alternative techniques:

- **Encapsulation:** Possible techniques are: Object, Active Object
- **Classification:** Possible techniques are: classes, prototypes, types, abstract data types, sets
- **Flexible sharing - How to share implementation / interface:** Possible techniques are: subclassing (inheritance), overloading, prototyping, polymorphism, subtyping, genericity, enhancement, subsets
- **Interpretation - How to decide which code to execute:** Possible techniques are: static binding, dynamic binding, delegation, static typing, dynamic typing
5.1.4 The Systems Integration Perspective used here

The notion of encapsulation is the most important aspect of object-orientation, seen from a systems integration perspective.

An object-oriented system is defined as a system of interacting objects. How the objects actually implements the interface is not of interest. Each object is encapsulated and has internally the knowledge it needs in order to respond to the services requested from it. Conformance-based inheritance is important for the ability of one object to substitute another object.

Control integration is in particular supported by object-oriented architectures to be presented in section 5.2, and experimented with in chapter 6.

Data integration is in particular supported by object-oriented databases to be presented in section ref00dl, and experimented with in chapter 7.

Presentation integration is in particular supported by object-oriented databases to be presented in section ref00p1, and experimented with in chapter 8.

5.2 Object-Oriented Architectures and Control Integration

The reports "Object-Oriented Architectures and Distributed Systems" [Ber91b] and "Object-Oriented Operating Systems" [Ber90b], give overviews of the use of an object-oriented approach for system-architectures. The article "Object Orientation in Heterogeneous Distributed Computing Systems" [NWM93] gives a good overview of this area.

Object-Oriented Architectures are particularly suited for supporting control integration, because of the focus on interaction between encapsulated objects.

Experiments with the use of an object-oriented approach to control integration are reported in chapter 6.

5.3 Object-Oriented Databases and Data Integration

The report "Object-Oriented Database Systems" [Ber91d], gives an introduction to object-oriented databases.

Object-Oriented Database Systems are suited for supporting data integration. Experiments with the use of an object-oriented approach to data integration are reported in chapter 7. The use of an object-oriented database-system for multi-model data integration is presented in section 4.6.

The assumption that object-oriented technology can be merged with database-technology has led to a lot of activity in this field the last 5 years. There have been two main approaches, either to extend a database-system with concepts from object-oriented technology or to extend a programming language with concepts from database systems.

Because the object-oriented model is not built on a set of theoretical principles such as the relational model, there has been a lot of discussion about what should be required from a database-system in order to call it object-oriented.

The Object-Oriented Database System Manifesto [MZ+89] gives the following rules:
Definition 5.14 Object Oriented Database:
A database system can be called object-oriented, if it supports the following 8 Object-Oriented rules, and the following 5 database rules. In addition there are 5 "optional" rules.

- The 8 Object-Oriented Rules
  - Complex Objects
  - Object Identity
  - Encapsulation
  - Types or Classes
  - Inheritance
  - Dynamic Binding
  - Extensibility
  - Computational Completeness

- The 5 Database Rules
  - Persistence
  - Disc Administration
  - Concurrency
  - Recovery
  - Ad-hoc queries

- 5 Optional Rules (Bonus-rules)
  - Multiple Inheritance
  - Type-checking and type-inference
  - Distribution
  - Design-transactions (long)
  - Versions and Variants

The following are some of the most dominant object-oriented database-systems on the market per mid 1993.

- The Client-oriented group:
  - ObjectStore from Object Design (C++ and C)
  - ONTOS from Ontos (C++)
  - Versant from Versant (C++ or Smalltalk)
  - Objectivity/DB from Objectivity (C++)
  - Itasca from Itasca Systems (Lisp)
  - Gbase from Object Databases (Lisp)

- The Server-oriented group:
  - Gemstone from Servio Logic (Opal, Pascal, C, Smalltalk, C++, 4GL)
5.4 Object-Oriented User Interfaces and Presentation Integration

- O2 from O2 Technology (Obj-C, C++)
- OpenODB from HP (C)

The client-oriented group consists of systems where the object-execution is done in a client-based cache. The server-oriented group consists of systems where object-execution is done on a server. This distinction is further discussed in chapter 7.

Chapter 7 will look closer into the solutions provided by object-oriented database-systems, and present experiences gained from experiments with some systems.

5.4 Object-Oriented User Interfaces and Presentation Integration

The course-notes [Ber91e] gives an introduction to object-oriented user interfaces.

Object-Oriented user interfaces are suited for supporting presentation integration. In particular because the modeling of the components in a user interface easily can be done by objects. The concept of application frameworks to be presented in the next section has successfully been used for the creation of user interfaces. The Model-View-Architecture introduced in the Smalltalk-80 user interface application framework [KP88] can be useful also for distributed systems.

Experiences with the use of an object-oriented approach to presentation integration is reported in chapter 8.

5.5 Object-Oriented Application Frameworks

An important advantage of object-oriented technology is the possibility to reuse existing classes, and to extend these through subclassing. Object-oriented application frameworks extends the reuse from single components to sets of interacting components making up the general design for a particular class of applications.

Definition 5.15 Application Framework:
An application framework is a set of classes for interacting components in a particular domain, where the control-structure of the interaction is embedded into predefined operations and interaction-sequences. New applications are made by subclassing specific classes and reimplementing given operations with code specific for the new application. □

An application framework incorporates the major parts of the design for a particular application-area. This reduces the work required to make new applications for that domain, because the application-design already is in place.

Application Frameworks require the support of object-oriented technology. References to objects in the general classes assume to interact with objects of certain types, that supports specific operations. Inheritance makes it possible to create subclasses/subtypes of the given type and dynamic binding makes it possible to execute the new application-specific code.

Class Simulation in Simula [BDMN73, DMN70] was the first example of an application framework. This was done for the application domain, Discrete Event Simulation. Lately the application framework approach has gained popularity in particular for the development of user-interfaces for modern windowing system. The second, and most well-known,
application framework is the Smalltalk Model-View-Controller framework [KP88]. The general "cookbook" for a user-interface application-framework is typically something similar to the following:

1. Define `YourView` as a subclass of `View`
2. Define `YourModel` as a subclass of `Model`
3. Implement the `draw`-operation of `YourView` with the necessary code to show application-specific
4. Implement the `menu`-operation with the creation of a menu with application-specific operations invoking operations on `YourModel`.
5. Implement the `buttonPress`-operation with the necessary code for handling the pressing of a button, if this is needed by the application.
6. Create an instance of `YourView` that knows about `YourModel`.
7. Start up the application by sending the message `run` to the created instance of `YourView`.

The concept of application frameworks is by many viewed as an important direction to follow for a larger number of application domains [WBJ90].

### 5.6 Object-Oriented Programming Languages

There are a number of different object-oriented languages. The course-notes [Ber91e] provides a comparison and evaluation of different object-oriented programming languages.

The languages can be categorized broadly into four groups: typed or untyped, and compiled or interpreted languages.

Examples of languages in each group are:

- **Typed and Compiled:** C++, Simula, Eiffel, Guide, POOL
- **Untyped and Compiled:** Objective-C
- **Untyped and Interpreted:** Smalltalk, Actor, Object/1, Self
- **Typed and Interpreted:** Can be useful for documentation. Experiments has been done with interface-typing in Smalltalk, but it is not a language in practical use.

The languages with most commercial success are C++ and Smalltalk. These languages represents the two main approaches, and both will be used later in this work.

### 5.7 Object-Oriented Analysis and Design

There is currently a lot of interest in methods for object-oriented analysis and design.

The report "Object-Oriented Analysis and Design - An Overview of Some Existing Methods and Techniques" [Ber90a], gives an overview and comparison of different methods.

The methods have been categorized in four different categories, based on their conceptual origin:
Based on our experiences with object-oriented system development, we have developed a method called OORASS [WBJ90, RB+92], which is classified in the OO-based category.

There is currently much activity in this area. Many consulting companies have made text-books and case-tools and provide seminars and courses for ER-based object-oriented methods.

Chapter 14 in part III of this work will present an ER-based extension to the OO-based OORASS-method, called COORASS, which integrates ER-concepts on top of a pure OO-based method.

5.8 Current Research in Object-Oriented Technology

5.8.1 Separation between Interface, Implementation and Subset

Most current object-oriented programming languages, like C++ and Smalltalk, use the class-concept for specifying both interface and implementation, as a factory for creating objects and as a manager for all instances of a class (Smalltalk). A current trend in object-oriented research is to define separate mechanisms for the different responsibilities that a class normally have.

The definition of a class in definition 4.2.7, shows that a class is responsible for:

- interface-definition
- implementation-description
- object creation - factory

\(^1\text{ER - Entity Relationship}\)
possible management of its instances (sets of objects)

As will be described in the following, it seems useful to make the following separations:

- The separation of interface and implementation
- The separation of intent (class) and extent (instances of a class)
- The separation of class and object-creation service

5.8.2 The Separation of Interface and Implementation

The separation of interface and implementation has its origin in the work on Abstract Data Types, [Gut81], and has been brought further in the context of object-oriented programming [Por92]. The separation between interface and implementation allows for multiple different implementations for one interface. The POOL programming language support a distinction between interface and implementation hierarchies [AvdL90].

The notion of type has commonly in the object-oriented literature been used as a synonym for interface [Obj91a, B+91b]. A type is then solely concerned with the external behaviour of objects, whereas a class also is concerned with how this behaviour is implemented.

The notion of generalized and specialized interfaces is then similar to supertypes and subtypes, as described in [B+91b].

Interface-based typing supports the principle of substituteability. An object can be substituted by another object which is conform to the interface of the first object. The rules for conformance was defined for the Emerald language [HM85b].

- Rule 1: The subtype provides at least the operations of the supertype. (The subtype might have more operations)
- Rule 2: For each operation in the supertype, the corresponding operation in the subtype has the same number of arguments and results.
- Rule 3: The types of the results of the subtypes operations conforms to the types of the results of the supertypes operations
- Rule 4: The types of the arguments of the supertypes operations must conform to the arguments of the subtypes operations (i.e. arguments must conform in the opposite directions)

This means that an interface cannot return a result which is more general than what is promised by an interface it shall be conform to, and that it should accept all arguments which are accepted by its generalized interface, e.g. not impose stronger restrictions.

A simple example which illustrates the principle of substituteability is the Interface-hierarchy: Person, Employee, (Engineer, Manager, TechMan and Department, TechDept

```java
Interface Person {
    String get_name();
    void set_name(String [in] newName());
}

Interface Employee: Person {
```
int salary();

Interface Engineer: Employee {
  ENUM Profession category();
}

Interface Manager: Employee {
  {int bonus();
}

Interface Department {
  Manager groupleader(Employee [in] emp);
  Department copy();
}

Interface TechDepartment: Department {
  // redefined the groupleader- and copy-operations
  // inherited from Department
  // Legal alternatives for groupleader and copy:
  Manager groupleader(Person [in] p); // receive Person
  // or receive Person - return TechManage
  TechManager groupleader(Person [in] p); r
  // return TechDepartment instead of Department
  TechDepartment copy();

  // Illegal alternatives for groupleader and copy:
  Employee groupleader(Employee [in] emp); // return Employee
  Person groupleader(Employee [in] emp); // return Person
  Company copy(); // return Company
};

There are systems, such as Emerald [RTL91], which implicitly create subtype relationships among defined types. This flexibility can sometimes lead to "mistaken" type matches, e.g. both Person and Wine might have the operations name() and age(). If the interface specification can be enhanced with semantic descriptions it is easier to make distinctions. This approach is the most flexible, because the introduction of new types do not require explicit changes in the type-hierarchy. There are, however, difficulties in mapping this to systems with explicitly defined subtypes, such as C++.

5.8.3 The Separation of Intent and Extent

It is also useful to separate the definition of an interface or an implementation, from the set of instances of the interface or the implementation.

In the database-literature this has been distinguished as the intent and the extent of a definition. Extent is used as the name of all instances of a type, type-extension, and all instances of a class, class-extension. A type-extension is all objects which support the interface defined by the type. A class-extension is all objects which have the same implementation, and thus also the same interface. The class-extent is thus a possible sub-set of the corresponding type-extent. An interface-extent includes all objects which provides that interface. An implementation-extent includes all objects which have that implementation, and thus also the same interface.

From a database-viewpoint it is most interesting to be able to manage the extent of types. Some natural categories of objects are, however, difficult to define with a conformance-based interface-system. The principle of substituteability puts restrictions on what kind
of specialization one can have in a sub-interface. For instance it will not be legal to define Teenager as a specialized interface of Person, if instances of Teenager are restricted to have their age-values in the 13-19 domain.

Predicate-restricted subsets are typically not a good candidate for making a specialized interface. Still it is very useful to model concepts like Teenagers. This might be done by introduction of categories, which are sets where all members satisfy a particular predicate. A special kind of category is interface-externt which satisfy the predicate that all members of the set provides a particular interface (either by being a direct implementation of this interface, or by being an instance of a specialized interface).

5.8.4 The Separation of Class and Object-creation Service

A notion of object-creation service, or factory, is useful for the creation of objects, in particular in distributed systems. This service might be provided by a class-object but it might also be provided by other objects.

From a client's point of view, there is no special mechanism for creating or destroying objects. Objects are created and destroyed as an outcome of requests. The outcome of an object-creation request is revealed to the client in the form of an object-reference that denotes the new object.

This distinction has been emphasized in the OMG-architecture [Obj91a].

5.8.5 Objects and Relations

There is no concept for relations or relationships in the state-of-the-art object-oriented languages. References between objects are maintained through the use of pointers in instance variables.

We have argued in [Ber86, Ber87, Ber88c] that a relationship is a semantic concept that should be explicitly supported in an object-oriented system. Others have reached the same conclusion [Rum87, RBP+91].

Some information-models have an inherent structure for how objects are related. This is in particular models which have a natural mapping to an entity relationship model. In database-/information-oriented applications it is useful to make this structure explicit through structural abstractions such as relationships and attributes. How these structures actually are represented should be hidden.

5.9 Object Oriented Technology and Systems Integration

This chapter presented an overview of object-oriented concepts and terminology. This terminology and concepts will be used in the rest of this thesis.

The survey of object-oriented technology showed that it can be applied to the three different integration areas. Object-oriented distributed system architectures may be used as a basis for control integration. Object-oriented databases may be used as a basis for data integration. Object-oriented user-interface frameworks may be used as a basis for presentation integration.

It can be argued that object-oriented concepts make the distinction between control integration and data integration less useful, because an object encapsulates both behaviour
and data. However, it is still useful to separate these areas, because they emphasize different aspects of a system, as can be seen by the distinction between object-oriented architectures and object-oriented databases.

Experiments with object-oriented technology have been conducted in each integration-area, to validate the thesis that object-oriented technology is better than traditional approaches in each area. Chapter 6 reports on experiences from experiments with object-oriented control integration. Chapter 7 reports on experiences from experiments with object-oriented data integration. Chapter 8 reports on experiences from experiments with object-oriented presentation integration.

The goal of these chapters is to evaluate an object-oriented approach to the three integration areas. The conclusion is that an object-oriented approach is better than traditional approaches in these areas.
Chapter 6

Experiments with Object-Oriented Control Integration

The goal of this chapter is to show that object-oriented control integration is feasible and better than traditional control integration. Object-oriented control integration has been experimented with in the SITE and CODEDISC prototypes, where object-oriented interaction mechanisms have been realized on top of traditional remote procedure call mechanisms. The SITE prototype shows an example of interoperation between process interaction engines and process interworking engines, following the ESF architecture as described in section 2.4.2 and in section 4.7. The CODEDISC prototype shows a development environment that supports object-oriented control integration.

6.1 SITE Prototype

6.1.1 Goal with SITE

A more detailed description of the experiences from SITE is presented in [Ber92c].

The goal of the SI Team Environment (SITE) prototype is to experiment with an object-oriented approach to control integration based on the ESF software bus architecture. The case has been to support interoperation between process interaction engines and a process interworking engine. This has been done by the creation of an environment for cooperation between a project manager and project members, with a flexible user-controlled Work Process-support. The ESF architecture has been described in section 2.4.2 and in section 4.7.

SITE has been developed at SI\textsuperscript{1} [OS89], with the author as project manager. In the validation of the architecture we wanted to try out interoperability among different programming languages (Smalltalk, C and Lisp) and the reuse of existing components. SITE was therefore developed following the ESF reference architecture of User Interaction Components that interacts with Service Components through a Software Bus.

The user-interface of the User Interaction Components are described in chapter 8. The project manager uses a Process Support Tool to define a hierarchical structure of tasks, ending up with a number of job-assignments. A team member receives job-assignments

\textsuperscript{1}Center for Industrial Research, SINTEF SI from 1. January 1993
through electronic mail, and automatically incorporates new jobs into the personal job management tool, if he chooses to accept the job. The project manager has access to the database for all jobs and the job-state will automatically be updated in response to changes in the service-component. A job might be associated with a work-context, which is an association between data/objects and the tool used to manipulate them. These work-contexts might also be passed between team members.

SITE is realized by dividing components into User Interaction Components and Service Components. The Process-support UIC is a user interface component showing a project manager's view of the planning- and process-model for a project, while the Job-management UIC is a user interface component showing the different jobs of one team member in a calendar-view. Both UICs are different views on two common underlying service components.

The SITE user-interfaces are illustrated in figure 8.1 in section 8.4 in chapter 8 on experiments with object-oriented presentation integration.

The Process Model SC contains information about project-activities, jobs and their status, and the Work-context storage SC has the ability to store and retrieve work-contexts. In the following section it will be illustrated how the software bus architecture has been used to create an integrated system out of these separate components.

6.1.2 Request-oriented Control Integration in SITE

Control integration was realized through a specialized implementation of the ESF software bus concept. The software bus of SITE was realized through building on top of a remote procedure call mechanism (ONC/RPC) and a network file system (NFS), with a change/update-service created in addition.

From Smalltalk the service-components are modeled as instances of classes, which can

\[\text{The work-context concept was developed by Gro Oftedal and Anne Lise Skaar}\]
receive a set of messages corresponding to the services the component provides. A message
to a local Smalltalk object results in an operation being invoked on a remote service
component. The interaction from Smalltalk is realized by primitive methods which call a
C-client-stub-routine which again connects with a server-stub-routine.

The rightmost service-component in the SITE architecture in figure 6.1 indicates a possible
pluggable service-component, which provides an alternative implementation for the Work-
context-storage SC. This component would be conform to the interface of a Work-context-
storage service, and would provide the same or an extended set of services.

6.1.3 Language Heterogeneity

Three of the components are implemented in Smalltalk, while one is implemented in Lisp.
The Lisp-system is an already existing process model component developed in a previous
project. It was, however, necessary to separate its existing user-interface and functional
part, and make its services explicitly available through a procedural interface. The remote
procedure call system that is used supports only the C-programming language directly.
However, both Smalltalk and Lisp can call C, so representation in C was used as the
common denominator. Object-references or pointers are not supported, and all interaction
is through "call by value".

6.1.4 Notification-oriented Control Integration in SITE

The functionality of the SITE update mechanism is modeled after the Smalltalk Model-
View-Controller change/update- mechanism, which is described in more detail in section
8.2 in chapter 8. A component might register itself as interested in changes in another
component. If such a change takes place, the components which have registered interest
will be notified.

Figure 6.2 shows how the change/update-mechanism works. There are some changes with
respect to the Model-View-Controller model, in particular that the change/update logic is
external to the components.

The example in figure 6.2 shows what happens if a programmer accepts a job through the
Job-management UIC and thereby changes the status-property of an activity-object from requested to accepted. This change will be reflected in the Process-support UIC in the project-manager's work-environment.

In 1A and 1B the process-support UIC registers interest in notification of changes in the Process-model SC. In 2 the Job-management UIC causes a change in the Process-model SC, and then sends an update-call with an update-aspect as parameter (3A). In 3B the update layer sends a notification to the update-box service-components which have requested to be notified. The "Update receiver" regularly polls its associated update-box SC (4). This mechanism makes it then possible to immediately reflect changes to job-status in the Process-support UIC. In 5A and 5B the update-receiver unregisters its interest in receiving update-notification.

This is a basis for a more general event/notification-mechanism which can be quite useful also for other means than change/update. Examples of tool-integration based on events and event-notification-mechanisms have been shown in the tool-integration-environments Field [Rei90a], HP Softbench [HP 89] and Sun ToolTalk [Sun91a].

6.1.5 Experiences from SITE

The experience from SITE is that the software bus architecture works well, but that it raises a number of questions and issues to be aware of.

- **Update of changes**
  Ideally we would like to avoid the polling-situation, and instead have components acting both as clients and servers, supporting a direct notification. One could also consider the Smalltalk model of letting the service-component causing the notification, not the update-causer. The advantage of letting the update-causer do it, meant that no changes were required for the reused service-component. However, a more sophisticated update-layer could have been made.

- **Need for service-database**
  In OMC/RPC the client explicitly requests an interaction with a numbered program-service. More flexibility will be given if this information is kept in a database-service, and the client instead requests a particular service to be offered. E.g. like the ANSA trader, [ans89].

- **Error handling**
  In the SITE prototype we have little support for consistent error-handling for different kinds of errors, i.e. it is difficult to catch and handle underlying communication errors. It would be useful to have a more elaborate error and exception-handling mechanism.

- **Need for development tools**
  It requires great effort to create such systems if all interaction-code needs to be coded manually. Fortunately one can get help from automatic stub-compilers, and there is a potential for creating a more complete object-oriented development environment. In such an environment the system-builder only sees available objects which might be reused directly as instances of types/classes in an object-oriented language, and all interaction-handling will be taken care of automatically.

- **Installation support**
  At run-time it would be useful to have component installation and removal tools, so that an environment-administrator could easily maintain the configuration of the environment.
6.2 CODEDISC Prototype

- **Language heterogeneity and data representation**
  It is important to handle component-implementations in different programming languages as transparently as possible. There are problems in passing pointers and references, and language-dependent data-types, but much more automatic support can be given for this. It should be possible to have a common semantic model for describing component-interfaces and component-interaction with different representations of the same concepts for different languages.

- **Object-Oriented Development methodology**
  The principle of designing a system as a set of interacting objects requires also a development methodology supporting this. We have developed a methodology called OORASS, Object-Oriented Role Analysis, Synthesis and Structuring, [RB+92] which supports the role(interface)-oriented view for creating component-based systems.

It is important to design components for the purpose of interacting with other components. This requires a well-defined interface and machine-readable return and error-messages. In order to create components from existing applications and systems, it should be possible to split these into a user-interface-part and a service-part. When this is not possible one might be able to create service-oriented filters on top of existing user-interfaces. The filter will interpret and transform input and output to fit the original interface. When doing this we have often experienced difficulties with providing good support for error-handling, which often is embedded in character-strings originally to be shown on a screen.

Based on experience from SITE it can be concluded that object-oriented control integration is feasible, and provides a higher conceptual level than traditional remote procedure call mechanisms. It is, however, a need for more sophisticated tools to make the development easier. In particular for developing UICs, good use have been found in powerful reusable class-libraries in Smalltalk. Interactive user interface tools have started to become available now, and this will make the development of UICs even easier. In the CODEDISC prototype to be presented in the next section the possibility for creating a better development environment has been explored.

6.2.1 Goal of CODEDISC

A more detailed description of CODEDISC can be found in [Ber92c, FST+90]. The CODEDISC prototype (Constructive Object oriented Development Environment for Distributed Software Components) was developed to demonstrate what a development environment that supports object-oriented control integration could look like. CODEDISC was developed in a project carried out at the Norwegian Institute of Technology in Trondheim, with the author as adviser [FST+90]. Based on the experiences from SITE this work focused on providing better development environment support. It elaborated on the experienced need for better automated tool-support, and put emphasize on a working environment for an application programmer who would like to use available services. It also emphasized the utilization of existing industry-standards like X-OSF/Motif, C++ and OSF/DCE [osf90].

CODEDISC is an environment for an application-programmer which develops integrated applications by selecting from pre-existing service components. These service components can viewed as objects, and can be interacted with as objects. A Browser tool is used for identifying available service-components which can potentially be used from a client-
Component Description SC
Tester SC
Installer SC
NIDL to C++ SC
Error handler SC
Help handler SC

Figure 6.3: CODEDISC components - using the architecture on itself

component. Components and their operations are classified into categories, similar to what is done in the Smalltalk-80 environment.

The Browser gives access to a description of available software components. Each component is described as an abstract class belonging to a certain class-category, and having a set of methods in different categories. This description is independent of programming language, and by choosing one component-description one can generate a class-implementation for a particular object-oriented programming language. Only C++ is supported in the current prototype, but the design is made in such a way that other object-oriented languages such as Smalltalk can be easily supported. The Browser UIC is shown in figure 8.2.

The Tester is a tool which automatically creates a test-environment for interactive testing of available components. The application-programmer can choose a service-component and automatically create a test-program for interactively trying out the different operations available. The test-program is automatically created from the interface-description of the component. The Tester UIC is shown in figure 8.2.

The Software Bus in this environment is realized on top of the remote procedure call component of OSF/DCE [osf90], which in its turn is based on NCS/RPC [ncs89]. It requires that the interfaces of components are described by an Interface-description in the NIDL-interface-language (Network Interface Definition Language), and uses that as a basis for the automatic generation of class-implementations. CODEDISC uses NIDL together with a description of class-category and method-categories as the language-independent component description.

In addition there is support for multiple inheritance of interfaces, and a possibility to reuse the implementation of parent-class components by directly delegating operations to these components.

6.2.2 The CODEDISC Architecture

The components of CODEDISC can be seen in figure 6.3. The interaction among the components of CODEDISC is realized by using a software bus based on the NIDL-based remote procedure-call-mechanism, so CODEDISC is itself adhering to the kind of architectures it
6.2.3 Support for Object-Oriented Component Interaction

Figure 6.4: Generation of C++-class from NIDL-based component description

supports the development of.

The NIDL-environment provides some other useful services, such as a location- broker for binding to services, and the notion of unique universal identifiers (uuid) for referring to objects within components.

Each tool will typically have one UIC-part interacting with one or more service components. The Reuse-browser UIC and the Tester UIC have already been described. The Reuse-Browser UIC interacts with a service-component which itself interacts with the database-service-component.

The Client generator automatically creates a C++-encapsulation of a NIDL-described interface, through creating a corresponding C++ class-definition and class-implementation based on stub-calls.

The Installer UIC helps installing new components into the component description service. The Help-handler and the Error-handler are services which are used to find and present help- and error- information.

6.2.3 Support for Object-Oriented Component Interaction

Figure 6.4 shows how a C++ class-specification and implementation are generated based on the component-description in the database. If a builder of a client- component wants to use a particular service, the specification of this component is selected in the browser, and a C++-header and implementation file for interacting with the service-component is automatically created.

The NIDL-file is retrieved from the database-service and parsed by the NIDL-to-C++ converter, which creates a corresponding C++ header (interface)- file and an implementation- file. The implementation for each method (member-function) calls a stub-routine for this operation, which is generated by the NIDL-compiler also based on the NIDL-file. The generated stub-routines are C-routines, and their definitions are used when compiling the
6.2 CODEDISC Prototype

implementation-part of the generated C++-class. The application-programmer will include the class definitions for the components that are chosen when the client-component is compiled and finally everything is linked together. Note that in general a component might serve both as a client and as a server. However, in the current CODEDISC prototype the focus has been on providing support for the client-aspects of components.

C++ object-constructors are made so that one can either automatically bind against a component of the required type, or supply a uuid for binding to a particular object. The uuid-notation for object-identification used by NIDL is supported directly.

It is a general problem to support inheritance of implementation in a distributed system. In CODEDISC the focus is on compatible interfaces or type-conformity. If a class is a subclass of another, it will support a superset of the operations of its parent class. If a component will implement all of the operations itself, all the operations will be specified in its interface, thus overriding the eventually inherited operations.

If some operations are not overridden, the generated class-implementation will automatically invoke this code in an instance of the parent-component. This mechanism then functions as a form for delegation. The generated class will have an implementation for each method that automatically will be directed to the corresponding component.

6.2.4 Experiences from CODEDISC

Some experienced problems and issues from the CODEDISC prototype are:

- **Need for activation-service**
  The current prototype requires that all services are active. They are started up manually and register themselves as available to the NCS Location Broker service. To have all services up and running, even if they are not currently used, is a waste of resources, and degrades the performance of the system. An activation-service which transparently activates a requested service if it is not currently up and running, should be developed.

- **Language independent descriptions**
  The component description is currently rather simple, using NCS-NIDL for the signature-descriptions, and a separate textual description for category-descriptions and information about superclasses. We would rather have had one abstract object-model language, which also transparently could generate necessary implementations for other interaction-mechanisms.

- **Granularity for objects**
  The current granularity of objects is that of active processes or applications. One could imagine also supporting the object-model for access to objects which are managed by these processes. This is partly supported by the uuid for objects, but could have been even more generally supported.

- **Persistent objects and object-references**
  Objects are currently supported by explicitly using uuids. This is a long sequence of characters, and is not well encapsulated in an object. It would be possible to give a more uniform support for persistent objects and references to objects, by adapting a persistent object viewpoint, and view the total environment as a large heterogeneous database-system with a uniform object-oriented model.

- **Need for server-support**
  The focus in CODEDISC has been on support for the client-aspects of a component.
6.3 Contribution and Conclusion

<table>
<thead>
<tr>
<th>Requirements to</th>
<th>SITE</th>
<th>CODEDISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCF-1: Interface-description</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>RCF-2: Interaction model</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RCF-3: Run-time</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RCF-4: Error-handling</td>
<td>÷</td>
<td>-</td>
</tr>
<tr>
<td>RCF-5: Multi-user</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCF-6: Version handling</td>
<td>÷</td>
<td>÷</td>
</tr>
<tr>
<td>RCF-7: Security aspects</td>
<td>÷</td>
<td>÷</td>
</tr>
<tr>
<td>RCF-8: Administration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCF-9: Heterogeneity</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: Evaluation for Request-oriented control-integration

<table>
<thead>
<tr>
<th>Requirements to</th>
<th>SITE Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCN-1: Event Notification</td>
<td>+</td>
</tr>
<tr>
<td>RCN-2: Event Posting</td>
<td>+</td>
</tr>
<tr>
<td>RCN-3: Activation</td>
<td>÷</td>
</tr>
<tr>
<td>RCN-4: Heterogeneity</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.2: Evaluation for Notification-oriented Control-integration

Similarly one would like to have automated support for the server-aspects of a component, e.g. by having a server-generator analog to the client-generator. This could also include support for making multi-user servers.

- **Integration with a programming environment**
  It would be useful to transparently integrate the CODEDISC tools into a comprehensive object-oriented programming environment.

- **Other useful tools**
  Other tools which would give good support for the component-oriented system building approach are a database or set of databases with information about generally available components, and interactive tools for creating client-interface code based on interactively making connections to services.

The overall experience from CODEDISC is that it is feasible to create an object-oriented development environment for the development of component-based systems. A good support environment is a prerequisite to do systems integration, based on a component-oriented approach.

6.3 Contribution and Conclusion

The contribution of the SITE and CODEDISC prototypes is the demonstration of practical interaction through an object-oriented control integration mechanism. Version-handling and security-mechanisms were not addressed. The SITE update-mechanism provided also a basic notification-facility. There was no support for direct activation of components. The experiences pointed out a number of areas where more support may be given, in particular in the use of a more powerful language for interface descriptions.
Both SITE and CODEDISC were created on top of traditional mechanisms for control integration. SITE was realized with Smalltalk object-encapsulation of Sun ONC remote procedure call mechanism, while CODEDISC was realized as a C++ encapsulation of OSF/DCE RPC.

The object-oriented perspective on control integration and interaction between system components makes development easier for an application programmer. Instead of having to write procedural code for interaction with remote services, these services are automatically encapsulated in objects which hide the interaction complexity. An object might easily be substituted by an object of a conformant subclass. Object-oriented control integration is then better than traditional control integration, because interacting objects provide a simpler conceptual framework. Much less code needs to be written by a programmer using objects, than by a programmer using a traditional remote procedure call mechanism. Much of the necessary code has already been generated and is encapsulated in the objects.
Chapter 7

Experiments with
Object-Oriented Data Integration

The goal of this chapter is to show that object-oriented data integration is feasible and better than traditional data integration.

The chapter presents experiences from the development and use of SOOM, Semantic Object Oriented Model, which is an object-oriented model with support for relationships. SOOM shows that it is possible to combine concepts from ERA models and object-oriented models.

The chapter then presents experiences from the development of and experiments with the "HyperModel Benchmark", for the evaluation of object-oriented database systems. This benchmark has been developed to be able to evaluate commercially available object-oriented database systems, to see if they meet the requirements for data integration in integrated environments. The results from the HyperModel benchmark shows that object-oriented data integration is better than traditional data integration, in some situations. The focus in this chapter is on single model data integration.

7.1  SOOM - Semantic Object Oriented Model and Tornado*

The work on SOOM and Tornado* has been presented in [Ber86, Ber87, Ber88c].

SOOM, Semantic Object-Oriented Model, and Tornado* are the datamodel and database-architecture supporting the object-oriented Taskon-Environment [Tas93]. The major component in this environment is a set of structure-oriented editors that manipulate a hypertext-like document-structure. The goal of SOOM has been to combine concepts from ERA models with concepts from object-oriented models. SOOM is thus a datamodel based on a merge of ideas from structural and behavioural object-oriented datamodels, where relationships are added to an object-oriented language, Smalltalk-80. Tornado* is an architecture based on one or more centralized object-servers and a local workspace on each workstation. The bottleneck of the initial caching from a centralized server to the local workspace has been identified, and a solution based on a distributed object-server is suggested.
7.1 SOOM - Semantic Object Oriented Model and Tornado

7.1.1 Introduction

Taskon-Environment is an Engineering Information System which has been developed in the research program "Effectiveness and Quality in Engineering" at Center for Industrial Research, in the period 1983 to 1988. The system is based on an object-oriented architecture in a distributed workstation/server environment [RS89, Tas93]. The main programming language for development of tools and applications has been Smalltalk-80.

The information management in Taskon-Environment is organized around a hypertext-like document-model of interconnected objects. The objects may contain different kinds of information, like text, pictures, drawings, spreadsheets, database-queries etc. The user-interface is managed by a set of structure-oriented editors which uses the structural schema of the document to prevent the user from creating illegal structures. The application-areas for the system have been preparation of bid-proposals in offshore engineering, and information management for software engineering.

The developed datamodel was named SOOM because of its integration of concepts from semantic datamodels and object-oriented programming languages, and the system was named Tornado because the Data-Store-part is integrated into our earlier system, Tornado [UMi82]. Tornado is a network-oriented database with variable-length records, originally developed for CAD/CAM-applications.

7.1.2 SOOM - Semantic Object Oriented Model

The datamodel was developed to support complex document-structures, and is a synthesis of ideas from semantic datamodels and object-oriented languages. An object-oriented model like Smalltalk does not support structural abstractions in the same sense as most semantic datamodels. Typical semantic datamodels support relationships to describe 1-to-1, 1-to-many or many-to-many relations between objects, and this is a useful construct for representing complex object-structures.

As has been pointed out in [BP86] and [Rum87], the notion of relationships is a useful semantic construct which is normally missing in object-oriented languages, but should ideally be supported by an object-oriented datamodel.

The following shows how the structural and behavioural aspects of SOOM are integrated in a multi-layered model.

7.1.3 Different Views on the Shared Data from Different Applications

Figure 7.1 shows different mappings for the same object, from storage-object to structural object to behavioural object, and some temporary display-objects which are used to present different views to the user.

Since different applications might want to present different aspects of a shared structure, there is not a need to share the behavioural part of objects. The structural part represents the information which needs to be persistent. Compared to the ANSI-SPARC 3-schema architecture, the structural schema corresponds to the conceptual schema, the way it is physically stored corresponds to the physical schema, and the aspect of the structural model which is used by an application corresponds to a view on the conceptual model.

It is the responsibility of the behaviour added in this view to maintain the integrity constraints described by the structural model. This layering gives a freedom to use 1-to-many mappings between conceptual objects and application objects.
The behavioural schema describes the message-interface-protocol for methods applicable to instances of the classes involved in an application. Such an instance is generated based on an instance from the structural model, but can be extended with class-dependent operations, in addition to those inferred from the structural type. The mapping between the structural schema and the behavioural schema is by default one-to-one. An object-type in the structural schema corresponds to a subclass of Model-object in the behavioural schema, and a relationship-type corresponds to a subclass of class Relationship.

The structural schema can describe 1:1, 1:N and M:N relations between objects. It is important to note that the structural schema describes a structural abstraction which can be realized by different physical data-structures, both in the shared database and in the application-views.

7.1.4 Structure and Behaviour in SOOM

The tension between encapsulation of structure in a behavioural object-oriented model, and the revelation of structure in a structural object-oriented model is resolved through making the structural abstraction of relationships explicit. Encapsulation of representation is still taken care of, because there are many ways to represent a relationship, and the actual representation will be hidden for the users of a class. The application-programmer will directly use relationships to traverse and manipulate the object-structure.

The structural schema is also represented as an object-relationship structure, and stored in the database in the same way as other structural objects. The same structure-oriented editors can now be used to create both the schema, and the structures which obey the schema. The description of legal structures is done in the structural schema, which reg-
ulates which relationships are allowed between pairs of objects. The structure-oriented editors interpret this as a grammar for legal structures.

For easy integration into an existing object-oriented language, Model-object and Relation are subclasses of Persistent-Object. Each object has a list of incoming and outgoing relationships and each relationship knows its type and object in each direction. Objects can be either Aggregates or Atomic. Atomic objects are used for new basic types like text and drawing.

The description of legal structures are done in the structural schema, which regulates which relationships are allowed between pairs of objects. The structural model will not represent the content of atomic types, but only relationships between objects. The content-part will be considered as a sequence of bytes which the behavioural schema knows how to interpret.

Both objects and relationships are identified by a surrogate-id which is unique throughout the system. It consists of a database-identifier and a unique identifier within the database. These identifiers are used instead of normal object-oriented pointers, when a message is to be sent to an object. The database-identifier is logical and only used for inter-database references. An object-reference without a special time-stamp is assumed to be the most recent one. If a special time is given, it indicates the object as of a particular time, and the same time is applied to the objects referenced from this object.

7.1.5 **Tornado* - An Architecture for Realization of SOOM**

**An Architecture for Workstations and Servers**

The supporting architecture takes advantage of the layer between structural objects and behavioural objects in SOOM. The structural objects according to the structural schema are shared and is stored on a central server, Data Store (DS). Each workstation has a local workspace, Working Store (WS), which maps the structural objects into behavioural objects.

The Data Store is a centralized Object Server that stores structural objects, and makes it possible to check-out different parts of the object-structure to different users.

The Working Store is a transaction-based private workspace or a local object-manager, and acts as a single-user database. The Data Stores and Working Stores communicates through a remote procedure call (rpc) mechanism. Each Working Store instance is identified by a WS-identifier, used to identify the Working Store in transactions against the Data Store. The Working Store will maintain a history of all changes done locally to an object. When the object is stored back in shared storage, the last version of the object will be stored as default.

The Data Stores represent an interface corresponding to the structural schema. The underlying physical schema, however, stores an object together with its relationships in a unit. The interface is based on object-by-object operations to be applied to a single object, or specified to be applied to each object in a set. The underlying storage-system stores objects in variable-length records. Each object is associated with access-attributes and a timestamp.

Since the structure-oriented editors are parameterized by the structural schema, the Data Store is ensured that only legal structures will be inserted, and does not have to verify that at the end of each transaction. It will, however, enforce referential integrity by checking that there are no relationships going to objects which are not stored in the database.
Figure 7.2: Application-Working Store-Data Store Interface

Access to Multiple Databases

Each Data Store will contain one structural schema, and there might be multiple Working Stores connected to it.

Each Working Store manager can simultaneously operate multiple logical Working Stores and there can be inter-model references between different objects belonging to different Working and Data Stores. An inter-model reference is represented through local proxy-objects which know the identification of the correct Data Store. There is always a one-to-many correspondence between Data Stores and Working Stores.

Access to remote databases of various kinds, like relational databases are handled by viewing them as remote objects with a certain interface e.g. SQL, and sending them messages. A special kind of atomic objects can store SQL-queries and represent the answer of a query.

7.1.6 Performance Suitable for Interactive Design-applications

The first version of the system fetched objects automatically when they were referenced, hidden from the application-programmer. Since a remote-procedure-call to retrieve an object can be expensive, it was decided that explicit handling of remote operations makes it easier to create good user-applications. In most cases the application can deduce which objects are to be operated on next, or the user is able to decide how deep into the network-structure he wants to go.

The object-relationship model lends itself naturally to specification of transitive closures, based on a root-object, a set of object-types to use, and a set of relationship-types to follow to a certain depth. This can be used for specifying prefetching of objects. A serial version-history is maintained for each object, and by the use of a timestamp, a Working
Store can check if the object it has is the current one.

Transactions

Normal transactions are considered to be short and to conform to serializability. They are used to ensure that a sequence of operations are atomic. Local transactions are performed on objects already checked out to the workstation, and does not have to involve any remote transaction. A remote transaction has to be started from a local transaction and is supposed to take a short amount of time.

Cooperative Work is supported through a notion of long transactions. To handle long transactions, a checkOut / checkIn - mechanism is used. It is associated with each object in a Data Store if it is checked out, and to which Working Store. The different checkOut modes are: Read, Write, and All. Read will always succeed, while Write and All will only succeed if the object is not checked out in any of these modes already. All gives also update-rights to the access-keys. The users are identified through their WSidentifiers, and the user-names are given each time a new Working Store is created, and kept in the corresponding Data Store.

7.1.7 Experiences from SOOM and Tornado*

Experience from the use of SOOM suggests that the uniform use of objects and relationships in SOOM could be extended to include attribute-properties which belongs directly to an object or a relationship. It has also shown that the kind of relationships which represents a part-of-hierarchy is extensively used. Both locking and clustering could take advantage of this hierarchy being more explicitly represented.

Interactive editing-applications require an access-performance which is comparable with access to objects in virtual memory. This means that caching/checking-out to a private workspace is essential. There is a significant difference in performance between access-time the first time an object is touched, and has to be brought into the private workspace, and the access-time when it already is cached.

The performance given by a Working Store after objects are checked-out from a Data Store is sufficient for most of our applications, with a retrieval-performance of around 500 objects of 100-1000 bytes per second.

The object-relationship structures used in our document-models are fine-grained, and a typical document will contain some thousand objects. The object-by-object interface between a Working Store and a Data Store is the bottleneck in the system, giving a retrieval-performance of around 5 objects of 100-1000 bytes each per second. This results in an operating mode where users tend to work on local structures in the Working Store, for a day or more.

It seems possible to enhance this performance by shipping larger segments from a server, and to distribute the Data Stores itself out to the workstations and only centralize what is necessary for lock-management and handling of name-tables.

The Tornado* system was developed because there were no commercially available object-oriented database systems before 1988. The HyperModel Benchmark was made to evaluate these systems, to see if they met the requirements for practical data integration for tools. The HyperModel Benchmark is presented next.
7.2 The HyperModel Benchmark

The motivation for the HyperModel Benchmark [B⁺89d, ABM⁺90, BA91] was to be able to evaluate commercially available object-oriented database systems, to see if they met the requirements for data integration in integrated environments. In particular the Tornado* system used for SOOM was a special purpose database, and it was a goal to see if any commercially available object-oriented database system could replace that. The benchmark was initially developed while the author was a visiting scientist at the Oregon Graduate Institute, during the "Database Year" 1987-1988. The benchmark was refined and used for the evaluation of a number of object-oriented and relational database systems from 1989 to 1992.

The requirements for data integration presented in section 3.4, have been used as part of the "HyperModel Benchmark" in order to evaluate how well different database systems meet the requirements. The benchmark specifies both functional and performance-oriented requirements.

The result from doing this benchmark for both traditional relational database systems and object-oriented database systems, showed that the first generation of object-oriented database systems, on the market from 1987 to 1989, did not meet the performance requirements. However, the second generation of object-oriented database systems on the market from 1990 was improved, and proved to be good enough to be used for practical purposes.

7.2.1 The HyperModel Benchmark Description

Hypertext is a generic graph structure consisting of nodes and links. The nodes may contain text or other kind of data such as bitmaps. The links are used to describe references between the nodes. Hypertext has been proposed as a good model for use in Computer Aided Software Engineering (CASE) because it is possible to store software and documentation as hypertext graphs. We chose hypertext as the basis for our evaluation strategy because the requirements it places on the DBMS are quite similar to the requirements for other kinds of engineering applications. For example, the PCTE development reported in [GMT86, HSE89] has shown that a datamodel to be used in software engineering environments is very similar to the node-link concept in hypertext models. We have extended the basic hypertext model with two aggregation relationships, a one-to-many and a many-to-many relationship, typical of those found in part-of hierarchies. Thus it is a model that incorporates the three most commonly found relationship-types, one-to-many, many-to-many and many-to-many with attributes. Together with operations to be performed on text-nodes and bitmap-nodes in the structure, it is a model that is mapable to a variety of engineering applications.

Historically, evaluation of DBMS systems has been concerned with performance issues [BDT83]. The most well-known database-benchmark aimed at engineering applications "Benchmarking simple database-operations" is found in [Rub87]. The application-model chosen here is document-authorship represented by documents and persons with a many-to-many relationship between them. The HyperModel has extended this to a more typical engineering-oriented model.
7.2.2 The HyperModel Application

The HyperModel is a high-level model that can be mapped into an implementation on different database-systems. The COORASS modeling technique described in chapter 14 is used as the diagram technique.

![HyperModel Diagram]

Figure 7.3: The HyperModel schema in COORASS

The model shows that a Node has a set of operations and that nodes are related by three different relations. Ovals are object-interfaces, while diamonds are relation-interfaces. Operations (or attribute-operations) are described by a circle with a number or letter connected to an object-interface or a relation-interface. The actual operations or attributes are listed separately for the appropriate number.

The model has three relations a one-to-many (1:N), a many-to-many (M:N) and a many-to-many with attributes (M:NATT). The relation parent/children is a one-to-many ordered relation. The black triangle in a relation-diamond shows a “many” end, and the arrow points towards aggregate-components. The "O" within the relation-diamond means that the relationship is ordered. The partOf/parts relation also describes an aggregation, but is many-to-many and allows for a Node to be partOf multiple Nodes. The refTo/refFrom relation is many-to-many with two attributes. There are two subtypes of Node, TextNode and FormNode, which contains text or bit-map contents with appropriate operations. The fact that some nodes contain text or a form (bitMap) is described by a generalization/specialization-relationship between the Node-class and the TextNode- and FormNode-classes. The triangle on the lines between the interfaces symbolizes generalization.

7.2.3 Benchmark-operations

The following ten categories of operations contains the operations to be executed on the structures in the test-database:

1. Name Lookup
   This operation finds the hundred-attribute of a node based on a reference to it.
   The reference can either be given by the value of a unique attribute (key), or by a system-generated identifier. Both kinds of lookup are measured if applicable.

2. Range Lookup
   This operation finds the nodes satisfying a range-predicate based on the values of the hundred- or million-attribute. The range-lookup has a selectivity of 10% for
the hundred-attribute and 1% for the million-attribute, and allows for the use of an indexing-mechanism on the hundred and million-attributes.

3. Group LookUp
In group lookup we follow the defined one-to-many, many-to-many, and many-to-many with attributes relationships from random nodes and measure the average time per node to retrieve a reference to the related nodes. The return-object contains references to the nodes found, and it is storable in the database. If there is an iterator construct that gives access to nodes one at a time, the corresponding timing should also be given.

4. Reference Lookup
This is the inverse of group-lookup, given by following the 1:N, M:N and M:NATT relationships in the direction opposite to that of groupLookup.

5. SequentialScan
This operation finds the average time per object to look up the ten-attribute when all objects of the test-structure are visited. The database should be allowed to have other instances of class Node, (e.g., a second copy of the test-database) so the direct extension allInstances-of-Node cannot be used to do the sequential scan. Thus one cannot use the Class property belonging to each type to iterate through its elements. This would give all instances of type Node, not only those belonging to the test-structure.

6. Closure Traversal
These operations will start with a random node, and find the nodes transitively reachable by a certain relationship from the given node. For all the operations, we measure the average time per node returned by the operation. These operations are defined to start in a random node on level 3 (id 31-155) in the test-database, and then to follow either the one-to-many relationship or one of the many-to-many relationships to a certain depth.

7. Closure Operations
These operations will start with a random node, and also perform operations on the nodes transitively reachable by a certain relationship from this node. For all the operations, we measure the average time per node involved in the operation. These operations do more than just return the nodes found in the closure, e.g. by summing up or setting values.

8. Editing
TextNode-Editing: The editing-operations are intended to demonstrate the power of the database-programming language and to test if any statements have to be executed in another programming language. The operation for TextNodes is to get a random textNode and substitute the occurrence of substring “Version1” with “Version-2” in the first run, then substitute back again in the second run. This requires that the text-structure be extended or replaced by a new structure. For systems that support handling of versions and variants, the editing-operations can also be used to demonstrate these features.

Bit-Editing: The FormNode is a node-type that contains a bitmap realized by the Form-type. The edit-operation is inversion of a sub-rectangle of the bitmap. The FormNode is defined as follows:

9. Create-and-Delete
These operations measures the time it takes to create and delete objects.
10. *Open-and-Close*
These operations measures the time it takes to open and close the database-system.

To prevent caching from earlier operations from having an effect on timing, the database is closed before each new operation. Each operation is run 50 times on randomly picked start-objects (cold run). To measure the effect of caching (warm run), the operation is then run once again on the same 50 objects.

### 7.2.4 Experiences and Results from the HyperModel Benchmark

![Graph showing performance comparison between OODBMS and RDBMS](image)

**Figure 7.4**: OODBMS vs RELDBMS performance on a Closure Traversal-operation

This section discusses the experiences from the different evaluation runs. The object-oriented database-systems which have been evaluated are Gemstone, ONTOS and ObjectStore. Comparisons have also been done with the relational database systems Sybase and Oracle.

Figure 7.4 shows the typical performance characteristics for a client-oriented OODBMS (ObjectStore) versus a RDBMS (Sybase) on a closure traversal operation (Get references to all nodes reachable from a random node on level 3, by following the refTo/refFrom relationship recursively to a depth of twenty-five). The performance is measured both in cold (nonCached) and warm (cached) runs, and as the size of the database increases from level 1 (781 Nodes/3906 Links) through level 2 (3906 Nodes/19531 Links) to level 3 (19531 Nodes/97655 Links). Typically the OODBMS have a much better performance in the warm run, when it is allowed to cache objects. This performance deteriorates, however, when the total size of the cached objects increases beyond the size of the memory of the machine. Then the virtual memory system is used, and if the access to objects is totally random it becomes more and more probable that the next object to be accessed has to be retrieved from virtual memory, eventually with page-trashing as a result. The lesson from this is that client-oriented OODBMSs can be used to an advantage for applications where the working set of objects easily can be controlled. Detailed measurement-results are presented in [ABM+90, Lar92, Inv92].
The main conclusion is that performance-differences are due to differences in architecture, not to differences in data model. In particular the main difference is the use of a client-oriented or a server-oriented architecture. This is discussed in more detail in the section 7.2.6.

A refinement and simplification of the HyperModel Benchmark for multi-user purposes has been done by Arne Bodin Larsen [Lar92]. The revised benchmark has been called TEP - Test Evaluation Procedure for DBMSs. A performance-evaluation study has also been done by Ivar Mølnevik, [inv92], in a comparison between object-oriented and relational database systems.

The HyperModel Benchmark has been used by others as a basis for later work on database benchmarking, in particular the inclusion of traversal-operations have been adopted. A revised version of the "Simple database benchmark" is reported in [CS92]. A recent benchmark, OOT, is presented in [CDN93]. Both of these are influenced by the HyperModel Benchmark.

The next section presents different object-oriented database systems and how they support the requirement areas for single model data integration.

### 7.2.5 Object-Oriented Databases and Single Model Data Integration

This section presents an overview of how the different object-oriented database systems meet the requirements for single model data integration.

<table>
<thead>
<tr>
<th>Requirements-area</th>
<th>Gemstone</th>
<th>Ontos</th>
<th>ObjectStore</th>
<th>Sybase</th>
<th>Ingres</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDS-1: Basic ERA-model</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-2: Complex objects</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-3: User-defined data types</td>
<td>+</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-4: Prog. language support</td>
<td>+</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-5: Versions/variants</td>
<td>✗</td>
<td>✗</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-6: Metadata/Schema</td>
<td>+</td>
<td>+</td>
<td>✗</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RDS-7: Views</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RDS-8: Rules / Triggers</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RDS-9: Query language</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RDS-10: Security</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RDS-11: Distributed Architecture</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-12: High Performance</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-13: Concurrency</td>
<td>+</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-14: Cooperation</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RDS-15: Logging - recovery</td>
<td>✗</td>
<td>+</td>
<td>✗</td>
<td>✗</td>
<td>+</td>
</tr>
<tr>
<td>RDS-16: Data Interchange</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 7.1: Evaluation for Single-model Data integration

**Basic ERA-model (RDS-1)**

The object-oriented data models do not support the notion of relationships and attributes directly.

In Ontos and ObjectStore it is possible to define attributes as public instance-variables.
7.2 The HyperModel Benchmark

This, however, violates the object-oriented encapsulation-mechanism. ObjectStore supports a macro-facility for defining relationship-maintenance functions, which automatically can maintain a two-way constraint between instance-variables.

**Complex Objects (RDS-2)**

There is no special support for complex objects/aggregation-relationship, but clustering may be specified at creation-time.

**User-defined Data Types (RDS-3)**

The class-concept makes it easy to create user-defined types.

**Programming Language Support (RDS-4)**

ONTOS and ObjectStore are persistent C++ systems with good support for C++, but weaker support for other languages. Gemstone has its own programming language, OFAL, with objects that might be accessed from Smalltalk, Pascal, C or C++.

**Versions/Variants (RDS-5)**

ObjectStore has a configuration-facility which can be used for the maintenance of versions and variants for objects.

**Metadata/Schema (RDS-6)**

Schema as metadata is supported by Gemstone and Ontos, but not by ObjectStore.

**Views (RDS-7)**

Object-Oriented database-systems do not naturally support the external schema level in the three-schema architecture view of databases.

The physical level is naturally mapped to class implementations and the conceptual level to class-interfaces, but there is not natural mapping for views in the external level. There is currently much research in the area of "Views in Object-Oriented Systems", but no common agreed upon model has yet appeared.

**Rules and Triggers (RDS-8)**

The first version of ONTOS, Vbase, had a notion of triggers when an attribute was accessed, but this is not supported in ONTOS. Rules are not supported by any of the systems.

**Query Language (RDS-9)**

The current OODBMSs on the market are typically more Persistent OO languages systems with navigational access, rather than DBMSs giving good support for (interactive) queries
on large amounts of data.

There has been some work on making an "Object-SQL", a language similar in flavor to normal SQL. ONTOS has a tool for doing this.

ONTOS Object SQL:

```
SELECT propname1, ..propnameN, procmameN
FROM aggregate1, ...aggregateN
WHERE expression
```

ObjectStore supports a predicate-oriented query-language for the retrieval of objects from collections.

A problem with query-optimization for object-oriented languages is that knowledge about implementation and indexes somewhat breaks the notion of encapsulation. E.g. An application-programmer should not know if age of a person is implemented as an attribute, or calculated by birthday and todays date.

Security (RDS-10)

There is not any sophisticated access-control for any of the systems. Access-control is on the level of databases, or on segments.

Distributed Architecture (RDS-11)

All the C++-based OODBMS-systems are based on a client/server architecture where the operation-execution is done in the client-process. Gemstone is executing operations in the server-process.

High Performance (RDS-12)

The second generation products have good performance when objects have been cached in the client.

Concurrency (RDS-13)

Different mechanisms for optimistic concurrency control are supported by the systems.

Cooperation (RDS-14)

The workspace-facility of ObjectStore may be used to support long transactions.

Logging, Backup and Recovery (RDS-15)

The systems have basic support for this.
Data Interchange (RDS-16)

No particular support is given for data interchange.

7.2.6 Client- and Server-oriented Architectures

**An Object-Oriented database system?**

![Diagram of object-oriented database system]

The programmer's responsibility

The database system's responsibility

Figure 7.5: The programmer’s responsibility and the database system’s approaches.

Answers to the first of the following three questions will separate OODBMS in two main groups: The Client-oriented and the Server-oriented groups. Answers to the next two questions might be derived from the group-characteristics.

- Where is the operation-executing process? (client-application or server-database)
- Who is responsible for the operations? (programmer or database)
- Which PL for methods - application? (one or many)

The operation-executing process is the layer where the operations associated with the objects are executed. Typically this is either at the client-side in the same process as the application - (the client-oriented group) or at a separate database-server - (the server-oriented group).

The Client-oriented Group.

The Client-oriented group represents systems where the operation-executing process is the same as the application process. This is typical for most current C++-based OODBMSs. This gives no impedance mismatch between the application programming language and the the database-programming language, and very high performance for access to cached objects - due to no interprocess communication overhead.

In the client-oriented approach the operation-executing process is the client application-process. This means that the operations and the application need to be written in the same language, typically C++ or Smalltalk. It is, however, possible to access other external procedures and functions, from the implementation-part of the operations.
Figure 7.6: Server-oriented (Extended DB) or Client-oriented (Extended PL) OODBMSs

It is important to note that a client-oriented system might be accessed again from an external client, and thus be viewed as a server-oriented system. The Distributed Object Management approach supported by the Object Request Broker infrastructure from OMG (Object Management Group) encourages this. Here an OODBMS might register which classes and objects it exports for external access. It seems natural that the client-oriented systems will evolve to also support this kind of access, as a move towards more truly distributed OODBMSs. The current multi-client - multi-server systems are really not truly distributed, they only give an opportunity for clients to cache objects from multiple storage-management servers.

The advantages of the client-oriented group are no impedance mismatch between application programming language and database programming language, and high performance for access to cached objects. The disadvantages are that the application-programmer often have to take responsibility for managing the methods (if there is not a database-system using operating-system support for dynamic linking) and that it is difficult to support multiple writers on the same object over a longer time.

The client-oriented group favors "engineering-design"-applications where one person might work intensively on a smaller sub-part of the total database for a longer period.

The Server-oriented Group.

The Server-oriented represents systems where the operation-executing process is separate from the application-program process. Examples of systems in this group are Open-ODB from HP, with the ObjectSQL-language, and Gemstone from Servio with the OPAL Smalltalk-like language. A full application might be written in these computationally complete languages, typically they are however accessed from applications in external languages like C, Pascal, C++ and Smalltalk. Neither OPAL nor ObjectSQL support objects for creating user-interfaces, and forms and screens are examples of a part of an application that needs to be written in another programming language.

This can to some extent be compared with the stored-procedure approach supported by some RDBMSs. More of the application, e.g. integrity-checking etc. might then be written
as a stored procedure with an interface of input/output-parameters. SQL-based languages for writing RDBMS stored procedures are not computationally complete, so that some parts of the application often has to be written in another programming language.

The server-oriented architecture does not have the same possibility for high-performance access to objects as the client-oriented architecture. In particular if one wants to have e.g. a person-object available in a C++-based or Smalltalk-based application and this object actually resides on the server there is an overhead involved in accessing this "remote" object, eventually from a "local proxy" object. A migration-path for resolving this is to have a true distributed object management system where objects can migrate from one execution-space to another - This, however, introduces many other problems such as having multiple compiled methods for different platforms etc.

The advantages of the server-oriented group are that it easier provides for access to objects from many different programming languages, and that it more naturally supports the notion of encapsulated objects providing services through their interface. The disadvantages of the server-oriented group is that it still might be an impedance mismatch between application programming language and database programming language, and that it does not have the same opportunity for high performance due to the overhead of interprocess communication.

The server-oriented group favors "centralized" applications where common information- and computation-resources needs to be accessed from many different "applications".

Client-oriented versus Server-oriented Architecture

Object-Oriented Databases as they exist today are a viable alternative for application-areas which involves cooperative work on complex information. It is suited for those applications which can take advantage of the current client-oriented architectures of most object-oriented databases. Note that this is an architectural issue - NOT a data-model issue, and in the future we might see both relational database-systems supporting a client-oriented architecture and more object-oriented database systems supporting a server-oriented architecture. Client-oriented OODBMSs are not well suited for typical batch/report-oriented applications on large amounts of data, if the access-pattern requires arbitrary navigation in a data-volume which is much larger than practical virtual memory-space of the client.

Server-oriented OODBMS architectures are more comparable with the client/SERVER-architecture of current relational systems. Object-Oriented and Relational databases will then probably not differ so much from a performance viewpoint, instead one should consider the added modeling power of object-oriented models.

7.2.7 Object Oriented Database-Systems - Conclusion

The benchmark-evaluations showed the performance differences and trade-offs between a client-oriented object-oriented database-system and a server-oriented relational database-system. A presentation of other positive and negative aspects is given here.

Some of the positive aspects of the object-oriented model are:

- Many high-level modeling primitives gives modeling power (Inheritance - Complex Objects).
- No "Impedance Mismatch" - same language for database and application
7.3 Contribution and Conclusion

- Support for reuse of code through inheritance
- Easier maintenance and extensibility through encapsulation and subclassing

Some negative aspects of the object-oriented model are:

- No standards (yet)
- Navigational access instead of Declarational access
- Query languages with indexes breaks encapsulation
- Integrity rules might also break encapsulation
- Not total Data Independence - some changes might require rewrite of access-methods. (No View-concept)

It is possible to enhance an object-oriented model to also support structural abstractions. The SOOM-model described in the previous section is an example of that.

7.3 Contribution and Conclusion

The goal of this chapter was to show that object-oriented data integration is feasible and better than traditional data integration.

The evaluation showed that the object-oriented database systems did not meet the requirements for support for relations and attributes (RDS-1), and complex objects (RDS-2). The development of the SOOM-model shows, however, that an object-oriented model can support both behavioural and structural object-orientation. Relationships is thus an extension which might be added to object-oriented databases in the future.

The HyperModel benchmark has shown that object-oriented database systems have a better modeling power than relational database systems, and a better performance for applications where the working set of objects easily can be controlled. The PCTE OMS also has a good modeling power, but do not support objects with behaviour, and do not have a very good performance for fine-grained objects.

The HyperModel benchmark showed that the second generation object-oriented databases are powerful enough to be of practical use in certain situations. Another contribution of the HyperModel benchmark is the understanding gained by both system-vendors and users, about how an object-oriented database system actually behaves. The strength and limitations of client-oriented versus server-oriented architectures are shown by the benchmark. The HyperModel benchmark has also been used as a basis for later work on benchmarking in [CS92, CDN93].

The conclusion is thus that object-oriented data integration is better than traditional data integration in situations where the applications need high performance and the modeling power offered by an object-oriented model, and are able to control the working set of objects. It is also better if it is a need for the modeling power of an object-oriented model and the performance requirements allows for the use of a server-oriented object-oriented database.
Chapter 8

Experiments with Object-Oriented Presentation Integration

The goal of this chapter is to show that object-oriented presentation integration is feasible and better than traditional presentation integration.

Support for an object-oriented approach to model-oriented presentation integration is introduced through a description of the model-view-controller architecture, and its application in distributed architectures like the ESF architecture. This separation also makes it possible to create multiple user-interfaces for one functional component and to more easily provide the functionality of multiple functional components through one user interface. The development of user interface components in the SITE and CODEDISC prototypes shows examples of model-oriented presentation integration.

The Smalltalk model-view-controller framework classes used in the SITE prototype, and the CommonView framework classes used in the PIONA system are examples on object-oriented support for display-oriented presentation integration.

8.1 Object-Oriented Presentation Integration

Object-oriented presentation integration is useful both from an environment user perspective and from an environment builder perspective.

From an environment user perspective it means that the user perceives the user interface as presenting a number of objects that offer operations. This can typically be presented through a desktop with objects.

From an environment builder perspective it means to use an object-oriented approach for the programming of the display-part of the user-interface. An object-oriented view can also be used for the interaction between the user-interface-part and the functional part of an application. This is exemplified by the model-view-controller architecture described next.
8.2 The Object-Oriented Model-View-Controller Architecture

The model-view-controller architecture has been developed in the context of the Smalltalk programming language [KP88, GR83], but the architecture has general applicability. It divides an application into three different parts: model, views and controllers.

- A Model is the functional part of the application. It is a single object, which typically manages a structure of other objects. The Model shall have no knowledge about the different user-interfaces, and make no preassumptions. - This makes it easy to dynamically add new user-interfaces. A Model is a subclass of Class Model, so it inherits behaviour for the change/update-protocol.

- A View is an object that arranges for output to be presented for the user. A View could highlight some aspects of a model, and suppress others. It is thus acting as a presentation filter.

- A Controller is an object that arranges for input from the user to the system. It handles and interprets mouse-events and keyboard-events, and acts upon these in order to send suitable messages to the model, or to the view.

The Smalltalk class-library has predefined classes for model, view and controller which contains the necessary functionality to create an "empty" application. This is more than a set of independently reusable classes, because the classes contains a complex interaction pattern for the interaction between instances of the three classes. This is called an application framework, because it contains the necessary facilities for building an application.

The object-oriented concepts of subclassing, inheritance and dynamic binding make the application framework approach particularly feasible. In order to create a particular system one creates subclasses of some of the predefined classes and reimplements some selected methods. These methods contains system-specific information, such as what to display in a view.

Model-View-Controller can be viewed as an example of a general principle of building an application-framework for a particular class of user-interface-oriented applications. The principle of making a design of default interaction among a set of objects, which can be customized by making subclasses reimplementing some selected methods can be applied more generally. Other areas where the application framework approach have been applied successfully are to develop a Simulation-Framework, a Financial-Application-Framework and a generic Graphical Editor Framework.

The Smalltalk Model-View-Controller framework is made to support consistency-update of views. In the MVC-triad there is one model-object and a number of interacting View-Controller pairs. When a view is associated with a model-object, the view and controller-objects will automatically be added as dependents of the model-object. Whenever the model-object executes a method that leads to a change of the state of the object, it sends itself the message: changed: #aspectX, this leads to the message update: #aspectX being sent to all the dependent objects. They may then check on the aspect-parameter and see if they need to change their presentation. This approach has been focused more generally in the context of notification-oriented control integration.

It is important to note that the separation between model and view/controller pair is different from the separation between X-client and X-server in the X-Window system. This system uses an X-server on each workstation that receives commands from and sends
events to a possibly remote X-client. The model-view-controller architecture might be
imposed on a typical X-application by splitting the X-client into two interacting parts, one
part concerned with the user interface and interaction with the X-server, and another part
concerned with the functionality of the application.

8.3 The ESF Separation of User Interaction and Service
Components

The ESF architecture described in section 2.1.2 and section 4.8 makes a distinction between
user interaction components and service components.

This is essentially a realization of a model-view-controller architecture, where views and
controllers are bundled into the user interaction component.

From a systems integration point of view this provides a good basis for presentation in-
tegration. The separation makes it possible to more easily create multiple user-interfaces
for one functional component and to more easily provide the functionality of multiple
functional components through one user interface.

8.4 SITE Presentation Integration Experiences

In the SITE prototype described in section 6.1 the ESF principle of separating user inter-
action and service components have been used in a practical case.

Figure 8.1 shows two different user interaction components, one for a project manager
and another for a project member. Both interaction components are using a common
underlying service component. This has been described further in section 6.1.

The main experience from this is a set of rules and guidelines for how to divide work and
functionality between the user interaction and the service components.

It is not always obvious how to divide the work between the service components and
the user interface components. A round-trip of an empty service-request took around
10-12 milliseconds on a Sun-3. This means that the separation of labour between a user-
interaction component and a service-component should be so that the functional operations
of the service component are of a granularity which is suitable for this performance.

Experience with SITE has given us the following guidelines for dividing responsibilities
between User Interaction Components (UICs) and Services Components (SCs):

- Information closely connected to the way the user interacts with the system should
  be included in the UIC.

- All permanently stored information and functionality which is essential to the in-
tegrity and consistency of the problem domain must be implemented in the SC.

- It is allowed to cache information in the UICs, but the state of the SCs must always
  be consistent, ensuring correctness when other components access it.

- The SC functionality may be used by several different UICs so therefore generality
  and consistency is essential.

The SITE prototype has achieved display-oriented presentation integration through the
use of the Smalltalk model-view-controller application framework. The resulting user-
interaction components are portable between different windowing systems such as Windows, X-Windows and Macintosh through the portability of Smalltalk.

The architectural separation of user interaction components and functional components provides the necessary basis for doing model-oriented integration.

8.5 CODEDISC Presentation Integration Experiences

Figure 8.2 shows the browser-user interface developed for the CODEDISC prototype described in section 6.2.

The CODEDISC architecture focused on the separation between user interaction and service components, and the user of non-object-oriented tools for the creation of the user-interface-components.

Initially the user-interface was developed through the interactive tool, TeleUSE, for inter-
active creation of X/Motif user-interfaces. It eventually turned out that his tool required a closer coupling between the user-interface and the functions than we wanted, so the final version of the user interaction components was developed directly in the Motif C-based toolkit.

The main goal of CODEDISC was to show that it was feasible and useful to separate the development of user-interaction components and service components. The development team early focused on defining the interfaces of the service components, and then user-interaction-components and service-components were developed in parallel by different people. This allowed some people to focus on the end-user presentation, while other people focused on the development of a stable and efficient service.

Some experienced problems and issues from the CODEDISC prototype was in particular the difficulty of how to divide work between the user interaction parts and the functional parts. It is not always clear where to split a tool into an UIC-part and a SC-part. For the Browser a special service-component was made to handle the interaction with the database. This currently contains some status information for the user-interaction, which
Table 8.1: Evaluation for Presentation Integration

instead could be moved into the UIC itself.

The CODEDISC prototype has achieved display-oriented presentation integration through the use of the TeleUSE interactive tool and the Motif user-interface toolkit. The architectural separation of user interaction components and functional components was particularly designed and supported through a remote-procedure-call interaction mechanism. The encapsulation of services in C++ objects made it particularly easy to access multiple service components, and achieve model-oriented presentation integration.

### 8.6 PIONA Presentation Integration Experiences

The separation between display-oriented and model-oriented integration is particularly useful for presentation integration of existing systems and applications.

Experiences from this have been gained through the development of a new user-interface part for an existing system in the SI PIONA project [BR90]. PIONA is a system for graphical display of the chemical substances of oil and gasoline. This was designed by the author according to the separation-principles of the model-view-controller architecture.

The object-oriented C++ library CommonView [glo91] was used for developing the user interface. This is an application framework that provides portability between Windows and Motif user-interfaces. The framework does not explicitly support a separation between the user-interface part and the functional part of an application. However, by applying the principles in the system-design it was easy to later separate the user-interaction parts and the functional parts of the system.

### 8.7 Contribution and Conclusion

It has been experimented with different object-oriented user interface frameworks, such as Smalltalk Model-View-Controller and CommonView. Table 8.1 shows the evaluation of the Smalltalk model-view-controller framework and the C++ CommonView framework with respect to the requirements of presentation integration.

These object-oriented frameworks meet the stated requirements for display-oriented presentation integration. The frameworks could improve in support for graphics, and copy/cut paste in different windowing environments. Interactive tools have started to become available, but have not been used in the experiments described here. Look-and-feel portability is supported by the portability of both Smalltalk MVC and CommonView between differ-
8.7 Contribution and Conclusion

ent windowing-systems. The SITE prototype has achieved display-oriented presentation integration through the use of the Smalltalk model-view-controller application framework. The resulting user-interaction components are portable between different windowing systems such as Windows, X-Windows and Macintosh through the portability of Smalltalk.

Object-oriented application frameworks for user interfaces have two advantages compared to traditional procedural toolkits. The possibility for dynamic binding makes it possible to specialize applications by re-implementation of selected operations, and the encapsulation of classes makes it easier for the framework vendors to provide frameworks which are portable to multiple windowing systems. Even procedural toolkits like X-toolkit for X-Windows or MS-Windows are based on an object-oriented philosophy around user interface elements like widgets and windows but do not yet offer this based on an object-oriented language.

The requirements for model-oriented presentation integration is met by an architectural separation of user interaction components and service components. This separation also makes it possible to create multiple user-interfaces for one functional component and to more easily provide the functionality of multiple functional components through one user interface. Experiences have been gained in an architectural separation of the user-interface part of an application from the functional part. Both the user-interface part and the functional part may or may not be developed by object-oriented technology. This is, however, a technique which also can be adopted by procedural toolkits.
Part III

COOP: A Unified Object-Oriented Approach to Systems Integration
Chapter 9

COOP - An Object-Oriented Framework for Systems Integration

This chapter provides a description of the COOP Integration Framework and an introduction to the COOP Object Model, COOM. COOP is an acronym for COOPerative systems. COOM is an acronym for COOP Object Model. First, background and motivation for COOP is given, based on the work presented in part I and part II, then the COOP vision and architecture is described. Next, an overview of the principles of COOM is given, finally the structure of the rest of part III is presented in section 9.9.

9.1 Background and Motivation for the COOP Integration Framework

Part I presented a taxonomy of problems, requirements and solutions related to systems integration in cooperative work environments. Systems Integration was divided into four main areas: process integration, control integration, data integration and presentation integration. The focus has been put on the last three of these.

Part II presented isolated experiences with an object-oriented approach to each of the three focused integration areas: control integration, data integration and presentation integration. The motivation for COOM is to unify these through one common model.

Figure 9.1 shows the refined Double "Toaster" model for systems integration from part I, with an added annotation for how an object-oriented approach has been used in each area.

As a basis for the COOP integration framework, it has been identified how an object-oriented approach can contribute in each integration-area.

Object-Oriented Control Integration: Chapter 6

Object-Oriented request-oriented Control integration has been evaluated through experiments with an "Object-Oriented Software Bus", an interaction-mechanism which facilitates request-oriented interaction [Ber92c]. The interoperability among components will be supported if they request and provide services through the mechanisms of a software bus [Ver91].

Distributed System technology is mainly dealing with control integration aspects. The main contribution from this area is in interface-definition languages for specifying the
interface of objects, and interaction-mechanisms for remote-procedure-call and message-passing. Examples of such interface-definition-languages are NCS NIDL [ncs89], OSF DCE IDL [osf90], ANSA IDL [ans89] and OMG Object Request Broker IDL [obj92a]. Object-Oriented notification-oriented control integration has been evaluated through experiments with a Smalltalk-inspired change-update mechanism [Ber92c].

Object-Oriented Data integration: Chapter 7

Object-Oriented single-model data integration has been evaluated through experiments with object-oriented database-systems [ABM+90, BA91, Ber91d] and experiences from the object-oriented model SOOM [Ber88c].

Database System technology is mainly dealing with data integration aspects. The contributions from this area are modeling-concepts and schema definition languages from both Semantic Datamodels [Che76] and from Object-Oriented Database-systems [LLOW91, PS87]. Classical abstraction mechanisms like Classification, Generalization/Specialization, Aggregation and Association have originated from the area of semantic data modeling.

Object-Oriented multi-model data integration has been partly evaluated through experiments with the use of an object-oriented model as a common model for both object-oriented and relational database-systems [Sol92, Tin92, Low92]. Also others have advocated for [CT91, LM91] and experimented with an object-oriented model in multi-databases [HMZ90, Ber91h, R+91].

Object-Oriented Presentation integration: Chapter 8
Object-Oriented presentation integration has been evaluated through experiments with object-oriented application-frameworks [KP88]. In particular a split between the user-interface and the semantic part of a tool makes it easier for one user-interface to interact and integrate with multiple semantic tool-parts, and for one semantic tool to be utilized from different user-interfaces.

**Process integration:**

Interoperation between process interaction engines and process interworking engines is supported through object-oriented control integration and object-oriented data integration. This has been illustrated through the SITE prototype in chapter 6.

**Unification in the COOP Integration Framework:**

The initial experiments with the different integration areas have shown that the object-oriented approach is better than traditional approaches for control integration, data integration and presentation integration. The COOP Integration Framework takes this a step further by providing a unified object-oriented approach for these integration areas, through the COOP Object-Oriented Model, COOM.

The goals and solution approach of COOP, the COOP architecture, and the COOM Object Model and is introduced next.

### 9.2 COOP Goals and Solution Approach

The COOP vision was described in section 1.3 as follows: "The COOP Vision is to be able to unify control integration, data integration and presentation integration through the notion of "Distributed Persistent Objects"."

![COOP Architecture Diagram](image)

**Figure 9.2: Principal COOP Architecture**

The goal of the COOP Integration framework is to facilitate integration and interoperability between heterogeneous systems. Object-Oriented technology is a good basis because it conceptually unifies control integration and data integration through objects which encapsulate services and data through an interface. It is quite natural to extend object-oriented technology with support for distribution and persistence.
The framework will make it much easier to create new systems and applications from pre-existing parts, because the already existing systems and data within them can be viewed through a unifying object model. The object model is called COOM, COOP Object-Oriented Model.

The main parts of the COOP architecture are shown in figure 9.2 and in figure 9.3.

- **Distributed Persistent Object Space** is a uniform world of objects that models available components through the new COOM object model. It consists of an object dictionary and a number of local object spaces.

- **Object Dictionary** contains a number of interface manager objects that are used to access objects and relations in the local object spaces.

- **Local Object Spaces** contains implementation manager objects and objects that reflects functionality and data in underlying service components or databases.

- **Service Components (SCs and DBMSs)** might be mapped into the object space either as one object (large grain) or by exposing a number of internal objects (fine grain).

- **Databases** are mapped into the object space as a number of objects reflecting the contents of the database (fine grain).

- **Tool User Interface Components (UICs)** interacts with the available objects in order to provide the required functionality for a user. These might themselves be mapped into objects.

![Figure 9.3: COOP related to the Double Toaster model](image)

Figure 9.3 shows how the COOP architecture is related to the integration areas shown in the Double Toaster model.

- **Request-oriented Control Integration** will be supported by a possibility to send requests to objects in the object space.

- **Notification-oriented Control Integration** will be supported by a possibility for objects and processes to register interest for events from objects in the object space.

- **Single model Data Integration** will be supported by a possibility to map the contents of a DBMS into objects and relations in the object space.
Multi model Data Integration will be supported by a possibility to manage the contents of multiple databases simultaneously.

Display-oriented Presentation Integration will be supported by a possibility to use an object-oriented user interface framework to develop portable user interface components.

Model-oriented Presentation Integration will be supported by a semantical split between user interface objects and functional objects, and the possibility for a user interface object to interact simultaneously with multiple functional objects.

Process Integration will not directly be supported, but the object space will be able to support interoperation between process interaction engines and process interworking engines as described in section 2.4.2.

It is not a goal to initially support all the requirement-areas described in chapter 3, but to provide a framework in which it will be possible to provide basic solutions, with a possibility for later extensions. Some of the requirements-areas which will be postponed for later extensions are version handling (RCF-6, RDS-7), security support (RCF-7, RDS-13, RDM-6) and transaction management (RDS-10, RDS-11, RDM-5).

Figure 9.4: COOP Architecture related to ESF Architecture

The COOP architecture can be viewed as an object-oriented extension to the ESF architecture. Figure 9.4 shows the mapping between these architectures. Tool user interaction components might be either general user interaction components, or a specialized process interaction engine.

- The Software Bus is substituted with a distributed persistent object space that provides for both control integration and data integration.
- Service Components are supported, both as large-grain components (SC 1) and fine-grain components (SC n).
- Databases are supported through a mapping to fine-grain objects.
- User Interaction Components are mapped into objects. They might internally be build by objects but these are then not externally available and are not shown.
Process Engines
Process interaction engines can be realised as interaction components, and process interworking engines (PIwE) can be realised as service components. An engine is then mapped into one object as a large-grain component.

The following describes the goals of COOP in the conflicting requirement areas described in section 3.1.

- **Heterogeneity versus Homogeneity**
  It is a goal to support heterogeneous components through a homogeneous object model.

- **Autonomy versus Common policy**
  It is a goal to support as much autonomy for integrated components as possible.

- **Distribution versus Centralization**
  It is a goal to hide distribution through the object model.

- **Fine-grain versus Large-grain**
  It is a goal to support both large-grained and fine-grained objects in a uniform way.

- **Efficiency versus Flexibility and Extensibility**
  It is a goal to give priority to efficiency, and not sacrifice too much flexibility.

- **Simplicity versus Coverage**
  It is a goal to achieve coverage and simplicity through the use of a uniform object model.

- **New language versus Existing languages**
  It is a goal to use existing programming languages, such as C++ and Smalltalk.

- **Variety of Components versus Functional Usability**
  It is a goal to support a large variety of different components.

- **Syntactical basis versus Semantical integration**
  It is a goal to provide a good syntactical basis for understanding of available functionality and data, as a basis for discussions for agreement on semantical integration.

### 9.3 COOP Architecture

Figure 9.5 shows the principal architecture of COOP both at development time and at run-time.

At development time the schemas or interfaces of existing systems are used as a basis for the creation of a corresponding COOM-schema in ODL (Object Definition Language). The development-support provides tools for making existing components and data in databases available as distributed persistent objects. The interfaces of existing system-components, and the schemas of existing DBMSs are used as a basis for generating a COOM-schema in ODL. This is a semi-automatic process with human support for semantic enrichment. A set of ODL-schemas might be used directly, or integrated through Schema-integration. The initial systems to be supported are relational database-systems, an object-oriented database, and distributed components with interfaces in OSF/DCE IDL [osf90] (Distributed Computing Environment Interface Definition Language), NCS NIDL [ncs89] or OMG Object Request Broker IDL [Obj92a]. The ODL-compiler produces interface-classes
and interface manager objects in a given object-oriented programming language (currently C++ or Smalltalk-80). The backends produce implementation-classes and implementation manager objects corresponding to the interface-classes with an implementation which is dependent on the target system. The interface-classes are the basis for an application-programmer who wants to use existing services. The application-programmer will be using predefined and interface-specific operations to interact with the objects, and might use OQL (Object Query Language) in order to retrieve existing objects.

At run-time, a tool/application written in an object-oriented language can make requests to available objects. Objects might be retrieved through queries expressed in OQL (Object Query Language) or created through a create-message to an interface-manager object. The objects serve as representatives or proxies for the real data and functionality in the underlying systems. The distributed persistent object space is maintained through an object-oriented DBMS. A notification service is used as a notification event manager. A query compiler and interpreter is used for the processing of OQL queries. The initial version of COOP does not support sophisticated transaction-management or security features.

The basic principles and concepts of the COOP Object Model, COOM, is presented next.
9.4 COOM - Principles and Concepts

9.4.1 COOM - Basic Principles

The main contribution of COOM is a merge and unification of behaviourally object-oriented concepts from object-oriented languages, OODBMSs and interface definition languages for distributed systems, with structural concepts from entity-relationship modeling. Behavioural concepts focus on encapsulated objects with operation-interfaces, while structural concepts focus on attributes of, and relationships between, objects. COOM supports multiple simultaneously implementations of interfaces.

The basic principles for COOM are:

- A model with support for both behavioural and structural objects
- To represent the functionality and data of heterogeneous systems and databases as encapsulated objects in a distributed object space.
- The separation between interface, implementation, extent and object-factory.
- The use of three languages: ODL - OML - OQL.

9.4.2 A Model with Support for both Behavioural and Structural Objects

COOM consists of three parts, $COOM_F$, $COOM_B$ and $COOM_S$. The COOM foundation model, $COOM_F$, is the basis for the COOM behavioural model, $COOM_B$, and the COOM structural model, $COOM_S$. The object-interface layer $COOM_B$ enhances $COOM_F$ with support for object management and predefined object-functionality.

The abstract attribute- and relation-interface layer $COOM_S$ enhances $COOM_B$ with the notions of abstract attributes and relations. A set of operations “belonging” together can express certain semantics, i.e. attributes and relationships. This is used to create an abstract structurally object-oriented model as an enhancement to a behaviourally object-oriented model. This model is abstract, because the actual representation of attributes and relationships is hidden behind a behavioural interface. This makes COOM a fully object-oriented model, according to the definitions in [Dit86].

COOM unifies control integration through $COOM_F$ and $COOM_B$ with data integration through $COOM_S$. A separation between user interface objects and functional objects supports presentation integration.

9.4.3 Encapsulated Objects in a Distributed Object Space

The basic concept in COOM is the encapsulation of data and functionality in underlying heterogeneous systems and databases as objects that have an interface provided as a set of operations. The basic model is “Interaction through message-passing between objects” and all interaction happens through messages sent to encapsulated objects. Encapsulation hides the actual implementation. Objects with the same interface might be implemented differently, and utilize existing functionality in underlying heterogeneous systems and databases.

\[1\] The words behaviourally and structurally are for COOM used according to the definitions in [Dit86]. A behaviourally object-oriented model focuses on operations, while a structurally object-oriented model focuses on structure. A fully object-oriented model gives support for both.
An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. Each object will exist in one object space. Each integrated system will typically have its own object space, although it is possible for multiple systems to share an object space. An object is characterized by a globally unique identifier (OID) independent of possible attribute values. However, different implementations of an interface might use different internal identifier-schemes, dependent on mechanisms in underlying systems. Object identifiers (OIDs) in COOM are heterogeneous, but might uniformly be converted to and from a string-representation.

9.4.4 Separation between Interface, Implementation, Factory and Extent

Figure 9.6 shows a hierarchy of interfaces, where interface B conforms to interface A. The extent of A is the extent of its two different implementations, A-1 and A-2, and the extent of the interfaces which are conform to these, are here the extent of interface B.

An interface is a description of a set of operations, while an implementation is the actual operation-bodies and local state and operations.

The interface extent is the set of objects having a certain interface, this is managed by an object interface manager. The implementation extent is the set of objects having a certain implementation, this is managed by an object implementation manager. Creation of new objects is done through the object interface manager. COOM supports these separations, and uses an interface-manager object as an object-factory. These separations are an important issue in current research in object-oriented technology and have been described in more detail in chapter 4.

COOM supports relationships between objects through relations. Relationships are managed by relation manager objects.

Figure 9.7 shows the object interface managers, metaA and metaB, with their respective
object implementation managers metaA-1 and metaB-1 in objectspace 1 and metaA-2 and metaB-2 in objectspace 2. Similarly the relation interface manager RelAB has a relation implementation manager RelAB-1 in objectspace 1 and a relation implementation manager RelAB-2 in objectspace 2.

9.4.5 The use of Three Languages: ODL - OML - OQL

COOM uses three different sub-languages, ODL (Object Definition Language), OML (Object Manipulation Language) and OQL (Object Query Language), which are briefly described in the following.

9.4.6 ODL - Object Definition Language

The basic principle behind COOM is to facilitate systems integration through encapsulated objects, using ODL for defining the interface of objects. The extent of an interface is all objects that support that interface. This includes the extent of all different direct implementations and the extent of all specialized interfaces, for this interface.

ODL is aimed at describing the interface of objects, as semantically complete as possible, and without any reference to implementations. ODL merges concepts from the distributed system world, where interfaces typically are described as a set of operations through an Interface Definition Language, and concepts from the ER-oriented database-world where entities, relations and attributes are described through a Data Definition Language.

ODL will lead to OML-based class-definitions where abstract attributes are mapped to put- and get-operations, and relations are mapped to relation-management objects with a corresponding set of operations. The following shows some of the ODL-syntax:

```plaintext
SCHEMA schemax {
IMPORT schemay;
/* Constants */
/* Datatypes */
/* OBJINTERFACE InterfaceZ { ... }; */
/* OBJINTERFACE InterfaceY { ... }; */
```
NOTIFICATION Not1: BasicNotification
{int n1; };

EXCEPTION Exc1: BasicException
{int e1; };

OBJINTERFACE InterfaceX: Super1, Super2
{ /* Exception-events: */
  /* Notification-events: */
  /* Operations: */
    InterfaceZ op1(IN int x1, IN String s1)
    RAISES(Exc1) NOTIFIES (Not1);
  /* Abstract Attributes: */
    String att1 NOTIFIES(Not2) IN(set);
    READONLY int att2;
  /* Unique attribute-combination: */
    UNIQUE(att2);
  /* Containment-relations: */
    CONTRELINTERFACE contx
    { #1; InterfaceY #N; };
}; /* End InterfaceX */

RELINTERFACE relationxy
{InterfaceX #N role1;
 InterfaceY #1 role2;
 Date relatt; };

}; /* EndSchema */

ODL will be described in more detail in chapter 10, 11 and 12.

9.4.7 OML - Object Manipulation Language

The Object Manipulation Language, OML, is not realized as a new language, but through mappings to existing object-oriented programming languages, in particular C++ and Smalltalk. This gives certain limitations with respect to the "ideal" OML, which should support a strong separation between interface and implementation and typing based on interface-conformance together with extent-management. The functionality of OML is defined in a set of classes that the specific object and relation-interfaces inherit from, with operations that are possibly reimplemented by the actual implementation-classes derived from the interface-classes.

The class-implementations will typically consist of mapping-code to the interfaces of underlying systems, such as database-systems or systems accessible through a remote-procedure-call mechanism. Efficient and flexible cache-handling for local proxy-objects is important. In order to efficiently support fine-granulated objects, these objects are maintained through an object-oriented database-system.

Example 9.1 C++ OML: printout att1 for the employees in department d1

    InterfaceY* y1;
    ...... /* Given a value for y1 */
try {
    /* Get objects of InterfaceX related to y1 through RelationXY */
    Set<InterfaceX*> sx1 = RelationXY->getToObj(y1);
    /* or directly through role-name */
    Set<InterfaceX*> sx2 = y1->role2();

    DO (sx1, InterfaceX*, x) /* Iteration-macro */
    
cout << x->att1();

ENDDD
}
catch(NotExist ne) {...};

ODL can be used both for generating new implementations, and for describing interfaces to existing systems. Different derivation-tools need to be made for different classes of systems, and might also include facilities for schema enrichment. There have been experiments with mappings for relational and object-oriented database-systems like Ingres and ObjectStore and for remote-procedure-call-based systems, like OSF/DCE and OMG Object Request Broker. COOM is a self-describing model with an Object Dictionary that provides information about interfaces, implementations and extents at run-time.

OML will be described in more detail in connection with relevant constructs in chapter 10 and 11.

9.4.8 OQL - Object Query Language

The Object Query Language (OQL) gives possibilities for selecting subsets of the existing collections of objects, based on predicates on their relationships, attribute-values and operation-results (for operations without side-effects). The queries are issued as query-strings to select-operations supported by the interface-objects.

Example 9.2 Find all objects with InterfaceX that have att2 between 20 and 30, and are related to object y1 through RelationXY:

Set<InterfaceX*> s = metaInterfaceX->find("att2 >= 20 && att2 <= 30 && RelationXY == <%", y1);

A query find is issued on the extent of an interface, which will delegate sub-queries to its different underlying implementations, after initial parsing and optimization. Each implementation will transform its sub-query to a corresponding query on the underlying system. OQL is logically a part of the OML, but is specified as a separate language which can be used from the OML-mapped languages C++ and Smalltalk.

9.5 COORASS Methodology

The COORASS methodology for the development of COOM-based systems is a structural extension to the behavioural-oriented OORASS methodology [RB+92]. This is similar to how COOMS provides a structural extension to COOMB.

The COOP framework and COOM supports a new way of system building, by greater emphasis on reusing existing system-components. System building is essentially the process
of gluing together pre-existing parts to fit new requirements. This system-building process involves a number of different user-roles, with a special emphasis on the user-roles of Tool-builders and Component-integrators.

9.6 Control and Data Integration for the Component Integrator

Component integrators are those that adapt information and services from underlying components to the COOM object-model and introduce these into the integration framework environment.

In this process the component integrator will provide implementations for the mappings between underlying systems and the COOM-schema with objects and relations. The COOM-support for control- and data-integration will be utilized for this.

9.7 Presentation Integration for the Tool Builder

Tool builders, or application-builders, are the providers of tools for the end-user. The tools are created by composing together different services needed by the user. The services are available through the COOM object model, as provided by the component integrators.

The functionality of the services is offered through customized user interfaces, by use of the support for presentation integration. Display-oriented presentation integration is supported by a possibility to use an object-oriented user interface framework to develop portable user interface components. Model-oriented presentation Integration is supported by a semantical split between user interface objects and functional objects, and the possibility for a user interface object to interact simultaneously with multiple functional objects.

9.8 A Scenario for the use of COOP and COOM

The goal of the COOP Integration Framework is to provide a basis for systems integration, as described through the requirement areas for control integration in section 3.3, data integration in section 3.4 and presentation integration in section 3.5.

To illustrate the principles throughout the text, a problem scenario is used. The scenario is two different organizations which have merged. Each organization has an employee management system. One is based on an Ingres RDBMS, while the other is based on an ObjectStore ODBMS. Both systems stores information about employees and which departments they are working in. In addition there is a project management system managing information about projects, activities and resources. There is a need to make a unified information system which provides information about which employees work in which departments, and how they are allocated to project-activities.

This is facilitated by the help of component integrators, who integrates components into the COOP Integration Framework. The schemas and interfaces of the components are used as a basis for manual or automatic generation of COOM ODL schemas, as described in chapter 15. These are again used as a basis for generation of classes in C++ or Smalltalk through the COOM compiler. These classes reflects available functionality and data in the underlying systems, and are in addition provided with functionality from base-classes in the COOP integration framework. The generated classes are then used by the tool
9.9 Structure of Part III

Chapter 9 has showed how the requirement-areas from Part I and the object-oriented integration approaches from Part II are used as a basis for the COOP Integration Framework. It also presented the vision and high-level architecture for COOP, and described the basic principles and concepts for COOM.

Different aspects of COOM and COOP will be treated in more detail in the following chapters:

Chapter 10 presents the COOM foundation model, \( COOM_F \), which is used as a basis for the COOM behavioural model, \( COOM_B \), and the COOM structural model, \( COOM_S \).

Chapter 11 presents \( COOM_B \) and its relation to control integration. Each of the basic constructs are defined, with its corresponding ODL (Object Definition Language) syntax and semantics and mapping to OML (Object Manipulation Language).

Chapter 12 presents \( COOM_S \) and its relation to data integration. Abstract attributes and relations are defined as a layer on top of the foundation model and the behavioural model. It is shown how COOM provides a foundation for further work on multi-model data integration.

Chapter 13 describes query processing through OQL - the Object Query Language.

Chapter 14 presents the COOP Development methodology, COORASS, and gives a description of a possible COOP-compliant development environment.

Chapter 15 illustrates the possible use of COOP and COOM in component integration and in tool building. Tool building also illustrates the support for presentation integration.

Chapter 16 evaluates COOP with respect to the requirements from Part I, and compares it to related work. Finally areas for future work are identified.

Appendix A presents grammar and definitions for COOM ODL. Appendix B presents the COOM OML interface-definitions. Appendix C presents the COOM OQL grammar.
Chapter 10

COOM Foundation Model

This chapter defines the COOM foundation model, $COOM_F$, as a basis for the later presentations of the COOM behavioural model, $COOM_B$, and the COOM structural model, $COOM_S$.

The concepts in $COOM_F$ are adopted from the interface definition languages NCS NIDL [ncs89] used in OSF/DCE [osf90] with additional constructs from the OMG CORBA interface definition language [Obj92a], and current research in object-oriented technology as discussed in chapter 5. A novel construct introduced by $COOM_F$ is the use of object-spaces for the management of the mapping from interfaces to implementations.

10.1 The COOM-model

![Figure 10.1: COOM Layers](image)

The COOM model$^1$ is defined as two layers above a base system-layer as shown in figure 10.1. The basis system-layer $COOM_F$ provides a foundation for objects with interfaces. The object-interface layer $COOM_B$ enhances $COOM_F$ with the notion of object-interface, which is an interface with predefined functionality and an interface manager. $COOM_B$ and $COOM_F$ together form the behavioural part of COOM which supports control integration. The attribute- and relation-interface layer $COOM_S$ enhances $COOM_B$ with the notions of attributes and relations. $COOM_S$ is the structural part of COOM that supports data

$^1$The word model is used as in the data modeling tradition, both for the modeling formalism and for a schema described in the modeling formalism. In presentation integration, model is used to describe a functional object, instantiated from a schema, as in the MVC paradigm.

157
integration. This chapter defines \( COOM_F \). Chapter 10 defines \( COOM_B \) and the support for control integration. Chapter 11 defines \( COOM_S \) and the support for data integration.

The basic foundation for the COOM-model, \( COOM_F \), is defined as follows:

**Definition 10.1** \( COOM_F \):

\( COOM_F \) is an n-tuple, \( COOM_F = \langle \text{IN}, \text{IM}, \text{OID}, \text{OP}, \text{NN}, \text{EN}, \text{CN}, \text{OSN}, \text{B}, \text{AT} \rangle \) where:

- \( \text{IN} \) is a finite set of Interface Names
- \( \text{IM} \) is a finite set of Implementation Names
- \( \text{OID} \) is a countable possibly infinite set of object identifiers
- \( \text{OP} \) is a finite set of Operation Signatures
- \( \text{NN} \) is a finite set of Notification Names
- \( \text{EN} \) is a finite set of Exception Names
- \( \text{CN} \) is a finite set of Context Names
- \( \text{OSN} \) is a finite set of Object Space Names
- \( \text{B} \) is a finite set of Basic Types \( B_1, \ldots, B_n \) such that \( B = \bigcup_{i=1}^{n} B_i \)
- \( \text{AT} \) is the set of all types such that \( \text{AT} = \text{IN} \cup \text{B} \)

Figure 10.2 shows the relationships between the different parts of the base model. An Interface has a number of operation signatures, which again might be associated with a
10.1.1 Values

set of input/output parameters, notifications, exceptions and a context. A parameter is
a type that might be either an Interface or a Basic type. For each interface there might
exist a number of implementations. Each implementation has a set of operation bodies,
and will be associated with one object-space. An object-space is a subset of the total
set of objects, where the objects are associated with a specific underlying implementation
platform. Definitions for each of these concepts will be given in the next sections.

In current research in object-oriented technology the term type has often been used for
interface and class has been used for implementation [AvdL90]. This special use of these
terms has caused confusion, because it is different from their historical use. Here it has
been decided to use the terms interface and implementation directly. The term type is here
used for both Interface types and Basic types.

10.1.1 Values

Definition 10.2 Values and Types:
There are 2 types of values, v

1. Every basic type value \( v \in B \) is a literal value for which there exists a textual
   representation. The type of a literal value (basic type value) is a Basic Type.

2. Every \( v \) with id \( \in \) OID is an object-reference. The type of an object is an Interface.

\( \Box \)

The symbol \( V \) denotes the set of all values.

A Basic Type value can be directly represented in a textual form, and does not have an
identifier of its own. It has been chosen to separate basic types and objects, even if it is
possible to also represent basic type values as objects, e.g. such as in Smalltalk[GR83].
The reason for this separation is that there is a semantic distinction between literals and
objects, and that it is possible to define a library-layer of types realized through objects if
that is needed.
Figure 10.3 shows the basic types in COOM. The different types are described in more detail in section A.3 in Appendix A.

**Definition 10.3 Basic Types:**

1. $S$ is a finite set of Basic Types called Simple Types $S_1, ..., S_n$ such that $S = \bigcup_{i=1}^{n} S_i$ and $S \subseteq B$

2. Every $v \in S$ is a literal value of a *simple type* for which there exists a textual representation.

3. $C$ is a finite set of Basic Types called Constructed Types $C_1, ..., C_n$ such that $C = \bigcup_{i=1}^{n} C_i$ and $C \subseteq B$

4. Every $v \in \{\text{Struct, Union, SetOf, ListOf, Array}\}$ is a value of a *constructed type* for which there exists a textual representation if $T \in B$.

**Definition 10.4 Object:**

An object is a triple $\sigma = (id, inn, imn)$ where $id \in \text{OID}$, $inn \in \text{IN}$, and $imn \in \text{IM}$, and $imn$ corresponds to $inn$. □

An object is an entity which provides the set of operations described by its interface, realized through its implementation. An object is an instance of its interface and also an instance of its implementation. The type of an object is given by its interface.

Based on an object $o$, the function $\text{res}(o)$ associates the set of all identifiers returned as results from operations on this object. The notation $o.id$, $o.in$ and $o.im$, is used, respectively to denote the identifier, the interface and the implementation for object $o$.

### 10.2 ODL - Schema Definitions

ODL, Object Definition Language, is the language used as a textual representation of definitions in the COOM model. Each concept in COOM will have a corresponding syntactical form in ODL. The following sections will present the syntactical form together with the definitions for each concept. The full syntax of ODL is given in Appendix A.

**Definition 10.5 ODL Schema:**

A **COOM ODL Schema** is defined as the tuple $s = (n, I, E)$ where

- $n$ is a schema name
- $I$ is a set of imported schema-names
- $E$ is a set of exported definitions

\(^2\text{Relations, relationships and attributes are defined in chapter 11}\)
A schema definition is a module for a related set of definitions, similar to a database schema. The syntax for a schema definition is:

```
SCHEMA <schemaName>
    
    [{
        [import "";"" ]*
        [export "";"" ]*
    }
```

A schema does not need to have any Import specifications. A schema must always have one or more Export declarations. An <export> specification may be one of the following:

```
<export> ::= <interface_declaration>
        | <object_interface_declaration>
        | <relation_interface_declaration>
        | <exception_declaration>
        | <notification_declaration>
        | <type_declaration>
        | <constant_declaration>
```

The order of Export declarations is arbitrary.

**Example 10.1 Example of ODL Schema-structure**

```
SCHEMA PersonSchema

{ import Standards;
  import Basics;

  INTERFACE Person
  { String name();
    void name(String [in] newName);
    int age();
    void age(int [in] newAge)
      RAISES(IllegalAge) NOTIFIES(ChangedAge);
    Person marriedTo();
    int personNumber();
    int personNumber(int [in] newPersonNumber);
  }; /* Interface Person */

  INTERFACE Employee: Person
  { int salary();
    void salary(int [in] newSalary);
  }; /* Interface Employee */

} /* PersonSchema */
```
10.3 Interface Definition and Syntax

Figure 10.4: Person and Employee interfaces

Figure 10.4 and the corresponding ODL-schema in example 10.1 shows the basis for the Person - Employee example that will be used to explain the different concepts throughout the text. This first example shows the specification of a Person and an Employee interface, where the Employee interface is a specialized interface from Person. The specialized interface Employee is conform to Person, meaning that all objects which are instances of Employee support the same operations as instances of Person.

In the following sections each of the schema-concepts will be described in turn, starting with interfaces and operations in section 10.3 and corresponding implementations in section 10.4. Interfaces and implementations are extended with manager objects and predefined operations in COOM$_B$ in chapter 10.

Notifications are defined in section 11.5, exceptions are defined in section 11.6 and contexts are defined in section 11.7. Basic types are described in Appendix A.3. Attributes and Relation-interfaces are described in chapter 11, as part of COOM$_S$.

### Definition 10.6 Interface:

An Interface$^3$ in is defined as the tuple in = $(inn, P, E, N, O, AO)$ where

- $inn \in \text{IN}$ is the Interface name
- $P$ is a sequence of parent interfaces (supertypes) of the form $in_1, ..., in_k$ where $in_i \in \text{IN}$
- $E \subseteq \text{EN}$ is a set of Exception-definitions
- $N \subseteq \text{NN}$ is a set of Notification-definitions
- $O \subseteq \text{OP}$ is a set of operation-signatures, defined directly for this interface.
- An $op \in O$ might redefine an operation defined in a parent interface, following the rules of substitutability for interfaces, see section 5.8.2.
- $AO \subseteq \text{OP}$ is the total set of applicable operation-signatures defined for this interface. How this is defined is described in section 10.3.2 on specialized interfaces and interface-conformity.

$^3$Object interface and relation interface are defined as special interfaces in section 11.1 and section 12.3 respectively.
The syntax for an `<interface_declaration>` is⁴.

```
<interface_declaration> ::= INTERFACE <interfaceID> [ [i.param_list] ] [ [ supertypes ] ] '{' '{' { interface_elem '{', '{' } * '{' } ]
| INTERFACE <interfaceID> /* forward */
```

`<ID>` is the name of the interface. An interface can inherit interfaces from parent interfaces that are imported, declared earlier or are declared forward. Those interfaces are supertypes to this interface. It is optional to declare supertypes. Parameterized interfaces with [i.param_list] is discussed in section 10.10. The syntax for `supertypes` is:

```
<supertypes> ::= '{', '{' <interfaceID> '{', ',' <interfaceID> '}' *
```

The `<interfaceID>` must be the name of an interface. The syntax for `<interface_elem>` is:

```
<interface_elem> ::= <operation_declaration>
| <notification_declaration>
| <exception_declaration>
```

The essential part of an interface description are the operation declarations. An operation might result in the posting of a notification, or in an exception. There is a possibility to define notifications and exceptions in an interface, or outside an interface. Notifications are described in section 11.5 and exceptions in section 11.6.

Example 10.1 showed an example on the use of this syntax for the definition of Person and Employee interfaces.

Operations are defined and described next. This is used as a basis for explaining conformity of interfaces, resulting from the use of parent-interfaces in an interface description.

### 10.3.1 Operation Definition - Syntax

**Definition 10.7 Operation-signature:**

An operation-signature is defined as a n-tuple \(op = (on,(m : S \times I \rightarrow R \times O),b,N,E,C)\) where:

- `on \in ON` is the name of the operation
- `m : S \times I \rightarrow R \times O` is a mapping from a product of a source type and input-types to a result-type and a product of output types, of the form `m : S \times I_1 \times I_2 \times \ldots \times I_n \rightarrow R \times O_1 \times \ldots \times O_m` where `S \in IN` and `I_i`, `R` and `O_j \in AT`.
- `b` denotes the semantics (behaviour) of mapping `m`.
- `N` is a sequence of the form \((n_1,\ldots,n_a)\) where `n_a \in NX`, being the set of notifications which might be posted from this operation ⁵.

---

⁴Parameterized interface, i.param_list, is described in section 10.10.

⁵See description in section 11.5.
10.3 Interface Definition and Syntax

- E is a sequence of the form \((e_1, ..., e_k)\) where \(e_k \in EN\), being the set of exceptions which might be raised from this operation \(^6\).

- C is a sequence of the form \((c_1, ..., c_k)\) where \(c_k \in CN\), being the set of context-values which might be added as parameters to this operation. \(^7\).

\[ \square \]

The operation \textbf{void age()} from interface \textbf{Person} from example 10.1 can be used to explain this:

\[
\text{void age(int [in] newAge) Raises(IllegalAge) Notifies(ChangedAge);}
\]

This is transformed to the following operation-signature: \(op = (age, (m: Person \times int \rightarrow void) \times \emptyset, b, \text{ChangedAge}, \text{IllegalAge}, \emptyset)\) by:

\begin{itemize}
  \item \textbf{Operation name, on} = age
  \item \textbf{Source type, S} = Person
  \item \textbf{Input types, I} = int
  \item \textbf{Result type, R} = void
  \item \textbf{Output types, O} = \emptyset
  \item \textbf{Semantics, b} = \emptyset (\textbf{See description in next section})
  \item \textbf{Notifications, N} = \text{ChangedAge}
  \item \textbf{Exceptions, E} = \text{IllegalAge}
  \item \textbf{Context, C} = \emptyset
\end{itemize}

The source-type, S, is the name of an interface in which this operation is defined, meaning that this operation will be offered by all objects having this interface. This is implicitly defined when an operation is defined as part of an interface. \(\emptyset\) is an empty set.

Figure 10.2 showed how an interface is associated with a set of operation-signatures. Each operation-signature is associated with a set of parameters, either of type \text{Interface} or of type \text{Basic value}, and potentially a set of notifications, exceptions and a context. For each operation in an interface, the corresponding implementation will provide an operation-body.

The ODL-syntax for an operation-declaration\(^8\) is:

\[
\begin{align*}
\text{<operation\_declaration>} & \quad ::= \quad \text{<type\_spec>} \quad \text{<operationID>} \\
\quad & \quad \quad \text{<parameter\_decls>} \\
\quad & \quad \quad \quad \quad \text{<[raises\_expr]>} \\
\quad & \quad \quad \quad \quad \text{<[notifies\_expr]>} \\
\quad & \quad \quad \quad \quad \text{<[context\_expr]>} \\
\text{<parameter\_decls>} & \quad ::= \quad \text{""} ('\) \quad \text{<[parameter\_list]>} \quad \text{""}\) \\
\text{<parameter\_list>} & \quad ::= \quad \text{""}, \quad \text{<parameter\_spec>} \\
\text{<parameter\_spec>} & \quad ::= \quad \text{""} \\
\text{<direction>} & \quad ::= \quad \text{""} ['\) \quad \text{<direction\_list>} \\
\quad & \quad \quad \quad \quad \text{[OUT \text{""}]""} \\
\quad & \quad \quad \quad \quad \text{[IN \text{""}]""} \\
\quad & \quad \quad \quad \quad \text{[\text{""}]""}
\end{align*}
\]

\(^6\) See description in section 11.6.

\(^7\) See description in section 11.7.

\(^8\) The syntax for operations is adapted from the operation-declaration in OSF/DCE. For completeness, the syntax shown also contains constructs that is introduced in the next chapter.
10.3.2 Conformance and Specialization of Interfaces

\[ \langle \text{ptypeID} \rangle ::= \langle \text{simple_typeID} \rangle \mid \langle \text{interfaceID} \rangle \mid \langle \text{objInterfaceID} \rangle \]

\[ \langle \text{declarator} \rangle ::= \langle \text{attributeID} \rangle \mid \langle \text{declarator} \rangle \langle \text{array_bounds} \rangle \]

\[ \langle \text{raises_expr} \rangle ::= \text{RAISES} \langle \text{exceptionID} \rangle \{ \langle \text{exception_id} \rangle \}^* \]

\[ \langle \text{notifies_expr} \rangle ::= \text{NOTIFIES} \langle \text{notificationID} \rangle \{ \langle \text{notification_id} \rangle \}^* \]

\[ \langle \text{context_expr} \rangle ::= \text{CONTEXT} \langle \text{string_literal} \rangle \{ \langle \text{string_literal} \rangle \}^* \]

Operations form the basis of COOM. Operation execution is done through the sending of a message to an object, which is a request for the invocation of an operation. COOM is a messaging object model where there is a recipient for the request, called the receiver.

The source-type for an operation will not be the declared reference type of the receiver object, as seen from the sender, rather the actual type of the receiver of the message. All operations will return a result-value, where the type might be either an interface or a basic type. Input- and Output- parameters are distinguished by the direction IN, OUT or INOUT between the parameter-type and the parameter-name. A parameter might be specified as an array through \langle \text{array_bounds} \rangle. The use of arrays is described in appendix A.3.2. Notifications and Exceptions are described in more detail in section 11.5 and section 11.6.

10.3.2 Conformance and Specialization of Interfaces

In COOM an object-type is called an Interface and a subtype is called a specialized Interface. A particular Interface subT is a specialized interface of T if subT provides the same behaviour as T. That is written as subT <= T. An object with interface subT can thus be used as if it is of type T because it is guaranteed to provide at least the operations of T. subT is then said to be substitutable for T. The principle of substitutability is an important issue in current research in object-oriented technology, and was described in more detail in section 5.8.2.

A specialized interface is in COOM defined as an interface that is specified with one or more parent-interfaces. New operations might be added, and redefined operations need to be conform with previously defined operations. The applicable operations, AO, for a particular interface is defined as all operations that objects with this interface can support.

The schema-example in section 10.2 showed examples of two interfaces, Person and Employee. In this example AO(Employee) will include the operation-signatures from Person transformed to use Employee as the source-type.

In addition to the inherited operation-signatures, it is possible to add new ones and to refine the existing one. Refined operation-signatures need to conform with the operation-signatures that are refined. Because of the addition of notifications, exceptions and context, the definition for conform operations is as follows:

Definition 10.8 Conform operation-signatures:
An operation-signature op is conform\textsuperscript{9} with another operation-signature op' iff:

- the name and number of arguments is the same
- there are no stricter restrictions on input-arguments
- the result and output-arguments are not more general in any signature
- there is no redefinition for "inout" arguments
- the behaviour is the same
- there are no new exceptions, but exceptions might be specialized
- there are no new context-arguments
- there might be other notifications generated in addition to those already specified

\[\Box\]

**Multiple inheritance and Conformance**

An interface might be defined with multiple parent interfaces. Operations with the same name and arguments will be merged into one operation. Conflicts might arise if the arguments are different. If the input-arguments are stricter, they are considered as different operations.

Figure 10.5 shows how conflicts with output are resolved. A problem occurs if Y and Z, as two different sub-interfaces of X, is refined differently when specializing the output-argument(s) (result) of the same operations, op1 and op2. A new interface, W, inheriting specification to be conform with both Y and Z, needs to redefine such arguments to be of a common sub-interface of the specialized arguments. In the example the result from op1 is specialized to W, which is specialized from Y and Z, and the result from op2 is specialized to subABT which is specialized from subAT and subBT.

\textsuperscript{9}An example of conformity rules is given in section 5.8.2.
Conflicts with choice of implementation is not a problem here, since the ODL specifications are concerned only with interfaces.

The rule for conform operation-signatures and multiple inheritance resolution is difficult to fully support, when mapping is done to C++. A derived class in C++ is not allowed to do any changes to only the result-type of an operation [Str91]. An example for this is a copy operation, `T copy()`. It is useful to have a generic copy-operation that returns a copy of an object with the same type as the object to be copied. This is discussed further in section 11.1.1. For this reason the C++ mapping transforms changed result-types and parameters to their base definition. In Smalltalk this is not a problem, since result and parameters are not typed at all. But then the advantage of interface-based typing is not provided either. The goal of supporting existing languages like C++ and Smalltalk, therefore limits the usefulness of a conformance-based model.

**Semantic Specifications of Operations**

In the definition of an operation, the term `b` was used to denote the semantics (behaviour) of an operation.

The signature of an operation does not give any information about its actual behaviour. Work on "algebraic specifications" [EM85, Gut81] has shown how it is possible to use axioms to specify the results of operations.

The semantics, `b`, can thus be described by a set of axioms that shows the result-relationships between the different operations. Axioms are not currently a part of the ODL-syntax, but this is seen as something which easily can be added in the future. It is a problem to verify that implementations actually conform to the axioms, but, nevertheless, axioms are a useful facility for describing the intended semantics of operations.

A typical example of such axioms for the Person interface could be to describe that the two operations for retrieving and setting of `personNumber` act together with the semantics of an abstract attribute.

```plaintext
personNumber(personNumber(i)) = i;
personNumber(i) -> personNumber() == i
```

A value `i` which has been set by the `personNumber(i)` operation will be returned by subsequent invocations of the `personNumber()` operation.

### 10.3.3 ODL Interface Preprocessing Facilities

ODL contains some constructs which make the definitions of Interfaces easier.

- The use of the type-specifier `Self`
- Parameterized Interfaces

**Definition 10.9 Self:**

Self is defined to be a template for the actual type of an Interface. When operations containing `Self` as the type of a result or output-parameter is inherited from a parent-interface, `Self` is substituted with the type of the actual child-interface. \[^{10}\]

\[^{10}\]Note that this is different from the use of `Self` in Smalltalk. In COOM `Self` is used on the interface specification level to refer to the current interface type, while in Smalltalk `Self` is used on the implementation level to refer to the current object.
Self is similar to the construct *like Current* in the Eiffel programming language [Mey88, Mey86]. Self can be used in the definition of operation-signatures in an Interface, where Self can be used as the type of the result of an operation, and as the type of output-arguments. It might not be used for input-arguments, because that will violate the principle of substitutability for operations.

```c
INTERFACE X {
  Self opx( int [in] i, Self [out] p);
};

INTERFACE Y: X {
  /* The following operation-signature will be defined: */
  Y opx(int [in] i, Y [out] p);
};
```

**Definition 10.10 Parameterized Interface:**
A parameterized interface is a template for an actual interface with arguments following the name of the interface in a parameter-list \( t = (\text{TYPE} \ t_1, ..., \text{TYPE} \ t_n) \). The arguments might be types or values to be used in the operation-definitions. A Parameterized interface is instantiated by giving the name of the interface followed by the actual types \( T_1, ..., T_n \) for the type-arguments as follows: Interfacename \( <T_1,...,T_n> \). A type-argument \( T \) might either be an object interface or a basic type. The declarations TYPE, INTERFACE, OBJINTERFACE or BASE can be used to restrict the arguments to all types, interface-types only, objinterface-types only or base-types only. In addition to type-parameters it is possible to have values of simple types, such as character strings, as parameters. In particular, integers, can be a useful parameter. □

The ODL-syntax for a parameterized interface is:

```c
<i_param_list> ::= "'('" <i_param> {"','" <i_param>}"'')'"

<i_param> ::= <i_param_type> <paramID>

<i_param_type> ::= TYPE | INTERFACE | OBJINTERFACE
               | BASE | <simple_typeID>

<paramID> ::= ID
```

This is illustrated in the following example:

```c
INTERFACE Stack(TYPE T) {
  void push( T [in] element );
  T top( );
  T pop( );
};
```

An interface can be specified as a parameterized interface as follows:

```
<interfaceID> ::= ID | ID "'('</' <inst_plist> "'/>')"

<inst_plist> ::= <inst_param> {"','" <inst_plist>}*
<inst_param> ::= <interfaceID> | <simple_typeID> | <const_exp>
```
Parameterized interfaces might be used in specialized interfaces as follows:

```java
INTERFACE FStack(TYPE T): Stack<T>
( Boolean full();
);
```

As a result-type from an operation, it might be used as follows

```java
Stack<Person> stackP();
```

Also values of simple types can be passed as arguments if the parameterized interface is extented for that.

```java
Stack(TYPE T, int size) ......
Stack<Person, 5> sp;
Stack<int, 10> si;
```

### 10.4 Implementation and Object Space

The separation between interface and implementation is an important part of the COOM-support for systems integration and mapping to heterogeneous systems. This facilitates for the use of interfaces with multiple implementations. For each interface there might exist more than one implementation.

The systems integration goal of this work is to be able to uniformly handle objects that actually reside in different underlying systems, in a way which hides mapping-details from the application programmer. The application programmer would like to be aware of which underlying system he is interacting with, but would not like to write the application program differently, depending on which system that will be used. The object-space\(^{11}\) is introduced to facilitate this independence, by instead using a higher level knowledge about which subsystem to use. An application-program should be written independently of which underlying implementation is chosen. Two different instantiations of the same application program should be able to use different implementations.

Figure 10.6 shows how a number of implementations might be associated with each interface. The diagram shows two implementations of each interface, one for Ingres and one for ObjectStore. The default name for implementations is the name of the corresponding interface, with the name of the underlying system that this is an implementation for as a suffix.

**Definition 10.11 Implementation:**
An Implementation \(\text{im}\) is defined as the tuple \(\text{im} = (\text{imn}, \text{inn}, \text{osn}, \text{OB}, f, \Omega)\) where:

- \(\text{imn} \in \text{IM}\) is the implementation-name.
- \(\text{inn} \in \text{IN}\) is the interface-name.
- \(\text{osn} \subseteq \text{OSN}\) is the Object-space names (see definition 10.13) for this implementation.
- \(\text{OB}\) is a set of operation-bodies for the operation-signatures of \(\text{inn}\).

---

\(^{11}\)see definition 10.13.
• $f$ is a correspondence-mapping between operation signatures and operation bodies.

• $\text{inn}$ corresponds to $\text{inn}$ when $f$ exists between the operation signatures in $\text{inn}$ and the operation bodies in $\text{inn}$.

• $\Omega$ is a set of objects called the extent of the implementation, that is the objects created to have this implementation, also denoted by $\text{extent(inn)}$.

Definition 10.12 Extent and Extent inclusion:
The set of instances of an Interface $i$, is called the extent of this interface and denoted $\text{extent}(i)$. This includes also the instances of interfaces that have defined $i$ as a parent, and therefore also have defined the operations that are defined for $i$. The direct extent, not including specialized interfaces, is denoted $\text{direct.extent}(i)$.

An implementation is required to provide all the operations defined by its interface. COOM does not put any restriction on how an implementation is realized. The implementation might reuse code from other implementations through use of delegation or some kind of implementation-inheritance for existing operation-bodies, or it might provide all the necessary operation-bodies itself. From the COOM point of view, an implementation will always contain all necessary operation-bodies for the operations defined in the object-interface of an object-implementation. A number of implementations might exist in parallel for one particular interface.

Each underlying system is associated with an Object-space, and one implementation for each type of interface that is supported by that system. Two different Ingres databases will be different object spaces, but might reuse the same implementation. Thus it is possible for one implementation to cover more than one object space, provided that the object spaces represents multiple identical subsystems.
Definition 10.13 Object Space:
An Object-Space is defined as the tuple \( os = (osn, IMP, \Theta) \)

- \( osn \in OSN \) is the object-space name.
- \( IMP \subseteq IM \) is the set of implementation names for the existing implementations in this object-space.
- \( \Theta \) is the set of objects formed by \( \Theta = \bigcup_{i=1}^{n} extent(im_i) \), where \( im_i \in IMP \), which is all objects being instances of the implementations in this object-space.

Figure 10.7 shows how the total object space is partitioned into different object spaces. The Object Dictionary is an object space with management-objects, while the object spaces Ingres2 and ObjStore1 contains objects for the information in underlying component systems.

10.5 Contribution and Conclusion

This chapter described \( COOM_F \), which is used as the basis for \( COOM_B \), to be described in the next chapter and \( COOM_S \), to be described in chapter 11. The main characteristics of \( COOM_F \) is the separation between interface and implementation, the use of conformance-based typing, and the use of object-spaces for object management.

The interface-specification is adopted from the interface definition language of NCS NIDL [ncs89] also used in OSF/DCE [osf90], enhanced with some constructs from the interface definition language of OMG CORBA [Obj92a]. The use of supertypes is adopted from
OMG CORBA, while the use of conformance-based typing is taken from the Emerald language [HM85b]. The use of parameterized interfaces is inspired by the template-construct of C++ [Str91]. The use of a struct-type with inheritance\footnote{The Struct-type is described more in Appendix A.} is trivially new in this context. The use of object-spaces for the management of the mapping from interfaces to implementations is novel in COOM. This is described in more detail in connection with manager objects in the next chapter.

The next chapter presents $COOM_B$, which extends $COOM_F$ with explicit support for object management and features for the support of control integration.
Chapter 11

COOM Behavioural Model and Control Integration

This chapter defines the behavioural part of the COOP Object Model, COOM_B, and its support for control integration through the COOM Object Definition Language (ODL) and corresponding Object Manipulation Language concepts (OML).

The COOM_B model is founded on the principles described in chapter 9 and 10. It is based on the experiences gained through object-oriented control integration and the use of Interface Definition Languages used in the software bus architecture of SITE and CODEDISC described in chapter 6. The motivation for COOM_B is to meet the requirements for request-oriented and notification-oriented control integration, as presented in section 3.3. A particular goal is to provide suitable support for these requirements through one mechanism. A basis for request-oriented control integration is adopted from the interface-definition-languages of OSF/DCE [osf90], ESF Software Bus [Eur91a] and OMG CORBA [Obj91a]. A basis for notification-oriented control integration is adopted from the HP Softbench/Field [HP 89, Rei90a] enhanced with a notation for the explicit specification of notifications in a schema. These mechanisms are integrated into a new object-management framework that separates between object interface managers and object implementation managers. The predefined functionality of objects and object managers is enhanced from functionality provided in the REBOOT datamodel [OH91].

<table>
<thead>
<tr>
<th>COOM</th>
<th>S</th>
<th>Abstract Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relation-Interface</td>
<td>Relation management</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COOM</th>
<th>B</th>
<th>Object-Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Object management</td>
<td>Notification, Etc. etc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COOM</th>
<th>F</th>
<th>Interface, Basic Types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implementation</td>
<td>Operations</td>
</tr>
</tbody>
</table>

Structural model for Data Integration

Behavioural model for Control Integration

Foundation model

Figure 11.1: COOM Layers
11.1 Object Interfaces and Manager Objects

An interface does not say anything about how an object is created, or how to find all instances of an interface, nor are there any predefined operations for instances of an interface. The only thing an interface provides is a facility for defining the operations that instances of this interface will provide.

The ODL syntactical construct, OBJINTERFACE, is introduced in order to provide explicit management of objects, through Meta-objects, and a predefined set of operations that are useful for all objects. An OBJINTERFACE specification leads to the declaration of a new interface, inheriting operation-definitions from the predefined interface ObjInt, and the creation of an interface manager object.

$COOM_B$ extends $COOM_F$ with constructs for interfaces with predefined operations - object interfaces, and the automatic creation of manager objects with functionality for object creation and object management:

**Definition 11.1 $COOM_B$:**

$COOM_B$ is an n-tuple, $COOM_B = < COOM_F, OIN, OIM >$ where:

- $COOM_F$ is the constructs defined in the COOM Foundation model.
- $OIN$ is a finite set of Object Interface Names, where $OIN \subseteq IN^1$.
- $OIM$ is a finite set of Object Implementation Names, where $OIM \subseteq IM$.
- For each interface name $inn \in OIN$ there is a corresponding object interface manager object, interface manager, with the name $metainn$ and type interface $META_{inn}$.
- For each implementation name $inn \in OIM$ there is a corresponding object implementation manager object (instance manager) with the name $metainn$ and type interface $META_{inn}$.

\[ \square \]

The possibility to define Object Interfaces is introduced through the OBJINTERFACE construct in ODL.

**Definition 11.2 Object Interface:**

An Object Interface $oi$ is defined as the tuple $oi = (in, ObjInt, META_{inn}, metainn, METAOP)$, where:

- $in$ is an interface $in = (inn, P, E, N, O, AO)$ as defined in 10.6, in which $inn$ is the interface-name.
- The interface $ObjInt$ is the implicit root interface in the transitive list of parent interfaces, $P$.
- If $in$ has the interface name $inn \in IN$, then an interface manager object with the name $metainn$ is automatically created. This manager object will have an interface and implementation defined by the interface and implementation $META_{inn}$, which is derived from $METAObjInt$ and the parameterized interface $M<inn>$.

\[^1IN\text{ and IM were defined in section 10.1}\]
11.1 Object Interfaces and Manager Objects

- \( \text{METAOP} \subseteq \text{OP} \) are operations defined in the object-interface with the prefix META. They become operations of \( \text{meta Inn} \), specified in \( \text{META Inn} \). This means that the operation will be defined for the instances of type \( \text{META Inn} \), in particular for the object-interface manager object named \( \text{meta Inn} \).

\[ \square \]

Definition 11.3 Interface Manager Object:
An interface manager object \( \text{meta Inn} \) is defined as an object with the interface and implementation \( \text{META Inn} \). There is no separation between interface and implementation for manager objects. The purpose of an interface manager object is to manage a set of objects with interface \( \text{inn} \), through managing a set of implementation managers with the interface \( \text{META Inn} \), where \( \text{inn} \) are the existing implementations corresponding to \( \text{inn} \). The manager objects have an OID, and there is a one-to-one mapping between the OID and the name \( \text{meta Inn} \). \( \square \)

![Diagram](image)

Figure 11.2: Type-and MetaType-hierarchy with operations for Person and Employee interfaces, and corresponding manager objects

Object Implementations and Implementation Manager Objects are defined in section 11.2. The manager objects are similar to class-objects in object-oriented programming languages like Smalltalk, with the exception of the distinction between interface manager objects and implementation manager objects. Some of the operations provided by manager objects are creation-operations like: \( \text{create, get ObjectFromOID, and create InObjSpace} \) described in section 11.2.3.
retrieval-operations like: exten tand find described in section 11.2.4, and management-
operations like addImplementation, removeImplementation, implementations, addSubType,
removeSubType, and subTypes described in section 11.2.2. An overview of these operations
is also given in the left part of figure 11.2.

An Object Interface definition, named inn, is an Interface-definition extended with a pos-
sibility for specifying META-operations, and with the automatic creation of a manager
object of type METAinn, with the name metainn. In addition, the syntax for an object-
interface definition is extended with constructs for defining attributes, unique attribute-
combinations, META-attributes and containment-relations. This belongs to the COOMS-
model and is further described in chapter 12.

The predefined manager interface, METAObjInt, defines a set of operations that are used
for creating and querying for objects with a particular interface. The manager objects are
not themselves using a split between its interface and implementation part. This is not
necessary since there will be only one implementation for these, in addition it saves the
need for yet another meta-level. The creation of manager objects is automatic and the
application programmer do not have access to any manager objects for manager objects.
The manager objects have an interface as defined in definition 10.6, but not an object
interface as defined in definition 11.2.

A parameterized manager-interface, M(Type inn) specialized from the METAObjInt is
used as a base for the interface of the object-interface manager-object for objects with in-
terface inn. In addition an object-interface manager will inherit definitions for the META-
operations defined for supertypes of inn. Note that this is use of non-conform inheritance.
In particular the operation Set<T> extent leads to a non-conform find-operation for
subtype that is not conform with the supertype, because it returns a set of objects that is
not conform with the set of objects returned by the supertype. This is discussed more in
section 11.2.4.

Operations are defined as for Interfaces, previously described in section 10.3.1, notifications
and exceptions are described in section 11.5 and 11.6. META-operations are operations
that will be defined for the manager-object metainn.

Figure 11.2 shows how the object interfaces Person and Employee in the ODL-schema from
example 11.1 are represented as two interfaces having OBJINT as the root.

In addition two interfaces METAPerson and METAEmployee are created from the pa-
parameterized interface M<T>, where T is the object interface Person or Employee. Two
manager objects metaPerson and metaEmployee will be created as instances of the in-
terfaces METAPerson and METAEmployee respectively. The operations provided by the
manager objects are shown in figure 11.2 and described in the sections 11.2.3, 11.2.4, and
11.2.2. Appendix B on COOM OML gives a more detailed overview of these operations.

The motivation for the object interface construct is the goal of providing a convenient
integration framework, which offers as much basic support as possible for the application
programmer. Interface-oriented frameworks like OMG CORBA [Obj92a] or OSF/DCE
[osf90] have interface-constructs similar to the interface-construct defined for COOM. They
do not have any constructs similar to manager objects, or predefined interface roots.

In order to make it easier to define object-interfaces, ODL provides a special syntax for
the definition of object interfaces:

The syntax for an <object_interface_declarations> is:

<object_interface_declarations> ::= 
OBJINTERFACE <objInterfaceID> [ <i_param_list> ]
11.1 Object Interfaces and Manager Objects

[<supertypes>] '{' '{' [<object_interface_elem> '{';'}'] * '{'}'}' |
      OBJINTERFACE <objInterfaceID>
      /* Forward declaration */

The syntax for <object_interface_elem> is:

<object_interface_elem> ::= [META] <operation_declaration>
                      | [META] <attribute>
                      .  | <control_interface_declaration>
                      | <exception_declaration>
                      | <uniques>
                      | <notification_declaration>

Example 11.1 Person-Department

SCHEMA PersonSchema {
  import Standards;
  import Basics;

  Notification Changed
  {;

  Notification ChangedAge: Changed
  {int age;
  };

  OBJINTERFACE Person {
    // The use of META-operations is an extension in COOM_B
    META Person create(String [in] name, int [in] age);
    META int averageAge();
    META int nrOfPersons();
    String name();
    void name(String newName);
    int age();
    void age(int newAge) Raises(illegalAge) Notifies(ChangedAge);
    int personNumber();
    int personNumber(int newPersonNumber);
  }; /* ObjInterface Person */

  OBJINTERFACE Employee: Person {
    META Employee create(String [in] name, int [in] age, int [in] salary);
    META int averageSalary();
    int salary();
    void salary(int newSalary) Raises(illegalSalary)
        Notifies(ChangedSalary);
  }; /* ObjInterface Employee */
}

/* SchemaPerson */

\footnote{attribute, uniques and control_interface are introduced in chapter 11.
The definitions of Person and Employee as object interfaces in ODL will generate Person and Employee as specialized interfaces of the interface named ObjInt. Person inherits operations directly from ObjInt, while Employee inherits operations from Person. Object interface manager objects with the names metaPerson and metaEmployee will be created. This is also shown in figure 11.2.

The name metaPerson is used as the name for an object that is used for managing and creating new instances of Person. The COOP-system has a name-service that supports retrieval of objects by name. For each object-interface inn defined there will exist an object interface manager-object with interface type METAinn. This object will be created in the Object Dictionary object space, where all interface manager objects are residing.

The interface type METAinn is used for managing the set of objects with interface inn. METAinn supports operations for creating new objects and for retrieval of existing objects. An important ability of METAinn is the potential for managing a set of different implementations of interface inn, and the selection of an implementation when a new object is created. METAinn will have knowledge of all implementations of interface inn and of all interfaces specialized from interface inn.

ObjInt-operations are described in the next section. Object-manager operations are described in section 11.2.1, together with the descriptions of the management of object implementations.

11.1.1 OML - Object Interface Operations

The COOM OML - Object Manipulation Language, is realized by mapping ODL interfaces to constructs in existing object-oriented programming languages such as C++ and Smalltalk, by the use of language specific constructs for the actual implementation. This mapping is described in section 11.4.

This section defines the functionality provided this way by objects which are defined to be of type object interface, through the ODL-construct OBJINTERFACE. An overview of the operations for the ObjInterface was given in figure 11.2. These operations fall into the following two groups:

- Copy and delete operations
- Information operations

The following gives a brief introduction to the functionality of operations in each group, a more detailed description can be found in Appendix B.

Copy and Delete Operations

The copy/delete-operations provides functionality for creating various types of shallow/deep-copies of an object. A copy-operation returns an object which is of the same type as the original object.

The following operations are defined:

- copy - deepCopy and shallowCopy
  The copy-operation returns a copy of the object, of its actual type. The default copy-operation is a shallowCopy, which returns a copy of the object itself. All contained
11.2 Object Implementation and Extent Management

objects (through containment relations\(^3\)) are shared between the new and old object. The `deepCopy` operation returns a copy of this object and all contained objects, reachable through containment relations. All these copy operations exist in two variants. One which is returning objects typed with the same type as the copied object, and another which returns the object typed as ObjInt. The last one is useful in situations where the expected type is not known beforehand.

- **delete**
  The `delete` operation deletes the object and its underlying representation.

The copy and delete operations are specified in more detail in the ObjInt Interface in Appendix B.

**Information Operations**

The Information operations will provide useful information about an object, e.g. what OID does it have, which implementation and interface does it have or what object space does it belong to. The operations defined are the following:

- **Information about interface and implementation managers**
  These operations return the interface manager object and the implementation manager object for an object: `(objIntMan, objImpMan)`.

- **Information about interface and implementation definitions**
  These operations return the Interface definition object and the implementation objects for an object. These are objects generated by the COOM compiler, that represents the definitions in the ODL schema.

- **Information about object space and OID**
  These operation return the Object Space that an object belongs to, and its OID represented as a string. The string representation of an OID can later be given to an interface manager in order to retrieve the object. All objects will provide its OIDString when requested: `(oSpace, oidAsString)`.

The information operations are specified in more detail in the ObjInt Interface in Appendix B.

11.2 Object Implementation and Extent Management

Figure 11.3 shows both interfaces and implementations with corresponding META-types, for the Person-Employee example. The META-types are defined by an interface and implementation with the same name.

The implementations PersonIngres and PersonOS are conform to the Person interface. There is a META type for each implementation similar to the way which there is a META type for each interface.

**Definition 11.4 Object Implementation:**
An Object Implementation \(\text{oim}\) is defined as the tuple \(\text{oim} = (\text{im}, \text{ObjImp}, \text{METAimn}, \text{metaimn}, \text{METAOP})\) where:

\(^3\)Containment relations are described in section 12.3.7.
11.2 Object Implementation and Extent Management

Figure 11.3: Type-hierarchy for Interfaces and Implementations

- im is an implementation im = (iimn, inn, OS, OB, f, Ω) as defined in 10.11, where
  iimn \in IM is the implementation-name, inn \in IN is the interface-name, OS \subseteq OSN is
  the Object space names for this implementation, OB is a set of operation-bodies for
  the operation-signatures of inn. f is a correspondence-mapping between operation
  signatures and operation bodies and Ω is the extent of the implementation.

- ObjImp is the transitive root for the implementation.

- If im has the implementation name inn \in IMN, then an implementation manager
  object with the name meta inn \footnote{It is not strictly necessary for the implementation
  manager objects to have names, as the application programmer never will refer to these. It is,
  however, useful to have a name when they are textually described as here.} will be created. This
  manager object will have an interface and implementation defined by the interface and
  implementation META inn, which is derived from METAObjImp and the parameterized interface
  MI< inn >, where inn is the interface for the implementation named inn.

- METAOP \subseteq OP are operations defined with the prefix META in the object interface
  definition. They become operations of meta inn, where the operation-bodies are
  defined in the interface and implementation META inn.

\[\Box\]

Definition 11.5 Implementation Manager Object:
An implementation manager object meta inn is defined as an object with the interface and
implementation META \textit{imm}. There is no separation between interface and implementation for manager objects. The purpose of an implementation manager object is to manage a set of objects with implementation \textit{imm}. The implementation manager object has an OID, and there is a one-to-one mapping between the OID and the name \textit{metaimm}.

Figure 11.3 shows four object implementations, PersonIngres, PersonOS\textsuperscript{5}, EmployeeIngres and EmployeeOS, with corresponding four META types. Both the implementations and the META types provide operation- implementations for the required operations, customized for the actual target system, i.e. Ingres or ObjectStore. These implementations are the basis for the actual instantiations of Person-objects or Employee-objects, being found in the Ingres-system or the ObjectStore-system. Implementation manager objects will be instantiated from the META-types, as shown in figure 11.4.

This figure shows two interface manager objects in the Object Dictionary object space, as instances of METAPerson and METAEmployee respectively. The interface manager objects have knowledge about their corresponding implementation manager objects, which reside in the same object space as the objects that they manage. There are two object implementation managers in each object space in the example, for the local implementations of Person and Employee respectively. Each object implementation manager takes care of the administration of each concrete implementation and its belonging instances.

Figure 11.5 shows the manager objects with their default names, for the different object spaces.

The application-programmers using ODL shall not have any knowledge of the actual implementations for each interface. The interfaces Employee and Person are the interfaces used as object-types by the application-programmer. The application-programmer will not be aware of the different implementations, only the interface.

When a new object is created, however, a decision needs to be made about which imple-

\textsuperscript{5}OS is used as an abbreviation for ObjectStore
11.2 Object Implementation and Extent Management

![Object hierarchy with named objects](image)

The notion of Object space has been introduced\(^6\) in order to provide a higher level of implementation-knowledge. The application-programmer will not know the actual name of an implementation, but he might know that he wants an object created in a particular object space.

Another possibility is to use context-dependent knowledge. Each process will be associated with a context, which essentially is a file of name-value mappings, where the values are strings. The createContext-operation will look up in the environment-file and choose the implementation-name that currently is associated with this interface-name. The nameservice provides the facility for retrieving the correct implementation. This is further described in section 11.7.

When ODL is used as an interface-description-language for already existing system components, the implementation-code will consist of the mapping to and from the existing system. Object Interfaces in COOM ODL are mapped to appropriate class-definitions in an underlying object-oriented programming language, such as C++ or Smalltalk. Object implementations are not specified in COOM. In the cases where the goal of COOM is to do systems integration of already existing systems, implementations already exist. The goal of an object-implementation is then to do appropriate mapping to and from underlying systems. The object implementations are either hand coded or generated from the ODL-compiler, based on knowledge about the interface and the underlying system that an implementation will be generated for.

11.2.1 OML - Object Manager Operations

The COOM OML - Object Manipulation Language, is realized by mapping ODL interfaces to constructs in existing object-oriented programming languages such as C++ and Smalltalk, and by the use of language specific constructs for the actual implementation, as described in section 11.4.

\(^6\)See the definition for Object Space in section 10.13.
This section defines the functionality provided this way by the manager objects which are automatically created as a result of OBJINTERFACE definitions. An overview of the operations provided by manager objects was shown for the M<T> interface in figure 11.2.

The functionality provided by manager objects fall into the following three groups:

- Implementation- and subtype- management operations
- Object-creation operations
- Object-retrieval operations

Figure 11.6 illustrates how an interface manager object has a knowledge of active object spaces and corresponding implementation managers, and of the interface manager objects of its subtypes. Each object space will have a corresponding implementation manager.

The interface manager object, metaPerson, has knowledge about two implementation managers metaPersonIngres and metaPersonOS, for the object spaces Ingres1 and Ostorel respectively. In addition it knows about the interface managers for its subtypes, here metaEmployee, which itself has knowledge about its object spaces with implementation managers and eventual interface managers for subtypes.

The following gives an overview of the functionality provided by the different operations. More detailed specifications, with operation-names and input and output parameters is given in Appendix B.

### 11.2.2 Interface Management Operations

The following operations are used for the management of the object implementations for an object interface, and for the management of specialized interfaces or subtypes. The operations are useful in order to create the structures shown in figure 11.6.
11.2 Object Implementation and Extent Management

• **Implementation management**
  The `addImplementation` operation adds a new implementation to the set of implementations for this interface, associated with an object space.

  The `removeImplementation` operation removes an implementation from the implementation set. The implementation manager interface type is a subtype of `MI<T>` where `T` is the interface type. This ensures that the operations will be correctly typed. The `implementations` operation returns the set of all implementations for an interface.

• **Subtype management**
  The following operations are for the management of subtypes. Note that it is not possible in advance to know the exact type of subtypes, since they per definition have not yet been defined. We do, however, know that the types are specialized from this type, and can use that knowledge to do a legal type-cast on objects typed by `ObjInt`. These operations are therefore defined for the `ObjIntManager-interface`.

  The `addSubType` operation adds a new subtype, by adding the object-interface manager-object which are managing an interface that is specialized from this, to the set of object managers for subtypes. The `removeSubType` operation removes a subtype, by removing the corresponding object-interface manager-object. The `subTypes` operation returns the set of all object-interface manager-objects which are managing interfaces specialized from this.

11.2.3 Object Manager Creation Operations - Interface and Implementation

The create-operations are defined for both the object interface manager and object implementation manager objects. The application programmer will only interact with the object interface manager object, named `metaInn` for the object interface named `inn`. The actual implementations of the operations will work differently for the object interface manager object than for the object implementation manager object. The purpose of the create-operation for the object interface manager object is to select the right implementation for creation. The purpose of the create-operation for the object implementation manager object is to create an object as an instance of the implementation that it manages, and to maintain and administrate the extent for this implementation:

• **Create operations**
  The application programmer sends a request for object-creation to an interface manager object. It is possible to specify an argument with which object space to create the object in. If no object space is specified, the default object space for this interface will be chosen.

  The create operation returns a new object of the type `Self`, and with the implementation that is managed by this object-implementation manager-object.

  Figure 11.7 illustrates how an implementation manager object has a knowledge of the objects. It has an internal representation which is suitable for the underlying implementation. Here the implementation manager `metaEmployeeIngres` is showed in more detail. Different implementation managers might choose different strategies for this, as described further in chapter 15. The interface manager object finds the implementation manager object for the chosen object space, and delegates the create request to this. Typically the internal representation in the interface manager object
Figure 11.7: Enhanced view of the metaEmployeeIngres implementation manager

is a dictionary-map from object spaces to corresponding implementation manager objects.

The implementation EmployeeIngres has knowledge about how each proxy object will retrieve and update values, or invoke operations, in the underlying database implementation.

Example 11.2 Object-creation:

ODL:

```
META Employee create(String [in] name, int [in] age);
```

C++:

```
metaEmployee->creationSpace(Ingres1); // Set default object space
Employee* p = metaEmployee->create("Arne- J. Berre", 34);
```

- `getObjectFromOIDString`

  An OIDString contains the necessary information to retrieve or create a proxy object. This operation returns the object corresponding to a given OID with this interface, based on delegating the operations further to the correct object implementation manager-object. The OID contains the name of the object implementation manager-object as part of the OID, which is used for finding the correct implementation.

  All interface managers are able to do this for their respective implementations, and to return an object of the correct type. If the interface is unknown, the metaObjInt manager will return the correct object with type ObjInt.

---

7 Typically an OIDString contains objectSpace, interface and implementation manager-name and an implementation dependent local identifier
11.2 Object Implementation and Extent Management

- `creation.Space(OSpace [in] os)`
  This operation gives a new value to the default object space used for creation-operations.

11.2.4 Object Manager Retrieval Operations - Interface and Implementation

The operations `extent` and `find` are used for retrieving objects. All objects with a given interface are retrieved by `extent` and selected subsets are retrieved by `find`:

- `find`
  This `find` operation parses an input queryString, represented in OQL-syntax\(^8\), does possible global optimization, and delegates corresponding queries or sub-queries to the underlying implementations and sub-types by `find(QTree [in] qTree)`, where `qTree` is an internal representation for the relevant part of the query. The `find` operation of the implementation manager returns the result of `find` for its particular underlying system. This operation returns the set of objects for which the predicate represented by the query-tree evaluates to true. It returns a set of objects with the same object interface as the interface manager. A special `findOne` operation returns only one object satisfying the predicate.

- `extent` and `directExtent`
  The `extent` operation returns the union of `extent` from all implementation managers for an interface together with the union of `extent` for all specialized interfaces of this interface. The `extent` operation of the implementation manager returns the set of all objects being instances of this implementation. The `directExtent` operation returns the union of `extent` from all implementation managers for this interface, but not from any subtypes. Only implementations from active\(^9\) object spaces are included.

- `cacheHint(String [in] cacheNames)`
  This operation is a potential future operation which gives a hint to the interface manager about which attributes and relationships that possibly might be accessed. The underlying implementations might use this to optimize the use of the cache. The cacheNames are names for abstract attributes, and relation-, or role-names\(^10\).

- `Object/Interface trading`
  The `trade` operation is a potential future operation which parses a tradeString, represented in a trading-language, and selects or delegates to the implementations. The current implementation does not support a separate trading-language, but the use of an ANSA-like language could be considered [ans89]. This will contain attributes for selection of objects based on e.g. min/max cost and/or time for the different interfaces provided. Currently the effect of a `trade` operation is achieved by a `create` or `findOne` operation.

The operations above also exist in a form where the results are typed by the root `ObjInt` instead of the actual interface. These operations are useful when operations are done on sub-types, which one does not know the actual type of. A specialized Set `< Employee >` is not a conform subtype of Set `< Person >`, and cannot be assumed as a result, where

\(^8\)See more description of OQL in section 13.
\(^9\)See section 11.3.
\(^10\)Abstract attributes and relations are described in chapter 11.
11.3 OML for Object Space Management and Transactions

Set< Person > is expected. This is because it is not allowed to add a Person object to a set of Employees, if the person is not also guaranteed to be an Employee. This fact also leads to the situation that an object interface manager for a subtype, i.e. Employee, is not conform with the object interface manager of a supertype, i.e. Person. It is useful to have operations like extent and find that returns a set of objects of with the actual interface. Because of this, there is a non-conform inheritance-relationship in the type-hierarchy for object interface managers.

11.3 OML for Object Space Management and Transactions

This section describes operations available for the management of object spaces and handling of transactions. The section also discusses the need for more advanced transaction models.

11.3.1 Object Space Management

Object space management is supported through the OBJINTERFACE OSpace. It is possible to dynamically activate and passivate object spaces. Only active object spaces will be included in the object management operations.

Definition 11.6 Object Space interface:

```
OBJINTERFACE OSpace {
    META Set<OSpace> activateFromCtx();
    META OSpace create(String [in] name);
    META OSpace getByName(String [in] name);

    Boolean activate();
    Boolean passivate();
};
```

11.3.2 Transactions

A Simple Transaction Model

The current specified transaction-support is a very simple ACID-model\(^\text{11}\), and assumes that the underlying system can support a 2-phase commit-protocol. In this model all system-specific transactions will be done as part of one global transaction.

Each transaction will have a read or update mode. When interaction with an underlying system is done, the transaction-mechanisms of this system might also be used. Updating of values might be done through write-through to the underlying systems, and we might get a situation with subtransactions on different underlying systems in parallel. A commit on a top-level transaction will then require a two-phase commit on the underlying systems. This is possible if the underlying systems can support a prepare-to-commit protocol.

\(^{11}\)ACID stands for the following four properties: Atomicity (or failure atomicity), Consistency, Isolation (or serializability), and Durability (or permanence).
Definition 11.7 Transaction interface:

```c
OBJINTERFACE Transaction {
    ENUM TType {Read, Write};

    META Transaction begin(TType [in] tt);

    Boolean commit();
    Boolean abort();

    addLocalTransaction(Transaction [in] tx);
    Boolean prepareToCommit();
};
```

A transaction is created by the Transaction::begin-operation. This creates a transaction-object which is used to keep information about subtransactions.

The GlobalTransaction-manager will be the coordinator for a 2-phase commit procedure. There will exist LocalTransaction-managers for each involved subsystem. In phase 1 the coordinator requests all participants to get into a state in which they can do either commit or abort, by the operation prepareToCommit. If the participants succeeds in this it returns OK.

If all replies are OK the coordinator broadcasts commit to all participants. The net effect is thus that all participants either commit or abort.

All operation-implementations will check if there is a current transaction for the local system associated with the current global transaction. If there is no local transaction, a local transaction will be initiated and a local transaction-object will be created and added to the list of local transactions.

Problems with a Simple Transaction Model

Transaction management has not been in focus in this work, but is nevertheless important for a future complete solution. The following provides a basis for future development of transaction management support in the COOP framework.

In general, all underlying systems will not support the ideal prepare-to-commit protocol, and some systems might not have a transaction mechanism at all. In such cases one might need to use compensating transactions to redo what has been done. Initially it is, however, focused on systems which can participate in a 2-phase commit.

The transaction structure may vary from flat transactions to closed nested transactions to open nested transaction, and combinations. The object structure may vary from simple objects to ADT-instances to complex objects to active objects, or combinations.

A particular problem is transaction handling for proxy objects versus transactions for the underlying systems.
11.3.3 Alternative Transaction Models

A conceptual basis for transactions is given in [Gra81]. Later work has focused on transaction-support for specific requirements, such as long-lived design-transactions, or transactions for multidatabase-systems. Different proposals have focused on different, often conflicting, requirements. A number of advanced transaction models have lately been proposed. A good overview is given in [Elm92].

The basic ACID-properties of a traditional transaction-mechanism might need to be relaxed for certain transactions. Many proposals for more advanced transaction mechanisms have introduced different kinds of subtransactions:

- A *compensating transaction* for a transaction T is a transaction that can semantically undo the effect of T after T has been committed.
- A *contingency transaction* is another subtransaction that implements an alternative action.
- *Non-vital transactions* are transactions that might be aborted without a required abort of parent-transactions.

Nested Transactions [Mos81] is a transaction model that allows for a parent transactions to choose between different actions if a sub-transaction aborts. The parent-transaction might ignore the condition, choose to retry the subtransaction, initiate a contingency transaction, or abort. If a subtransaction commits, aborting one of its superiors will undo its effects.

Sagas [MS87] are based on compensating transactions. A Saga is a long-lived transaction that is based on a set of relatively independent steps, T₁, T₂, .., Tₙ. Sagas preserve the atomicity and durability properties of traditional transactions, but relaxes the isolation-property since partial results are revealed to other transactions.

The most promising approaches being investigated are S-transactions [EVT88], open nested transactions [Wei91], Polytransactions [RS91b], and the DOM transaction model [MH*92]. These all involve the use of compensating transactions and contingency transactions.

The S-Transaction-model [EVT88], was introduced to support cooperation of the international banking system, where local autonomy is a major concern. A component system is entitled to execute a subtransaction of an S-transaction in any way it wishes. In case of failure, a component system can issue an equivalent request to an alternative system. Like in Sagas, S-transactions use compensating transactions for recovery, each subtransaction will commit when it completes.

Open Nested Transactions [Wei91] is a generalization of multilevel transactions. The key idea for multilevel transactions is to exploit level-specific semantics for concurrency control.

Polytransactions [RS91b] focuses on how to maintain interdatabase dependencies in a multi-database-system. This is based on having an interdatabase dependency schema with data dependency descriptors. When a transaction T is executed, the data dependency descriptors are checked to see if actions need to be taken to maintain the consistency of the database. The appropriate actions might result in other actions, and the transaction T is considered to be a root in a polytransaction.

The DOM Transaction Model [MH*92] is among the most promising for adoption to the COOP integration framework. The goal of the DOM project is similar to the COOP goal, by supporting application development in a distributed object-oriented environment that integrates various component systems. The DOM Transaction Model allows closed nested
and open nested transactions and combinations of the two. Closed nested transactions are similar to traditional nested transactions, while open nested transactions allow the partial results of a transaction to be viewed by other transactions. This can be combined with compensating and contingency transactions. A formal correctness theory for this model has not yet been developed.

The EPOS project has addressed problems related to cooperative transactions [Ng92]. Other proposals described in [Elm92] include the ConTract model, Split-Transactions, Flex Transactions and others.

It is interesting to note that mechanisms from process integration support, i.e. as described in section 4.1.5, might be used as part of more advanced transaction models. There is ongoing work at NTH [Hag92, CM91] on the evaluation and development of different transaction models for integrated environments.

## 11.4 Mapping from COOM$_B$ to Object-Oriented Languages

COOM has been designed to be independent of any particular object-oriented language. In particular, mappings have been done for both a statically typed compiled object-oriented language, C++, and an untyped interpreted object-oriented language, Smalltalk.

Most object-oriented programming languages do not support a strict separation between interface and implementation as focused by COOM. Instead of adopting one of the experimental languages, such as Emerald [HM85b] Guide [Tea90] or POOL [AvdL90], it has instead been chosen to map COOM to much used object-oriented languages such as C++ and Smalltalk. The reason for this is that none of the experimental languages gives the needed support for object management and structural extensions. They are not in much use and thus not supported by services like object-oriented database systems or object-request broker technology, which is needed by the COOP architecture. Experimentation with COOP and COOM is easier motivated by popular languages like C++ and Smalltalk.

An overview of mapping-strategies from COOM$_B$ to C++ and Smalltalk is given in the following. More details about the C++ mapping can be found in [Sol92]. More details about the Smalltalk mapping can be found in [Øst92b, Øst92a].

### 11.4.1 Mapping from COOM$_B$ to C++

The following sections give an overview of the mapping from COOM basic types, interfaces and object-interfaces to C++.

**Basic Types, Interfaces and Implementations**

The base types have simple and direct mappings to C++.

C++ does not textually separate between interfaces and implementations. Some of the intended effects can be achieved through the use of abstract super-classes called interface-classes, and concrete derived classes, called implementation-classes.

In C++ it is necessary to define implementation-classes as derived classes of interface-classes in order to let the application-programmer know and use only the interface-classes. All operations in the interface-class need to be declared as public and virtual, and are implemented with an appropriate body in the implementation-class. The interface-classes will then form a directed graph, and the implementation-classes will always be derived
11.4.1 Mapping from COOM to C++

from an interface-class. Possible inheritance of code and internal representation between implementation-classes is not part of the COOM-model. The representation for each implementation-class will be defined in the private part of the class. All references to objects are managed through pointers to objects of the corresponding interface-class. The implementation-classes are either hand coded or generated from the ODL-compiler, based on knowledge about the interface and the underlying system that an implementation will be generated for.

All interfaces in COOM are passed as references, which will be viewed as pointers in C++. This means that all interface-references in ODL will be replaced by object-pointers when mapped to C++. Simple types are passed as values. Constructed types and the String simple type will be passed as pointers.

C++ does not give a fully satisfactory support for conformance-based typing as specified for COOM in section 10.3.2. The C++ mapping is therefore restricted in this area, e.g. an inherited copy-operation will always be typed to return an object of type ObjInt only. This is because it is not legal in C++ to redefine only the result-type of an operation. This could have been circumvented by defining copy-operations to take the type of the object itself as a parameter, but that would have created a different copy-operation for each interface. This would create a problem when iterating through a set of objects. If one would like to make a copy of each object in the set - one would not be able to request the correct copy-operation.

The following shows some of the generated C++ classes for OBJINTERFACE Person in example 11.1.

Example 11.3 C++ interface-class and implementation-class

```cpp
Class Person: public ObjInt { /* Interface class */
    public:
        virtual String* name();
        virtual void name(String* newName);
        virtual int age();
        virtual void age(int newAge) throws(IllegalAge);
        virtual int personNumber();
        virtual void personNumber(int newPersonNumber);
    }

Class Employee: public Person /* Interface class */
    { ... };

Class PersonIngres: public Person /* Implementation class */
    { ... };
```

Object Interfaces

Object Interfaces introduce interface manager objects of type METAinm, which takes care of object management and services related to a set of objects, rather than a single object. In C++ [Str91] there are no class-objects or predefined root class. It is necessary (useful) to use meta-objects instead of static variables and static functions to provide for persistent
storage of and extensible handling of implementation classes. The use of meta-objects allows for the use of inheritance and dynamic binding for the use of correct operations, when a new implementation is introduced\(^\text{12}\).

There exist a number of proposals for the extension of C++ with class-objects, or meta-objects. None of these give support for the separation of interface-classes and implementation-classes with parameterized typing, so a specific scheme has been developed for COOM.

It is possible to hide the use of meta-objects from the application-programmer by instead providing the necessary operations through static functions in the object interface class. The application-programmer does not need to be aware of the object management going on behind the scenes.

In the C++ mapping it is possible to encapsulate the functionality of META\textit{inn} in static member functions and to use the class name \textit{inn} only. However, it is useful to have an object for maintenance of the internal structures.

**Example 11.4 C++ interface-class and implementation-class**

```cpp
Class METAObjInt {
public:
    virtual ObjInt* createOI();
    virtual Set<ObjInt*>* findOI();
    virtual Set<ObjInt*>* extentOI();
};

Template<Class T>
Class M : METAObjInt {
public:
    virtual T* create();
    virtual Set<T*>* find(String*);
    virtual Set<T*>* extent(String*);
    virtual void addImplementation( Mi<T>* imp, OSpace* for);
    virtual void removeImplementation( Mi<T>* imp, OSpace* for);
    Set<Mi<T>>* implementations();
};

Class METAPerson: public M<Person> {
public:
    Person* create(String* name, int age, OSpace* os);
    int averageAge();
    int nrOfPersons();
};

Template<Class T>
Class MI: M<T> {
public:
    /* re-implement interface-defined functions */
};

Class METAPersonIngres: public MI<Person>{
```
\(^{12}\)It is possible to achieve the same effect by the use of an array of function-pointers, but this then becomes a lower-level implementation of the same concept.

public:
    /* re-implement interface-defined functions */
};


The object interface manager, metaPerson, and the object implementation managers, metaPersonIngres1 and metaPersonOS1, are created as persistent variables in the Object Dictionary. The interface manager might also be accessed through a static function in the interface-class, Person.

Example 11.5 C++ named variables

```c++
persistent<od> METAPerson* metaPerson = METAPerson();
persistent<od> METAPersonIngres*
    metaPersonIngres1 = METAPersonIngres("Persontab", "id");
persistent<od> METAPersonIngres*
    metaPersonOS1 = METAPersonOS("Personxx");
```

Example 11.5 shows how named objects are created in the Object Dictionary, od, by using the persistent variable facility in ObjectStore.

11.4.2 Mapping from \textit{COURM} to Smalltalk

This section gives a short overview of the mapping from \textit{COURM} to Smalltalk. A detailed description can be found in \cite{Ost92b, Ost92a}.

In Smalltalk "everything" is an object, also the COOM basic types. These are, however, not given their own OID. The COOM basic types are mapped to similar Smalltalk classes. Object interfaces are mapped to Smalltalk classes. The meta-class structure in Smalltalk automatically creates a meta-class with a class-object as the single instance for each defined class. META-operations in \textit{COURM} become class-operations in Smalltalk.

Since Smalltalk is an untyped language, there is no possibility to use typing in the operation-definitions. In Smalltalk there are no such typing-conflicts, and there does not need to be a class-relationship between interface- and implementation-classes. All objects might be used, as long as they support the operations required by the interface type.

A Smalltalk-programmer will, however, always be interested in the expected interfaces of objects, and the information in the ODL-schema will provide that knowledge. Since the COOP Integration Framework provides a generative approach, where implementations will be generated automatically, it is possible to assure that the interface-types will remain valid.

Example 11.6 Smalltalk interface-class and implementation-class
11.4 Mapping from COOM to Object-Oriented Languages

```
ObjInterface subclass: #Person
instanceVariableNames: ''
classVariableNames: 'ImplementationDict'
poolDictionaries: ''
category: 'PersDepInt'

!Person methodsFor: 'attributes'
  age
  age: anInt
  name
  name: aString
  personNumber
  personNumber: anInt

!Person class methodsFor: 'instance creation'
  createName: aName
  age: anAge
  os: anOs
  averageAge
  nrOffPersons

!Person class methodsFor: 'impinstallation'
  addImplementation: aClass
  for: anOs
  removeImplementation: aClass
  for: anOs
```

Example 11.6 showed the Smalltalk-representation for the object interface Person.

11.4.3 COOP as an Application Framework for Systems Integration

Section 5.5 described the general principles of object-oriented application frameworks. An application framework was defined to be a set of classes for interacting components in a particular domain, where the control-structure of the interaction is embedded into predefined operations and interaction-sequences, and where new applications are made by subclassing specific classes and reimplementing given operations for the new application.

These principles are applied for COOP, in order to make COOP an Application Framework for Systems Integration. It provides an abstraction which covers both communication with services through control integration and databases through data integration.

There is a default set of operations for objects and another set of operations for manager-objects. Some of these operations need to be reimplemented by the component integrator.

The following are the steps to follow:

1. Generate interface classes in C++ or Smalltalk, by running the ODL-definition through the ODL-compiler.
2. If a backend for the target-system exists, it is possible to get automatic support also for the generation of implementations.
3. If no backend exists, only empty implementation classes might be generated.
4. Implement the operations for the object implementation class (e.g. copy + user-defined operations)
5. Implement the operations for the object implementation manager class (e.g. extent, find + user-defined operations).

6. Define the object space that the implementation will reside in.

7. The component integrator will get the management-objects installed in the Object Dictionary and the correct Object Space, and the ODL and interface-definitions available for application/tool-builders.

8. Create an instance of a user interface class YourView that interacts with YourModel. YourModel is an application-specific class which serves as a manager for the access to required functionality and information.

9. Retrieve initial objects by an extent- or find-operation for the relevant object interface managers.

The concept of object-oriented application frameworks was described in chapter 4. Chapter 13 and 14 will give a more detailed overview of the process of component integration and application building.

A similar application-framework-approach to an object-oriented integration framework is described in [HD92].

11.5 Notifications

Notifications are introduced in order to support the requirements for notification-oriented control integration, stated in section 3.3.3. Notifications is a mechanism for providing asynchronous indirect interaction among objects. The mechanism provides a facility for objects to notify about events, and for objects to register their interest for being notified about events. The possibility to specify a hierarchy of notifications in the schema is novel in COOM.

The specification of potential notification events, is supported by ODL, because notifications generated from an object is an important part of the external knowledge about an object. ODL provides a syntax for defining notifications with value attributes in a directed graph, and to specify which notifications might be generated after which operations from objects supporting a certain interface.

11.5.1 Notification Definition and Example

Definition 11.8 Notification:
A Notification not is defined as the tuple not = (nn, PN, ATT) where

- \( nn \in \text{NN} \) is the notification name.
- PN is a sequence of parent-notifications, from which ATT-definitions are inherited.
- ATT is a set of (type,name)-pairs where type \( \in S \), simple types.

Example 11.7 shows the declaration of a notification, when the age of a person changes.
Example 11.7 Notification:

```
Notification Changed
{}

Notification ChangedAge: Changed {
    int age;
};

OBJINTERFACE Person {
    ....
    void age(int newAge)
        raises(IllegalAge) notifies(ChangedAge);
    ....
};
```

Example 11.7 shows use of the notification, ChangedAge, which is generated after a new age has been set by the age(newAge) operation by an object with the object interface type Person. The new age is returned as an attribute of the notification.

### 11.5.2 Notification Declaration

The syntax for `<notification_declaration>` is:

```
<notification_declaration> ::= 
    NOTIFICATION <notificationID> 
    [ <super_notifications> | 
    { '' <not_attributes> '' } ]
```

Notification declarations permit the declaration of struct-like data structures which may be returned to indicate that an event has occurred.

The syntax for `<super_notifications>` is:

```
<super_notifications> ::= 
    { '' <notificationID> [ "" <notificationID> ] }*
```

A notification that is specialized inherits all attributes from all its supertypes. If a name becomes ambiguous because of overloading, ambiguities can be resolved by qualifying (prefixing) a name with the name of its base-notification name.

The syntax for notification attributes is:

```
<not_attributes> ::= 
    { <not_attribute> ',' }*

<not_attribute> ::= <simple_typeID> <attributeID>
```

The syntax for declaration of possible notifications from an operation is as follows:
11.5.3 OML - Notification Management

\[
\text{<operation\_decl>} \quad ::= \quad \text{<type\_spec>} \quad \text{<identifier>} \quad \text{<parameter\_decls>}
\quad \quad [\text{<raises\_expr>}] \quad [\text{<notifies\_expr>}] \quad [\text{<context\_expr>}]
\]

\[
\text{<notifies\_expr>} \quad ::= \quad \text{NOTIFIES} \quad \{(\text{<notificationID>} \quad \{\text{<notification\_id>} \quad ,\quad \text{<notification\_id>} \quad \})\}\quad \}
\]

Notifications are organized as a directed graph, through the use of parent notifications. Objects which have registered interest in a particular notification will also receive all notifications which have been derived from that.

The generic root node Notification defines a printString as a common part of each notification. Attribute declarations only allow simple types as attribute domains. Interfaces might be referenced through their OIDString.

11.5.3 OML - Notification Management

Figure 11.8 shows an example where a client object, cX, in Process\textsuperscript{13} X registers interest in notifications from object, p1, on the event \texttt{ChangedAge}. In another Process Y a client object cY register interest in the notification \texttt{Changed} from the same object. The two objects p1 are proxy-objects for the same p1, actually residing in the object space Ingress1.

The notification management is handled by a number of management objects. The interfaces of the following different management objects are described in more detail in Appendix B:

- LocalNotificationManager
- NotificationReceiver
- Notification
- GlobalNotificationManager
- LocalNotificationManagerPort

The LocalNotificationManager provides an interface where it is possible to register interest for certain notifications from specific objects. It is possible to register a call-back operation, which is the one which will be called as a result of a notification. The default result is a call of the notify-operation defined in the NotificationReceiver interface. The LocalNotificationManager interface might be extended with more functionality, such as requesting notifications from all objects with a certain interface.

In figure 11.8 cX and cY register interest for notifications \texttt{ChangedAge} and \texttt{Changed} respectively, both from object p1. A register-operation results in an update of the local-interest knowledge and in a register-operation sent to the local proxy-object GNMPProxy, which represents the GlobalNotificationManager. The LocalNotificationManager will transform objects to their OIDStrings, which will be forwarded to the GlobalNotificationManager, through the GNMPProxy.

A \texttt{ChangedAge} notification is generated from p1 in ProcessX after p1 has committed the operation age(30). This is done by sending a notify-message to the LocalNotificationManager, having knowledge about all locally registered interest in this process, and handling

\textsuperscript{13}A Process is here an operating system process
local notifications directly. The LocalNotificationManager will transform the notification to its string-representation by the writeAttToString-operation supported by all notifications, and forward the notification to the GlobalNotificationManager, through the GNMPProxy. The encoding and decoding of information into a generic string format is so that each supernotification might read the parts it knows about, and skip the rest.

The GNM takes care of notification-management across different processes. The GNM does not deal directly with interface objects as arguments, but instead with OIDStrings which represent the objects. When it receives a notification, it finds out where notifications should be sent. It uses knowledge from the object dictionary to search the directed graph of notification-definitions to find parent interfaces. For optimization this might be done once for each notification-type, and a list of parents might be maintained instead. The GNM does not need to interpret the contents of the notification-string, which is rather interpreted by the receiver knowing the notification-type.

In the example the GNM will find that the object cX has registered interest for notification ChangedAge from object p1, but that this cX is in the same process as the p1 that has given out notification and therefore will be handled by the LocalNotificationManager. The object cY has registered interest for notification Changed, which is a parent to ChangedAge. A notify-message will be sent to port2, with the fromObject p1String and a string-representation for the ChangedAge notification.

The LocalNotificationManager will convert from OIDStrings to local object-pointers. The Changed-notification registered by the cY object will be able to decode the string representation of ChangedAge, to retrieve the attributes of relevance. Port2 acts as a mailbox for the GNMPProxy, either managed by a separate process, or a thread within ProcessY, or
monitored through an event-loop. It is possible to embed the event-loop of the notification-management in the X-event loop used for X-based user-interfaces. This has been done for the ESF Event Multi Caster service [Eur91b].

If the underlying integrated systems will be used from applications outside of the COOP framework, notifications should be generated directly from the underlying system. If not, it would be possible for example to change a person’s age from outside, without the generation of any notification-events. A rule-mechanism in a relational database system might be used to generate notification-messages, when certain events occurs. This requires the knowledge of how to generate the appropriate OIDString and notification string-representation in the underlying system.

The GlobalNotificationManager might be accessed from any system, through its port-interface. The architecture then becomes quite similar to the notification-mechanism of HP-Softbench/Field described in section 4.4 and ESF Event Multi Caster described in section 4.7.

These kind of events are similar to the notification/broadcast-events in tool-integration systems like Field/HP-Softbench and Forest. Request for notification is dynamic, as any object might register interest in an event. The potential generation of events is, however, statically defined in the schema. Note that this is somewhat different from the trigger/event-concept in active database-systems[DS86]. In COOM the main aim is to facilitate integration based on event-notification.

11.6 Exceptions

If an operation execution results in an error, an exception will be raised. An operation specification includes a possible sequence of exceptions that might be raised. How the exception is raised depends on the actual language mapping. Both C++ 3.0 and Smalltalk provides mechanisms for exception handling.

11.6.1 Exception Definition and Example

**Definition 11.9 Exception:**
An Exception $e$ is defined as the tuple $e = (en, PN, ATT)$ where

- $en \in EXN$ is a name for this exception
- $PN$ is a sequence of parent-notifications, from which $ATT$-definitions are inherited.
- $ATT$ is a set of (name,type)-pairs where type $\in S$, simple types.

The definition of an exception is conceptually similar to the definition for notifications, but an exception is managed in a different way than a notification.

**Example 11.8 Exception:**

```cpp
Exception IllegalAge {
    int age;
}```
};

Exception IllegalEmpAge: IllegalAge {
    int lower;
    int upper;
};

OBJINTERFACE Person {
    ......
    void age(int [in] newAge)
        raises(IllegalAge) notifies(ChangedAge);
    ......
};

OBJINTERFACE Employee {
    ......
    void age(int [in] newAge) raises(IllegalEmpAge) notifies(ChangedAge);
    ......
};

Example 11.8 shows use of the exceptions, IllegalAge and IllegalEmpAge, which might be raised if an illegal age is given to the age(int [in] newAge) operation of Person and Employee respectively.

The IllegalEmpAge adds two attributes for lower and upper age, to the age-attribute already specified for IllegalAge. In addition all exceptions have a printString, which might be used to print out information about the exception. The default content of the printString is the name of the exception.

11.6.2 Exception Declaration

The syntax for <exception_declaration> is:

<exception_declaration> ::= EXCEPTION <exceptionID> [super_exceptions] "{" "exc_attributes" "}"  

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exception has occurred.

The syntax for <super_exceptions> is:

<super_exceptions> ::= "":"" <exceptionID> ["","<exceptionID>]*

A specialized exception inherits all attributes from all its supertypes. If a name becomes ambiguous because of overloading, ambiguities is resolved by qualifying (prefixing) a name with the name of its base-exception name.

The syntax for <exc_attributes> is:
11.6.3 OML - Exception Management

Exception handling in C++ and Smalltalk is a mechanism working in the context of one executing process. In a systems integration situation an object might be a proxy object for a remote implementation. In such cases the exception-information is transformed to an error-structure which is handled as an extra parameter to an operation. It is then the responsibility of the proxy-object to raise the exception, based on information received as operation arguments from the underlying implementation.

Figure 11.9 shows an example of exception handling for the C++ mapping. A client-object cX gives a new age to a person and an employee-object, inside a try-clause. It will catch exceptions of type IllegalAge, and also then IllegalEmpAge, since IllegalEmpAge is a specialization of IllegalAge.
The p1 and e1 proxy-objects are in this case representing information in an Ingres database. When crossing process-boundaries the exception-handling needs to be broken down to the handling of normal sql-errors\(^\text{14}\). The implementation for these objects will generate in-process exceptions when necessary. This is here done through the \texttt{throw} statement of C++. The : prefix for some of the variables in the figure is for mapping from local variables to embedded SQL-variables.

A similar mechanism exists for the COOM to Smalltalk-mapping, as described in [Ost92a].

### 11.7 Context

A context might be associated with an operation, as an additional way of passing information. This is most useful for passing information about environment values, which the application programmer should not be concerned about. The context concept is adopted from OMG CORBA [Obj92a].

#### 11.7.1 Context Definition and Example

**Definition 11.10 Context:**

A Context \(c\) is defined as a list of pairs \(c_i = (n_i, s_v)\) where:

- \(n_i\) is a context.name \(\in\) CIN
- \(s_v\) is a String-value
- for each \(c_i\) the environment will associate a string-value \(s_v\) with \(n_i\) and deliver it as an extra argument to the operation.

Contexts provide a mechanism for associating values with arguments, without requiring that the application-programmer provides the values. They are typically retrieved from a separate file, associated with the current process, although other techniques might also be used. This gives a facility for environment-specific parameterization of values. This mechanism is based on the context-mechanism from OMG Object Request Broker [Obj92a].

**Example 11.9 Context:**

```
Exception IllegalCtxValue {
    String ctx_item_name;
    String ctx_item_value
};

OBJINTERFACE Person {
    META Person createCTX(String name)
        raises(IllegalCtxValue) context('Person');
};
```

\(^{14}\)Newer operating-systems like Windows NT supports exception-handling external to processes.
Example 11.9 shows a create-operation with a context with the context_name Person. This will create a stub-operation which retrieves an environment-value associated with Person. In this case the value could be the name of the ObjectSpace to be used for the creation-operation.

11.7.2 Context Declaration

The syntax for `<context_declaration>` is:

```plaintext
<operation_decl> ::= <type_spec> <identifier> <parameter_decls> 
                   [ <raises_expr> ] [ <notifies_expr> ]
                   [ <context_expr> ]

<context_expr> ::= CONTEXT "'" <string_literal>
                   { "'", <string_literal> }* "'",
```

11.7.3 OML - Context Management

The result of an ODL-specification of an operation with a context, is the creation of a new operation with a parameter-list extended with string-values for the context_names. A stub-operation is generated for the original signature. This stub-operation will retrieve the appropriate values from the environment and insert these as arguments to the new operation.

Figure 11.10 shows an example of context management, where invocation of the same operation, metaPerson->create, results in different environment-values for different processes.
11.8 COOP Architecture

11.8.1 COOP Logical Architecture

The COOP architecture provides run-time support for the concepts in COOM.

Figure 11.11: COOP Architecture

Figure 11.11 shows the different parts of the COOP logical architecture. The logical architecture might be realized through different physical architectures, as will be discussed in the next section.

The main component within COOP is the distributed persistent object space, which is partitioned into different parts. A description of the different parts of the architecture is given in the following:

- **Application/Tool**
  An application (or tool) is developed as a user-interface part interacting with a functional model-part. The functional model is realized by utilizing objects and relations provided through the Object Dictionary.

- **Object Dictionary**
  The Object Dictionary contains object interface managers and relation interface
managers used to access the available objects. It also contains information about
the definition and representation of interfaces and implementations. This information
is generated from the parsing of ODL-specifications.

- **Interactor Mechanism**
The interactor mechanism provides support for communication between objects.

- **Local Object Space(s)**
The local object spaces contain the implementation managers and their managed
objects, with the actual implementation for the mapping for the integrated compo-
ments. There is typically one object space for each integrated component with local
authority and name-space:

  - **RDBMS**
    A relational database will contain tuples in tables, that are partly mapped to
    objects and partly to relations. Mappings for relational DBMSs have been
described in [Low92].

  - **OODBMS**
    An OODBMS contains related objects that are mapped to COOM-objects and
    relations. Mappings for object-oriented DBMSs have been described in [Fin92].

  - **ORB**
    An ORB-environment will contain objects that will be mapped to COOM-
    objects. Mappings for ORB-technology have been described in [Øst92a].

  - **DCE**
    An OSF/DCE-environment will contain functional services with interfaces that
    can be mapped to COOM-objects. Mappings for OSF/DCE have been
described in [FST*90, Øst92b].

Mappings for different components are further discussed in chapter 14.

- **Integration Space**
The Integration Space is a storage space in particular for relationships between ob-
jects in different object spaces, and for enrichment-objects.

- **Model Space**
The model space can be used by applications for storage of objects related to the
functional model-part of an application.

- **OQL Query Processor**
The Query Processor is a component that is used during runtime to process OQL-
queries. The goal of the query processor is to do query optimization and to split and
distribute queries and to merge answers. This is discussed further in section 13.

- **Notification Manager**
The Notification Manager is a component that is used for the management of notification-
events, as discussed in section 11.5.

- **Transaction Manager**
A Transaction Manager is a component which e.g. provide two-phase commit and
other transaction mechanisms. This was discussed further in section 11.3.2.

- **Security Manager**
A Security Manager can be realized to give support for authorization and authenti-
cation. Security Management has not been addressed further in this work.
• Exception Manager
An Exception Manager will be notified about all exceptions, and can be used to maintain a log about such events. This was discussed in section 11.6.

11.8.2 COOP Physical Architectures

Figure 11.12: Different Physical COOP Architectures

Figure 11.12 shows three different possible COOP physical architectures. They are different with respect to in which process object-operations are executed, and how the different objects interact with each other.

• Distributed Client-oriented OODBMS
  This architecture will cache objects from distributed servers and databases into the client-process, where execution for both the object-mapping-code and the manager objects take place. The interactor-mechanism is provided by the object-interaction support given by the OODBMS. The characteristics of client-oriented OODBMSs was discussed in section 7.2.6. The C++-based COOP prototype described in Appendix D has been using the ObjectStore client-oriented OODBMS.

• Distributed Server-oriented OODBMS
  This architecture will use local proxy-objects for accessing objects in separate server-processes, where operation-execution will take place. Different object spaces might reside in the same, or in different servers. The interactor-mechanism is provided by the object-interaction support given by the OODBMS. The characteristics of server-oriented OODBMSs was discussed in section 7.2.6.

• Object Request Broker with component-adaptors
  This architecture will need to be extended with component-adaptors for the different components to be integrated. This can either be done through the Basic Object Adaptor, or through specialized adaptors, e.g. for relational and object-oriented
DBMSs. The Object Dictionary will be integrated with the ORB interface and implementation repository. The interactor-mechanism is provided by the ORB. The characteristics of the OMG Object Request Broker technology was discussed in section 4.3.3.

All three physical architectures can be used to support the COOP Integration Framework. The distributed client-oriented OODBMS-approach has been experimented with in the COOP prototype, in particular because this kind of OODBMSs was early available. Recently, both server-oriented OODBMSs and Object Request Broker products have been put to the market, and it will be interesting to explore the COOP approach for these architectures in the future.

11.9 Contribution and Conclusion

This chapter presented the basic concepts for $COOM_B$. $COOM_B$ provides support for both request-oriented and notification-oriented control-integration.

The basic functionality for objects is adopted from functionality of objects in OMG CORBA and the REBOOT data models. The use of exceptions and context is adopted from OMG CORBA. The use of inheritance for exceptions is new in this context, but the inheritance-structure is adopted from the exception-scheme used in C++ [Str91].

The use of an explicit level of interface and implementation manager objects for extent management and object creation is new. Although this is similar to the use of class-objects in Smalltalk, it has not been used for the explicit management of interfaces and implementations. The use of notifications as part of the definitions in a schema is novel. Other work, such as HP Softbench and Field, realize notifications around a string-type which is interpreted at run-time. The meaning of notifications needs for these systems to be agreed upon outside of the support system, while in COOM this is supported through ODL.

The COOP architecture is realized as a merge of concepts from distributed system architectures like the ESF Software Bus and the OMG CORBA architecture, and object-oriented database architectures. The use of explicit object spaces for the mapping from interface to implementation is new.

An example of the use of $COOM_B$ for control integration is given in section 15.6, where it is shown how the interface of a project management system reachable through a remote procedure call mechanism can be integrated into COOP. How the constructs of $COOM_B$ give a basic support for the requirements of request-oriented and notification-oriented control integration is evaluated in section 16.3.1.

The next chapter presents $COOM_S$, which extends $COOM_B$ with support for data integration.
Chapter 12

COOM Structural Model and Data Integration

This chapter presents extensions to the COOM-model for abstract attributes and relations, as a foundation for the support of data integration. This is also used as a basis for the definition of the Object Query Language, OQL, presented in chapter 13. Abstract Attributes, Relations and OQL provide support for single-model data integration. The last part of the chapter discusses COOM with respect to multi-model data integration.

Chapter 2 in part I stated a set of requirements for a data-model, which included the requirements for the support of composite objects and relations and attributes, similar to an ERA-model. Chapter 7 in part II described experiences with using an object-oriented approach to databases, including the definition of the SOOM model [Ber88c], and experiences with object-oriented databases. The SOOM model included the use of relations as an added construct to a behaviour-oriented object-oriented model. The COOM structural model aims at meeting the requirements for data integration stated in chapter 2, using structural concepts from SOOM and the REBOOT datamodel [OH91] as an extension to the COOM behavioural model. The HyperModel benchmark in chapter 7 has showed that commercial object-oriented database systems can provide the necessary underlying functionality and performance.

12.1 The COOMS structural model

<table>
<thead>
<tr>
<th>COOM</th>
<th>S</th>
<th>Abstract Attributes</th>
<th>Structural model for Data Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Relation-Interface</td>
<td>Behavioral model for Control Integration</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Relation management</td>
<td>Foundation model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| COOM | F | Interface, Basic Type Implementation Operations |

Figure 12.1: COOM for Data Integration
This chapter will define $COOM_S$. The structural attribute- and relation-interface layer $COOM_S$ enhances $COOM_F$ and $COOM_B$ with the notions of abstract attributes and relations to support data integration.

In some kinds of applications it is natural to use the concepts of entities, relations and abstract attributes as basic modeling constructs. The group of applications this is valid for is typical database-oriented applications, where the goal is to update and retrieve information from the model itself. The focus is on the structure of the model, not on the behaviour and interaction between the various objects.

However, a support for structural modeling should not violate the encapsulation principles of a behaviour-oriented model. An important goal is to still keep the possibility for having different implementations of an interface at the same time. This is useful if a relation or an attribute is represented differently in different underlying systems.

The ERA-model concepts are adapted from the EPOSDB datamodel [LCE+89] and the REBOOT datamodel [OH91] which are structurally object-oriented models. The REBOOT datamodel inherits many concepts from the FOOD datamodel [OB90, Bra91]. The contribution in COOM is how this is realized as an extension to a behaviour-oriented model with a separation between interface and implementation of objects. In particular a new contribution is a framework for management of multiple parallel implementations of a relation.

In COOM this is realized as the possibility to describe abstract attributes and relationships, which will be automatically transformed to a corresponding set of operations/access-functions. The separation between interface and implementation emphasizes that we will not know how an abstract attribute is implemented. I.e. an age might be stored directly or it might be calculated by a birth-date and the current-date. Each abstract attribute implies two access functions with the same name but with different signatures, $attType att()$ and $void att(attType newatt)$.

The structural COOM-model, $COOM_S$, is a structurally object-oriented model defined as follows:

**Definition 12.1 $COOM_S$:**

$COOM_S$ is an n-tuple, $COOM_S = < COOM_B, AN, RIN, CRN, RIM >$ where:

- $COOM_B$ is defined in definition 11.1.
- $AN$ is a finite set of Abstract Attribute Names.
- $RIN$ is a finite set of Relation Interface Names, where $RIN \subseteq \text{IN}^1$.
- $CRN$ is a finite set of ContainmentRelation Interface Names, where $CRN \subseteq RIN$.
- $RIM$ is a finite set of Relation Implementation Names, where $RIM \subseteq \text{IM}$.

□

### 12.2 Abstract Attributes

#### 12.2.1 Motivation for Abstract Attribute Support

In the systems-integration situation that COOM aims at, possible attributes will be stored in the underlying systems, so there is no need for specifying any representation for at-
tributes. Indeed, it is a goal to not specify any representation, so that there is as much flexibility as possible for the mapping to underlying systems. What is needed is to be able to specify that objects conceptually do have attributes. This is provided through the specification of abstract attributes. Relations are used for the abstraction of composite objects and objects that relates to other objects.

The REBOOT datamodel also provides support for attributes and relations, but do not emphasize the need for encapsulation. As stated in the Scandinavian "modeling-oriented" definition of object-oriented programming, in definition 5.1: "... each object is characterized by attributes and a sequence of actions .." Languages like Simula and C++ also support the possibility for object-attributes, but do this normally by breaking the encapsulation of an object and giving direct access to data-members.

In COM, certain pairs of operations such as int age() and void age(int newage) in example 10.2, might be considered as an interface to an abstract attribute of an object. A client of this object will then not know if age is implemented as an instance-variable (data-member), or if it is calculated at runtime. The specification of an abstract attribute, attType att, is then a short-hand for the specification of two operations (get and put) provided by an object: attType att() to get the value and attType att1(attType newValue) to put a new value. These two operations conform to the semantics of attributes, e.g. the last value set by a conceptual put-operation is the one to be retrieved by the next get-operation. It is important to note that this can be a conceptual operation. An underlying implementation might choose to represent a person's age with the birthdate and calculate the age based on the current date. A person's age will then change on his birthday without any explicit put-operation taking place. This can be viewed as if the system always might do a put-operation before any get-operation. If an event is to be generated when an attribute changes, it is up to the implementation to generate this, in this case for instance with a time-trigger.

It is possible to specify that an abstract attribute is readonly, thereby creating only the get operation, and to specify that one abstract attribute, or a combination of such, is unique.

The domains for abstract attribute-types are basic data-types as described in section 10.1.1. Both simple and constructed types might be used. Relations should be used if there is a need for referring to objects2.

It is possible to create user-defined types based on basic data-types, so that user-defined abstract attribute-domains can be specified. It is strongly encouraged to use abstract attribute-domains, e.g. to specify that the abstract attribute height of a person is given in centimeters and not in inches, define the attribute domain heightcentimeter as integer.

12.2.2 Abstract Attribute Definition - Syntax

Definition 12.2 Abstract attribute:
An abstract attribute is defined as an n-tuple \( att = (aname, t, O, E, N) \) where:

- \( aname \in AN \) is the attribute-name.
- \( t \in B \) is the type of the attribute.
- \( O \subseteq OP \), is a pair of operations \( (t \ aname(), \ void \ aname(t [in] newan)) \) so that the following axiom is valid for an object \( o \) with attribute-name \( aname: o->aname(x); y := o->aname(); \implies y == x. \)

2See section 12.3 about relations
• $N$ is a sequence of the form $(n_1, \ldots, n_j)$ where $n_i \in NN^3$.

• $E$ is a sequence of the form $(e_1, \ldots, e_j)$ where $e_i \in EN$, being the set of exceptions which might be raised from a put or get operation. (See description in section 11.6).

Abstract attributes are defined as part of an object interface, and the syntax for object interface is enhanced with the possibility for the specification of abstract attributes, see section 11.2. An abstract attribute specification might be prefixed with “META” to specify an abstract attribute of the interface manager.

The syntax for abstract attributes is:

```plaintext
<attribute> ::= [READONLY] <basic_typeID> <attributeID> [<array_bounds>] [raises_aexpr] [notifies_aexpr]

<raises_aexpr> ::= RAISES "('' <exceptionID> 
{ '' , '' <exceptionID> }* '')

'IN' '(' ''get'' | ''put'' | ''get,put'' '')')

<notifies_aexpr> ::= NOTIFIES "('' <notificationID> 
{ '' , '' <notificationID> }* '')

'IN' '(' ''get'' | ''put'' | ''get,put'' '')')

<array_bounds> ::= ''['', ''']'',
| ''['', '*' '', ''']'',
| ''['', <const_exp> ''']'',
| ''['', '*' '', '*' '', ''']'',
| ''['', '*' '', '*' '', <const_exp> ''']'',
| ''['', <const_exp> '', ''']'',
| ''['', <const_exp> '', '*' '', ''']'',
| ''['', <const_exp> '', '*' '', ''']''
```

As for operation-arguments, an array-specification might be attached to the declarator. Arrays are discussed in section 10.1.1 and appendix A.

12.2.3 Example and OML for Abstract attributes

Example 12.1 Abstract attribute:

```plaintext
OBJINTERFACE Person
{
    String name;
    int age RAISES(IllegalAge) IN (put) NOTIFIES(NewAge) IN (put);
    readonly int personNumber;
    UNIQUE(personNumber);

    ENUM Hobby {tennis, soccer, golf, other};
```
12.2.4 Unique Attribute-combinations

```java
SET<Hobby> hobbies;

META Person create(String [in] name, int [in] pNr, int [in] age);
META readonly int nrOfPersons; /* Meta attribute */
}; /* ObjInterface Person */
```

Example 12.1 shows some examples of the definition of the person abstract attributes name, age, personNumber and hobbies.

The use of notifications and exceptions can be stated for an abstract attribute, with an IN clause stating if it will apply for the get- or put-operation, or both.

The age attribute of the Person object-interface of example 12.1 is represented in ODL as:

```java
int age Raises(IllegalAge) IN (put) Notifies(ChangedAge) IN (put);
```

which is equivalent to the following two ODL-specifications:

```java
int age();
void age(int newAge) Raises(IllegalAge) Notifies(ChangedAge);
```

which results in the following two operations in a C++ mapping:

```cpp
virtual int age();
virtual void age(int newAge) throws(IllegalAge);
```

It is possible to use the defined abstract attribute-names in the specification of create-operations. This is a way to give initial values to the attributes.

12.2.4 Unique Attribute-combinations

**Definition 12.3 Unique Attribute Combination:**

A Unique Attribute combination uac for an object interface oi is defined as a list of abstract attribute names of the form $a_1, \ldots, a_k$ where $a_i \in AN$, such that for each object of interface $oi$ each combination of values will exist for only one object at a time. □

Uniqueness is useful to state on the external interface-level, e.g. when creating an external index. The uniqueness property can be maintained by the object interface manager, in cooperation with the object implementation managers and underlying implementations.

The syntax for declaring a unique attribute-combination for an object interface is:

```
<uniques> ::= UNIQUE "'(' <key> { '','<key>' })* '',''

<key> ::= <attributeID> { '','<attributeID>' }*
```

---

*Exceptions are handled by the new C++ 3.0 try and throw expressions.*
The <attribute_id> is a previously defined abstract attribute in the current object interface, or one of its parents. A key is some combination of one or more abstract attributes.

OBJINTERFACE Person2
{
    .......
    String firstName;
    String lastName;
    Date birthDate;
    int personNumber RAISES(IllegalArgumentException) IN(put);
    UNIQUE(personNumber; firstName, lastName, birthDate);
      .......
}; /* ObjInterface Person2 */

In this example personNumber is a unique readonly abstract attribute, while the combination of firstName, lastName and birthDate also is unique.

12.3 Relations

12.3.1 Definition and Motivation for Relation-support

Relations and relationships are abstract notions which are supported by the use of relation-interface manager objects and relation-implementation manager objects.

The relation-concepts are adopted from the REBOOT datamodel [OH91], but enhanced with the explicit separation of interface and implementation for relations, and the possibility to have multiple parallel implementations of one relation.

**Definition 12.4 Relation:**
A relation is a subset \( R \) of the Cartesian Product: \( \text{extent}(F) \times \text{extent}(T) \times A_1 \times \ldots \times A_n \) where

- \( F \) is the interface of the from-object in a relationship.
- \( T \) is the interface of the to-object in a relationship.
- The cardinality of a relation may be 1:1, 1:N, N:1, or M:N.
- Relationships might have attributes.
- \( A_1 \ldots A_n \) are the attribute-types (domains) of a relationship.

The extent of a relation is its set of relationships. \( \Box \)

**Definition 12.5 Relationship:**
A Relationship is a tuple \( t \) from a relation \( R \) such that: \( t \in \text{extent}(F) \times \text{extent}(T) \times A_1 \times \ldots \times A_n \). \( \Box \)

Relations and Relationships are in COOM supported through the specification of relation interfaces, which leads to the creation of relation interface and implementation managers.
A Relation-Interface describes an association between two object-interfaces, and is defined through the ODL-construct, RELINTERFACE. It implicitly defines a relation interface manager with operations for managing a set of relationships.

A relation-interface is specified separately, independently of the two involved object-interfaces. A relationship will ensure reference integrity by maintaining a reference between both objects involved in a relationship. If the relation-interface definition gives the names of roles in each direction (in addition to types), object-dependent access and storage functions will automatically be created, for the involved object interfaces.

The relation is the set of all relationships of the corresponding relation-interface. A relation can be managed through a special kind of manager-object which provides the service of defining relationships between objects. For each relation-interface there will be an object which provides the service of storing and retrieving the actual relationships. It is possible to define role names, and to create access functions in the involved object types. The provided operations for relationship management are: insert, remove, getFromObj, getToObj, getIterator, getIteratorFrom, getIteratorTo, getIteratorTo, getCardinality, changeCardinality and nrRelationships. These operations are shown in figure 12.3 and they are described in section 12.3.5.

There might exist multiple implementations of a relation at the same time, e.g. as a separate table, as pointers within each object, or in other ways. Sometimes it is convenient to look at a relation separately, while sometimes it is natural to start from one object, and through its role in the relation find associated objects. The generation-part of COOM supports both views. Separate relation management is always supported, while object access can be supported through the definition of role names. The role name will be used to generate access-functions for the involved object interfaces.

RELINTERFACE definitions leads to the creation of Relation Interface and Implementation Managers. The same structure as between an object interface and object implementation managers, exists between a relation interface and relation implementation managers. Each object space representing a relation will have a relation implementation manager. Relation Interface and Implementation Managers are similar to Object Interface and Implementation Managers, with the exception that the relationships managed by Relation Managers are not objects. A relation interface manager is then an object that manages all relationships through delegation to relation implementation manager objects for the different object spaces.

Part-of structures might be specified through Containment relations. Containment relation is defined in section 12.3.7. A subset of a relation might be specified through sub-relations. Sub-relation is defined in section 12.3.8.

The provided operations for the management of relation implementations and sub-relations are: addImplementation, removeImplementation, implementations, addSubRelation, removeSubRelation and subRelations. These operations are shown in figure 12.3, and they are described in section 12.3.9.

A relation implementation manager will have freedom to select how to represent the relationships. This can be done through a centralized representation or by distributing the representation to the involved objects. The access- and storage operations will, however, look the same in all different implementation-cases.

12.3.2 Relation Example

Figure 12.2 shows three different relations, a 1:N AssociatedWith between Department and
Figure 12.2: COORASS Person-Department example

Person, a 1:N WorksFor relation between Department and Employee with relationship-attribute startDate, and a 1:N containment-relation, TheDepts, between Company and Department.

The ODL-schema for the new parts of the diagram in figure 12.2, is shown in example 12.2. The COORASS diagram notation is described in section 14.1.4

Example 12.2 Relation:

```
Schema PersonDept {
import PersonSchema;
import Basics;

OBJINTERFACE Department {
    Location loc;
    String name;
};

RELINTERFACE AssociatedWith {
    Person #N associates;
    Department #1 associated;
};

RELINTERFACE WorksFor < AssociatedWith {
    Employee #N employees;
    Department #1 works_in;
    Date startDate;
};

OBJINTERFACE Company {
    CONTRELINTERFACE TheDepts {
        #1;
        Department #N;
    };
    Location loc;
```
String name;
}; /* Company ObjInterface */

}; /* EndSchema */

### 12.3.3 Relation Interface Syntax specification

A Relation Interface is defined in ODL by use of the following syntax:

```
RELINTERFACE <relInterfaceID> [ "'" <relInterfaceID> ]
   "'" [roles] [rel_attributes] "'"
```

<relInterfaceID> is the name of the relation. It is possible to have attributes associated with a relation, as described in section 12.2.

```
<rel_attributes> ::=  
   { <rel_attribute> "';'" }+  
   <rel_attribute> ::= <simple_typeID> <attributeID>
```

The attributes together with the two related objects are the key for finding a relationship, and must therefore be unique within a relation. The syntax for roles is:

```
<roles> ::=  
   <objInterfaceID> "'#'" <cardinality> [roleName] "' ';'"  
   <objInterfaceID> "'#'" <cardinality> [roleName] "' ';'"
```

<cardinality> ::= "'i'" | "'N'"

The first line specifies the object interface the relation goes from, and the second line specifies the object interface the relation goes to. <objInterfaceID> is a name of a previously declared object interface. <cardinality> is either '1' or 'N' representing one or many, respectively.

### 12.3.4 Relation Interface and Implementation Manager

**Definition 12.6 Relation Interface:**
A Relation Interface is defined as a tuple

\[
ri = (i, s, (from, fromca, fromrole), (to, toca, torole), A, R<from, to, A>, RName) \]

where

- \( i \) is an interface \( i = (rn, P, E, N, O, AO) \) as defined in 10.6, where \( rn \in IN \) is the name of the relation interface, \( P \) is empty and \( AO \equiv O \). \( AO \) is the operations defined for this relation-interface, with possible use of new exceptions \( E \) and new notifications \( N \). \( P \) is empty and \( AO \equiv O \) because there is a different inheritance (sub/super-relation) semantics for relations than for interfaces.

- \( s \) is an optional name for a RelationInterface which \( ri \) is a sub-relation of.

---

5The semantics of relation-inheritance (sub/super-relations) is described in section 12.3.8.
• from is the name of the Object Interface of the from-object, fromca is the cardinality of the from-object as seen from the to-object and fromrole is a possible role name for the from-object(s) as seen from the to-object.

• to is the name of the Object Interface of the to-object, toca is the cardinality of the to-object as seen from the from-object and torole is a possible role name for the to-object(s) as seen from the from-object.

• A is a set of relationship attribute-definitions. These are pairs \( a = (n, \tau) \), where \( n \in AN \) is the attribute-name and \( \tau \in B \) is the attribute-type (domain). The attributes are bundled together in a struct with the name of the relation interface, rn, prefixed with A. The COOM-support for inheritance in struct-definitions, allows for a specialization of this set in sub-relations.

• If ri has the interface name \( rn = RName \), then a relation interface manager object with the name RName is automatically created.

• Role name access-functions might be generated for the involved from- and to-object interfaces.

\( \Box \)

Definition 12.7 Relation Interface Manager
A Relation Interface Manager is an object created through an ODL RELINTERFACE definition rd. If rd has the interface name \( rn = RName \), then a relation interface manager object with the name RName is automatically created. This manager object will have an interface and implementation defined by the interface and implementation \( R < from, to, A > \). There is no separation between interface and implementation for manager objects. Cardinality for RName is automatically set to the appropriate initial value, 1:1, 1:N, N:1 or M:N. The purpose of the relation interface manager RName is to manage a set of corresponding relation implementation managers. The manager objects have an OID, and there is a one-to-one mapping between the OID and the name RName. \( \Box \)

A relation interface manager has an interface for management of relationships which is realized through delegation of operations to relation implementation managers for the involved object spaces. The relation interface manager is the object that provides the operations defined for the relation interface.

Definition 12.8 Relation Implementation:
A Relation Implementation rim is defined as the tuple
\( rim = (im, OS, (s), (from, fromca, fromrole), (to, toca, torole), A, RI<from, to, A >, OP, RName, U) \) where:

• im is an implementation \( im = (imm, inn, osn, OB, f, \Omega) \) as defined in 10.11.

• If im has the implementation name \( imm = X \in IMN \), then a relation implementation manager object with the name RLX will be created. This relation implementation manager object will have an interface and implementation defined by the interface and implementation RLX, which is derived from the parameterized interface RI<T >.

Note that \( U \) is a set of relationships, while \( \Omega \) is a set of objects.
It is not strictly necessary for the relation implementation manager objects to have names, as the application programmer never will refer to these. It is, however, useful to have a name for reference to it.
Definition 12.9 Relation Implementation Manager:
A Relation Implementation Manager is an object that manages one particular representation/implementation of relationships for a relation RName. A relation implementation manager is related to one object space, and will typically relate objects from this. Relation implementation managers in the integration space will relate objects from different object spaces. This manager object will have an interface and implementation defined by the interface and implementation RI\textit{inn}< from, to, A >, a particular implementation of a relation. There is no separation between interface and implementation for manager objects. The manager objects have an OID, and there is a one-to-one mapping between the OID and the name RName. □

Definition 12.10 Integration Space:
The Integration space is a specific object space with the purpose of maintaining relationships between objects in different object spaces. It contains its own relation implementation for each RELINTERFACE, for this purpose. □

Figure 12.3 shows how the relation interfaces AssocWith and WorksFor in the ODL-schema from example 12.2 are represented as two instantiations of the parameterized relation interface R(F, T, A). R is parameterized with the from- and to-types and the attribute-structure. The figure shows a list of the general relation-operations described in section 12.3.5 and the management-operations described in section 12.3.9.
The relation interfaces are the type for relation interface managers, which will be automatically created. The relation interface managers AssocWith and WorksFor will be created as instances of the interfaces R<Employee, Department, AAssocWith> and R<Employee, Department, AWorksFor> respectively.

Figure 12.4: Type-hierarchy for Relation Interfaces and Relation Implementations

Figure 12.4 shows the type-hierarchy for relation interfaces and corresponding relation implementations. For each relation interface, there will exist one or more relation implementations.

Figure 12.4 shows three relation implementations for each relation interface. The implementations are based on the three parameterized implementations for the Ingres RDBMS (RIIngres), the ObjectStore OODBMS (RIOS), and the special integration space (RIISpace).

These implementation-types are used as the basis for the creation of relation implementation manager objects. The automatic creation of the manager objects will be provided with information for correct initialization of cardinality, and necessary information for the mapping to the underlying system. The cardinality might change dynamically, and is therefore not a parameter for the interface-type. The relation implementation managers conform with the type RI(F, T, A) which is the type used by the corresponding R(F, T, A) relation interface manager. Note that each attribute-structure will be named uniquely for each relation interface, so there will be one unique parameterization of RI(F, T, A) for each relation interface.

A relation implementation might be derived from a generic implementation, or it might be implemented totally from scratch. There might exist different alternative generic implementations for the same system.

Figure 12.5 shows the object-hierarchy for the relation interface manager objects and the relation implementation manager objects which are generated from the schema in
Figure 12.5: Relation interface managers and Relation implementation managers

example 12.2. The figure also shows the involved object interface managers and object implementation managers.

There are two relation interface managers in the Object Dictionary object space, one for AssocWith and one for WorksFor. Each relation interface manager is here associated with three relation implementation managers, one for each of the object spaces Ingres1, ObjStore1 and IntegrationSpace.

When an insert-relationship operation is invoked on a relation interface manager, it will delegate the operation further to a relation implementation manager dependent on which object spaces the related objects exists in. If the two related objects are in the same object space, a relation implementation manager for this object space will be used if it exists. The relation implementation manager in the integration-space will be used if no other alternative is available. Inter-Object Space relationships will then always be managed by the relation implementation manager in the integration-space.

12.3.5 OML - Relationship Operations

- Insert and Remove
- Get related objects
- Iterate

There are no update-operations, since a different value to either objects or attributes logically leads to a new relationships. An update is done by a deletion of the old relationship,
and a creation of a new one. The following operations are defined:

- Insert and remove relationships

  The insert- and remove -operations provide functionality for management of relationships. The application programmer will invoke these operations for the appropriate relation interface manager. The relation interface manager will interact with its corresponding relation implementation managers.

  The operation insert(From [in] from, To [in] to, Att [in] a) inserts a new relationship. It checks whether the object space of the from- and to- objects are the same. If a relation implementation manager exist for such a shared object space, the relationship is inserted here. If they are in different object spaces, or no relation implementation manager exists in the object space, the relationship is handled by the corresponding relation implementation manager for the integration-space. The exception "CardinalityViolation" will be raised if the cardinality-constraint is violated.

  The insert of the same relationship several times will not lead to "CardinalityViolation".

  The operation remove(From [in] from, To [in] to, Att [in] a) deletes the relationship. It checks whether the object space of the from- and to- objects are the same. If a relation implementation manager exists for such a shared object space, the relationship is removed from here. If they are in different object spaces, or no representation-object exists, the relationship is removed from the relation implementation manager for the integration-space. The exception "NotExist" will be raised if the relationship, or any of the objects, does not exist.

Example 12.3 Update of a relationship:

```c
/* Relationships cannot be directly updated, they need
to be deleted and a new one inserted */

Employee* e1;
Department* d1;
AWorksFor aw;
...... /* Given values for e1, d1 */
try {
    WorksFor->remove(e1, d1, aw);
    aw.duration = aw.duration + 40;

    WorksFor->insert (e1, d1, aw)
};
catch(NotExist ne) {
    ....};
```

- Get related objects

  The get-operations provide functionality for finding related objects. The following operations are defined:

  The operation Set<To> getFromObj(From [in] f) retrieves the set of objects which are related to the From-object from. The operation is delegated to all relation implementation managers and all sub-relation interface managers.

  The operation Set<From> getToObj(To [in] t) is analogous to the getFromObj operation, only in the opposite direction.
Example 12.4 Get related objects:

```c++
/* Get employees in department d1 */
Employee* e1;
Department* d1;
&WorksFor aw;
...... /* Given values for e1, d1 */
try {
    Set<Employee*> emps = WorksFor->getToObj(d1);
    /* or directly through role-name */
    Set<Employee**> emps2 = d1->employees();

    DO (emps, Employee*, e) /* Iterates through the result */
        cout << e->name;
ENDDO
}
catch(NotExist ne) {...};
```

The example above shows the retrieval of employees in department d1. This is done either by accessing the related objects through the `getToObj` operation of the relation interface object, `WorksFor`, or by direct access through the role-name access-operation, `employees()`. An exception is raised if there are no relationships for this object. A `DO-ENDDO` macro allows for an iteration through the returned set of Employees.

- Iterate relationships - `RelationIterator`

A `RelationIterator` is an object which iterates through a subset of a relation. A `RelationIterator`-type is implicitly defined for each relation interface type. Each relation implementation manager will also have a corresponding relation iterator.

A Relation Iterator will be an instance of the generic interface `RIT<F, T, A>`, where F and T are the object-interfaces of the `from`- and `to`-objects, and A is the attribute-structure of the relation.

![Figure 12.6: Relation Iterator - Interface and Implementations](image)

Figure 12.6 shows the interface a relation iterator, with three different implementations corresponding to the relation implementations described earlier.

**Definition 12.11 Relation Iterator operations**

A Relation Iterator will provide the following operations:
INTERFACE RIT (OBJINTERFACE From, OBJINTERFACE To, BASE RAtt) {
    Boolean atEnd();
    first();
    next();
    From from();
    To to();
    RAtt att();
}; /* End RIT */

The operation first() sets the internal iterator-cursor to the first relationship in the set. At that point the operations from(), to() and att() can be used to retrieve the respective parts of the relationship.

Example 12.5 Relation Iterator

    /* Iterator-example: Print out name and startDate for
     Employees working in department d1 */
    try {
        RIT<Employee, Department, AWorksFor> it =
            WorksFor->getIteratorTo(d1);
        
        while (! it->atEnd() )
        {
            cout << it->from->name() << it->att.startDate;
            it->next();
        }
    }

The operation RIT<From, To, Att> getIterator() returns an iterator for all relationships in the relation.

The operation RIT<From, To, Att> getIteratorFromTo(From [in] fromObject, To [in] toObject) returns an iterator for all relationships between the From and the To object.

The operation RIT<From, To, Att> getIteratorFrom(From [in] fromObject) returns an iterator for the relationships going from the From object.

The operation RIT<From, To, Att> getIteratorTo (To [in] toObject) returns an iterator for the relationships going to the To object.

A getIteratorXY operation invoked on a relation interface manager will delegate the operation to all its relation implementation managers and all subrelation interface managers. Each relation implementation manager will return a relation iterator of the correct type. The sub-relations, however, would return a relation iterator that might have specialized from and to object-interfaces and attribute-structure. Since these cannot be known in advance, there is a special getIteratorOl-operation that returns from and to typed as the root objInt and att typed as the root Att. This can be used to do a safe type-cast to the correct interfaces and attribute-structure.
12.3.6 Relation Interface Functionality

A relation interface manager could reflect cardinality and attributes in operation interfaces, or use a generic interface for all cardinalities and attributes. In a 1:1 relation the result of following a from-relationship will always be one object, while a 1:N relation will result in a set of objects. To reflect cardinality and attributes in operation-interfaces will give a more comprehensive specification, with less possibilities for errors, but will be much less resistant to changes and evolution. If a 1:1 relation is changed to a 1:N relation, or attributes are added, all code using the old operation-interfaces then need to be rewritten. The alternative is to have a generic operation-interface which uses exception-handling for error-situations. Because of the importance of support evolution, the generic operation-interface approach is chosen for COOM.

It is often useful to access relationships directly from one of the involved objects. If role-names are given it is possible to automatically create access-functions for the involved objects.

All relation interface managers are defined to be an instance of the interface \( R<F, T, A> \), which is a parameterized interface for a generic M:N relation with attributes. \( F \) and \( T \) are the object-interfaces of the from- and to- objects respectively, while \( A \) is a structure for the attributes. If no attributes exists for this relation, the attribute-structure is empty. All attribute-structures inherit from the empty structure \( \text{Att}^8 \).

Example 12.6 Relation-operations

```plaintext
ENUM Cardinality (C11, C1N, CN1, CMN);
Struct Att { }

INTERFACE R (ObjInterface From, ObjInterface To, Base Att): ROI
{...
  changeCardinalityTo(Cardinality [in] card);
  insert(From [in] from, To [in] to, Att [in] a)
    Raises(CardinalityViolation) Notifies(Inserted);
  SetTo> getFromObj(From [in] from);
  ...
};

Struct AWoksFor: Att {
  Date startDate;
};
```

Example 12.6 shows some parts of the generic relation-interface, \( R \), and the definition of the example-relation WorksFor. The relation interface manager WorksFor will automatically be created, and initialized with cardinality \( CN1 \).

The attribute-structure \( AWoksFor \) contains the relationship attribute, \( \text{startDate} \). An insert-operation will receive from and to objects and an attribute-structure as arguments. If the cardinality is violated, the exception \( \text{CardinalityViolation} \) will be raised.

---

\(^8\)The use of a struct to represent attributes is due to a proposal by Per Solberg in [Sol92]. He has also proposed an extension of COOM to support N-ary relations, which is not discussed here.
The operations supported by the relation managers are described in more detail in section 12.3.9. The operations getFromObj will return a Set<Department> even if only one object should be expected. However, if a cardinality-change later allows a person to work for multiple departments, no source-code changes are necessary.

12.3.7 Containment Relations

Part-of structures might be specified through Containment relations. A Containment relation is a relation that has a special part-of semantics, useful for modeling complex objects. The containment relation concept is adopted from REBOOT [OH91].

Definition 12.12 Containment relation:
A containment-relation is a relation RC: extent(F) × extent(T) × A₁ × ... × Aₙ, as defined in definition 12.4 where

- F is the interface of the from-object in a relationship. A containment-relation definition, CONTRELINTERFACE, is given as part of an object interface definition, OBJINTERFACE. F is implicitly defined as the interface where C is defined.
- T is the interface of the to-object in a relationship.
- Also M:N relations are supported, i.e. object sharing is allowed.
- A special semantic is defined for copy-operations for an object O with the object interface where RC is defined. A copy operation also makes a copy of all to-objects in the containment relation from O, and inserts these in the containment relation as related to the new copy of O. A deep-copy operation will then recursively copy the entire complex object.
- A special semantic is also defined for delete-operations for an object O with the object interface where RC is defined. A delete operation also deletes all to-objects in the containment relation from O, unless these objects also participates in other containment relations.
- The name of a containment relation is the name of the object interface where it is defined, followed by the name in the CONTRELINTERFACE definition.
- The definition of a containment relation is inherited in specialized object interfaces. It is allowed to specialize an inherited containment relation by the creation of a sub-relation\(^9\).
- Except for the above, an ODL CONTRELINTERFACE specification is a relation interface specification as defined in definition 12.6

The extent of a relation is its set of relationships. ☐

A containment relation interface, such as TheDepts in example 12.2, has the same syntax as a relation interface, except that the "from" object interface is implicit. The syntax for defining roles is then:

\(^9\)Sub-relation is defined in section 12.3.8.
<control_interface_declaration> ::= 
CONCRETEINTERFACE <relInterfaceID> [ "'<' <relInterfaceID> ]
  "'{' ' '#' <cardinality> [<roleName>] '',''
  <objInterfaceID> ' '#' <cardinality> [<roleName>] '',''
  [<rel_attributes>] ''}' 

The object interface where the containment relation interface is declared, is called the container. The object interface associated with <object_identifier> is called the contained. A containment relation interface can have attributes just like a relation interface. It is allowed to specialize a containment relation interface in the same way as for a relation interface.

12.3.8 Sub- and Super- relations

A relation interface S can be a specialized sub-relation of another relation interface R. The sub-relation concept is adopted from REBOOT [OH91], but refined to be a concept for sets of relationships, instead of a typing concept.

Definition 12.13 Sub-relation:
A sub-relation is a subset S: extent(FS) × extent(TS) × A₁ × ... × An of one super-relation R: extent(FR) × extent(TR) × A₁ × ... × An where

- FS is the interface of the from-object in a relationship in S, it may be an interface that is specialized from FR.
- TS is the interface of the to-object in a relationship in S, it may be an interface that is specialized from TR.
- The cardinality of S may be restricted to 1, where it has been specified as N in R.
- The set of relationships of the super-relation will be extended with the set of relationships from all its sub-relations. The relationships of S becomes a subset of the relationships of R.
- New attributes might be added, in addition to those inherited from R. Attributes can not be redefined.
- Only one super-relation is allowed.
- A containment relation is not allowed to be declared as a sub-relation of an ordinary relation, and vice versa.
- Except for the above, a sub-relation specification is a relation interface specification as defined in definition 12.6

Insert and removal of relationships have to be done directly on the sub-relation S, or one of its own sub-relations.

The syntax for the specialized relation interface is:

"'RelInterface' 'ID' "'<' "<Super_relation>
  "'{' [<roles>] [<rel_attributes>] '}'


<super_relation> must be a name of a previously defined relation interface. Note that "<" is used instead of ":" to indicate a super-relation, to indicate that this super/sub-relation has a subset-semantics as opposed to the interface-conformity semantics of the super/sub-relation used for object-interfaces.

If no <roles> are specified, then the specialized relation interface uses the same roles as the super relation. If <roles> are specified, then the cardinality can be changed from N to 1 and object_identifier can be changed to a subtype of the original object interface.

The <roleName> need to be changed if they are to be used.

Example 12.7 Sub Relation:

```c
RELINTERFACE WorksFor < AssocWith {
   Employee #N employees;
   Department #1 works_in;
   Date startDate;
};
```

In example 12.2 WorksFor is a subrelation of AssocWith. The from-role is specialized from Person to Employee, while the to-role remains a Department. In addition startDate is added as a new relationship attribute.

All employees working for a department will then be included in the AssocWith relation. The attribute startDate will not be shown through the WorksFor relation.

12.3.9 OML - Relation Management operations

- Management operations - implementation
- Management operations - subrelation
- Management of cardinality

The management-operations provide functionality for the explicit management of relation implementation managers belonging to a relation interface manager. These operations are used by the environment administrator, not by the tool builder or application programmer.

The following operations are defined for all relation interface managers:

- Management operations - implementation
  The operation `void addImplementation( RI<FT,A> [in] aRelImp, OSpace [in] for)`, facilitates adding a new relation implementation manager to a relation interface manager. The added relation implementation manager `aRelImp` is an instance of a subtype of RI.

  The operation `void removeImplementation( RI<FT,A> [in] aRelImp, OSpace [in] for)` inversely removes a relation implementation manager.

  The operation, `Set< RI<FT,A> > implementations()` returns the set of relation implementation managers for this relation interface manager.
• Management operations - subrelation
As mentioned, it is not possible to know the interface type of all potential subrelation interface managers in advance. This creates a problem for the type to be used for the sub-relation interface managers. Both the from type and the to type might be specialized, and new attributes might be added. This means that the type of a sub-relation interface manager will not be a proper sub-type. In the example, WorksFor of type R<Employee, Department, aWorksFor> is not a proper subtype of AssocWith of type R<Person, Dept, aAssocWith>. WorksFor cannot be used as a substitute for AssocWith because it puts stricter requirements on the input arguments. However, it is known that all the parameters individually will be proper sub-types. It is thus possible to do a safe type-cast internally in a relation interface manager. This is because inheritance also works for structures, such that aWorksFor might add attributes to aAssocWith but can always be used as aAssocWith also. The operations for management of sub-relations are therefore typed by ROI.

The operation `void addSubRelation( ROI [in] relIntMan, OSpace [in] for)` adds a relation interface manager to the set of subrelations. Inversely the operation, `removeSubRelation( ROI [in] relIntMan, OSpace [in] for)` removes a relation interface manager from the set of subrelations. The operation `void Set<ROI> subRelations() returns the set of all sub-relation interface managers.`

• Cardinality-constraint operations
The cardinality-operations provides functionality for checking and manipulation of the cardinality of a relation. The following operations are defined:

The operation `Cardinality getCardinality()` retrieves the cardinality for this relation. Underlying representations are not consulted. Cardinality is an enumerated type, as defined in example 12.6.

The operation `changeCardinality(Cardinality [in] newc)` changes the cardinality for this relation. If the existing relationships in the relation violates the new cardinality, no change is made and the exception "CardViolation" is raised. This operation is only available for the environment administrator.

Example 12.8 Management of implementations and sub-relations

```
persistent<od> R<Person, Department, AssocWith>* AssocWith;
persistent<od> R<Employee, Department, aWorksFor>* WorksFor;

persistent<ingres> R1Ingres<Employee, Department, aWorksFor>*
    WorksForIngres1;
AssocWith->addSubRelation( WorksFor);  // Useful with generic types for management tools - schema installer */
try {
    WorksFor->changeCardinality(CIN);
}
WorksFor->addImplementation( WorksForIngres, Ingres1);
AssocWith->insert(e1, d1, aw);
}
catch{NotExist ne) {
    }
}
catch() { ... ;
```
12.4 Mapping from COOM$ _S$ to Object-Oriented Languages

Example 12.8 shows the use of relation interface manager objects, that are referenced as named variables, through the ObjectStore persistent variable facility.

12.4 Mapping from COOM$ _S$ to Object-Oriented Languages

An overview of mapping-strategies from COOM$ _B$ to C++ and Smalltalk was given in section 11.4. More details about the C++ mapping can be found in [Sol92]. More details about the Smalltalk mapping can be found in [Øst92b, Øst92a].

12.4.1 Mapping from COOM$ _S$ to C++

The mapping of object-interface attributes to pairs of put/get operations is straightforward. The same name can be used for both the put and get-operation, because their signatures are different. However, the same name can not be used for private instance variables.

Relation interfaces and implementations are mapped to classes. The manager objects are created as named instances of these classes.

The template-mechanism in C++ is used for the typing of relation interface managers. Each new relation interface manager will be instantiated as an instance of a corresponding class R<From,To,ARname>.

Example 12.9 Relations in C++

template<
class F, class T, class A>
class R {
    public:
        virtual Set<T>* findObjFrom1(F*);
        virtual Set<F>* findObjTo2(T*);
        virtual void insert(F* f, T* t, A* a);
        virtual void remove(F* f, T* t, A* a);
        virtual RIT<F,T,A>* getIterator(F* f, T* t, A* a);
    }

class RAtt {
}; /* Struct RAtt */

class AWorksFor: public RAtt {
    public:
        Date* startDate;
    }

// Declaration of relation managers:
persistent<od> R<Employee, Department, AWorksFor>* WorksFor;
persistent<od> R<Person, Department, AWorksFor>* WorksFor;
persistent<od> RIOS<Employee, Department, AWorksFor>* WorksForOS1;
persistent<od> RIngres<Employee, Department, AWorksFor>* WorksForIngres1;
// The relation managers are made persistent through the
// ObjectStore persistent variable facility

12.4.2 Mapping from COOMS to Smalltalk

The mapping of attributes to pairs of operations is the normal way of doing attribute-access in Smalltalk, since instance-variables cannot be accessed directly.

Relation interfaces and implementations are mapped to classes. Since a class-object is automatically instantiated, this can be used as the manager object.

12.4.3 Relations and System Autonomy

Relationships may exist between two objects. If one object is deleted, it is necessary to reflect this in the relations that the object is involved in. Relations will be represented in the appropriate structures of underlying systems so that internal deletions in an underlying system automatically will be handled. Relationships between different object spaces are managed by a relation implementation manager in the integration space. In order to maintain a consistent relation here, it is necessary to inform the relation implementation manager about all relevant object deletions. This restricts the autonomy of the underlying systems. An alternative is to always check if the objects exists, before giving out information about a relationship.

12.4.4 Relations and Notifications

It is useful to be able to monitor changes made to a relation. This is done through the notification management facility. All insert-operations will result in the generation of a notification of type Inserted from the relation interface manager.

```
Notification inserted
{OIDString relIntMan;
 OIDString from;
 OIDString to;
};
```

By giving values to the OIDStrings from and to, it is possible to monitor relationship-insertion for specific objects.

12.5 Possible extensions to COOM structural model

This section describes some useful extensions to the COOM Structural model.

- *Ordered Relations*
  Quite often there is a need for applying a certain ordering on the relationships, i.e. when one is modeling the different chapters of a document and wants to keep the sequence of chapters in order. A relation-interface might then be specified as an
ordered relation. When it is ordered it means that there is a sequence applied on the involved relationships. Currently, this will be handled with a sequence-number attribute. However, the insert of a new relationship between two existing requires an increment to all following sequence numbers. This could more easily be handled by an ordered relation, where the the operation-set had been extended with operations for insertBefore and insertAfter.

- **Max and Min Cardinalities**
  To give more comprehensive support for integrity-constraints, it could be possible to also state values for minimal and maximal cardinalities for relations.

- **N-ary Relations**
  Some relations are by their nature ternary or n-ary. An example of this is the Coach-relation between Person, Sport and Club: A person X is a coach for Sport Y in Club Z, and also a coach for Sport M in Club N. An extension from binary to N-ary relations for COOM is specified in [Sol92].

- **Relationship structures**
  Relationships are not objects, as they have no OIDs. It is possible to enhance the relationship attribute structure to also contain the from and to objects. Then a relationship could be handled in one piece, and access-operations and iterator-operations could return full relationship-structures. This does not make relationships objects, but facilitates the handling of a set of relationships, such as in: `Set<Works_In**, w relship = el->activities();`

- **User-defined operations**
  It could be useful to support special-purpose user-defined operations for relation interface managers.

- **User-defined notifications and exceptions**
  If it becomes possible to have user-defined operations, it will be useful to also have user-defined notifications and exceptions.

### 12.6 COOM and Data Integration

#### 12.6.1 Single model data integration by structural abstractions

Single-model data-integration is the degree to which systems are able to share common data that are stored and manipulated through one single data model and a corresponding storage service. COOM supports this through viewing existing data-items within systems and database-systems as objects, and references between data-items as relationships.

COOM succeeds in supporting most of the requirements for single-model data-integration. Versioning, long cooperative transactions and security-aspects have not been dealt with in this work.

An evaluation of COOM with respect to the requirements given in section 3.4 is given in section 16.2.

#### 12.6.2 COOM and Multi-model Data Integration

Multi-model Integration was defined in section 3.4.3 as: *the degree to which tools are able to access and share common data that are stored and manipulated through multiple data models with corresponding multiple storage services.*
The current work has not aimed to resolve all problems related to multi-model data integration, but instead to develop a framework where such problems can be investigated further\textsuperscript{10}.

Schema Integration as described in RDM-3 will be commented on in the next section.

12.6.3 Schema Integration and semantic interoperability

This section will give some directions for how further work on schema integration might evolve. Such work has already started, as commented on in section 16.6.

Problems with schema integration arise when there is a need for having a unified view on information in different object spaces. The easy case, implicitly supported by COOM, is when the schemas are totally disjoint and the new schema becomes the union of the schemas for the involved object spaces. COOM also supports schema integration when equivalent object and relation interfaces have disjoint objects and relationships.

In general there is a difference between identical and equal objects. Identical means that they have exactly the same object identifier, equal means that their attributes, operation results and relationships are the same. Equivalent objects is here defined to be real-world identical objects, which means that two objects are equivalent if they represent the same real world object. The problem is then to determine when they are real-world identical. In the following discussion Equivalent objects is then defined to be objects that are "real world" identical. This means that they are meant to model the same real world phenomenon. Disjoint object are objects that model different "real world" phenomenon. Equivalent interface means that the objects have the same interface while Different interface means that they are not totally conform with each other.

In general, the following situations might arise:

1. Disjoint objects - Equivalent interface.
2. Disjoint objects - Different interface.
3. Equivalent objects - Equivalent interface.
4. Equivalent objects - Different interface.
5. Different representation of information: objects, attributes, relations.
6. Evolution and changes.

Problems and solutions related to these different cases are discussed in the following.

1) Disjoint objects - Equivalent interface

An example of this is: Different person-objects might exist in different object spaces, but there is globally only one person-object for each physical person.

This is the situation currently handled by COOM. Objects in different object spaces might use different local object identification schemes, since the identification scheme is handled by the particular implementation.

The following are potential problems with respect to the management of object identifiers, OIDs:

\textsuperscript{10}See description about future and ongoing work in section 16.6.
12.6 COOM and Data Integration

- **OID uniqueness** - How to generate globally unique OIDs, so that there is no conflict when databases are interoperating? The global uniqueness is taken care of by the creation of unique proxy-objects, which represents a local object in a given object space.

- **OID equivalence** - How to determine when two OIDs refer to the same real world object? This is discussed in section 12.6.3.

- **Locatable objects** - How to determine an object location from its OID? The OIDString encodes knowledge about the implementation manager and the object space.

- **Local autonomy** - How to preserve local autonomy in creating OIDs, while preserving global uniqueness and preserving object equivalence? The local autonomy is limited to a local selection of an object identification scheme. Object equivalence is discussed in section 12.6.3.

The COOM object identification scheme requires secure local object identifiers. A secure OID is a unique and immutable identifier which always can be used to retrieve the object. Many systems do not provide secure object identifiers. In a relational database system the "object identifier" will at best be the key-attributes for a tuple in a table. To assure stability, local autonomy needs to be restricted by either disallowing changes to key-attributes, or to require that a notification of such changes is given. There are some object-oriented DBMSs that offers stable object identifiers within a session\(^{11}\), but not between sessions. Root-objects have secure object identifiers, but related objects have session-based object-identifiers.

Problems related to the stability of local object identifiers are discussed further in chapter 14.

2) **Disjoint objects - Different interface**

An example of this is: Different person-objects might have different interfaces, such as student and employee. There is no person which is both a student and an employee.

The normal case is that independently developed object interfaces will be different for the same real-world objects, due to different focus and needs.

Some of the following differences might arise:

- The same information might be represented by different names - "Synonyms".
- Different information might have the same name - "Homonyms".
- Different levels of aggregation or scales might be used, e.g. weight in kilos or pounds.
- Loss of information from detailed to less-detailed descriptions.

When a new component is integrated into an environment in which a stable schema already exists for that particular domain, it is possible to make uniform interfaces during the schema enrichment and schema reconciliation phase.

An implementation-mapping might transform to and from the representations in an underlying system, and the needed interface. The difficulty is in discovering when semantic

\(^{11}\) A session might last between the begin and commit of a transaction, or between the open and close of a database
conflicts do exist. A semantic conflict exists, for instance, if the price of a product is given without taxes in one system and with taxes in another system. The main problem is how to identify and state such conflicts. The implementation mapping might then try to resolve the conflict.

The loss of information from detailed to less-detailed descriptions might also be difficult to handle, if one should want the most detailed description.

3) Equivalent objects - Equivalent interface

An example of this is: *The same physical person might be represented as different person-objects in different object spaces.*

In the context of an object-oriented common data model, the question of object-equivalence becomes important. The fundamental problem of equivalent objects is to determine the criteria for object equivalence.

For instance if an Ingres database with Ingres1 object space and an ObjectStore database with os2 object space both contains objects with object interface Person, it might happen that two objects both represent the same real-world person. The criteria for object equivalence might be that two objects have the same social security number. If two or more objects have the same social security number they should be presented to the application-programmer as one object.

The goal is that the application-programmer should only see one object. This can be resolved by creating a new layer of "integrated" objects, hiding the underlying objects. The "integrated" objects are created in the integration space, and will obtain knowledge about which objects they are "integrating". The criteria for object equivalence need to be determined as a mapping between different equivalent objects. A possible mapping-function could be identical social security numbers. If no mapping-functions exist, an externally maintained mapping table could be made. One way of resolving this is to state a criteria for object equivalence in the object interface, e.g. *Object Equivalence On SocSecnumber;*.

When equivalent objects exist, the following problems arise:

- **OID equivalence** - What is the criteria for object equivalence? Should the application-programmer see one or multiple objects?

- **Certitude and Discrepancies** - What to do when information in two objects differs? E.g. if there are two different birthdays for the same person, could information in one of the objects have a higher certitude?

4) Equivalent objects - Different interface

An example of this is: *The same physical person might be represented as a student in one object space and as an employee in another object space.*

In this case, a physical object is fragmented and partly replicated into two objects. The goal is to present this as one object to the application programmer. If students and employees are generalized into persons, the Person object interface should reflect the intersection of their interfaces. It is difficult to present the union of the interfaces, because relevant information will not be available for persons who are either a student or an employee, but not both.
When component schemas already exist, it might be possible to use generalization, and create a new object interface as a parent interface for the actual object interfaces. This requires an object-oriented programming language and/or environment which supports generalization. Unfortunately, most current object-oriented programming languages only support specialization (subclassing).

That is, current object-oriented programming languages typically do not allow to create super-classes of existing classes. Such a generalization (supertype) could be implemented in different ways. One way is to directly reuse instances of the existing types Student and Employee, and to manage the extension of Person such that for equivalent instances of both types (e.g. objects representing same physical person being both Student and Employee), only one object is retrieved from the extension. This requires an underlying object-oriented programming language where it is easy to support evolution and upward inheritance [SN88]. An alternative is to use representation-objects. These are new objects with the required interface, but with the knowledge of how to delegate messages further to the underlying objects. This does not require any changes to the underlying interfaces and implementations. Which alternative to choose depends to some extent on the facilities of the actual object-oriented programming language.

The first problem is how to identify when two objects are equivalent. This can be handled similarly to the case of equivalent objects with equivalent interfaces, as presented in the previous section. In addition comes the problem of how to deal with different interfaces. A common generalized interface needs to be decided. This is a way of doing generalization, where a new object is made as a representative for more specialized objects. This would mean that the specialized objects should not been seen in the new schema.

If there exists two different databases, Students and Employees, one physical person might be represented in both by playing both roles. E.g. it can be decided that it is the same person, if both representations have the same social security number. Here one might see a need for doing generalization which is not generally supported in existing object-oriented programming languages. From two existing types Student and Employee, derive a common type Person, where the extent (set of instances) is the intersection of the set of instances of subtypes.

By declaring the object-type Person to be a generalization of Student and Employee, and stating that object equivalence is based on (student.persnr = teacher.socnr) an implementation of the Person class might automatically be generated. If no object equivalence is stated, it is assumed that the extensions are disjoint. The interface supported is what is common for the interfaces of the subtypes. New operations might also be introduced, such as total.salary(), but this needs then to be specially implemented. It would be possible to have a generic object-integration-language to resolve simple conflict-situation such as synonyms and homonyms. The implementation will automatically take care of mapping to two objects that represent the same person-object.

**Example 12.10 Generalization**

```java
OBJINTERFACE Person
{ Generalization of {Student, Employee}

    Object Equivalence IS {Student.persnr == Employee.socnr};
    String name is {if Student {Student.firstname concat Student.lastname}
                      elseif Employee {name}
                      elseif equivalent {Employee.name}};

    String address; // ok in both special types
```
Money salary IS {if Student {grant} elseif Employee {salary}};
Real totalSalary() IS {Student.grant + Employee.salary};
};

Example 12.10 shows some parts of a possible ODL-syntax for the support of generalization.

It has to be considered what to do in the different cases when the object is represented as either student or employee, and when there exists one student and one employee object which are equivalent (real-world identity).

The generalized interface is to automatically generate an implementation-class which takes care of the most common mappings.

The alternative of representation-objects seems promising for languages which do not directly support generalization. This requires a special equality-operator which returns true if two different objects are representing the same "real" object. Some work in this direction has been presented in [RS91a]. The work by Svein Erik Bratsberg [Bra93] at the Norwegian Institute of Technology has addressed some aspects of generalization.

5) Different representation of information: objects, attributes, relations

An example of this is: A person-activity might be represented as a relationship between a person-object and an activity-object in one object space, and as an object in itself in another object space, and as a string-attribute, person-number, for an activity in a third object space.

The conceptual difference between objects, attributes and relations creates another possibility for schema integration problems.

A related problem occurs when something is modeled on the schema level in one schema and on the instance level in another schema. An example of this is: A person might be specialized into two object interfaces, Man and Woman, in one schema. Another schema might represent this difference in an object-attribute, sex (M/F), on the instance level.

These problems might be resolved by defining a new conceptual schema, and view the different existing schemas as underlying representations which it is to be mapped to and from. Issues related to such mappings need to be explored further.

6) Evolution and changes

An example of this is: A relation person-activity might be extended from 1:N to M:N, if it has been decided that many persons can participate in the same activity.

Evolution and changes to the schema is bound to happen. In particular due to real world domain evolution.

Some examples on domain evolution are:

- Cardinality changes: 1:N -> M:N
- Granularity changes: location (room - building)
12.7 Contribution and conclusion

This chapter presented the basic concepts of abstract attributes and relations in $COOM$s. $COOM$s provides support for single model data integration and some aspects of multi model data integration. Attributes and relations also provide a necessary basis for the query language, OQL, which will be introduced in the next chapter.

The concept of basing a structural data model on top of a behavioural data model is different from the approach used by others, i.e. the REBOOT model [OH91, Rum87]. The most used approach is to extend a structural data model with support for operations. These structural models tend to provide attributes as data members with values, and relationships as data members with pointers. This gives less flexibility with respect to variations of how these are mapped to different underlying systems. The behavioural basis, with only an operational interface, gives a better basis for the integration of heterogeneous systems, by an encapsulation of how the caching and management of attributes and relationships is done. An operational access makes this transparent for the programmer. Note that the important point here is full encapsulation. This goal can also be achieved by layering a behavioural model on top of a structural model, if all attributes and relations are encapsulated and reflected in the operational interface. The SOOM model presented in chapter 7 realized a behavioural model on top of a structural model, and enforced full encapsulation. The special typing of relationships used in SOOM, made it difficult to map the model to various underlying systems. This is more generally supported in COOM. The strategy for operational access to attributes and relationships is adopted from message-oriented object-oriented languages, like Smalltalk [GR83] and is also used in the OMG CORBA model for attribute-access.

The functionality for relations is adopted from the structural REBOOT data model [OH91]. The separation of relation interface and implementation with support for multiple implementations of a relation interface is novel in COOM. The dual support for both role-name access and relation-access is new in COOM. Role-name access provides the same encapsulation of actual implementation as a relation interface manager.

The possibility for the association of exceptions and notifications with attribute-specifications is novel. The possibility for the specification of unique attribute-combinations in an interface is also new, this was introduced in the initial COOM-compiler [Ber91g].

In COOM a relation is an object with the functionality of managing relationships between objects COOM introduces a clear distinction between sub-relations and specialized interfaces. A sub-relation in COOM is explicitly stated as a subset of relationships.

The management of multiple parallel implementations of object interfaces and relation interfaces provides a basis for multi model data integration. The current handling of
generalized interfaces is through the explicit creation of a new object interface with new objects.

An example of the use of $COOM_S$ for data integration is given in chapter 15, where it is shown how a RDBMS and an ODBMS is integrated into COOP. How the constructs of $COOM_S$ give a basic support for the requirements of single model and multi model data integration is evaluated in section 16.3.1.

The next chapter presents OQL, and the support for query processing. Abstract attributes and relations as introduced in this chapter, is the basis for OQL-queries.
Chapter 13

COOM OQL - Query Processing

This chapter presents OQL, COOM Object Query Language, and the related support for query processing. OQL is based on the definitions of abstract attributes and relationships from $COOM_S$, and provides a facility for flexible access to objects.

13.1 Motivation for OQL

In order to efficiently manage and retrieve a large number of objects, there is a need for a powerful query-language and corresponding query processing facilities. It is important to be able to select and retrieve existing objects, based on intentional predicates on their characteristics and relationships with other objects.

There exists already a number of object-oriented query languages, [LLOW91, B+91a], but none of these support the concept of abstract attributes and relations as defined in COOM.

OQL is based on the associative query language AQL developed for the REBOOT data model [CK90]. AQL supports attributes and relations as defined in $COOM_S$. It has, however, been necessary to extend AQL with support for objects as values, query arguments, nested queries and operations. This has been done in order to show how the separation of interfaces and implementations in COOM gives a new basis for parallel query processing in multiple underlying systems. In particular the use of objects as values was necessary for this.

OQL is based on the constructs in the structural model $COOM_S$. The goal of OQL is to be declarative, efficient and independent of programming language. The same OQL-queries can be used for different programming language mappings.

The OQL is based on an object-calculus, where a predicate is conceptually applied to all objects in a set. A query is an object calculus expression of the form $\{o|\psi(o)\}$ where $o$ is the only free variable in $\psi$. In the predicate $\psi$ the following logical connectives might be used: $\forall, \exists$.

The initial basis for a selection is the extents of object interfaces and implementations managed by object interface managers and object implementation managers respectively. An extent represents the set of all instances that belongs to the interface or the implementation. A query on this extent will return the instance-subset that conforms to the predicate in the query. The predicate might specify restrictions on attribute-values or relationship-links.

By keeping OQL at the object-calculus level, it might be a possibility to later utilize
results from current work on object-algebras and related query-optimization-techniques [S790, SO90].

The current OQL is just a minimal version in order to show the general principles of query processing. Extensions are discussed in section 13.5. SOQL [Low92] is a Static SQL-inspired OQL developed to explore a more extensive query language, and will be briefly presented in section 13.5.

Without a query-language it would be necessary to select objects by first retrieving the extent of a needed object type, and then do individual tests for each object. The example shows how this can be done for the selection of teenagers.

Example 13.1 Selection based on iteration

```c
Set<Person>* teenAgers = new Set<Person>();
Set<Person>* persons = metaPerson->extent(); // All persons
Set<Person>* persons = Person::extent(); // Access through static operation
  // hides meta-objects from
  // the tool-builder
/* iterate the set through a DO-END macro */
DO(persons, Person*, p)
  if (p->age() >= 13 && p->age() <= 19)
    { cout << p->name(); // prints the name of a teenager
      teenAgers->add(p);
    }
END;
```

This requires the retrieval and checking of each person-object. Underlying systems might have facilities for doing such selection more efficiently. A query will be translated into a set of queries on the underlying systems.

All object interface managers support the operation `find("predicate")`, which will be executed by delegation to relevant object implementation managers.

Example 13.2 Selection based on an OQL-query

```c
Set<Person>* teenAgers = metaPerson->find("age >= 12 && age <= 19");
```

13.2 OQL Principles

The Object Query Language (OQL) gives possibilities for selecting subsets and combinations of objects from the existing collections of objects, based on predicates on their attribute-values and relationships. The type-externt are the basic collections for initially retrieving objects from the database.

The conceptual view of a query is that it is equivalent to a "foreach" loop-construct that evaluates the predicate-construct for all objects in the set in turn, and returns the subset of these objects for which the predicate evaluated to true. This is inspired from the `select:`- and `collect:`-methods from Smalltalk collections [GR83, BKK88]. A query might be more efficiently implemented than actually doing the iteration through all the objects.
The goal of the query-facility is to transform a query formulated in OQL to a (set of) efficient queries on the underlying DBMSs.

A query might in principle be object-preserving or object-generating. An object-generating query might result in a new object which did not exist prior to the query. An object generating query might also dynamically define the interface of the new object. An object-preserving query will always return existing objects, with a predefined interface. The current OQL is object-preserving, as shown in table 13.1.

A query might in principle be statically or dynamically defined. A static query is predefined and might be optimized by a query-compiler at compile-time. A dynamic query might be created at run-time, and must thus be interpreted and optimized at run-time.

Section 13.5 describes a suggestion for an OQL, SOQL, which is object-generating for statically defined queries.

<table>
<thead>
<tr>
<th>Parse-time Object</th>
<th>Compile-time</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Preserving</td>
<td>OQL</td>
<td>OQL</td>
</tr>
<tr>
<td>Object-Generating</td>
<td>SOQL</td>
<td>(\div(C++))</td>
</tr>
</tbody>
</table>

Table 13.1: OQL - Object-Preserving/Generating at Compile/Run-time

The query might either be parsed and generated/optimized at compile-time, or at run-time. OQL is supporting this at run-time, with a possibility to create query-objects which are parsed, and which might be bound to different variables for input. The \(\div\) mark for object-generating at run-time is because run-time generation of new classes is not possible in C++. Object-generation from existing classes is possible.

The same OQL-queries might be used in all languages supported by COOM, e.g. C++ and Smalltalk. Object-oriented query languages have often broken the encapsulation of objects to reveal which attributes exist. If one knows about actual attributes it is possible to create indexes on the attributes to speed up query processing. In COOM this is handled on the implementation manager level. The separation between interface and implementation makes it possible to create implementation dependent indexes without reflecting this in the interface.

Figure 13.1 shows the principal use of the OQL compiler for static OQL and the OQL interpreter for dynamic OQL.

Both the compiler and the interpreter are getting OQL-statements as input and produce queries for one or more target systems as output. The knowledge about the ODL schema and the mapping between ODL and constructs in the target systems is retrieved from the Object Dictionary database. Information about schemas and mappings is placed in the Object Dictionary database by the ODL compiler. These are stored as interface-definitions and implementation-definitions in association with interface and implementation managers. These are not shown in the figures.

Static OQL is embedded in the C++ or Smalltalk application program source code by the application programmer. Then this is run through (preprocessed by) the OQL compiler, which generates necessary code against the underlying systems.

SOQL uses a compiled approach, while the current OQL uses an interpreted approach. It is in principle possible to do an initial parsing and optimization of an OQL query, and later then only bind in actual values for the query.
13.3 OQL Constructs and Syntax

Queries are supported through the `find("predicate")` operation which is provided by all object interface managers. In principle a similar `constrain("predicate")` operation could be provided by all collection-objects, but this has not been realized yet. An extra complexity for this is sets containing objects from many different object-spaces, thus having different underlying implementation and representation. The "predicate" will "conceptually" be applied to all objects of the collection. The result is the subset of these objects for which the predicate evaluates to true.

The OQL Grammar is given in Appendix B. The following gives a short description of the most important elements of a predicate.

```plaintext
<predicate> ::= <atomic Predicate> | <predicate> "&&" <predicate>
             | <predicate> "||" <predicate>
             | "(!" <predicate> ")"
             | "!" <predicate>

<atomic_predicate> ::= <comparison> | <quantifier>

<comparison> ::= <lvalue> <operator> <rvalue>

<lvalue> ::= <dot_list>
```

Figure 13.1: Query Processing - OQL Compiler and Interpreter
13.3.1 Comparisons

\[ <\text{rvalue}> ::= \]
\[ \quad <\text{dot_list}> \]
\[ \quad | <\text{const_exp}> \]
\[ \quad | <\text{nestedQuery}> \]
\[ \quad | <\text{argument}> \]

Simple comparison-predicates can be combined into more complex ones by use of the logical operators '&&' (AND), '||' (OR) and '!' (NOT). Brackets () can be used to explicitly indicate sequence of execution, which is done from left to right with highest priority to NOT and equal priority to AND and OR.

13.3.1 Comparisons

The following operators can be used in comparisons:

\[ <\text{comparison}> ::= <\text{lvalue}> <\text{operator}> <\text{rvalue}> \]
\[ <\text{operator}> ::= <\text{identop}> | <\text{orderop}> | <\text{setmembop}> \]
\[ <\text{identop}> ::= '==', '!=', '===' \]
\[ <\text{orderop}> ::= '<', '<=', '>', '>=', '<' \]
\[ <\text{setmembop}> ::= \text{IN} | \text{CONTAINS} \]

The identity operators check on identical values, while the order operators check on relative order for types where ordering is defined. The operators IN and CONTAINS are used for set inclusion and containment.

Example 13.3 Comparisons:

SetOf Hobby hobbies;

Person.hobbies == { 'Tennis' };

Person.hobbies CONTAINS { 'Tennis' };

Person.hobbies IN { 'Tennis', 'Golf', 'Soccer', 'Basket' };

A left-hand side literal, such as in 'Tennis' IN Person.hobbies, is not legal.

13.3.2 Comparison of Objects

OQL allows also for comparison of objects with the identity operators and the set member operators. The order operators are not initially supported for objects\(^1\).

The use of IN and CONTAINS for objects checks on object membership based on object identity, while == similarly checks on object-identity. A possible later extension is to also provide an equality operator, EQ, which can check if two objects are value-identical but not necessary object-identical.

\(^1\)It could be possible to support ordering among objects, if order operations are defined for an object interface.
13.3.3 Dot Notation

Dot notation can be used to access attributes or operations of an object or attributes of a relationship, or the object(s) in a relationship. Legal identifiers are names on attributes, roles and relations.

\[
\begin{align*}
\text{<dot_list>} & \quad ::= \\
& \quad \text{<dot_element>} \\
& \quad | \quad \text{<dot_list>} "," \text{<dot_element>}
\end{align*}
\]

\[
\begin{align*}
\text{<dot_element>} & \quad ::= \\
& \quad \text{<relInterfaceID>} \\
& \quad | \quad \text{<relInterfaceID>}_\text{INV} \\
& \quad | \quad \text{<roleName>} \\
& \quad | \quad \text{<attributeID>} \\
& \quad | \quad \text{<operationID>}
\end{align*}
\]

The teenager example can be extended to include relations, i.e. by finding all persons between 20 and 30 working in the "Computer Science" department.

**Example 13.4** Find all employees of age between 20 and 30, working in the Computer Science Department:

Set\{Employee\}* s = metaEmployee->find("age >= 20 && age <= 30 \\
& worksFor.name == 'Computer Science' ");

The example shows comparison of values, where a dot notation is used to access the name of a department, through the worksFor relation. The use of relations is described below.

13.3.4 Relations

A relation can be specified through its name, or through a possible role-name. To traverse a relation in the to-from direction instead of from-to, the relation name is used with "INV" as a suffix.

Containment relations are handled in the same way as ordinary relations.

Set\{Department\}* ds = 
metaDepartment->find("Company_TheDept_INV.[name == 'SINTEF'] ");

13.3.5 Multi-valued Relations - Exists and ForAll Semantics

When a relationship is multivalued, as with 1:N or M:N cardinalities, the result of traversing a relationship is a set of objects.

A predicate can be true for all relationships or for at least one relationship. It is a difference between finding all departments with at least one employee between 20 and 30 and finding all departments with all employees between 20 and 30.

A bracket-notation is used to distinguish these two different cases. Exists-semantics is denoted by [... brackets, and ForAll semantics is denoted by {...} brackets. The base for the predicate within the brackets is the set of objects referenced in the relationship.
This is syntactically stated as a quantifier, which is a dot_list with a predicate which will evaluate to true or false.

\[
\text{quantifier} ::= \text{dot_list} \{ \text{predicate}\} \{\}\text{ /* ForAll */}
| \text{dot_list} \{\text{predicate}\} \{\}\text{ /* Exists */}
\]

**Example 13.5 Exists:**

Find all departments where there exists at least one employee with an age between 20 and 30:

```java
Set<Department> s1 =
    metaDepartment->find("worksFor_INV[age >= 20 && age <= 30] ");
/* Or use role-access through employees instead: */
Set<Department> s2 =
    metaDepartment->find("employees[age >= 20 && age <= 30] ");
```

**Example 13.6 ForAll:**

Find all departments where all employees have an age between 20 and 30:

```java
Set<Department> s3 =
    metaDepartment->find("employees{age >= 20 && age <= 30} ");
```

### 13.3.6 Relation Attributes

Attributes on relationships are handled similarly to object attributes.

Access to a relationship-attribute:

```java
Set<Employee> s =
    metaEmployee->find("worksFor.startDate > '1.1.92' ");
```

Access to an object-attribute:

```java
Set<Employee> s =
    metaEmployee->find("worksFor.name == 'Computer Science' ");
```

If the relationship and the object have attributes with the same name, the relationship-attribute will be selected\(^2\).

### 13.3.7 Nested Queries

A comparison value can be the result of a new query based on a predicate on the extent of another object interface. The name of that object interface is then given, followed by a predicate in brackets.

A bracket-notation is used to enclose the predicate. Two different bracket types are used with the same "ForAll" (\{\}) and "Exists" ([\[]\]]\) semantics as presented in 13.3.5.

\(^2\)A possible solution to resolve attribute names in relationships and related objects is to use ",->$" instead of ",." notation for access to object attributes.
13.3 OQL Constructs and Syntax

\[ <\text{nestedQuery}> ::= <\text{objInterfaceID}> \{ \text{'' predicate ''} \} \]

// Return Set of objects of type <\text{objInterfaceID}>
| <\text{objInterfaceID}> \{ \text{'' predicate ''} \}
// Return one object of type <\text{objInterfaceID}>

Example 13.7 Return Set:
Find the employees that are employees of a Company that have a Sales department.

Set<
Employee>* s =
metaEmployee->find(" department IN Company(TheDepts.name=='SALE') ");

These queries might also be rewritten to be directly based on value-comparisons, such as

Set<
Employee>* s =
metaEmployee->find("department.name == 'SALE' ");

This possibility may make nested queries unnecessary.

13.3.8 Query Arguments

It is useful to parameterize a query with dynamic arguments. Arguments are in the
predicate prefixed with % and a letter stating the type of the argument. %o means an
object while %b is a basic type. The types should correspond to the types stated in the
schema.

Set<
Employee>* s = metaEmployee->find("(age >= %b & age <=%b) 
& department == %o ", minAge, maxAge, d1);

Person->find("children IN %o", personSet)
Person->find("department == %o", d1);

The number of arguments depends on the number of % in the predicate. An unknown
number of arguments can in C++ be supported by the (,...) open function declaration
and in Smalltalk by a collection of arguments.

If an argument is used many times it must be given as input the same number of times. A
more flexible approach is to define a named keyword structure with argument-name and
a bounded value, vs keyword arguments (ObjectStore)

The query-parser needs to check that provided arguments is of correct type, by checking
against the definitions in the object dictionary.

13.3.9 Other Query Operations

In addition to find("predicate") which returns all objects satisfying a predicate, there are
three other useful query-operations:

- \text{findOne("predicate")}.
  Returns one object satisfying the predicate.
13.4 OQL Query Processing

- \textit{exists}(\textit{"predicate"}).
  True if there is at least one object satisfying the predicate.

- \textit{forall}(\textit{"predicate"}).
  True if all objects satisfy the predicate.

13.4 OQL Query Processing

Figure 13.2: Query Processing - Different steps

Figure 13.2 shows the different steps involved in query-processing.

First, a parse-structure of the query is created. This is syntactically checked against the OQL Grammar. Next, the parse-tree is checked against the schema definitions in the Object Dictionary. All this is managed by the relevant interface manager object, which might delegate further to a generic query-processor. The query should be globally optimized based on the knowledge about the relevant implementation managers. Relevant query-fragments is then given to the different implementation managers which will transform the generic query-tree to a form which is suitable for the native queries of the underlying systems.

The next step is query-processing in the underlying systems, where the result is mapped back to objects by the implementation managers. Each implementation will know about how to map between COOM objects and relations and the representation used in the underlying system.

The interface manager must perform the final preparation of the result, by merging the
different results. Some parts of a query might also be necessary to perform on the global interface manager level.

Figure 13.3: Query Processing - Interface and Implementation Managers (Objects and Relations)

Figure 13.3 shows how interface managers are related to implementation managers. Initial retrieval of objects is typically done through a find-operation of an interface manager. The interface manager will delegate this further to the implementation managers and its subtypes. Each implementation manager will get its part of the parse-tree for the query and transform it to a form which is convenient for the actual implementation. Typically an implementation done for a relational DBMS will transform it to a select-query on the corresponding tables. This is described more in chapter 14.

13.5 Issues and Possible Extensions for OQL

OQL is a minimal language developed only to demonstrate the basic concepts of query processing in COOM.

There are a number of issues that requires further investigation, and a number of possible extensions that could be made. Some issues and possible extensions are described in the following.

- Aggregate functions
  It would be useful to be able to select results based on aggregate functions like (Min, Max, Sum, Average, Count).

- Arithmetic operations
  It could be useful to support arithmetic operations in a predicate, e.g. age < age0 + 10.
Null values and Pattern Matching
The issue of Null values needs a closer treatment. It is possible to enhance string-comparisons by pattern matching.

General Operations
Operations might have side-effects, such as creating new objects or changing attribute values. It is difficult to have a consistent query if such operations are included. Thus, only operations without side effects should be allowed within OQL queries. Currently there is no way of distinguishing such queries in COOM. Many underlying systems, such as relational database-systems, might not support general operations. In such cases operations will be executed in the object-space level, and not directly in the relational database-system.

Categories and Views
There is an important difference between value-based specialization and interface specialization based on conformity. A subset of the extent of an object interface could be selected based on attribute-values. E.g. teenagers are all persons with an age in the interval between 13 and 19. Such predicate-based subsets could be defined as special category-sets which the system could give special support for. This could be part of a view-concept which could be useful. A special kind of category is interface-exterms which satisfy the predicate that all members of the set provide a particular interface (either by being a direct implementation of this interface, or by being an instance of a specialized interface).

Queries versus Trading
The trading-concept for matching offering of services with request for services has been developed for the matching of components in distributed systems, [ans89]. This is a kind of query which, however, often refers to non-manifest attributes of an object such as performance and cost. An integration of query and trading-concepts would be useful.

SQL-inspiration - SOQL
There has been arguments that SQL is "Intergalactic Data-speak" [Sto90] and all query-languages need to conform to its basic concepts. An SQL-inspired version of OQL for COOM is described in [Low92]. The goal of this approach has been the following:

- Support the basic SQL SELECT-FROM-WHERE construct
- Make it possible to only cache selected attributes
- Make it possible to generate new objects as results
- Support aggregation operations like Min and Max
- Support other operations such as Update, Insert and Delete
- Provide for multiple input and output parameters

The approach has been to define SOQL as a kind of embedded OQL. This is done by embedding a text after the construct EXEC OQL, which a pre-processor transforms to a new function, which later can be called with input and output parameters. The following gives some examples of SOQL.

Example 13.8 SOQL:
EXEC OQL
Persons1(Set<Person> [out] persons) /* return-function */
  SELECT persons = ["p.(name, age)] /* [ ] means return of Set */
  FROM Person p
  WHERE "p.age > && /* "p is a query-variable */

EXEC OQL
Persons2(Set<Person> [out] persons, String [in] dname)
  SELECT persons = ["p.(name, age)]
  FROM Person p
  WHERE "p.age > &&
  "p.department.name = dname

EXEC OQL
raise()
  UPDATE Employee e
  SET salary = "e.salary * 1.1
  WHERE "e.worksFor.name = "Computer Science"

EXEC OQL
SaveActivity (Activity [in] newActivity)
  INSERT INTO Activity
  WITH hasPersons
  VALUES newActivity

EXEC OQL
salaryInformation(Tuple<float, int, int> [out] salaryInfo)
  SELECT salaryInfo = (avg("e.salary), min("e.salary), max("e.salary) )
  FROM Employee e
  WHERE "e.department

13.6 Contribution and Conclusion

This chapter presented the object query language, OQL, and strategies for query-processing in the COOP architecture.

The contribution of OQL is to show a basis for object-based query processing in an environment supporting multiple parallel implementations of object and relation interfaces.

The novel contribution in this work is a framework with multiple parallel implementations for both object and relation interfaces, and a query processing strategy that utilizes this.

OQL is based on the REBOOT AQL [OH91], and extends it in four areas:

1. Objects as values
2. Query Arguments
3. Nested queries
4. Operations
These extensions are not unique, and may also be found in other object-oriented query languages, such as ObjectStore DML [LLow91] or OSQL [FBC+87]. These languages do not, however, provide the necessary support for attributes, relations, and separation of interface and implementation.

The separation between interfaces and implementations in COOM provides a basis for utilizing different local query-processing strategies for different implementations.

An implementation of an OQL compiler/interpreter and query processor for the AQL subset of OQL has been made by Tor Stenvaag [Ste92]. This work has shown that the query-translation approach is feasible for an OODBMS. Work in the SEROO-project showed this similarly for a RDBMS [FkJL+91].

OQL provides a solution for the requirements for query-support, RDS-9 and RDM-4, as stated in section 3.4. There are a number of challenging issues with respect to extensions to OQL and OQL Query Processing. Future work with respect to OQL and query processing is discussed in section 16.6.

The next chapter presents the COORASS diagram-technique and development methodology, and a specification of a COOP-compliant development environment.
Chapter 14

COOP Development Methodology and Support Environment

This chapter presents the COORASS methodology and a description for the CODE development environment, and shows how the behavioural and structural aspects of COOM described in chapter 10, 11 and 12 are supported at development time.

The COORASS methodology for the development of COOM-based systems is a structural extension to the behavioural-oriented OORASS methodology [RB+92]. This is similar to how COOMS provides a structural extension to the behavioural-oriented COOMB.

The CODE development environment description shows how the COORASS methodology might be supported through a COOP-compliant development environment.

14.1 COORASS - COOP Methodology

The COORASS methodology, COOM-extended OORASS, has been created as an object oriented development methodology that supports the development of COOM-based systems.

The motivation for COORASS has been to create a diagram technique and methodology that support the concepts in COOM, in particular both the behavioural concepts in COOMB and the structural concepts in COOMS.

Existing methodologies tend to be biased towards either a structural model or a behavioural model. Most of the most popular commercial methods, such as OMT [RBP+91], OOA/OOD [CY90] and OOA-IE [MO92] have taken a structural data modeling based approach. As described in chapter 4, these methods extends an ER-model with object-oriented concepts such as inheritance and operations.

A few methods, such as OORASS - Object Oriented Role Analysis, Synthesis and Structuring Method [RB+92], have taken a more pure object-oriented behavioural approach. The author was a member of the team that initially developed OORASS. Further development of the method and accompanying tools is ongoing at the Taskon-company, under the name OOram.

COORASS has taken the untraditional approach of extending a pure object-oriented behavioural-oriented method, OORASS, with concepts from ER-oriented data modeling,
14.1 COORASS - COOP Methodology

This is in line with how $COOM_B$ is extended with structural concepts in $COOM_S$.

14.1.1 Different User-roles

The COOP framework and COOM supports a new way of system building, by greater emphasis on reusing existing system-components. System building is essentially the process of gluing together pre-existing parts to fit new requirements. This system-building process involves a number of different user-roles.

- **End-users**
  
  End-users are the final user of tools. They might participate in the specification of functionality and user-interface.

- **Tool-builders**
  
  Tool-builders, or application-builders, are the providers of tools for the end-user. The tools are created by composing together different services needed by the user. The services are available through the COOM object model. The functionality of the services is offered through customized user interfaces.

- **Component-integrators**
  
  Component-integrators are those that adapt information and services from underlying components to the COOM object-model and introduce these into the integration framework environment. A Component-integrator might also install the components in the environment, or that might be handled by an Environment-administrator.

- **Component-builders**
  
  Component-builders are the providers of underlying general services, such as populated databases or computational services.

- **Framework-builders**
  
  Framework-builders are the providers of the integration framework.

The COORASS methodology is based on a structural extension to the generic OORASS methodology, and is further specialized for Tool-builders and Component-integrators.

14.1.2 OORASS - $COOM_B$ Modeling Support

OORASS focuses on the roles that objects play in a structure of collaborating objects. OORASS gives support for the behavioural part of COOM, $COOM_B$, from chapter 9 and 10. COORASS, COOM-extended OORASS, extends OORASS to also give support for the structural part of COOM, $COOM_S$, from chapter 12.

To facilitate flexibility and reuse, it is crucial that objects can play different roles in different settings. A role is characterized by a specific interface which an object playing that role have to support.

Three different abstractions of objects are supported by OORASS: The Why, the What and the How of objects. The Why is a new concept called a role, which represents the responsibility of the object within an organized structure of collaborating objects. The What is the interface (type) of the object, describing its externally observable behaviour. The How is the implementation (class) of the object.

The OORASS methodology is based on four basic ideas:
1. Letting *structures of collaborating objects* represent phenomena of interest.

2. *Generally useful objects*, which are objects that may be reused in a variety of different positions in many different structures.

3. *Different abstraction levels*, where *synthesis* is used to reuse collaboration structures, object interfaces and implementations.

4. *Late collaborator binding* is an extension of traditional late binding which facilitate dynamic creation of many different structures from predefined objects.

- *Structures of collaborating objects.*

![Role-diagram "Person-role"](image)

![Role-diagram "Employment"](image)

![Legend](image)

Figure 14.1: OORASS Role Diagrams

A system is viewed as a set of collaborating objects. An analogy can be drawn between this and an "ideal" bureaucracy as described by Max Weber [PS64]. Objects are like clerks in an organization, they are organized in a structure of collaboration, where each clerk has its own responsibility within the organization. A clerk has a position in the organization, an object plays one or more roles in the structure of objects. A clerk has a certain competence needed to fill his position, an object has an interface (type) that defines the behaviour it needs to play its roles.

Figure 14.1 shows three OORASS role-diagrams "Person-role", "Planning" and "Employment" which is used to describe structures of collaborating objects. The ovals are roles, while the small circles attached to the ovals are interaction-ports. One small single circle means that the attached role can interact with (send messages to) one object playing the role in the other end, while a small double circle means that it might interact with several objects in the other role. The messages which can be sent are annotated with a protocol-name, (r1, p1, etc.,) being described separately.

- * Generally reusable objects.
An object might be used on many different places in a structure of interacting objects, if it provides the interface required for each role it might play. The nature of the implementation does not matter as long as the required interface is provided.

- *Different abstraction levels* and the use of *synthesis*.

```

PClent
   p1

Department
   e1

Person
Employee
Resource
   d1

Activity
   rl

Project
   al

Figure 14.2: Synthesis of OORASS Role Diagrams
```

It is possible to combine different role-models into one model through synthesis. Figure 14.2 shows the three different role-models from figure 14.1 combined into one role-model.

The COORASS-extension allows for the description of a conformance-relation between two different roles, through a triangle-connection. This has the same semantics as overlaying two roles into a new role. This is supported through conformance between interfaces in *COOM_B*.

- *Late collaborator binding*.

An object might act as a given collaborator as long as it is capable of playing the expected role. The binding of collaborators should be flexible and done as late as possible. The binding of a given collaborator has three steps: Bind the collaborator's role to an interface, bind the interface to an implementation and instantiate the implementation. These bindings might occur at five different times:

1. At *role modeling time* - by specifying which interfaces collaborating roles should have.
2. At *interface specification time* - by specifying the expected interface of input and output arguments of operations.
3. At *compile time* - by specifying which interface to expect from objects used in the program-code.
4. At *configuration time* - by specifying a process-context which contains information about which implementation (object space) to use for the creation of objects with a certain interface in this local context.
5. At *run time* - by choice of implementation based on given object space or from context-information.

The four ideas are supported by *COOM_B* and the OORASS-methodology as follows:
14.1.3 From OORASS Role Models to COOMB ODL

The idea of describing structures of collaborating objects is supported by role-models showing the interactions between objects playing different roles. The description might be enhanced through message-sequence diagrams.

The idea of generally useful objects is supported by the possibility to specify the interfaces of objects.

The idea of different abstraction levels and use of synthesis is supported by the possibility to specify new interfaces as being conform to existing interfaces, by conformity inheritance.

The idea of late collaborator binding is supported by the separation of interface and implementation, and the possibility to choose implementation as late as possible.

14.1.3 From OORASS Role Models to COOMB ODL

OORASS role models can be used as a basis for generating object interface specifications in COOM ODL.

The current interfaces in COOM are one-way interfaces, describing which operations that are provided at an interface. There is no description of two-way interactions similar to contracts [Hol90]. An ideal interface should describe required two-way interactions. Separate work to address this is ongoing [Nor93, AR92]. Message-sequence diagrams can be used to show some of the dynamic aspects.

14.1.4 COORASS - OORASS Extended to COOMS Modeling Support

COORASS extends OORASS with the support for structural abstractions in the same way as COOMS extends COOM.

The OORASS-methodology is aimed at the specification of role-interaction, and there is no concept for structural abstractions, such as relations or attributes. The arguments for the use of structural abstractions in chapter 10 is also valid for a corresponding modeling methodology. In some application-areas the concept of attributes and relations is a natural modeling mechanism, in particular for database-oriented environments.

The conceptual understanding of attributes, relations and relationships is the same as described in chapter 11.

![Diagram symbols]

Figure 14.3: COORASS diagram symbols

Figure 14.3 shows the new diagram-symbols for attributes and relationships. A COORASS-diagram contains the necessary information to transform a schema into a corresponding
ODL-specification.

Attributes

Attributes are shown as a short line with a name at the end, connected to a role or a relation-symbol. The name is the attribute-name; the attribute-type is specified separately together with the separate description of operation-signatures.

Relations

A relation is shown as a diamond-square, which might have black triangles to show a "many"-end in the relation. The name of the relation is shown close to the relation-symbol. A circle on a relation means that the connected object knows about the relationship through the role-name for the opposite object. The role-names might be given close to the circle, or separately.

A containment-relation is shown through an arrow pointing towards the container object. Relationships with attributes are shown by attributes associated with the diamond-square.

Sub-Relations

Sub- and Super-relations are connected through a triangle-symbol with an enclosed s, for subset.

Interface Conformity - Synthesis

As an alternative to direct synthesis, it is possible to show the synthesis by a conformance triangle in the diagram.

Figure 14.4: COORASS
Figure 14.4 shows the COORASS-diagram corresponding to the ODL-schema of the generic example. The triangle in the connection between Person and Employee means that Employee is conform with Person. The triangle with an s between AssocWith and WorksFor means that the latter is a sub-relation of the former.

Mapping from COORASS to COOM ODL

The COORASS-concepts of roles and relations map into ODL object interfaces and relation interfaces. The operations required by a role is mapped into operations supported by the corresponding object interface.

14.1.5 Basic OORASS Methodology

![Diagram of OORASS Methodology]

Figure 14.5: The basic OORASS methodology

Figure 14.5 shows the basic OORASS methodology as described in [RB*92].

- **Role Modeling - Analysis and synthesis**
  Analysis is done by simple role modeling for the different areas of concern. Synthesis is composite role modeling, showing how an object might play different roles by combining simple role models.

- **Interface (Type) Specification**
  Composite role-models contain the necessary information to derive type specifications for the involved objects. A type describes the external observable behaviour of an object. A minimal description of a type can be given through a COOM ODL specification.

- **(Class) Implementation**
  The implementation step is to create the programs that implement the interfaces described in the previous step.

- **Structure Configuration**
  Object-interfaces and implementations can be made so that objects might be configured in a large number of different structures. Structure configuration makes it possible to restrict the possible structures to the ones that are meaningful to the user. The Taskon system [Tas93] supports this by a grammar-based information model that can be used to control object-instantiation.
• **Run-time support**
  The run-time support is provided by the framework-makers, through mechanisms to do remaining bindings between interfaces, implementations and instances.

The unique aspect of OORASS is the initial focus on role-modeling with analysis and synthesis. The focus is then strictly on interfaces, with no concern for implementation. The goal is to specify generic roles, such that one object might play different roles, and one role might be played by different objects. The basic OORASS-methodology is aimed at a the user-role of a general programmer. The COORASS-methodology specializes this to distinguish to separate between the user-role of component-integrator and tool-builder.

### 14.2 COORASS Methodology for the Component-integrator

![Diagram](image)

Figure 14.6: COORASS methodology for the Component-integrator

The COORASS methodology for a component-integrator shown in figure 14.6 specializes the generic OORASS methodology with steps for schema reconstruction, schema integration and enrichment and mapping implementation.

• **Schema Reconstruction**
  This step creates a COORASS role-model from the schema of the component to be integrated into the COOP Integration framework. Some more details about this step is given in section 15.2. This is in COORASS extended to also include the modeling of relations in addition to roles for objects. This step might be supported by a schema reconstruction tool.

• **Interface Specification**
  This step specifies the object and relation interfaces from the previous step in COOM ODL.

• **Schema Integration and Enrichment**
  This step compares the new interfaces with interfaces in existing schemas, to see if any object- or relation-interfaces should be integrated or if there are obvious possibilities for additional enhancement. This step could also be the responsibility of a person in the schema-integrator user-role.
• **Mapping Implementation**
  This step implements the necessary mapping-code from COOM-objects and relations to the underlying system. This step might be supported by a mapping tool for automatic creation of mapping implementation. Some more details about this step is given in section 15.2.

• **Configuration**
  This step provides necessary information for context-specific configuration. An end-user tool might be differently configured for different users with respect to which mapping-implementations to use, by getting different configuration-information from a context-file.

• **Runtime Support**
  The runtime support is provided by the COOP Integration framework.

The focus of a *component-integrator* is to integrate a new component into the COOP Integration framework. This makes the component available for the tool-builder, through the COOM-model.

### 14.3 COORASS Methodology for the Tool-builder

![Diagram](image)

**Figure 14.7: COORASS methodology for the Tool-builder**

The goal of a tool-builder is to create a tool that provides the functionality and information needed by the end-user, through a uniform user interface.

This implies methodology-support for presentation Integration Presentation integration as defined in 3.5 is the degree to which the actual set of tools provides access to the functionality needed by the user through a uniform look-and-feel interface. This was divided into two sub-areas:

*Display-orientated Presentation Integration*: the degree to which a common look-and-feel is provided by the tools which are used. *Model-orientated Presentation Integration*: the degree to which the functionality presented through the display is accessed and combined from one or more underlying functional models. A model is here defined as a functional part of a system, without any knowledge of input and output devices.

The COORASS methodology for a tool-builder (figure 14.7), specializes the generic OORASS model with steps for user interface specification, model specification, schema integration
and enrichment, and model and user interface implementation:

- **User-interface Specification**
  This step supports *display-oriented presentation integration*, by specification of the user interface part of the tool.

- **Model Specification**
  This step supports *model-oriented presentation integration* by specification of the interaction between the user interface part and model part of a tool. The interface of the model is specified in COOM ODL.

- **Schema Integration and Enrichment**
  This step compares the new interfaces with interfaces in existing schemas, to see if any object- or relation-interfaces should be integrated, or if there are obvious possibilities for additional enhancement. The step is similar to the same step for the component integrator.

- **Model and User-interface Implementation**
  This step implements the user interface part and model part of a tool, based on the specification from the first steps.

- **Configuration**
  As before.

- **Run time support**
  As before.

The focus of a *tool builder* is to create a tool by utilizing available functionality and information, provided by components that have been made available by the component integrator.

### 14.4 CODE - COOP Development Environment

The COOP architecture can itself be used as a basis for the realization of a development environment. The CODEDISC student prototype described in chapter 5 demonstrated that it was possible and feasible to create a development environment using the principles of a communication-based "software-bus" architecture.

The following specifies how the COOP architecture can be utilized in a development environment for cooperative systems, CODE - COoperative systems Development Environment. CODE is an extension to CODEDISC which focused on the development of client programs in a component-based software bus environment [FST+90, Ber92c]. The goal of CODE is to make it easy to develop COOP compliant systems. CODE uses the same principles as used in CODEDISC, extended with support for data integration.

CODE is specified to consist of a number of tools available for different users. Two main users are foreseen: The Component-integrator, who is responsible for plugging new components in and out of the environment and monitor the use of the environment itself and the Tool-builder who will be developing tools for an end-user, providing necessary functionality and data through presentation integration.

Figure 14.8 shows some possible tools for CODE. They will be described in the following, in the use-context of a component integrator or tool builder.
14.5 CODE - Support for the Component-integrator

The following are tools which mainly will be used by a person in the Component-integrator user-role.

- Schema Reconstructor and Mapper
- Client Generator
- Schema Integrator
- Component Installer

Short descriptions of these tools are given in the following.

14.5.1 Schema Reconstructor and Mapper

Schema reconstruction means "to find and/or recreate the necessary information for making a best possible semantic ODL schema". A schema reconstructor tool gets a schema-description for the underlying systems as input and produces an ODL-schema as output. The tool might also produce the necessary mapping-code for access to the underlying system.

There will be different schema reconstructors for different underlying systems. CODEDISC described in chapter 5 contains a OSF/DCE NIDL to C++ converter that produces a C++ schema from a NIDL schema. This tool can easily be changed to produce an ODL-schema, since the operations in a NIDL schema are compatible with the ODL operations. Other systems are OMG/ORB, RDBMSs, OODBMSs and network DBMSs.

There is ongoing work to produce a schema reconstructor and mapper for COOM and relational databases [rst93]. Much work to build on exists in this area [CS91, AR90, Ber91h],

Figure 14.8: COOP Development Environment
together with work on reengineering tools. Rules for translating relational schemas with constraints into object-oriented schemas are given in [Lin91].

Concepts for schema reconstruction and mappings between COOM and the models of different underlying systems will be described further in chapter 15.

14.5.2 Client-Generator

A client-generator will take an ODL component description as input and produce as output a component representation description for either C++ or Smalltalk. The output is in form of a class-definition, with necessary implementation.

The client-generator produces the C++ or Smalltalk framework-classes to be used by a tool builder. The client-generator might be a part of the schema reconstructor and mapper tool. The ODL-compiler [LS92] has been used as a basis for the generation of class-defintions in C++ [Sol92] and Smalltalk [Øst92a].

14.5.3 Schema Integrator

A schema integrator is used to create an integrated and unified view on two or more ODL schemas. This tool needs to deal with the multi model issues discussed in section 12.6.2.

There is ongoing work to produce a schema integrator tool for COOM in [Kor93, Kin93].

14.5.4 Component Installer

This tool is for actually installing and testing of new components, for removing or replacing components, and for general administration of the environment.

One version of such a tool was created for the CODEDISC environment described in chapter 5.

14.5.5 Component-integrator Support in the different COORASS Steps

The different tools might be used as follows in the different steps in the COORASS methodology for the component integrator.

- \textit{Schema Reconstruction} uses Schema Reconstructor and Mapper.
- \textit{Interface Specification} uses uses Schema Browser Editor\footnote{Schema Browser/Editor is described in section 14.6.2.}.
- \textit{Schema Integration and Enrichment} uses Schema Integrator.
- \textit{Mapping Implementation} uses Client Generator or implements manually.
- \textit{Configuration} uses Component Installer.
- \textit{Run-time support} uses COOP Integration Framework.
14.6 CODE: Support for the Tool-builder

The following are tools which mainly are to be used by a person in the Tool-builder user-role.

- User-Interface Development Tools
- Schema Browser/Editor
- Schema Tester
- Integrated Programming Environment (C++ and Smalltalk)

Short descriptions of these tools are given in the following.

14.6.1 User-Interface Development Tools

These are tools to aid in developing the user interface part of a tool accessing functional service-components. The preferred approach for COOP is to use an object-oriented application framework, with interactive tools for the specification of the layout of the user interface.

A number of such tools already exists in the market, and experiences with some of these have been described in chapter 7. The CommonView 3.0 application framework is used for C++, while the Smalltalk 4.0 Model-View-Controller framework is used for Smalltalk. These tools support both the development of Windows and X/Motif user interfaces.

14.6.2 Schema Browser Editor

A Schema Browser Editor is the main tool for managing ODL-schemas. This tool presents the content of the Object Dictionary through a browser that lets the user select and edit ODL schemas, object interfaces and relation interfaces. One version of this tool, the Component Browser, was created in CODEDISC and described in section 6.2, section 8.5 and figure 8.2. This tool supports the editing of schemas specified in NCS NIDL, but could similarly support COOM ODL.

It is possible to extend this tool to support graphical diagrams also, for instance by making an extension to the OOram tool from Taskon that already supports the basic OORASS notation.

14.6.3 Schema Tester

A schema tester can be used for interactive testing and experimentation with available objects. The test-environment can be automatically created, based on the ODL-schema.

One version of this tool, the Component Tester, was created in CODEDISC and described in section 6.2, section 8.5 and figure 8.2.

14.6.4 Integrated Programming Environments (C++ and Smalltalk)

In order to program mapping-implementations and tools, there is a need for a good programming environment. The ODL-schemas will be made available in C++ or Smalltalk.
A number of different programming environments for C++ and Smalltalk exists, and can be used together with CODE.

The COOP prototypes have been using the ObjectWorks environment for Smalltalk and the GNU environment for C++.

### 14.6.5 Tool-builder Support in the Different COORASS Steps

The different tools might be used as follows in the different steps in the COORASS methodology for the tool builder.

- *Model Specification* uses Schema Browser/Editor and Schema Tester.
- *Interface Specification* uses Schema Browser/Editor.
- *Schema Integration and Enrichment* uses Schema Integrator.
- *Configuration* uses File-editor for context-file.
- *Runtime support* uses COOP Integration Framework.

### 14.7 Contribution and Conclusion

This chapter has described the COORASS methodology and a specification for a COOP-compliant development environment, with particular emphasis on support for the component integrator and the tool builder roles.

The novel approach in COORASS is to extend a pure object-oriented behavioural-oriented method, OORASS, with concepts from ER-oriented data modeling. This provides support for the concepts in both COOM$_B$ and COOM$_S$. The motivation for extending from an object-oriented model is that an object-oriented model with operations provide an encapsulation that facilitates for multiple implementations, and hiding of underlying mappings and caching-strategies. Examples of approaches that uses an ER-oriented basis normally put less emphasis on encapsulation, and tend to think about physical attributes and relationships, instead of abstract structures. These approaches do not have a basis in interaction modeling, such as OORASS, and are less suited for modeling of distributed systems. OORASS on the other hand provides only support for interaction modeling. It is possible to model attributes as pairs of operations, relations as separate objects with predefined interfaces, and relationships as operations with Set-arguments. This, however, puts the focus on a lower level and hides the conceptual model of attributes and relations.

It has been shown how a development environment, CODE, can be realised as a toolset compliant with the COOP architecture itself. Particular emphasis has been given to the specification of tool-support for the component integrator and the tool builder roles. The contribution of the specification of CODE is to serve both as an example of a COOP-compliant working environment, and as an illustration of the functionality needed by a component integrator and tool builder.

The next chapter will illustrate the tasks of the component integrator and tool builder user-roles in more details.
Chapter 15

Component Integration and Presentation Integration

This chapter illustrates the intended use of COOP and COOM, by looking at the component-integrator role and tool-builder role, for the scenario described in section 9.8. Presentation integration and tool-builder support is illustrated through an example of a part of a resource planning tool, that also shows the principal use of COOM.

15.1 A Scenario for the Use of COOP and COOM

A scenario for the use of COOP and COOM was described in section 9.8. The scenario described the problem of component integration of a RDBMS, an ODBMS, and a project management service component. This chapter will describe the tasks of the component-integrators and the tool builder for this scenario.

The goal of the component-integrator is to introduce a new component into the integration framework. This is done by using the support for control integration and data integration provided by COOM, as described in chapter 10 and 11. Figure 15.1 shows that component integration might be done with different degrees of support. General aspects of schema reconstruction, mapping and object-identity issues are discussed first, then examples of integration-strategies for the different categories of systems are given. Examples of this is given for relational DBMSs, object-oriented DBMSs and large-grained and fine-grained functional services.

The goal of the tool-builder is to provide a new tool, by utilizing services provided by the integrated components, and present this tool through a uniform user interface. This is done by creating a tool that consists of a user-interface part, for presentation integration, and a model-part that extracts the necessary functionality from underlying services. The scenario case is to create a browser tool that allows for selection of departments, to get information about employees working in that department. The browser is a user interface component and also demonstrates presentation integration. The last part of the chapter describes how a tool-builder can do presentation integration, and use the COOP integration framework to access information and functionality provided by the components that the component integrator has made available.
15.2 Component Integration

Component integration focuses on how to integrate a new component into the COOP integration framework. The steps in the component-integrator process was described in the previous chapter.

This chapter will focus on issues related to schema reconstruction, mapping implementation and schema integration and enrichment. The issues will be discussed with respect to experiences with component integration for a relational DBMS, an object-oriented DBMS, and a functional service based on OSF/DCE.

Schema reconstruction and mapping is done by representing the provided services of the component in a COOM ODL schema, and to create a mapping between corresponding objects and relations in the COOP object space, and in the component. The generation of the ODL schema and necessary implementation-code is done through a schema reconstruction process, followed by a mapping-process.

Figure 15.1 shows three different alternatives for doing schema reconstruction and mapping. These alternatives are described in the following.
15.2.1 Manual Schema Reconstruction and Mapping

The first step when integrating a new component into the COOP integration framework is to create an ODL-schema, this will reflect the object interfaces and relation interfaces to be used. Schema reconstruction can be done manually by inspecting the interface and schema of the component to be integrated. This is called schema reconstruction, because some semantic information might not be represented in the interface or schema, and needs to be reconciled by consulting other sources. Information about the mappings between the existing schema and the ODL schema needs to be kept, for the process of doing mapping-implementation.

The derived ODL-schema is then run through the ODL-compiler, which generates interface-classes and skeleton implementation-classes in the desired language, i.e. C++ or Smalltalk. These classes serve as a basis for manually implementing the necessary mapping-code, in the body-part of the predefined operations.

The interface-classes will be used for establishing interface manager-objects in the Object Dictionary, and serve as the abstract interface classes to be used by the tool-builder/application programmer. The implementation-classes will be used for establishing object- and relation implementation managers and object implementations, in the local object space to be used for this component.

15.2.2 (Semi)-Automatic Schema Reconstruction and Mapping

A schema reconstruction and mapping process might be (partly) automated for certain categories of components. For often recurring categories of components, where the same reconstruction and mapping principles are used many times, this will be quite useful. Examples of such categories are relational database systems (RDBMSs), object-oriented database systems (OODBMSs), and remote-procedure-call-based services.

The second alternative in figure 15.1 shows a schema-reconstructor tool. This assists in schema-reconstruction and provides the ODL-schema and mapping-information which the ODL-compiler needs, to automatically generate interface-classes and implementation-classes with the necessary mapping-code.

15.2.3 Generation of New Implementations

Systems integration is based on utilizing existing services and database-systems. For certain categories of systems, such as database-systems, it is also possible to generate new implementations based on information in the ODL-schema.

The third alternative in figure 15.1 shows an ODL-compiler with a backend, that generates a suggestion for a schema for the underlying system, and mapping-code that matches this suggestion. This is for instance possible for both relational and object-oriented DBMSs, for the parts of the schema that represents objects, attributes and relations and default operations. It is not possible to generate implementations for user defined operations, as the operations-definitions do not contain information about the semantics of these operations.
15.2.4 Mapping Principles from and to Different Systems

The tool-builder, or application-programmer, will write code that uses the object interface managers and relation interface managers stored in the Object Dictionary. These will provide the interface to the actual objects and relationships. The tool-builder might activate and passivate different object-spaces, but do not have any direct knowledge of the different object implementation managers and relation implementation managers.

![Diagram of Object Dictionary and local object spaces]

Figure 15.2: Object Dictionary and local object spaces

The component integrator is responsible for integrating new components into a local object space, by providing necessary mappings between COOM and the local systems in implementation classes. Experiments have been carried out for mappings from and to a RDBMS, an OODBMS, and service-components reachable through remote procedure call mechanisms. Mappings to/from COOM and Local data-models and services/applications are realized as an implementation of the specified interfaces. The mappings are handled through object implementations, object implementation managers, and relation implementation managers in the local object spaces.

Figure 15.2 shows the Object Dictionary object space, the Integration Space, and three local object spaces for a RDBMS, an OODBMS and a project-planning service. The Object Dictionary object space contains the object and relation interface managers.

15.2.5 Object Interface Management

The task of the object interface manager is to manage a set of object implementation managers in different local object spaces. The available operations have been described
in section 11.2.1. The most important operations with respect to interaction with the underlying system are `extent`, `find` and `create`.

If there exist multiple implementations of the same interface in parallel, this might mean that the total extension of an interface is the extensions of all implementations (and eventually the extensions of its possible subtypes). A query sent to the type will then be delegated to all implementations, e.g.:

```c
foreach( impl in implSet)
    { result ->add( impl->select(query)); }
```

The object interface manager will manage objects with a certain interface. The object interface is the type that the tool-builder assumes the object to have. The available operations of all objects have been described in section 11.1.1. The most important operations with respect to interaction with the underlying system are attribute-operations and user-defined operations.

It is possible to access relationships directly from the participating objects, if role-names are defined. The definition of role-names in a RELINTERFACE-specification, will lead to the creation of relationship-access operations in the involved object-interfaces.

### 15.2.6 Relation Interface Management

The role of a relation interface manager is to manage a set of relation implementation managers. The available operations have been described in section 12.3.5. The most important operations with respect to interaction with the underlying system are `insert`, `getX` and `getIteratorX`.

The following sections will describe implementation-mapping strategies for different underlying systems, for the interfaces described above.

### 15.3 Object Identity and Autonomy

The use of COOM as a canonical data model provides objects with a strong identity. Identity is the property of an object which distinguishes it from all other objects [KC86]. In COOM this is inherent in each object. When COOM is used to interoperate and integrate with existing systems, it becomes a question how object identity is maintained in the underlying systems.

The different underlying systems might have different ways of identifying objects. This leads to a two-level OID-scheme where an identifier consists of a global part and a local part. The global part contains information about which object-space, interface and implementation the object belongs to, while the local part consists of necessary information for local identification. The local identifier might be different for different implementations. This means that there will not be one OID-structure common for all objects, instead each implementation will choose its own local structure, based on what is natural for the underlying system. However, it is useful to be able to externally store and communicate object-references for later retrieval. This is done through an OIDString, where each implementation supplies a mechanism for converting the internal representation to and from a string-representation.

Typically for a relational system, the local identifier will be the database-name, table-name and primary key for the table. For a CODASYL-system it might be the address
(File, Block, Record) as a hint and a unique name in addition. For a foreign object-oriented database it might be the OID used in that system. However, not all OODBMSs support OIDs externally e.g. in ObjectStore an external OID can only be provided by giving the object a name. In the OMG Object Request Broker (ORB), the ORB will support object-references (objref...) which can function as OIDs.

In [EK91] a distinction is made between three different support levels of object identity that may be provided to the federated system by component systems. These are permanent (immutable) object identifiers, value-based identifiers and temporary (session-based) object identifiers:

- **Permanent identifier**
  A Permanent OID is a global immutable identifier that will not change during the lifetime of an object. This is a secure identifier that always can be used to retrieve the object.

- **Value based identifier**
  A value-based identifier is based on the value of a key-attribute, and the identifier will change if the value is updated. This is an unsecure identifier if no precautions are taken. Secure object identification at the global level can be provided if components are willing to relax their requirement for update autonomy. A component has update autonomy if it may unilaterally make updates to its information. Either this can be done by not allowing updates of key-attributes, or by keeping the global level informed about when key-attributes are updated. If the global level is informed about changes, it might use that knowledge to update the local identifiers used.

- **Temporary identifier (Session-based identifier)**
  A temporary identifier might change between sessions. This is an unsecure identifier if no precautions are taken. It is an interesting observation that not all object-oriented database-systems support the strong notion of identity, because they do not make the identifiers available. E.g. ObjectStore [LLOW91] and O2 [B+91a]. For session-based identifiers it is necessary with an extra mechanism to make the identifiers secure. A possible solution is to introduce a dictionary-table in the local system, which provides a mapping between external names and internal identifiers.

A strong notion of identity is required at the global level, if there is a need to make stable relationships between different objects. The OIDString serves as an external name for identifying an object. The COOP prototype uses an object-oriented database-system as a basis for the persistent object space. The tool-builder will normally use the session-based object-identification-scheme provided by this system, but needs to use the OIDString if an object-reference is to be communicated between sessions.

In [HD92] it is suggested that instead of requiring strong identity, one could make a distinction between safe and unsafe use of identifiers. This can be supported by limiting the available functionality, to operations that do not violate a weaker identity. Temporary identifiers might e.g. allow for the creation of relationships between objects in the same object space, but not if they are in different object spaces. It might be system-dependent, if operations like create, delete and update can be supported. This could be reflected in an object interface-hierarchy, where generic operations like create, find, delete and update are incrementally supported. Another group of identifiers, imaginary identifiers, is introduced for objects where no direct counterpart exists in the underlying system. These objects are imaginary in the sense that they have been created based on underlying aggregated information, but cannot directly be created or updated.
The discussion on object identity showed a need for relaxing local autonomy in certain situations, to assure secure object identifiers. There is a conflict between the participation in a federation of components in a COOP integration framework, and having full autonomy for the components. Autonomy is the possibility to unilaterally change behaviour and policies in a certain areas. COOP impose restrictions on autonomy by e.g. requiring that the underlying interface-schema shall remain unchanged and by not allowing for updates of value-based identifiers, or by requiring that such updates should result in notifications. If full autonomy is required, it could be provided at the expense of the usability of the objects.

15.4 Component Integration for Relational DBMSs

Relational database systems represent information as tuples stored in tables. The interface to these systems is by giving SQL-statements as input, and receiving tables as output. Some relational database systems also provide access through stored procedures, where the input is arguments to predefined operations.

A relational DBMS can be integrated as a component at two levels. The first level is to view the relational DBMS itself as an object, with operations equal to the interface provided for the relational DBMS. This is similar to the interface-strategy used for remote-procedure-call based components and is further described in section 15.6. The interface can then either be a SQL query-interpreter, or a set of schema-specific routines realized as stored procedures.

The second level, which is targeted here, is to hide the component level, and instead provide access to the information stored in the system as a set of fine-granulated objects and relations. The implementation of the second level will typically use the functionality specified in the first level, but the first level will be hidden from the application-programmer.

15.4.1 Experiences with Relational DBMS Integration

The principles of mapping between COOM and relational systems have been tried out through experimental work reported in [Low92]. This was an extension to the work done in the SEROO-project [FkJL91], which again was based on an early version of the ODL-compiler [Ber91g]. This work has concentrated on mapping-strategies between COOM and relational schemas, and on the mapping between OQL and SQL. The experiences from this work is that mappings between COOM and relational schemas are feasible, but that it is important to pay attention to flexible cache-strategies in order to achieve high performance.

15.4.2 Relational DBMSs and Object Identity

A relational database system is an example of a system where value-based identifiers are used. In order to map this to objects in COOM, it is necessary to restrict the autonomy of the relational system, by disallowing changes to tuple-keys. Eventually, all changes of tuple-keys should result in notifications of change to the COOP integration framework.
15.4.3 Schema Reconstruction and Mapping for Relational DBMSs

In order to discuss the mapping between COOM ODL and a relational database system, the object interfaces Person, Employee and Department, and the relations AssocWith and WorksFor are again used as examples.

```sql
create table persons
(persnumber int not null,
 name varchar(50),
 age int,
 dnumber int /* assoc_with */ )

create table employees
(persnumber int not null,
 name varchar(50),
 age int,
 salary int )

create table department
(dnumber int not null,
 name char(12),
 )

create table works_for
(persnumber int,
 dep int,
 Date startDate
 )
```

Figure 15.3: Integration of a Relational DBMS

There are many ways to represent a conceptual schema in table-form in a relational DBMS. A schema reconstruction process needs to reconstruct the information which is lost in the design process, from a conceptual schema to a relational schema in third normal form.

Figure 15.3 shows how the local object space is related to the underlying component. The database in this case contains information about persons, employees, departments and their relationships. Figure 15.2 shows how the Object Dictionary contains interface manager objects for the management of implementation manager objects in the different local object spaces.

Some particular design choices which will have been made, include how to represent relations, how to represent generalized/specialized objects, and how to represent aggregate objects. In the example in figure 15.3 the 1:N relation associated_with is represented as an attribute, dnumber, in the person table, while the 1:N relation works_for is represented in a separate table.

Each object- and relation implementation manager and each object, will have the necessary knowledge about the underlying representation to provide the required functionality, as discussed in the following.
15.4.4 Object Implementation Management for Relational DBMSs

The task of the object implementation manager is to manage persistent and active representation of the objects.

The operations to be provided by an object implementation manager, are described in section 11.2.1. The most important operations with respect to interaction with the underlying system are `extent`, `find` and `create`.

Extent and Find Operations

The `extent`-operation will retrieve all objects with a certain implementation. This can be done by doing a Select-query on the table(s) representing the relevant information for the object.

The management of subtypes is done at the object interface level. It is assumed that an extent-operation only returns the direct extent of that object implementation. In the example of finding the extent of Person, the object interface manager will add the extent of Employee since that is a subtype of Person. It might have been the case that the tables in the relational databases was modeled differently, so that the Person-table also contained the identifiers of the subtypes. The select-operation should then do a difference between the Person and Employee tuples, or else the Employee-objects would be created as Persons.

```
SELECT p.persnumber, p.age, p.name
FROM Person p
```

This will result in a buffer-table, with the existing local identifiers and possibly relevant attributes and relationships. For each tuple a COOM-object will be created if it not already exists. The local identifier is stored in the private instance-variable localID in each created object. Attributes might be stored in each object together with a reference to the transaction-object representing the transaction, when the attributes were cached. Alternatively the attributes might be kept in a global bufer, with the localID as an index. The cached values are only valid within a transaction, so the current transaction needs to be compared with the caching-transaction to check if the values are safe to use.

Existing COOM-objects, which are not found in the local system through an extent-operation, can be assumed to have been deleted in the local system, and can then also be deleted at the COOM-level. The extent-operation will only return objects which currently exist in the local system.

Which information to retrieve, depends on the chosen cache strategy. It is useful to cache as much as possible of the information that will be used. A cacheHint operation is provided by each interface manager. This operation provides the interface manager with knowledge about which information the tool-builder intends to use. The hint-string may contain names of attributes and relations that it is planned to use.

The result of an extent- or find-operation will be a set of objects. It is possible to postpone the activation of objects, and delegate this responsibility to the set-object itself. This can be done by creating a special implementation of Set for each object implementation, in this relational database-system. The Set-object will be initialized with a declaration of a cursor:

```
EXEC SQL
DECLARE CursorX FOR
```
SELECT p.socsecnr, p.age, p.name
FROM Person p

OPEN CursorX

EXEC SQL FETCH CursorX INTO :xx

Each next-operation on the Set, would result in a new fetch by the cursor. This will be followed by a check if the corresponding object already exists, otherwise the object will be created.

Sometimes the reason for executing an extent-operation is that one would like to traverse through all the objects. This might be done sequentially, without any need for going back to previous objects. As an optimization, it is then possible to reuse the same local object as a buffer for the information retrieved from each fetch. This avoids the overhead of reusing internal space, for objects that will not be used any more. This gives a basis for a special set-iterator implementation which for each next operation, uses the same object as a buffer. This, however, violates the assumption that all objects are unique, and needs to be handled with care. A special extent-operation, extentMulti, could be defined to handle this case. If the implementation of all relationship operations always uses the OIDString as a basis for the representation, this will avoid the situation of referring the wrong object.

The find-operation will be similar to the extent-operation, with the extra facility for filtering the retrieved objects, based on a query-predicate expressed in OQL.

The query-operation find will be invoked with a parsed query-tree as an argument. This needs to be transformed to a local SQL-query on the underlying tables.

Q: Find all employees of age between 20 and 30, working in the Computer Science Department:

OQL: Set<Employee> s = metaEmployee->find("age >= 20 && age <= 30 && worksFor.name == 'Computer Science' ");

SQL:
SELECT p.socsecnr, p.age, p.name
FROM Person p, Department d
WHERE p.age >= 20 and p.age <= 30 and
    p.deptnr = d.deptnr and
    d.name = "Computer Science"

The table resulting from the query can be treated as the result from an extent operation, as discussed in the previous section.

The Object Dictionary contains the necessary information about the defined object- and relation-interfaces. The information needed in a query-mapping, will be found through an implementation definition associated with each implementation manager object. This will be used to dynamically find the local names and mappings for the names used in the query.

Create Operation

A create-operation will be transformed to an insert-operation on the appropriate table(s). Initial values for primary key, and other attributes, are provided as input-arguments to
the create-operation.

A COOM-object with the primary key as a local identifier will be created and returned. This will happen within a transaction on the underlying system, and the newly created object will not be visible for others until the transaction is committed\(^1\).

### 15.4.5 Object Implementation for Relational DBMSs

The object implementation implements each object. It contains the necessary code for dealing with the object-specific operations, such as attribute access and caching. How this is done depends on the caching strategy chosen by the object implementation manager. Caching might be done to avoid to always retrieve information from the underlying system. This needs, however, to be synchronized with the transaction mechanism, or some kind of change/update mechanism to remain consistent.

The most important operations are attribute-access operations and user-defined operations.

#### Attribute Operations

The attribute operations get and set the values of each attribute.

For instance, say that the reason for selecting the persons, was that one would like to display the name and age of each person, as in the following iteration-loop:

```c++
for (p = s->first(); p; p = s->next())
{
    cout << "name: " << p->name() << " age: " << p->age() << "\n";
}
```

This requires access to the name- and age-attribute of each selected person. The default approach will be to fetch the necessary information each time it is required. But this might be optimized through caching.

```c++
name()
if cache-name != nil and (current.Transaction == cache.Transaction)
return cache-name
else
    "SELECT p.name
     FROM Person p
     WHERE p.sec = socsecnr"
    cache-name = name
    cache.Transaction = Current Transaction
    (invalidate older cache-info)
```

The setting of new values to an attribute, will result in an insert-operation on the respective table. This will be done within a transaction on the underlying system.

\(^1\) More sophisticated transaction-mechanisms might deviate from this. This is described more in section 11.3.2.
User-defined Operations

It is difficult to give general support for user-defined operations, since RDBMSs do not support operations. Such operations might be supported by a stored procedure facility provided by some RDBMSs.

15.4.6 Relation Implementation Management for Relational DBMSs

The role of a relation implementation manager is to manage a set of relationships. The implementation of a relation implementation manager contains the necessary code and representation-structures, to efficiently deal with management of relationships between objects.

A relation might be represented in different ways in a relational schema. 1:1 and 1:N relations might be represented in the same table as the entity with cardinality 1, while M:N relations require a separate table. Attributes might be added in both representations.

Relationships are not objects. The internal representation of relationships in a relation implementation manager, will depend on how the relationships are managed in the underlying system. In relational systems they are represented in tables, and a natural mapping would be to have a cache of the relevant parts of the table. That table will contain localIds which needs to be converted to object-references, when provided as the result of operations.

The most important operations are insert, getFrom/getTo and getIteratorFrom/To operations. The first and last of these are discussed in the following.

Insert Operation

The insert-operation in the relation interface manager, will check on the object-spaces of the objects, for which a relationship shall be created. If the relationship is to be created between two objects in the same object space, the insert-operation will be delegated to the relation implementation manager for that object space, if it exists:

\[ \text{WorksFor} \rightarrow \text{insert} (e1, d1, aw); \]

The implementation of the insert-operation will convert the input-objects to their localIds, which will be similar to the keys that will be used as table attributes.

\[
\text{EXEC SQL}
\]

\[
\text{INSERT INTO works_for (pers, dep, date)}
\]

\[
\text{VALUES (:e1->localId, :d1->localId, :aw.date)}
\]

GetIteratorTo Operation

The getIteratorX-operations are getIteratorFromTo, getIteratorFrom, getIteratorTo. These operations return an iterator-object instead of a set of from- or to-objects.

The following getIteratorTo-operation results in an iterator used for finding the name and startDate, for all employees working in a particular department d1. The iterator provides the operations described in section 12.3.5.
RIT<Employee, Department, AWorksFor*> rit = Works_for->getIteratorTo(d1);

WHILE (!rit->atEnd())
{
    cout << "name: " << rit->from->name() << " startDate"
        << rit->a.startDate() << "\n";
    rit->next();
}

The getIteratorTo operation returns a new RIT ingress WorksFor instance, conform with the type RIT<Employee, Department, AWorksFor*> , which uses an SQL-cursor to manage the iterations.

EXEC SQL
   DECLARE CursorY CURSOR FOR
   SELECT w.persnumber, w.startDate
   FROM works_for w
   WHERE w.dep = :d1->localID
EXEC SQL OPEN CursorY
EXEC SQL FETCH CursorY INTO :xx

Each new fetch-operation will fill the iterator-object with a person-object, from, a department-object, to, and an attribute-structure, att - based on the retrieved values. In this case, it is an interesting question if any attribute-values should be retrieved for the person-object. This would have required a join with the person-table.

Such incremental retrieval would also be quite convenient, if the underlying DBMS was a network-DBMS and the set represented the relationships of a certain type, going from an object. In the underlying network-dbms this would typically be represented by a CODASYL-set construct, which is efficiently traversed in sequence, but not so efficient if each participating object is accessed directly by its unique key/name.

15.5 Component Integration for Object-Oriented DBMSs

There is a number of object-oriented database-system on the market, as described in chapter 7. They have been divided into client-oriented or server-oriented, based on their architecture. They can further be divided into language-specific, or language-generic, dependent on how they are coupled with different programming languages. Their datamodel is typically based on the datamodel of an object-oriented programming language, like C++ or Smalltalk.

15.5.1 Experiences with Object-Oriented DBMS Integration

The principles of mapping between COOM and an object-oriented DBMS, have been tried out through experimental work reported in [Tin92, Ste92, So92].

This work has been using the client-oriented C++-based ObjectStore system. In [Tin92] it was shown how ObjectStore could be used as a target-system for the REBOOT datamodel. COOM uses this datamodel as part of the its structural model COOMs. In [Ste92] it has
been shown how an OQL-query can be translated to a query on an underlying system, in particular the ObjectStore DML query expressions. In [Sol92] it was shown how the ObjectStore system could be used to manage the COOP Integration framework.

15.5.2 Object-Oriented DBMSs and Object Identity

It is interesting to note that although the object-oriented model inherently supports object-identity, many of the current object-oriented database-systems do not expose this identity externally to the system.

Objects can, however, be given names which can be used to uniquely retrieve objects between sessions. This requires an extra dictionary-mechanism, which maps from names to objects. Most systems support this, but it requires some overhead compared to going directly through the local pointers.

15.5.3 Schema Reconstruction and Mapping for Object-Oriented DBMSs

The data definition language for many object-oriented database-systems, such as ObjectStore, is based on C++. The mapping from the schema to COOM ODL then equals a mapping from C++ to ODL.

The most notable difference with respect to COOM, is that these models do not have any explicit construct for relations or relationships. Objects are referencing each other through instance-variables with pointers. In addition to having relations as an explicit construct, COOM allows for the use of role-names, so that relationships can be requested directly from the participating objects.

The example to be used to illustrate OODBMS reconstruction and mapping, is the same schema as used for the RDBMS, creating a situation where objects and relations with the same interface might exist in either an OODBMS or a RDBMS.

Figure 15.4 shows that there is one ObjectStore C++ class for each ODL Object interface, the AssociatedWith relation is represented through a pointer in the Person-class, while the WorksFor relation is represented through a separate class.

Each object- and relation implementation manager, and each object, will have the necessary knowledge about the underlying representation, to provide the required functionality, as discussed in the following.

15.5.4 Object Implementation Management for Object-Oriented DBMSs

The object manager-operations extent, find and create are discussed in the following.

Extent and Find Operations

The extent-operation will retrieve all objects with a certain implementation. This requires that the underlying representation is able to provide the set of all instances of a particular representation. The ObjectStore system do not have an automatic support for extent management. Extent-management is normally added by the application-programmer, by the creation of a persistent set of objects, which is maintained for new and deleted objects. This means that the management of subtypes also needs to be explicitly handled.
Since ObjectStore also is used as the system for the internal COOP object support, it is possible to use the ObjectStore classes directly as implementation classes. In the general case, however, the object implementation classes will result in instances that references the objects in the OODBMS, through an external reference mechanism provided by, or created for, the OODBMS. ObjectStore supports a client-oriented architecture, which means that the referenced objects will be cached in the client work-space.

The find-operation will be similar to the extent-operation with the extra facility for filtering the retrieved objects, based on a query-predicate expressed in OQL. This will be transformed to an equivalent query on the underlying system, in this case expressed as an ObjectStore DML query-expression (OS query).

Q : Find all employees of age between 20 and 30, working in the Computer Science Department:

OQL: Set<Employee*> s = metaEmployee->find("age >= 20 && age <= 30 && worksFor.name == 'Computer Science'");

OS query: os_Set<OSEmployee*>* os = OSEmployee::extent().query("OSEmployee e="", e.age >= 20 && e.age <= 30 && e->deptHired->dept->name == "Computer Science");
The OS query is issued on the extent of all OSEmployee instances, and will return a set of OSEmployees, which will be referenced from corresponding EmployeeOS-objects in the ObjStore1 object space. The extent operation, or a similar facility for retrieving instances, needs to be implemented by the classes that shall participate as full object interface types.

Create Operation

A create-operation will be transformed to a new operation for the appropriate ObjectStore class, in the appropriate ObjectStore database. A COOM-object with an ObjectStore OID as a local identifier will be created and returned.

15.5.5 Object Implementation for Object-Oriented DBMSs

The object implementation contains the necessary code for dealing with the object-specific operations, such as attribute-access and user-defined operations.

Attribute Operations

The attribute operations are for getting and setting of the values of each attribute. The setting of new values to an attribute will result in an update to an instance-variable, or in an operation on the corresponding object in the OODBMS.

User-defined Operations

The user-defined operations will be mapped one-to-one between the COOM-objects, and the objects in the OODBMS, so this is handled by a direct delegation from the COOM-object to the corresponding OODBMS-object.

15.5.6 Relation Implementation Management for Object-Oriented DBMSs

The role of a relation implementation manager is to manage a set of relationships. In an OODBMS this is typically handled directly as pointers between objects. If relationships with attributes are to be supported, they need to be created as separate objects.

The relation operations are insert and getIteratorTo are discussed in the following.

Insert Operation

The insert-operation will know that this is an insertion of two objects in this object-space.

\[
\text{WorksFor}\rightarrow\text{insert}(e1, d1, aw);
\]

The implementation of the insert-operation will convert the input-objects to their localIds, which will be the external OID, localId, used for the OODBMS. In the case of ObjectStore it will be a reference to the ObjectStore object itself.
Hired* h = new(ostoreDB) Hired();
h->startDate = aw.startDate;
h->emp = e1->localId;
h->dept = d1->localId;

e1->localId->deptHired = h;
d1->localId->employees->add(h);

The WorksForOS object will know the local representation of relationships, and create the necessary connections, as shown above.

GetIteratorTo Operation

The following getIteratorTo-operation results in an iterator that is used for finding the name, and startDate, for all employees working in a particular department. The iterator provides the operations described in section 12.3.5.

RIT<Employee, Department, AWorksFor>*
    rit = Works_for->getIteratorTo(d1);

    rit->first();
    WHILE (!rit->atEnd() )
    {
        cout << "name: " << rit->from->name() << " startDate"
        << rit->a.startDate() << "\n";
        rit->next();
    }

The getIteratorTo operation returns a new RITOSWorksFor instance, conform with the type RIT<Employee, Department, AWorksFor>*, which uses the ObjectStore representation of the relationships, when managing the iterations.

15.6 Component Integration for Large-grain Services

Functional service-components are typically provided through remote-procedure-call systems like NCS/NIDL [ncs89] or OSF/DCE [osf90], or through object-based systems like OMG/ORB [obj92a].

These systems offers remote access to executing processes, with interfaces that provide services through a set of procedure calls.

Large-grain functional services is the target-level for systems like OSF/DCE and OMG/ORB. A functional service is equivalent to an application with certain functionality, i.e. a spelling-checker or a mail-service. An RDBMS or OODBMS encapsulated as a server can also be viewed as a large-grain functional service. The interface to the service is then given by the set access-operations that is provided. The examples of component-integration for a RDBMS and an OODBMS, was done on a fine-grain level. It was the content within the DBMS that was made accessible, not the database-server itself. The large-grain service was hidden by the implementation of the fine-grain level. Also other large-grain services might be embedded this way, as discussed in the next section.
15.6.1 Experiences with Integration of Large-grain Services

The principles of mapping between COOM and functional services have been tried out, through experimental work reported in [FST+90, Øst92b, Øst92a].

This work has been using the OSF/DCE-compatible NCS/NIDL-system.

In [FST+90] it was experimented with a mapping between C++ and OSF/DCE. In [Øst92b] it was experimented with a mapping between COOM in Smalltalk and OSF/DCE. In [Øst92a] it was shown how to map between COOM in Smalltalk and OMG ORB.

15.6.2 Large-grain Services and Object Identity

A remote procedure call mechanism do not inherently support object identity. At the highest level, an object would represent a process that executes, and offers certain services. A process might be stateless, such as a numerical computation service, or state-dependent, such as a project-management-system. Object-identity is not particularly useful in a stateless service, since it would not be possible to distinguish different instantiations of the service.

Support-mechanisms like OSF/DCE Location Broker or OMG/CORBA Object Request Broker, offer support for matching and identifying requested services and objects with available offerings. These mechanisms make it possible to request a particular service, and to connect to that service.

Sometimes it is useful to expose the identity of information managed within a state-dependent service. Both OSF/DCE and OMG/CORBA have support for object-identity, through unique object-identifiers. A component-programmer might utilize this facility to expose objects with permanent identifiers, outside of a component. A request to a broker would result in an interaction with the required object.

15.6.3 Schema Reconstruction and Mapping for Large-grain Services

The mapping from interface-definitions in OSF/DCE, or OMG/ORB, to object interfaces in COOM ODL, can be done in a one-to-one mapping. The operation-declaration part of COOM ODL is initially based on the operation-declaration of NCS NIDL [ncs89], which was used as the basis for OSF/DCE.

Figure 15.5 shows a project-planning system, with a remote-procedure-call based OSF/DCE interface. The operations in the interface can be used directly, or be a basis for the creation of fine-grain functional services, as shown in the next section.

A Schema-reconstruction tool was created as part of the CODEDISC prototype described in chapter 5. This was the NIDL-C++ converter, which parsed NIDL-files and generated a C++ encapsulation of the interfaces.

An example of a component available through OSF/DCE is a project management system, with an interface for managing activities within a project. Parts of an IDL-specification of the interface is showed in Figure 15.5. Running this ODL-schema through the ODL-compiler will generate an interface-class and an object interface manager, and an implementation class and an object implementation manager. The operations for the implementation-class and object implementation manager will map to and from the underlying system, and need to be written particularly for this implementation. The interface-class acts as the type to be used from an application-programmer, while the object interface manager will be automatically generated.
15.6.4 Object Implementation Management for Large-grain Services

The object manager-operations extent, find and create can still be used, but might have a slightly different semantics. The normal operation mode for RPC-based systems is to identify a particular service, based on the interface that is needed, and then bind to this by getting the port-address used for communicating with it.

The extent-operation will retrieve all objects with a certain interface. This requires that the underlying representation is able to provide the set of all instances of a particular representation. It is possible to change the semantics for the extent-operation to return the set of all instances that are available, instead of all that exists.

The find-operation is similar to the extent-operation, with the extra facility for filtering the retrieved objects based on a query-predicate expressed in OQL. A find-operation could be extended with a possibility to do a query on system-related attributes, such as time and performance.

A create-operation will return an object that represents a functional service, with an interface described through the object interface type.

If the functional service do not maintain any state, e.g. such as a mathematical calculation service, it might not be critical to distinguish between two different processes offering the same service.

The object that is returned will contain necessary information, to be able to identify and
rebind to the same service on later requests. If the functional service do maintain a state, the RPC-system needs to provide information to reactivate a service with that state. This is provided through the unique identifier (uid) construct in OSF/DCE.

15.6.5 Object Implementation for Large-grain Services

The object implementation contains the necessary code for dealing with the object-specific operations, such as attribute-access and user-defined operations. The implementation will contain the necessary stub-routine code, to access the functional service. Such stub-routines will normally be automatically generated by a stub-compiler. In the case of interaction through OMG CORBA the IDL-compiler will provide this.

15.6.6 Relation Implementation Management for Large-grain Services

Relationships between components are normally not supported by a remote-procedure-call system like OSF/DCE. It is possible to maintain a relationship through a relation that is implemented in the Integration Space.

15.6.7 Conclusion and Future Work

The mapping between an object interface in COOM and a service provided through an interface in a functional service, provided through OSF/DCE or OMG/CORBA, is one-to-one.

The binding semantic is currently embedded in the create and find-operations. Binding through a predicate with non-represented attributes, such as time and cost, could later be done through the use of a new trade-operation, similar to the one found in the ANSA system [ans89].

15.7 Component Integration for Fine-grain Services

There are two possible levels for the integration of functional services. The first level is the large-grain integration of a component as one object. The interface of the component is then mapped to an ODL object interface, and the component is manipulated as one object, as shown in the previous section.

In some cases, a large-grain functional service provides the necessary basis for a fine-grained view.

An example of this is a project management system that contains information about projects with activities, and resources allocated to the activities. It is possible to also view the items within a large-grained component as objects with relationships between. In the project management system example just described, it could for instance be possible to view the activities as objects related to each other, and to resource-objects. The way to achieve this would generally depend on the functionality of the large-grained component, and would need to be done manually on a case-by-case basis.

A requirement for this is that it is possible to realize the required operations for each identified object and relation interface, through the large-grain interface. This is not true in general, so fine-grain support is limited to certain special cases. Note, however, that service components always can be trivially integrated at the large-grain level.
15.7.1 Experiences with Integration of Fine-grain Services

The work done with the integration of a RDBMS, is actually an example of an integration of a fine-grain functional service. The large-grain service is the SQL-level interface, that is accessed as part of the implementation of the fine-grain access. In the experiment-mapping, an embedded SQL was used, instead of interaction with a SQL server-object.

15.7.2 Fine-grain Services and Object Identity

The general discussion on object-identity is also valid for fine-grain services. In order to support a mapping to objects it is necessary to find, or create, a unique stable identifier. The way this is done will vary between different systems.

In the project management system example, unique identifiers are given by integer identifiers for projects, activities and resources.

15.7.3 Schema Reconstruction and Mapping for Fine-grain Services

![Diagram](image)

Figure 15.6: Integration of a fine-grained functional service component

Figure 15.6 shows a fine-granulated view on the project planning system. The object-interfaces Project, Activity and Resource are related through the relations Activity In Project, Resource In Activity and Activity follows Activity.

These are implemented by the use of the basic functionality provided through the PMS-service as described in the previous section.

15.7.4 Object Implementation Management for Fine-grain Services

The create-, extent- and find-operations will be implemented by a mapping to the equivalent functionality provided through the large-grain interface. Some mappings might
be trivial, while others might involve more work. A find-operation can be implemented by an iteration that checks on each object, if there is no support for queries.

15.7.5 Object Implementation for Fine-grain Services

An object implementation will be using the large-grain interface for implementing the mapping for the operations for each individual object.

15.7.6 Relation Implementation Management for Fine-grain Services

A relation implementation will be done similarly to an object implementation, based on functionality provided through the large-grain interface.

15.8 Schema Integration and Enhancement in the IntegrationSpace

There might often be a need to introduce new objects or relations, that do not exist in any of the underlying systems. This is particular useful when relating information from different sources. In the Employee-Department example it would be useful to introduce Company as a new object interface, and relate Departments to their respective companies.

Such new object and relation interfaces can be specified and realized through object and relation implementations in the integration space. Storage for this integration space is provided directly, by the object-oriented database used for the distributed object space.

15.9 Tool-builder Support and Presentation Integration

The steps for the tool builder role was presented in the last chapter. A tool-builder does not have to be concerned with the details of how components are made available through the integration framework. A tool-builder needs only to be concerned with COOM object interfaces and relation interfaces, not with implementations. A tool-builder will use functionality and information from the integrated components when a tool is to be created.

The scenario described in section 9.8 and section 15.1 is used as an illustration for how a tool-builder is supported. The case is to create a browser tool that allows for selection of departments, and to get information about employees working in that department. The browser is a user interface component and also then demonstrates the use of COOM for presentation integration.

Figure 15.7 shows a resource-planning tool for a department-manager. The tool has a window with a list of departments, and a list of employees for each selected department. A dialogue-window provides information about an employee, with name and age from the person databases, and allocated and available hours from the project management system.

The description will follow the tool-builder process described in the previous chapter:

- *User-interface Specification*
- *Model Specification*
- *Interface Specification*
Figure 15.7: The user interface of the example tool

- Schema Integration and Enhancement
- Model and User-interface Implementation
- Configuration
- Run time support

15.9.1 User-interface Specification

Figure 15.7 shows the user interface of a part of the resource planning tool. An interactive user interface development tool could be used, to rapidly get agreement with the end-user about the user interface layout and provided functionality. The functionality will be provided by the model-part of the tool.

In this case, the user interface is a window with one list for departments and one list for employees in that department. There are menu-options for the creation of new departments...
and employees. A double click on an employee, will display a dialog-box with information about that employee.

15.9.2 Model and Interface Specification

```cpp
OBJINTERFACE DeptEmpModel: Model {
    Set<Department> getDepartments();
    Set<Employee> getEmployeesInDept(Department [in] d);
    int getSalaryForEmp(Employee [in] e);
    int setSalaryForEmp(Employee [in] e, int newSalary);
    Department createDepartment(String [in] name ...);
    Department createDepartment(String [in] name);
    Employee createEmployeeInDept(Department [in] d,
        String [in] name, int [in] pn;
        Date [in] startDate);
    Set<Employee> findEmployeeInAgeRangeInDept(
        int [in] min,
        int [in] max, Department [in] d);
};
```

Figure 15.8: User Interface - Model interaction

Figure 15.8 shows the interaction between the user interface part and the model part of the tool, together with an ODL-specification for the model-part of the tool. The model-part can also be created directly in C++, if the functionality is not be made available to other tools. Appendix B.4 shows some of the C++ code for the implementation of this. The model-part supports model-oriented presentation integration.

15.9.3 Schema Integration and Enhancement

The next step is to study the available schemas, to identity if the necessary functionality and information is directly available. The schema consists of interface definitions for Person, Employee and Department, with the relation WorksFor between Employee and Department, as showed in figure 14.4. The employees might exist in different object-spaces, either in Ingres or ObjectStore. No particular integration or enhancement is done in this case.

15.9.4 User-interface Implementation

The user interface part of the tool will be created through the use of a portable user interface application framework. In C++ the CommonView framework is used, while in Smalltalk the Model-View-Controller framework is used.

The following shows some of the C++ code for the creation of the user-interface-object, UIWin, more of this code is shown in appendix B.4.

```cpp
void App::Start()
```
15.9.5 Model Implementation

{ 
  DeptEmpModel* m = metaDeptEmpModel->create();
  UIWin tui(m);
  tui.Show();
  Exec();
}

A model-object is created and used by the user-interface object, UIWin. This user-interface part, UIWin, supports display-oriented presentation integration.

15.9.5 Model Implementation

The model implementation implements the operations defined in section 15.9.2, by use of COOM OML for accessing the required objects and relations.

/*******************************************************************************/
Set<Department>* DeptEmpModel::getDepartments()
{
  return Department::extent();
  // return metaDepartment->extent(); /* direct use of interface manager */
}

/*******************************************************************************/
Employee* DeptEmpModel::createEmployeeInDept(Department* d,
String* name, int pnr, int age, Date* startDate)
{
  ...
  // Sets object space for creation
  Employee::creationSpace(os1);

  Employee* e1 = Employee::create(name, pnr, age, salary);
  // Employee* e1 = metaEmployee->create(name, pnr, age, salary);

  AWorksFor* awf = new AWorksFor();
  awf->startDate = Date::today();

  // insert in relation
  WorksFor->insert(e1, d, awf);
  return e1;
}

The tool-builder uses directly the interface-classes for Employee and Department, their interface manager objects are hidden through static access-operations in the classes. The relation, WorksFor, is manipulated directly through the relation interface manager, WorksFor, which is an instance of class R1Person, Department, AWorksFor. More of this code and definitions are shown in appendix B.4.

15.9.6 Configuration and Run time support

The configuration step tailors the default object-spaces for each object-interface for each individual user. This is done through a context-file, with information about default object
spaces for each object interface.

// Initiate Object-Spaces in ContextFile:

"os1 Ingres1
os2 ObjStore1
DefaultOS Ingres1
Person Ingres1"

Here the internal os1 is used for Ingres1, os2 for ObjStore1 and default object space is Ingres1.

The run time support is provided by the COOP Integration framework, through classes in C++ or Smalltalk. More details on this can be found in the OML-specifications and examples in appendix B.

15.9.7 Contribution and Conclusion

This chapter has given an illustration of the intentional use of COOP and COOM, by looking at the component-integrator role and tool-builder role, and how the goals of these roles are supported, for the scenario described in section 9.8.

The first part of the chapter showed how different components like RDBMSs, OODBMSs and both large-grain and fine-grain functional services could be integrated by a component integrator. The last part of the chapter showed how presentation integration can be achieved by a tool-builder. This also serves as a general example for the use of COOM.

The basis for this chapter was the basic concepts in COOM and COOP, and the COORASS methodology. Presentation Integration is based on the experiences from presentation integration discussed in chapter 7. The discussion on object-identity has been adopted from [EK91] and [HD92]. There has been practical experiments with mappings for different kind of systems. Mapping for a RDBMS have been tried out in [FjKL91, Low92]. Mappings for an OODBMS have been tried out in [Tin92, Ste92]. Mappings for a Functional service have been tried out in [FST90, Öst92a].

The contribution of this chapter is to illustrate the principles involved in doing component integration, and how presentation integration can be achieved by a tool builder. Tool building with presentation integration is used as a general case to illustrate a more comprehensive use of COOM and COOP. The example-tool for resource-planning based on a RDBMS, an OODBMS and a project management system illustrates the COOM and COOP possibilities for systems integration.

This chapter concludes the presentation of COOP and COOM. The next chapter will evaluate and discuss the results of this work with respect to the requirements from Part I.
Chapter 16

Evaluation, Future work and Conclusion

This chapter gives concluding remarks on the work, and discusses areas for future work. The contributions of this work are highlighted and evaluated with respect to the initial requirements. Experiences from the development of COOP-related prototypes are presented, then the work is compared with other similar efforts. Finally, remaining problems and issues for future work are discussed.

The goal of this work has been to supply methods, formalisms and tools that make it easier to cope with the problems of integration and interoperability in heterogeneous software systems.

The goals of the research have been:

- To identify different systems integration areas and their requirements for solution.
- To evaluate the use of an object-oriented approach in each individual integration-area.
- To develop a unified object-oriented integration framework to meet the requirements from each integration-area.

The structure of the work has thus been divided into three parts, which will be discussed in the following.

- Part I: Systems Integration - A Taxonomy of Problems and Solutions
- Part II: Experiments with Object-Oriented Approaches to Systems Integration
- Part III: COOP - A Unified Object-Oriented Approach to Systems Integration

16.1 Contribution and Evaluation of Part I

The goal of the research presented in part I "Systems Integration - A Taxonomy of Problems and Solutions" has been to identify different systems integration-areas and their requirements for solution.

Chapter 2 gave an introduction to the problem of systems integration seen from the perspectives of an environment user and an environment builder. The Double Toaster model
was presented as a reference model for integration, as a refinement of the ECMA/NIST Toaster model based on concepts from the ESF architectural reference model.

Chapter 3 presented more detailed definitions and related requirement areas for the three integration-areas: control integration, data integration and presentation integration. These three areas as a part of the Double Toaster model has been refined into six areas by refining each area into two sub-areas: request-oriented and notification-oriented control integration, single model and multi model data integration, and display-oriented and model-oriented presentation integration. The fourth area, process-integration, was described in chapter 2 as an area where support could be given by process interaction engines interoperating with process interworking engines. It has been a goal of this thesis to provide architectural support for such interoperating.

Chapter 4 described a number of proposed systems integration architectures with respect to the Double Toaster model.

The contribution of this work is the refined Double Toaster model based on a refinement of the ECMA/NIST Toaster reference model and requirements related to the different integration areas. The Double Toaster model is a good basis for evaluation and understanding of different proposals for integration frameworks and systems integration architectures. This work has been presented at different seminars [Ber92d, Ber93], and published in [Ber92b].

The contributions from part I can be summarized as follows:

1. Double Toaster Model with integration requirement areas
2. Use of the Double Toaster Model for description of systems integration frameworks

### 16.2 Contribution and Evaluation of Part II

The goal of the research presented in part II "Experiences with Object-Oriented Approaches to Systems Integration" has been to evaluate an object-oriented approach for each individual integration-area.

This has been done by independent experiments in the three integration areas, control integration, data integration and presentation integration.

Chapter 6 reported on experiences from using an object-oriented approach for control integration. This was based on prototypes of an object-oriented "Software Bus" interaction-mechanism that facilitated both request-oriented interaction and notification-oriented interaction. The SITE prototype showed an object-oriented approach to both request-oriented and notification-oriented control integration based on Smalltalk. The CODEDISC prototype gave experiences with the automatic generation of C++ classes for request-oriented control integration, in a development-environment based on C++. These experiences have been published in [Ber92c].

Chapter 7 presented experiences from object-oriented data integration. Object-Oriented database-integration is shown through the development and experiences from the object-oriented model SOOM, and the "HyperModel" benchmark for database-system-evaluation. The SOOM model showed the usefulness of relationships as a structural abstraction in an object-oriented model. The "HyperModel" benchmark gave insight into the architecture and performance of object-oriented databases. The results from benchmark-runs showed that the first generation of object-oriented databases systems did not have sufficient performance, but that this significantly improved in the second generation of such systems.
The current generation of object-oriented database-systems have sufficient performance and modeling power to be a basis for data integration. These results have been published in [Ber88c, B+89d, ABM+90, BA91].

Chapter 8 reported on experiences from object-oriented presentation integration, based on the use of object-oriented user-interface application-frameworks. The SITE prototype and CODEDISC prototype showed that the separation of user interaction components and service components is useful and provides a potential for more flexible and task-oriented tools. Experiences from this have also been published in [Ber92c].

The contribution of these results is a validation of the hypothesis that object-oriented technology is a suitable tool for facilitating systems integration, and a summary of experiences related to the different areas of integration.

The contributions from part II can be summarized as follows:

1. *Experiences with an Object-Oriented Software Bus*
2. *Combined Structural and Behavioural model in SOOM*
3. *HyperModel Benchmark for OODBMS evaluation*
4. *Experiences with Object-Oriented Presentation Integration for Distributed Objects*

### 16.3 Contribution and Evaluation of Part III

The goal of the research presented in part III "COOP: A Unified Object-Oriented Approach to Systems Integration" has been to develop a unified object-oriented integration framework to meet the requirements for solution from each integration-area.

The main contribution of the COOM model is the combination of different features in a fully object-oriented model as defined in [Dit86], realized as an abstract structural model layered above a behavioural model, with a separation between interface and implementation. The COOM model unifies different previously independent mechanisms for control integration, data integration and presentation integration in one model.

The following evaluates the different contributions of COOP and COOM, with respect to different problem areas.

#### 16.3.1 COOP/COOM and Control Integration

Chapter 10 defined $COOM_F$ as a basis for $COOM_B$ and $COOM_S$, and introduced the novel use of object-spaces for the management of the mapping from interfaces to implementations.

COOM and control integration was described in chapter 11. The contribution of COOM with respect to control integration is the separation of interface and implementation used as a basis for explicit object management through object interface managers and object implementation managers. The COOM ODL is mappable to language-specific realizations in object-oriented languages like C++ and Smalltalk.

A new contribution by COOM is the introduction of notifications as part of the schema description. The use of object-space specific interface and implementation managers contributes to an integration framework where it is possible to add new implementations.
16.3 Contribution and Evaluation of Part III

\[
\begin{array}{|l|c|}
\hline
\text{Requirements to} & \text{COOP/COOM} \\
\hline
\text{RCR-1: Interface-description} & + \\
\text{RCR-2: Interaction model} & + \\
\text{RCR-3: Run-time} & + \\
\text{RCR-4: Robustness and error-handling} & + \\
\text{RCR-5: Multi-user} & + \\
\text{RCR-6: Version handling} & \div \\
\text{RCR-7: Security aspects} & \div \\
\text{RCR-8: Administration} & - \\
\text{RCR-9: Heterogeneity} & + \\
\hline
\end{array}
\]

Table 16.1: Evaluation of COOP/COOM for Request-oriented Control-integration

The COOM ODL compiler has itself been developed as an object-oriented application framework, in order to make it easy to add new backends. Mapping strategies for a particular type of implementation, e.g. a relational database system might be realized first as reimplementation of operations in the framework, then added as a backend to the ODL compiler.

**Request-oriented Control Integration**

Request-oriented control-integration is the extent to which systems are able to interact directly with each other, by requesting and providing functional services. COOM supports this through viewing existing systems as objects with a certain interface. The view of having everything in the environment available as objects with well-defined interfaces gives a good basis for an integrated environment. The environment consist of a set of objects providing services to each other, and requesting services from each other.

Table 16.1 evaluates COOP and COOM with respect to request-oriented control integration. Interface-descriptions (RCR-1) are provided through ODL. The COOP framework provides an interaction-model (RCR-2) with run-time support (RCR-3) for both requests and notifications. Error-handling (RCR-4) is supported through the exception-mechanism. Multi-user support (RCR-5) is provided through a possibility for having multi-thread components. More support can be given for administration and installation tools (RCR-8). There is support for heterogeneity transparency (RCR-9) for distribution, access-mechanism and location. The implementation-mapping provides support for underlying systems written in different programming languages.

Remaining areas are in particular version handling (RCR-6) and security aspects (RCR-7) which, as stated in section 9.8, have not been addressed in this work.

**Notification-oriented Control Integration**

Notification-oriented control-integration is the extent to which systems are able to interact by sending out notification about certain events. Other systems might register interest for being notified about these events. This gives a looser and more flexible integration-facility, since the systems which is generating notifications do not need to know about the receivers. COOM supports this through Notifications.

Table 16.2 evaluates COOP and COOM with respect to notification-oriented control in-
16.3.2 COOP/COOM and Data Integration

Table 16.2: Evaluation of COOP/COOM for Notification-oriented Control-integration

<table>
<thead>
<tr>
<th>Requirements to</th>
<th>COOP/COOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCN-1: Event Publishing</td>
<td>+</td>
</tr>
<tr>
<td>RCN-2: Event Subscribing</td>
<td>+</td>
</tr>
<tr>
<td>RCN-3: Heterogeneity</td>
<td>-</td>
</tr>
</tbody>
</table>

tegration. There is support for event publishing and subscribing (RCN-1/2) through the notification mechanism. A novel contribution in COOM is the specification of notification-types as a part of the schema, and the use of inheritance between notification-types. The traditional approach is to embed information in a string which is interpreted at run-time. The meaning of the string needs to be specified and agreed upon separately. There is a possibility for implementation of the notification-mechanism for different heterogeneous systems (RCN-3), but this has not been tried out. More work can also be done on the use of multi-thread and light-weight processes to support the handling of notification-events.

16.3.2 COOP/COOM and Data Integration

COOM and data integration was described in chapter 12 and 13. The contribution of COOM with respect to data integration is the realization of a structural model on top of a behavioural model. This is done in a way where it is possible to transform a structural model to an equivalent behavioural model. The important lesson to learn is that an object-oriented model should be fully encapsulated and reflect structure only as structural abstractions. This provides a freedom to change the implementation of underlying structures.

Chapter 12 presented the COOM structural model and its relation to data integration. Attributes and relations was defined as a layer on top of the basic The separation of relation interface and implementation with support for multiple implementations of a relation interface is novel in COOM. The dual support for both role-name access and relation-access is new in COOM. Role-name access provides the same encapsulation of actual implementation as a relation interface manager. The possibility to specify attributes with exceptions and notifications is novel. The possibility for the specification of unique attribute-combinations in an interface is also new, this was introduced in the initial COOM-compiler [Ber91g].

It has also been shown that COOM provides a foundation for further work on multi-model data integration.

The main contribution of OQL and its query processing, presented in chapter 13, is to show how it is possible to realize a query processing facility as part of the integration framework. The contribution of OQL is to show a basis for object-based query processing in an environment supporting multiple simultaneously implementations of object and relation interfaces. The novel contribution in this work is a framework with multiple parallel implementations for both object and relation interfaces, and a query processing strategy that utilizes this. OQL is based on the REBOOT AQL, and extends it in four areas: Objects as values, Query Arguments, Nested queries and Operations.
Table 16.3: Evaluation of COOP/COOM for Single-model Data-integration

<table>
<thead>
<tr>
<th>Requirements to</th>
<th>COOP/COOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDS-1: Basic ERA-model</td>
<td>+</td>
</tr>
<tr>
<td>RDS-2: Complex objects</td>
<td>+</td>
</tr>
<tr>
<td>RDS-3: User-defined data types</td>
<td>+</td>
</tr>
<tr>
<td>RDS-4: Prog. language support</td>
<td>+</td>
</tr>
<tr>
<td>RDS-5: Versions/variants</td>
<td>∨</td>
</tr>
<tr>
<td>RDS-6: Metadata/Schema</td>
<td>+</td>
</tr>
<tr>
<td>RDS-7: Views</td>
<td>∨</td>
</tr>
<tr>
<td>RDS-8: Rules / Triggers</td>
<td>-</td>
</tr>
<tr>
<td>RDS-9: Query language</td>
<td>+</td>
</tr>
<tr>
<td>RDS-10: Security</td>
<td>∨</td>
</tr>
<tr>
<td>RDS-11: Distributed Architecture</td>
<td>+</td>
</tr>
<tr>
<td>RDS-12: High Performance</td>
<td>+</td>
</tr>
<tr>
<td>RDS-13: Concurrency</td>
<td>-</td>
</tr>
<tr>
<td>RDS-14: Cooperation</td>
<td>∨</td>
</tr>
<tr>
<td>RDS-15: Logging - recovery</td>
<td>+</td>
</tr>
<tr>
<td>RDS-16: Data Interchange</td>
<td>+</td>
</tr>
</tbody>
</table>

COOM and Single-model Data Integration

Single-model data-integration is the extent to which systems are able to share common data and information that are stored and manipulated through one single data model and a corresponding storage service. COOM supports this through viewing existing data-items within systems and database-systems as objects. A relation is viewed as a special kind of object which provides the service of defining relationships between objects. For each relation-interface which is defined there will be an object which provides the service of storing and retrieving the actual relationships. If the relation-interface definition gives the names of roles in each direction, object-dependent access and storage functions will automatically be created.

Table 16.3 evaluates COOP and COOM with respect to single-model data integration.

COOMS provides support for an abstract ERA-model (RDS-1) with complex objects (RDS-2), based on the behavioural encapsulation basis of COOMB. COOMB gives support for user-defined data types (RDS-3), with mappings to different programming languages like C++ and Smalltalk (RDS-4). The Object Dictionary provides access to an interface repository with schema information (RDS-6). Triggers (RDS-8) are partly supported through notifications. A query language (RDS-9) is provided through OQL. The distributed (RDS-11) client-oriented architecture of ObjectStore, used in the COOP-prototype, gives support for high performance (RDS-12). Concurrency (RDS-13) is supported through a simple transaction-mechanism. This mechanism needs extensions, as suggested in section 11.3.2. Mechanisms for logging/recovery (RDS-15) and data interchange (RDS-16) are those provided by the ObjectStore OODBMS.

Version and variants (RDS-5) has, as stated in chapter 9, not been looked into in the scope of this work. A view-mechanism (RDS-7) has not been developed, and is still an open research issue. Some ideas for a solution based on representation objects is discussed in section 12.6.2. Security-aspects (RDS-10) and long cooperative transactions (RDS-14) has not been looked into in the scope of this work.
### 16.3.2 COOP/COOM and Data Integration

<table>
<thead>
<tr>
<th>Requirements to</th>
<th>COOP/COOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDM-1: Common (Canonical) Data Model</td>
<td>+</td>
</tr>
<tr>
<td>RDM-2: Schema Mapping and Transformation</td>
<td>+</td>
</tr>
<tr>
<td>RDM-3: Schema Integration</td>
<td>-</td>
</tr>
<tr>
<td>RDM-4: Query Processing</td>
<td>+</td>
</tr>
<tr>
<td>RDM-5: Transaction Management</td>
<td>-</td>
</tr>
<tr>
<td>RDM-6: Security</td>
<td>÷</td>
</tr>
<tr>
<td>RDM-7: Autonomy and Evolution</td>
<td>-</td>
</tr>
<tr>
<td>RDM-8: Functional Components</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 16.4: Evaluation of COOP/COOM for Multi-model Data Integration

COOM succeeds in supporting most of the requirement-areas for single-model data-integration. Some areas, such as RDS-5, RDS-7, RDS-10 and RDS-14 have not been focused on in this work and require more research and development.

#### COOM and Multi-model Data Integration

Multi-model Integration was defined in section 3.4.3 as the degree to which tools are able to access and share common data and information that are stored and manipulated through multiple data models with corresponding multiple storage services.

How COOM meets the requirements for multi-model data integration given in section 3.4.3 is shown in table 16.4.

The COOM model is a suitable canonical data model (RDM-1), and the separation of interface and multiple implementations provides a good basis for schema mapping and transformation (RDM-2). Schema integration (RDM-3) is partly provided through the

The use of attributes and relations as concepts in addition to objects, creates an extra level of complexity with respect to schema integration. This is commented on in section 12.6.3.

Remaining areas are in particular RDM-3: Schema Integration, RDM-5: Transaction Management, RDM-6: Security and RDM-7: Autonomy and Evolution. Another relevant area is strategies for schema-reconciliation and enrichment (RDM-2). There is also still work to be done on query optimization (RDM-4).

Schema Integration as described in RDM-3 has been commented on in section 12.6.2. Query Processing has been shown in chapter 13. Transaction Management has been discussed in section 11.3.2. Security and support for Autonomy and Evolution has been given less attention. Functional component-support in RDM-8 is given by the possibility to represent components as objects, and the possibility to also represent information managed by a component as objects.

The main focus has been to provide a foundation-layer for later work on multi-model data-integration. Issues for this was discussed in section 12.6.2. The COOM model is a suitable canonical data model, and the separation of interface and multiple implementations provides a good basis for schema mapping.

The current work has not aimed to resolve all problems related to multi-model data integration, but instead to develop a framework where such problems can be investigated further. Some of the requirements for multi-model integration (RDM-2, RDM-3, RDM-4,
16.3 Contribution and Evaluation of Part III

RDM-5) are explored further in PhD thesis and Master thesis as described in section 16.6.

16.3.3 COOP/COOM and Process Integration

It was stated in section 3.6 that it has been a goal to provide architectural support for process integration.

Process integration can be supported through a set of interworking process engines, where some of these are supporting individual users while others are responsible for the coordination between users. This has been exemplified by the ESF process interaction engines and the ESF process interworking engines described in section 2.4.2, and further described in section 4.7.6. This approach is a combination of architecture 2 (Agent-to-Agent connections) and architecture 3 (Bus-oriented communication), as described in section 4.1.5 and in [PR92]. Support for this is offered through the COOP/COOM support for request-oriented and notification-oriented control integration. The COOP/COOM support for data integration can provide architectural support for process integration according to architecture 1 (centralized database).

A limitation to the solutions offered by COOP/COOM for process integration is that the interoperation support is embedded in an object-oriented model. As discussed in section 4.1.5, it is a current research goal to experiment with different, often hybrid, modeling-formalisms [Sch91]. The engines might, however, be encapsulated as service components and made accessible as objects.

Experiments with object-oriented control integration in the SITE environment described in section 6.1, showed an example of object-oriented control integration for process interaction engines interworking with a process interworking engine. Section 16.6 discusses possible future work with respect to COOP/COOM and process integration.

16.3.4 COOP Architecture

The COOP architecture was described in chapter 11. The contribution of the COOP architecture is the notion of a distributed persistent object space, handled through the integration framework. The architecture provides a basis for a transparent handling of both large-grained and fine-grained objects. This is done by the use of an object-oriented database that supports efficient handling of fine-grained objects. Large-grain components might easily be mapped to and manipulated from a fine-grain object. The contribution of the COOP Integration framework is to provide a unified approach to integration combining data integration, control integration and presentation integration.

16.3.5 COORASS methodology

The COORASS methodology was described in chapter 14. The contribution of the COORASS methodology is modeling-concepts for the basic behavioural-oriented model $COOM_B$ in the basic OORASS layer, and enhancements for the structural-oriented model $COOM_S$ in the additional COORASS-layer. The COORASS-layer for $COOM_S$ can be mapped to an equivalent OORASS-model in the same way as a $COOM_S$ model can be mapped to a $COOM_B$ model. The OORASS methodology is unique with respect to its focus on interfaces through roles. The basic OORASS-methodology has been published in [RB*92].
16.3.6 CODE specification

The CODE development environment was described in chapter 14. The contribution of the CODE specification is an example of how a development-environment for the COOP Integration Framework could be build, and an example of an environment that is using the COOP architecture. The basic concepts has been validated through the experiences with the CODEDISC prototype, described in [Ber92c].

16.3.7 Mapping and Systems Integration

The COOP mapping and systems integration was described in chapter 15. The contribution of COOP/COOM with respect to mappings and systems integration is to show that it is possible to map to and from different underlying systems. The discussion on object identity shows that the use of separate object implementation managers facilitates for different local identity-schemes in different underlying systems.

16.3.8 COOP/COOM and Presentation integration

COOM and presentation integration was described in chapter 15. The contribution of COOP/COOM with respect to presentation integration is to show that it is easy to create different tool user-interfaces interacting with the same underlying model-objects. COOM facilitates for model-oriented presentation integration. Display-oriented presentation integration can be realized by different object-oriented user-interface frameworks.

<table>
<thead>
<tr>
<th>Display-oriented</th>
<th>COOP/COOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPD-1: Look-and-Feel and Portability</td>
<td>+</td>
</tr>
<tr>
<td>RPD-2: Dialog-support</td>
<td>+</td>
</tr>
<tr>
<td>RPD-3: Graphics-support</td>
<td>-</td>
</tr>
<tr>
<td>RPD-4: Interactive Tools</td>
<td>-</td>
</tr>
<tr>
<td>RPD-5: Copy/Cut and Paste</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model-oriented</th>
<th>COOP/COOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPM-1: Model-interaction</td>
<td>+</td>
</tr>
<tr>
<td>RPM-2: Multi-Model-interaction</td>
<td>+</td>
</tr>
<tr>
<td>RPM-3: Change/Update</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 16.5: Evaluation of COOP/COOM for Presentation integration

Table 16.5 evaluates COOP and COOM with respect to requirements for display-oriented and model-oriented presentation integration, given in section 3.5.

This work has focused on the model-oriented part of presentation integration. Display-oriented presentation integration has been addressed through the use of state-of-the-art object-oriented user-interface application frameworks, CommonView for C++ and the Model-View-Controller framework for Smalltalk.

The requirements for display-oriented presentation integration have been met as follows:

Look-and-Feel and Portability (RPD-1) is supported through the use of object-oriented user-interface application frameworks, such as CommonView and Smalltalk Model-View-Controller. Both frameworks are supported at different platforms, such as Unix/X-Windows with OSF/Motif Look-and-Feel, or DOS/Windows with Windows Look-and-Feel. An ap-
plication is easily portable between these platforms. Dialog-support (RPD-2) is provided through the available user-interface frameworks in CommonView and Smalltalk MVC. Graphics-support (RPD-3) is partly provided in both application frameworks. Interactive tools (RPD-4) is supported through 3rd-party tools, such as Window-builder for Smalltalk and Case:W for CommonView. Copy/Cut and Paste (RPD-5) is partly supported through the use of mechanisms in the underlying window-system.

The requirements for model-oriented presentation integration have been met as follows:

Model-interaction (RPM-1) is provided through the creation of a model-object that interacts with the objects made available through COOM. Multi-Model interaction (RPM-2) is provided through the use of COOM to interact with different underlying systems. Change/Update support (RPM-3) is provided through the $COOM_B$ notification mechanism.

16.3.9 COOP and COOM Summary

The main contribution of this work is the COOP Integration framework which supports control integration, data integration and presentation integration within the same modeling paradigm.

The contributions from part III can be summarized as follows:

1. An Object-Oriented Integration Framework that unifies control, data and presentation integration
2. An Abstract Structural model realized through a Behavioural model
3. Support for both large-grained and fine-grained objects, through a distributed object space
4. The introduction of separate interface and implementation managers
5. Support for multiple implementations of a relation interface
6. Notifications as part of the schema description
7. Attribute specifications with notifications, exceptions and unique-specification
8. Query Processing with multiple implementations
9. Foundation for multi-model data integration
10. Methodology for Integrated Behavioural and Structural modeling

Section 3.1 described a number of conflicting requirements and section 9.2 described the goal of COOP in these areas. The following describes to which degree these goals have been achieved.

- **Heterogeneity versus Homogeneity**
  The object-oriented model provides a homogeneous view on underlying heterogeneous components. Heterogeneity is limited by the requirements to give up some autonomy, and to provide functional support for object identity and object creation, as discussed later.
- **Autonomy versus Common policy**
  It has not been possible to provide full autonomy for the integrated components. It is required that they provide a basis for object identity that is maintained, and that notification is given about deletion of objects and relationships.

- **Distribution versus Centralization**
  A centralized view is provided on the distributed components through the object-oriented database used for the maintenance of the object space. The experiences are limited to the client-oriented architecture supported by this system. Future development in distributed object-oriented database-systems might give more comprehensive support.

- **Fine-grain versus Large-grain**
  The support for fine-grain objects with high performance provides a good basis also for the support of mapping of objects to large-grain components. The fine-grain support puts requirements on the components to be integrated, as discussed in the section on "Variety of Components versus Functional Usability".

- **Efficiency versus Flexibility and Extensibility**
  ODL generates interface and implementation classes in object-oriented languages automatically. The focus have been on efficiency rather than flexibility. It is possible to also make a dynamic interface similar to the dynamic API of the OMG CORBA architecture to allow for more flexibility. The trade-offs here are similar to what is found in CORBA, and these have been resolved by the offering of both a static and a dynamic interface.

- **Simplicity versus Coverage**
  The COOM object model provides a simpler model than the alternative of having three different model-approaches for control integration, data integration and presentation integration. The presentation of COOM done here is, however, rather complex and detailed, and aimed at integration framework builders. COOM should be presented in different ways, with increasing simplicity, for the framework builder, the component integrator and the tool builder. It is possible to hide most of the complexity of implementations for the tool builder, and let the tool builder only work with interfaces, and the notion of object spaces.

- **New language versus Existing languages**
  Neither Smalltalk nor C++ provides an ideal mapping for the concepts in COOM. Smalltalk because it does not support typing on interfaces, and C++ because it does not support conformance-based typing. This has resulted in a lack of typing-support for the Smalltalk mapping, and a restricted interface-typing for the C++ mapping.

- **Variety of Components versus Functional Usability**
  The possibility to have the functionality to create new objects with object identity and do a find-query on object-sets means that there are restrictions on which components that can be integrated. Only components where this can be realized can be integrated. Sometimes this functionality might not be necessary. It could be considered to create a hierarchy of increasing functional support similar to what have been suggested for the ZOO1FF [HD92] framework. Another interesting question is what to do with functionality that not automatically can be supported by COOM. The query-language of a RDBMS is more powerful than the current OQL, with respect to certain operations such as finding a sum, and max/min for values. If such functionality is needed, it may be supported by predefined SQL-queries in stored procedures, mapped into operations. Another alternative is to increase the functionality of OQL, as suggested by SOQL in section 13.5.
16.4 Experiences with COOP-related Prototypes

- **Syntactical basis versus Semantical integration**
  COOM ODL provides a comprehensive syntactical support for the description of both object interfaces and relation interfaces. This is, however, not sufficient to achieve integration. System vendors need to provide open systems, in particular to support a split between user interface components and service components, so that interfaces can be described through ODL schemas. They still do not provide enough semantical information to be fully understood. A particular missing part is descriptions of the semantics of operations. In addition such schemas need to be discussed and understood, preferably in user groups and consortiums like CASE Communique and the CASE Interoperability Alliance.

Further problem areas and issues for future work are described in section 16.6.

The following papers have been published as results of this work: [Ber91c, RB+92, Ber92b, Ber92a].

### 16.4 Experiences with COOP-related Prototypes

A number of COOP-related prototypes have been developed by the author, or in student projects and MSc-work at the Norwegian Institute of Technology with the author as adviser.

One of the aims of these prototypes has been to validate and get experience with different aspects of COOP and COOM. This section provides an overview of the relationship between the different prototypes, and a short description of the experiences with each of these.

Figure 16.1 shows the relationship between some of the different COOP-related prototypes.

- CODEDISC [FST+90, Ber92c]
- Initial COOM Parser [Ber91g]
- OODBMS mapping for ObjectStore [Tin92]
- RDBMS mapping in SEROO [FkJL+91]
- COOM Compiler Framework and Frontend [LS92]
- OQL Compiler and Processor [Ste92]
- Initial COOP Smalltalk Framework [Ber91a]
- COOM Smalltalk and OSF/DCE Backend [Øst92b]
- COOM Smalltalk and OMG/CORBA Backend [Øst92a]
- C++ Framework Backend [Sol92]
- RDBMS Backend and SOQL for Ingres [Low92]
- Final prototype set-up (see Appendix D)

These prototypes will be briefly described in the following:
• **CODEDISC**
  The experiences from the CODEDISC project [FST+90, Ber92c] was described in section 6.2. The goal of CODEDISC was to create a development environment for component-based systems with an object-oriented encapsulation. Components reachable through the OSF DCE remote procedure call system was encapsulated and made available through C++ objects. The NIDL2C++ compiler developed in CODEDISC gave the first basis for the COOM-compiler. The OSF DCE compatible IDL-syntax was used for the operation-specifications in COOM.

• **Initial COOM Parser**
  The initial COOM-parser was developed by the author [Ber91g] as a merge of concepts from OSF DCE IDL [osf90] used in CODEDISC, the REBOOT datamodel [OH91] and concepts from OMG CORBA [Obj91b]. This COOM-parser was used as a basis for further work on the COOM-compiler, in particular in [Tin92] and [LS92].

• **OODBMS mapping for ObjectStore**
  Mappings for the ObjectStore OODBMS was realized in MSc-work by Roy Tingstad [Tin92], with the author as adviser. The REBOOT-compatible part of the initial COOM-parser was used as a basis for this work. The ObjectStore-mappings was reused by Tor Stenvaag in [Ste92].

• **RDBMS mapping in SEROO**
  The SEROO-project developed a mapping for an extended REBOOT model and a RDBMS. The Initial COOM Parser [Ber91g] was used as a basis for this.

• **COOM Compiler Framework and Frontend**
  The COCom project [LS92] by Petter Lowzow and Per Solberg in the spring of 1992,
with the author as adviser, realized a COOM-compiler based on an object-oriented framework design. The COOM-compiler framework is built around an "active parse tree". A front-end parses the input ODL-file and creates a parse-structure of C++ objects representing the syntax-structure. The code is generated by back-end specific classes. The first step when implementing a back-end is to create back-end specific subclasses for the code-generating classes. The code-generating methods, genCode(), genDeclarator(), and genType(), of these classes need to be reimplemented. This follows the principle of object-oriented application frameworks, described in section 5.5. The purpose of the COOM-compiler was to be a basis for future development of different back-ends. The COOM-compiler was built using the lex and yacc compatible tools Flex and Bison, and GNU C++.

- **OQL Compiler and Processor**
  An OQL Compiler and processor was made by Tor Stenvaag, with the author as adviser, in the OQL-project [Ste92]. This compiler generated code for the ObjectStore mapping made by Roy Tingstad in [Tin92].

- **Initial Smalltalk Framework**
  An initial COOP framework was created as a Smalltalk prototype by the author [Ber91a]. The purpose of the initial Smalltalk Framework was to validate the use of object and relation interface managers with corresponding multiple implementation managers for different object spaces. The Smalltalk Framework was used as a basis for the work in [Øst92b, Øst92a].

- **COOM Smalltalk and OSF/DCE Backend**
  The initial Smalltalk framework was used by Viger Østensen, with the author as adviser, as a basis for a specification of mappings between COOM ODL and Smalltalk, and for experimentation with a mapping to components reachable through the remote-procedure call mechanism of OSF/DCE in [Øst92b].

- **COOM Smalltalk and OMG/CORBA Backend**
  The Smalltalk-specification in [Øst92b] was used as the basis for realization of a Smalltalk-backend for the COOM-compiler from [LS92], in MSc-work by Viger Østensen [Ost92a], with the author as adviser. Instead of backend-generation for OSF/DCE, the dynamic interface of OMG CORBA was used.

- **C++ Framework Backend**
  A C++ framework backend for the COOM-compiler was realized in MSc-work by Per Solberg [Sol92], with the author as adviser. This work validated the object-oriented framework architecture of the COOM compiler, by providing a new backend for the existing front-end.

- **RDBMS Backend and SOQL for Ingres**
  A Relational backend for the COOM-compiler and an SQL-oriented version of OQL, SOQL, was made by Petter Lowzow in MSc-work [Low92], with the author as adviser. This work was done as an extension to initial work on RDBMS-mappings in the SEROO-project [FKJL+91].

### 16.5 Comparison with other related work

This section relates COOM and the COOP integration framework to other similar work.
16.5.1 Eureka Software Factory

The Eureka Software Factory project, described in section 4.7, has taken a comprehensive approach to integration. The project has addressed the four integration areas: process integration, control integration, data integration and presentation integration, in a number of different sub-projects.

One of the goals of the COOP integration framework has been to unify three of these integration areas: control integration, data integration and presentation integration through a common object model, COOM.

COOP and COOM provides a unified object-oriented framework for the integration-problems where ESF provides three different solutions: Software Bus for control integration, object storage services for data integration and user-interface tools for presentation integration. The Software Bus CDL language provides for a way to specify components by describing which interfaces it provides and which interfaces it requires. The COOM ODL interface concept provides a basis for making such descriptions. COOP and COOM provides concepts for how a more unified approach to integration can be taken in the Eureka Software Factory project.

16.5.2 COOM related to the REBOOT Datamodel

The REBOOT [OH91] and EPOSDB [LCD+89] datamodels are structurally object-oriented datamodels. The COOM structural model $COOM_S$ is based on the structural aspects of the EPOSDB and REBOOT datamodels. These models do not have the strict encapsulation view that is enforced in COOM.

The COOM ODL (Object Definition Language) is based on the REBOOT DDL, but extends it to include specification of operations, notifications and exceptions, in addition to the possibility of defining constants and basic datatypes. This extension is based on the syntax of NCS NIDL - Network Interface Definition Language, as it is used in the RPC (Remote Procedure Call) facility in OSF/DCE (Distributed Computing Environment).

The behavioural basis contains concepts like operations, exceptions and notifications which do not exist in the REBOOT model. The support for manager-objects with attributes and operations is also different, as well as the focus on interfaces and the possibility for parameterized types.

The Object Query Language, OQL, is an extension to the REBOOT AQL, by allowing objects to participate in queries in addition to basic values. The separation of interface and implementation managers in COOM leads to a different strategy for query processing. Some other extensions are the use of arguments, and the possibility for nested queries.

COOP and COOM preserves the basic concepts in the REBOOT model and extends it with new concepts, in particular for object management and the use of a behavioural basis. COOP and COOM provides concepts that could be utilized in extended versions of the REBOOT model, in particular the concepts for behavioural support and full encapsulation would add a possibility for control integration support in addition to data integration.

16.5.3 OMG

The Object Management Architecture of the Object Management Group (OMG), described in section 4.3 has focused in particular on request-oriented control integration.

The OMG Object Model is a behaviourally object-oriented model. COOM is conform with
the described semantics of the OMG basic object model, in addition COOM supports
notifications as an explicit concept. The OMG model supports attributes similarly to
COOM, but there is no support for relations or queries, as in the COOM structural model.

The goal of the OMG architecture is to be a framework for the interaction between objects
in a distributed heterogeneous environment [Obj91a]. However, the OMG architecture
is initially aimed at single-object interactions, with no support for queries or explicit
management of set of objects.

COOP and COOM is conform with the basic concepts of the OMG architecture and object
model, and extends it with new concepts, such as explicit management of objects, relations
and support for queries. RFPs for new object services like life-cycle support, persistence,
events, relationships, transactions and others have been issued from OMG. COOP and
COOM provides an approach to these services based on a realization in object-oriented
languages, as opposed to the generic C-based interfaces focused on in OMG.

16.5.4 Other work

The Field/HP Softbench [Rei90a] environments have taken a similar communication-
oriented approach to integration, but does this only through the support for events. This
system does not support an explicit schema-facility, like ODL, in order to specify provided
services or notifications. It does not offer any support for data integration.

The EIS-project [HSZ91, HZ90] and DOM [Man89, MH+92] are early approaches to the
use of object-oriented technology for systems integration. COOP and COOM is similar to
this, but uses a classical object model instead of a generalized functional object model.

Object-Oriented database-systems are a good basis for COOP and COOM. COOM has a
structural model with relations, not commonly found in object-oriented database systems.
In addition the explicit separation of interface and implementation managers provides
an extra facility for the handling of heterogeneous underlying systems. The operation-
definitions of COOM with specification of in/out-direction for arguments and possibilities
for notifications, exceptions and context, provides a better foundation for distributed com-
puting. A group of object-oriented database companies is currently working on a speci-
fication of a common object model, expected to be released in late 1993. Some of the
concepts in COOM could be interesting with respect to this.

Distributed computing environments like ANSA [ans89] provides a basis for viewing an
environment as a set of interacting objects. COOP and COOM is similar to this, but
adds extra support for the mapping to underlying heterogeneous systems and support for
database-systems.

There has been much work in the area of heterogeneous database-systems and interoperable
database-systems [LA86], [SL90], lately there has been much interest in the object-oriented
approach to this [CT91, KDN90, KLP91]. COOP and COOM provides a basis for a hetero-
genous database-system which also might incorporate services from non-database
components.

ZOOIF1 [HD92] is an object-oriented integration framework with a framework architecture
similar to COOP. A difference is that COOP supports a generating approach for multiple
programming languages and backends based on COOM ODL, while ZOOIF1 is a more
direct integration framework for C++.

COOP and COOM builds on a combination of the approaches taken by distributed com-
puting environments for control integration, and object-oriented database-systems for data
integration.
16.6 Problem-areas and issues for future work

The work on COOP and COOM has focused on the establishment of an overall unified integration framework. There are a many remaining issues to be resolved within parts of this framework.

Some remaining areas where work needs to be done were pointed out in section 16.3. The following are the most important problem areas to address in future work. As pointed out, some of these areas are already addressed in work that takes place as MSc and PhD thesis-work with the author as adviser, or co-adviser.

- **Transaction Management (RDS-10, RDS-11, RDM-5)** is an area which explicitly has been left out, but which needs to be addressed to achieve a fully working system. Marianne Hagaset [Hag92] has started PhD-work in this area at the Norwegian Institute of Technology. This work will look into more advanced transaction-mechanisms that might be useful in COOP. The use of concepts from process modeling and process integration might be an interesting approach here.

- **Multi-user support** is related to transaction management. There is a number of problems related to deletion and consistency of the objects used by different users. In particular the current architecture have manager objects as a critical resource. This problem is similar to the problem of supporting class-objects in object-oriented databases, and might take advantage of solutions from this area.

- **Support for Dynamics and Evolution** is necessary for a long-lived system. Version handling (RCF-6, RDS-7) has explicitly been left out of this work. Some of the problems here are similar to problems for evolution in object-oriented databases, and results from this area should also be applicable to COOP. Problems and solutions related to versions and variants with respect to schema evolution are currently being explored further in separate PhD-thesis work by Svein Erik Bratsberg [Bra93] and Erik Oudberg [Odb93]. It is a hope that these results can be utilized later.

- **Schema reconstructor and mapper** are necessary tools for a fully working system. A prototype of a schema reconstructor and mapper tool is under specification by Geir Kjerstad, [rst93], as part of MSc-work at the University of Oslo.

- **Schema integrator tool and Multi-model issues** need to provided and resolved in a fully working system. A prototype of a schema integrator tool is under specification by Steinar Kindringstad [Kin93] and Espen Koren [Kor93] as part of MSc-work at the University of Oslo. This work also looks into problems and solutions for semantic interoperability, and how generalization or representation objects, as described in section 12.6.2, can be realized in COOP/COOM. This will be used to resolve the problems in the normal situation, where there is semantical differences between data and functionality in different systems.

- **Query Processing** needs to be more comprehensively addressed. The current solution is only a framework for future work. Ole Jørgen Anfindsen [rA93] has started on PhD-work at the University of Oslo, in the area of Object-oriented query-processing. This work will look into object-oriented query-processing strategies that might be useful in COOP. The work is done in cooperation with Televerket, GTE Laboratories and SINTEF SI.

- **Security Support (RCF-7, RDS-13, RDM-6)** is needed for future commercial use integration frameworks. This has explicitly been left out in this work. Possible
future approaches includes the exploration of access-control-lists and capabilities for objects.

- **Integration with a trading language** would be interesting to try out. Object-oriented distributed systems like ANSA [ans89] and ISO/ODP includes the concept of a trader. The trader will return an object of a certain type, based on a predicate expressed in a trading language. A similar functionality is offered through the COOM interface manager objects. Create and find operations could be extended to accept a full trading-predicate.

- **Process Integration** has been pointed out as an important area that is not explicitly addressed in this work. COOP currently aims at support for the interoperation between process interaction engines and process interworking engines.

It could also be a possibility to use some of the modeling concepts of COOM as part of a process-modeling formalism. The SPELL process modeling formalism [CJM+92, Maz92] uses a structural object-oriented formalism as a basis for extensions for process-description. It could be interesting to see how the behavioural parts of COOM could add extra power to this. Earlier work by the author, DYNAMO-P [BL85] has also taken an object-oriented approach to the modeling of organizational processes.

Experiments on how object-oriented models and integration frameworks can be used as part of process modeling support is ongoing in MSc-work by Ketil Andenes [a93] at the University of Oslo. Investigations into how object-oriented dynamic models can be used as part of process modeling support is ongoing in PhD-work by Geir Magne Heydalvix [yda93] at NTH. It would also be interesting to see how process models can provide a foundation for advanced transaction mechanisms.

- **Semantics of behaviour and object-interaction** needs to be better defined. There is ongoing PhD-work in this area at the University of Oslo, which could provide interesting results, [Nor93, AR92].

- **A total integrated COOP framework and Tool-set** is necessary to develop in order to get more real-life experiences with the integration framework, and to try out all aspects of the framework. As have been mentioned, a number of the tools are already in development in various MSc-work. The recent availability of basic technologies, such as OMG CORBA implementations and distributed object-oriented databases, gives a good basis for future work. Some research and development projects where parts of this work will be done, have already started at SINTEF SI.

### 16.7 Conclusion

The contribution of this thesis is the COOP Object-Oriented Integration Framework that unifies control, data and presentation integration through the COOP Object Model, COOM. COOM is a fully object-oriented model that supports control integration through encapsulation and event-notification, supports data integration through structural abstractions for attributes and relations, and supports presentation integration through explicit separation of user-interface objects and functional objects.

A basic feature of the COOP architecture is the notion of a distributed persistent object space, handled through the integration framework. The architecture provides a basis for a transparent handling of both large-grained and fine-grained objects, through this object space. The novel introduction of separate interface and implementation managers for both objects and relations, provides a basis for transparent access to different underlying
16.7 Conclusion

systems. It is also a foundation for multi-model data integration and query processing. COOM introduced the novel use of object-spaces for the management of the mapping from interfaces to implementations. This means that a tool dynamically can change between different implementations, by a change of object space. Another contribution in COOM is the specification of notification-types as a part of the schema, and the use of inheritance between notification-types. An important feature in COOM is the extension of a pure object-oriented behavioural model with abstract structures, based on ER-oriented constructs. This facilitates for data integration in addition to control integration. The COORASS methodology does the same structural extension to the behavioural-oriented OORASS methodology. The different features of COOP and COOM forms a unified basis for control, data, and presentation integration, and also provides some architectural support for process integration.

Part I introduced requirements for the different integration areas, based on the new Double Toaster Model, refined from the ECMA/NIST Toaster model. The Double Toaster Model can be used as a basis for a high level description of systems integration frameworks.

Part II reported on experiences from experiments with object-oriented approaches to integration. These experiments concluded that an object-oriented approach is better than traditional approaches for control-, data-, and presentation-integration.

COOM and the COOP Integration framework have addressed a large problem area. The focus on the overall integration framework has meant less focus on interesting sub-areas such as query-processing, transaction management, multi-model mapping and others. In these areas, there is still a number of problems to be solved. Some of these problems, such as model-mapping and translation, schema integration and semantic interoperability, object-oriented query-processing, multi-system transaction management and object-oriented process-integration, are addressed in ongoing research with the author as adviser or co-adviser. It is also a goal of this work to bring together these contributions in a total integrated COOP framework and tool-set, which is needed in order to get practical experiences on a broader scale.

It is a growing industrial interest in an object-oriented approach to systems integration and interoperability. In particular the announced support for the OMG Object Request Broker-technology from a number of large system vendors is an indication of this. COOM and the COOP Integration framework should provide an interesting extension to the OMG technology, by showing additional support for notification-oriented control integration, data integration and presentation integration, in addition to the basic request-oriented control integration support defined by OMG.

It is a hope that COOM and the COOP Integration framework will provide a basis for a better understanding of different aspects of systems integration, and also continue as a platform for future research in the identified problem areas.
Part IV

Appendices
Appendix A

COOM ODL

A.1 The Object Definition Language

This appendix gives a more detailed description of COOM ODL (Object Definition Language). The syntax of ODL is based on a merge of constructs adapted from OSF/DCE IDL [osf90], OMG CORBA IDL [Obj92a] and REBOOT DDL [OH91]. The following is based on the current version of the COOM ODL compiler, as described in [LS92], with additions for some concepts like meta-operations and meta-attributes, the self-type, notifications, and parameterized interfaces.

The following presents first lexical conventions and details on basic types, then the ODL grammar in EBNF-format, and an example of the use of ODL.

A.2 Lexical Conventions

This section presents the lexical conventions of ODL. It defines tokens in an ODL specification and describes comments, identifiers, keywords, and literals.

A.2.1 Tokens

The tokens used in ODL are:

- **identifiers**: An arbitrarily long sequence of letters and digits.
- **keywords**: See table A.1 for a list of reserved identifiers.
- **literals**: See section A.2.5 for a description of literals.
- **others**: One or more consecutive "whitespace", tab, newline, form feed and comment (see below) are ignored, i.e. they may only serve to separate identifiers, keywords and literals.

A.2.2 Comments

A comment starts with the characters "/*" and ends with the characters "*/". It is not possible to nest comments. A comment can be used between any tokens.
A.2.3 Identifiers

An identifier is an arbitrarily long sequence of letters and digits. Letters are the characters [a..z A..Z]. Digits are the characters [0..9]. The first character must be a letter. Upper- and lower-case letters are not treated as the same. For ODL all characters are significant, but the target language may have restriction on numbers of significant characters.

A.2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<table>
<thead>
<tr>
<th>BITSET</th>
<th>INT</th>
<th>RELINTERFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>INTERFACE</td>
<td>SELF</td>
</tr>
<tr>
<td>BYTE</td>
<td>INT8</td>
<td>SETOF</td>
</tr>
<tr>
<td>CASE</td>
<td>INT16</td>
<td>SHORT</td>
</tr>
<tr>
<td>CHAR</td>
<td>INT32</td>
<td>SMALL</td>
</tr>
<tr>
<td>CONST</td>
<td>INT64</td>
<td>STRING</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>LONG</td>
<td>STRING0</td>
</tr>
<tr>
<td>CONTRELINTERFACE</td>
<td>LONGFIELD</td>
<td>STRUCT</td>
</tr>
<tr>
<td>DATE</td>
<td>META</td>
<td>SWITCH</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>MONEY</td>
<td>TRUE</td>
</tr>
<tr>
<td>DUBLE</td>
<td>NOTIFICATION</td>
<td>TYPE</td>
</tr>
<tr>
<td>ENUM</td>
<td>NOTIFIES</td>
<td>_TYPEDEF</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>NULL</td>
<td>UNION</td>
</tr>
<tr>
<td>FALSE</td>
<td>OBJINTERFACE</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>ONEWAY</td>
<td>UNSIGNED</td>
</tr>
<tr>
<td>HYPER</td>
<td>OUT</td>
<td>VOID</td>
</tr>
<tr>
<td>IMPORT</td>
<td>RAISES</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>READONLY</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Keywords in ODL

A keyword may be written with only lowercase letters, only uppercase letters or the first character as uppercase letters and the rest as lowercase letters. Keywords that are a combination of two words like CONTRELINTERFACE may be written with uppercase letter first in each word (ContrelInterface).

Characters used for punctuation are:

: ; { } [ ] , .. = ( ) *

A.2.5 Literals

Boolean Literals: A boolean literal is the string TRUE or the string FALSE. Only uppercase letters is allowed.

Character Literals: A character literal is a character enclosed in single quotes, as in 'q', or a character can be an escape sequence. All escape sequence starts with the character "\" followed by

- a number consisting of three digits.
• the character “x” and two characters from [0..9a..fA..F] where a..f means the numbers from 10 to 15.
• a single character. The table A.2 list allowed characters.

Date Literals: A date literal consists of four digits, a decimal point, two digits, a decimal point and two digits. The first four digits is the year. The next two digits is the month and the last two digits is the day. (yyyy.mm.dd)

Floating Point Literals: A floating literal point consists of:

• An optional integer part which consists of an optional minus followed by one or more digits.
• A fraction part which consist of a decimal point and one or more digits.
• An optional exponent part which consist of an optional minus followed by e or E and an integer part.

Integer Literals: An integer literal consist of a sequence of digits. An integer literal is taken to be decimal (base ten). The type of an integer literal depends on its value and the target language.

String Literals: A string literal is a sequence of characters. A string is surrounded by double quotes, as in “This is a string literal” or by single quotes , as in ’This is a string literal’. An escape sequence within a string can be written in three ways, all of them starts with a back-slash

• followed by three digits, as in “\214” or “\013”.
• followed by an “x” meaning a hexadecimal number and two characters from [0..9a..fA..F] where a..f means the numbers from 10 to 15.
• followed by one character, as in “\n” meaning newline. The table A.2 list allowed characters.

Within a string the double quote can be used if the string is surrounded by single quotes, and opposite a string can contain single quotes if it is surrounded by double quotes.

A string literal may not contain the character ‘\\’.

<table>
<thead>
<tr>
<th>Description</th>
<th>Abbreviation</th>
<th>Escape</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line</td>
<td>NL(LF)</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT</td>
<td>\t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT</td>
<td>\v</td>
</tr>
<tr>
<td>backspace</td>
<td>BS</td>
<td>\b</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR</td>
<td>\r</td>
</tr>
<tr>
<td>form feed</td>
<td>FF</td>
<td>\f</td>
</tr>
<tr>
<td>alert</td>
<td>BEL</td>
<td>\a</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
<td>&quot;</td>
</tr>
<tr>
<td>single quote</td>
<td>'</td>
<td>&quot;</td>
</tr>
<tr>
<td>double quote</td>
<td>“</td>
<td>&quot;</td>
</tr>
<tr>
<td>hexadecimal number</td>
<td>hh</td>
<td>\xhh</td>
</tr>
</tbody>
</table>

Table A.2: Predefined escape sequence
A.2.6 Preprocessing

In ODL there is the keyword Import. Before a schema is compiled a preprocessor must be run. The preprocessor shall for every import specification replace the import specification with the imported schema (file). If a schema is imported more than once, then only the first substitution shall be done.

A.3 Basic Types

A.3.1 Simple Types

A simple type may be one of the following: Boolean, Byte, Char, Date, Double, Enum, Float, Integer, Longfield, Money, String or Void.

- **Boolean:** A type where value domain is TRUE or FALSE.
- **Byte:** A byte is an eight bit type that can take the values 0 to 255.
- **Char:** In ODL a char defaults to a signed eight bit type. A char can have a value in the range from -128 to 127. It is possible to precede char with modifier Unsigned. It will then take the values 0 to 255.
- **Date:** will be mapped to an object with an interface that supports operations for getting and setting of year, month, day, weekday, week, leapyear, and arithmetic operations for difference, subtraction, addition, increment, and decrement. A C++ mapping for this is specified in [Sol92]. A String-form is used for external representation.
- **Double:** Double represents the IEEE 754\(^1\) double-precision floating numbers. IEEE 754 describes the precision offered by Float and Double.
- **Enum:** An enumerated type consists of an ordered list of identifiers. The first identifier is associated with zero, the second with one and so on. The syntax is:

\[
\langle\text{integer}_\text{size}\rangle \ 'enum' \ [\langle\text{ID}\rangle] \\
\{'\','\langle\text{enumerator}\rangle\}^{*} \ '}'
\]

The \langle\text{integer}_\text{size}\rangle can be small, short, long or hyper (8, 16, 32 or 64 bits). That means that a small enum can have \(2^8 = 256\) different identifiers and a short enum can have 65536 different identifiers. \langle\text{ID}\rangle is a local type for that enum-declaration.

- **Float:** Float represents the IEEE 754 single-precision floating numbers.
- **Integer:** The ODL supports small, int8, short, int16, long, int32, hyper and int64. All of these may be modified with the modifier Unsigned. Table A.3 lists the size of the integers.

All integers are by default signed if unsigned is not used. The size of an unmodified int is not specified, it depends on the target language and the machine it is supposed to run at.

- **Longfield:** This type is a pointer to a file with path and filename. There are operations for copying the contents of a longfield to and from a file-representation in the local environment.

---

\(^1\)IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985
A.3.2 Constructed Types

<table>
<thead>
<tr>
<th>Integer type</th>
<th>Number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>small int</td>
<td>8</td>
</tr>
<tr>
<td>int8</td>
<td>8</td>
</tr>
<tr>
<td>short int</td>
<td>16</td>
</tr>
<tr>
<td>int16</td>
<td>16</td>
</tr>
<tr>
<td>long int</td>
<td>32</td>
</tr>
<tr>
<td>int32</td>
<td>32</td>
</tr>
<tr>
<td>hyper int</td>
<td>64</td>
</tr>
<tr>
<td>int64</td>
<td>64</td>
</tr>
</tbody>
</table>

Table A.3: Integer size

- Money: will be mapped to an object with an interface that supports operations for adding and subtraction of money-values. A String-form is used for external representation.

- String: A string is usually an array of characters or a pointer to a character followed by arbitrarily many characters, where the last character is escape-character 0. Some languages support both. In those systems string will be an array of characters and string0 will be the null-terminated string pointer. The syntax for string is:

  `<string> ::= "string" <string_bounds>

  <string_bounds> ::= "[" ""]"
  | "[" "*" "]"
  | "[" <ID> "]"
  | "[" <integer_literal> "]"

  <ID> must be the name of a constant. The type of <ID> must be evaluated to an integer.

- Void: This type is the same as void in C and C++. An operation returning nothing will have void as the returning type.

A.3.2 Constructed Types

A `<constructed_type>` may be one of the following: Array, BitSet, List, Set, Struct or Union.

- Array `<T>`: An array is not specified as a new type. Instead all types might be extended to an array, by providing a suffix of array bounds to the name of a variable for a type. This is a syntactical convention instead of specifying an array-type directly.

The syntax for Array is

  `<type_spec> <declarator> <array_bounds`

  `<array_bounds> ::= "[" "]"
  | "[" "]"
  | "[" <const_exp> "]"
A type spec can be a basic type or an interface type. If the type spec is an interface type, then the array will be an array of references to objects of the interface type.

```c
Person p[1..10];
double floatd[*];
```

There are seven possibilities for defining array bounds in combinations of specified/unspecified size and specified/unspecified index type. These can be grouped into three categories:

- Arrays with unbounded indices
- Arrays with partially bounded indices
- Arrays with bounded indices (when size, rather than upper and lower bounds, is given, indices are assumed to start at zero).

<const.exp> used in array.bounds is described in section A.3.4. Note that the evaluated type of the <const.exp> that is used must be an ordinal type. That implies that in table A.5 float, string and string0 cannot be used in an <array.bounds>.

Mappings to C++ and Smalltalk are described in [Sol92, Øst92a].

- BitSet: A bitset type consists of an ordered list of identifiers. Each identifier can have the value 0 or 1, or FALSE or TRUE. The first identifier is associated with the least significant bit in the declarator of this type. The second identifier is associated with the second bit and so on. The syntax is:

```c
<integer.size> '{' BitSet '}' [ID]
  '{','<enumerator> {','<enumerator>)* '{'}
```

The <integer.size> can be small, short, long or hyper (8, 16, 32 or 64 bits). That means that a small BitSet can have 8 different identifiers and a short BitSet can have 16 different identifiers and so on.

- SetOf: The syntax for SetOf is:

```c
{SetOf} <type.spec>
```

A <type.spec> can be any basic type, either a simple type or a constructed type.

```c
SetOf double floatSet;
```

floatSet is an attribute representing a set of doubles. The operations that are offered for the set are the same as those offered for sets of objects. There is a distinction between these, because SetOf contains basic values, while Set<T> contains object references.

- Set<T>: Set is a parameterized interface, which is used for set of objects. SetOf is used for basic types.

```c
{Set} '<' <ObjInterfaceID> '>'
```
Set<Person> persons;

The operations provided by a Set are described in Appendix B.

- **ListOf**: The syntax for ListOf is:

  `'ListOf' <type_spec>`

  A `<type_spec>` can be any basic type, either a simple type or a constructed type.

  ```
  ListOf double floatList;
  ```

  `floatList` is an attribute representing a whole list. The operations that are offered for the list are the same as those offered for lists of objects. There is a distinction between these, because ListOf contains basic values, while List<T> contains object references.

- **List<T>**: The syntax for List is

  ```
  'List' '<' <ObjInterfaceID> '>'
  ```

  List<Person> persons;

  The operations provided by a List are described in Appendix B.

- **Struct**: The syntax for struct is:

  ```
  <struct_type_spec> ::= 
  STRUCT [ <structID> ] 
  [ <super_structs> ] '{' <struct_attributes> '}'
  ```

  `<super_structs> ::= 
  ':' <structID> [ ',', <structID> ]`

  `<struct_attributes> ::= 
  { <struct_attribute> ';', }`

  `<struct_attribute> ::= <basic_typeID> <attributeID>
  ```

  The identifier is optional. If used, it will be a local type representing the defined struct. A struct might inherit definitions from one or more predefined structs.

  A `<basic_typeID>` is a simple type or a constructed type.

- **Union**: The syntax for union is:

  ```
  'Union' <tag> 'Switch' '{' <ordinal_type> <identifier> '}'
  ```

  `<union_cases> ::= 
  ```

  The tag is an optional identifier which can be used as a local type later. An ordinal type can be a boolean, byte, char, date, enum type, integer or a scoped name. A scoped name must be a previously defined ordinal type. `<union_cases>` is:
A.3 Basic Types

<union_cases> ::= { <case>* <union_member> ',', }*

<case> ::= 'case' <case_label> ':'
| 'default' ':'

This means that it is possible to write:

case 1:
case 2: int a;
case 3: float a;
default: string[100] a;

A default case can appear at most once. The <case_label> must match or be automatically castable to the defined type of the switch identifier. Table A.4 shows the matching rules.

<table>
<thead>
<tr>
<th>Switch Identifier Type</th>
<th>Matched By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Any integer value in the value range of the integer</td>
</tr>
<tr>
<td>Char</td>
<td>Any character in the ASCII alphabet</td>
</tr>
<tr>
<td>Byte</td>
<td>Any integer value in value range of byte</td>
</tr>
<tr>
<td>Boolean</td>
<td>The values TRUE or FALSE</td>
</tr>
<tr>
<td>Enum</td>
<td>Any enumerator for the switch identifier type</td>
</tr>
<tr>
<td>Date</td>
<td>Any legal date</td>
</tr>
</tbody>
</table>

Table A.4: Case label matching

A.3.3 Type Names

Types can be given new names by the TypeDef <type_declaration>. The syntax for <type_declaration> is:

<type_declaration> ::= 'TypeDef' <type_spec>
<declarator> { ',', <declarator> }*

A <type_declarations> starts with the keyword “Typedef” followed by a <type_specification> and a list of one or more declarators.

A.3.4 Constant Declaration

The syntax for <constant_declaration> is:

'const' <const_type> <ID> '=' <const_exp>

A <const_type> can be a simple type, enum type or a scoped name. A scoped name must be a previously defined simple type or enum type. <ID> is the identifier associated with the value on the right side of the equal sign. <const_exp> must evaluate to the same type as the specified <const_type>. Table A.5 is mapping <const_type> to literals.
### A.3.5 Names and Scoping

The following list contains rules for the three different name-spaces: Global, Schema and Local.

- The name of a schema belongs to a global name-space. The name may be used for naming files, databases, etc and can therefore be in conflict with other schemas.
- Types, enumerators, constants, object interfaces, relation interfaces, etc. declared in a schema are in the same name-space.
- Structs, unions, notifications, exceptions, object interfaces, relation interfaces, containment relation interfaces and operations have their own local name-spaces.
- Union- and struct-members are only in a local name-space.
- Declarations inside object interfaces, notifications, exceptions and relation interfaces are in the local name-space.

### A.4 ODL Grammar

The COOM-compiler has been implemented with lex and yacc. The lex and yacc code is not so easy to read, so the grammar is here instead presented in a more readable form, using the following notation:

CAPITUL letters are keyword-symbols.
"..." literal from "" to ""
[ ... ] Optional from "[" to "]"
{ ... } Grouping from "{" to "}"
* zero or more
+ one or more
<symbol> to be expanded
::= expansion symbol
| or

/******************** Schema ********************/
SCHEMA <schemaName>
   '{',
   [import ' ;' ]*
   [export ' ;' ]+
   '{', '}',

<export> ::= <interface_declaration>
   | <object_interface_declaration>
   | <relation_interface_declaration>
   | <exception_declaration>
   | <notification_declaration>
   | <type_declaration>
   | <constant_declaration>

/******************* Interfaces *******************/

@interface_declaration> ::= INTERFACE <interfaceID> [ <i_param_list> ]
   [supertypes] '{',
   {interface_elem ' ;' } * '{'},
   INTERFACE <interfaceID> /* forward */

<i_param_list> ::= ' ( ' <i_param> ' ; ' , ' <i_param> )* ' ) ' 

<i_param> ::= <i_param_type> <paramID>

<i_param_type> ::= TYPE | INTERFACE | OBJINTERFACE
   | BASE | <simple_typeID>

<paramID> ::= <id>

<supertypes> ::= ' : ' <interfaceID> ' ; ' , <interfaceID>)*

@interface_elem> ::= <operation_declaration>
   | <notification_declaration>
   | <exception_declaration>

@interfaceID> ::= <id> | <id> ' < ' inst_plist ' > ' 

<inst_plist> ::= inst_param { ' ; ' , inst_plist }*

<inst_param> ::= <interfaceID> | <simple_typeID> | <const_exp>

/******************* Operations *******************/

<operation_declaration> ::= <type_spec> <operationName>
   <parameter_decls>
   [raises_expr] [notifies_expr] [context_expr]

<parameter_decls> ::= ' ( ' [ <parameter> { ' ; ' , <parameter>)* ] ' ) ' ;
<parameter> ::= <typeID> <direction> <declarator>
<direction> ::= "[''] IN ['']"
| "[''] OUT ['']"
<typeID> ::= <simple_typeID> | <interfaceID> | <objInterfaceID>
<declarator> ::= <attributeID>
| <declarator> <array_bounds>
<raises_expr> ::= RAISES "('' <exceptionID> 
{ '',' <exceptionID> }* '"
<notifies_expr> ::= NOTIFIES "('' <notificationID> 
{ '',' <notificationID> }* '"
<context_expr> ::= CONTEXT "('' <string_literal> 
{ '',' <string_literal> }* '"

/***************************************************************************/

<object_interface_declaration> ::= 
  OBJINTERFACE <objInterfaceID> [ i_param_list ] 
  [ <supertypes> ] '{'} 
  {<object_interface_elem> ';'} * '{'}
  | OBJINTERFACE <objInterfaceID> /* Forward declaration */

<supertypes> ::= '',' <objInterfaceID> {',' <objInterfaceID>)*

<object_interface_elem> ::= 
  [META] <operation_declaration>
  | [META] <attribute>
  | <control_interface_declaration>
  | <exception_declaration>
  | <uniques>
  | <notification_declaration>

/***************************************************************************/

<notification_declaration> ::= 
  NOTIFICATION <notificationID> 
  [ <super_notifications> ] '{'} <not_attributes> '{'}

<super_notifications> ::=

```
"::" <notificationID> ["",""]<notificationID>]*

<not_attributes> ::= 
{ <not_attribute> """" }

<not_attribute> ::= <simple_typeID> <attributeID>

/******************************** Exceptions ****************************/

<exception_declaration> ::= 
EXCEPTION <exceptionID> 
[super_exceptions] "{" <exc_attributes> "}

<super_exceptions> ::= 
"::" <exceptionID> ["",""]<exceptionID>]*

<exc_attributes> ::= 
{ <exc_attribute> """" }

<exc_attribute> ::= <simple_typeID> <attributeID>

/******************************** Attributes ****************************/

<attribute> ::= [READONLY] <basic_typeID> 
<attributeID> [array_bounds] 
[raises_aexpr] [notifies_aexpr]

.raises_aexpr ::= RAISES "{" <exceptionID> 
{ "" },<exceptionID> } star ""

['IN', "{" "get" | "put" | "get,put" }"

.notifies_aexpr ::= NOTIFIES "{" <notificationID> 
{ "" },<notificationID> } star ""

['IN', "{" "get" | "put" | "get,put" }"

.array_bounds ::= ["","]" |
["","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*","]*" 
```

/******************** Unique Attributes ****************************/

```
<uniques> ::= UNIQUE "('" <key> {"'," <key> }*) '(',')', ');

<key> ::= <attributeID> {"," <attributeID> }*

/************* RelInterfaces *************/

<relation_interface_declaration> ::= RELINTERFACE <relInterfaceID> ["'" <relInterfaceID> ]
 '(', [<roles>] [<rel_attributes>] ')

<roles> ::= <objInterfaceID> "'#'", cardinality [<roleName>] "',"
 <objInterfaceID> "'#'", cardinality [<roleName>] "',"
 <cardinality> ::= '"1"' | '"N"

<rel_attributes> ::= {
 <rel_attribute> "'," }*
 <rel_attribute> ::= <simple_typeID> <attributeID>

<control_interface_declaration> ::= CONTRELINTERFACE <relInterfaceID> [<"'" <relInterfaceID> ]
 '(', '"#", cardinality [<roleName>] "',"
 <objInterfaceID> '"#", cardinality [<roleName>] "',"
 [<rel_attributes>] ')

/************* TypeDef *************/

<type_declaration> ::= TYPEDEF <basic_typeID>
 <declarator> {"','" <declarator> }

<basic_typeID> ::= <simple_typeID> | <constructed_typeID>

/************* Simple types *************/

/* Boolean, Byte, Char, Date, Double, Float, Integer, Longfield, Money, 
   String and Void. */

<simple_typeID> ::= BOOLEAN
 | BYTE
 | CHAR
 | UNSIGNED CHAR
DATE
FLOAT
DOUBLE
<integer_type_spec>
LONGFIELD
MONEY
<string>
<enum_typeID>

VOID
SELF

<integer_type_spec> ::= 
<integer_modifiers>
<integer_modifiers> INT
INT
UNSIGNED INT

<integer_modifiers> ::= 
<integer_size_spec>
UNSIGNED integer_size_spec
integer_size_spec UNSIGNED

<integer_size_spec> ::= 
SMALL | INT8
SHORT | INT16
LONG | INT32
HYPER | INT64

<string> ::= "string" <string_bounds>

<string_bounds> ::= "[", "]"
| "[", "+", "]"
| "[", <int_const_exp> "]"
| "[", <integer_literal> "]"

<enum_type_spec> ::= 
<integer_size> ENUM [<simple_typeID>]
"{" <enumerator> "," <enumerator> }* ""

/************** Constructed types *******************/
BitSet, Enum, List, Set, Array, Struct, Union,

<constructed_type_spec> ::= 
<bitSet_type_spec>
| <struct_type_spec>
| <union_type_spec>
| SETOF <basic_typeID>
| LISTOF <basic_typeID>

<bitSet_type_spec> ::= 
    <integer_size> BITSET [<constructed_typeID>] 
    '('|' ') <enumerator> {'|', ' ') <enumerator>* '|

<struct_type_spec> ::= 
    STRUCT [<structID>] 
    [<super_structs>] '(' <struct_attributes> ')'

<super_structs> ::= 
    '|' <structID> ['|', <structID>]*

<struct_attributes> ::= 
    '{' <struct_attribute> '}'*

<struct_attribute> ::= <basic_typeID> <attributeID>

<union_type_spec> ::= 
    UNION <constructed_typeID> SWITCH '('|' ') <ordinal_type> <identifier> ')' 
    '(' ')
    <union_cases>+
    '|

<union_cases> ::= { <case>+ <union_member> ' ; ' }*

<case> ::= CASE <case_label> ' ; ' 
    | DEFAULT ' ; '

<ordinal_type> ::= 

/****** Constants **********/

CONST <simple_typeID> <ID> '=' <const_exp>

<const_exp> ::= 
    <character_literal> 
    | <date_literal> 
    | <double_literal> 
    | <float_literal> 
    | <integer_literal> 
    | <string_literal> 
    | <enumerator> 
    | TRUE /* Boolean */ 
    | FALSE /* Boolean */ 
    | 0...255 /* Byte */
The use of parameterized types gives automatic support for the predefined interfaces LIST<> and SET<>.

A.5 ODL Example

This example shows the complete ODL-schema for Person, Employee, Department, and their relations.

SCHEMA PersonSchema {
    import Standards;
    import Basics;

    Notification Changed {
    }

    Notification ChangedAge: Changed {
        int age;
    }

    Exception IllegalAge {
        int age;
    }

    Exception IllegalEmpAge: IllegalAge {
        int lower;
        int upper;
    }

OBJINTERFACE Person {
    META Person create(String [in] name, int [in] personNumber, int [in] age);
    META int averageAge();
    META readonly int nrOfPersons; /* Meta attribute */

    String name;
    int age RAISES(IllegalAge) IN (put) NOTIFIES(NewAge) IN (put);
    readonly int personNumber;
    UNIQUE(personNumber);

    ENUM Hobby {tennis, soccer, golf, other};
    SET<Hobby> hobbies;
}; /* ObjInterface Person */

OBJINTERFACE Employee: Person {
    META Employee create(String [in] name, int [in] personNumber,
        int [in] age, int [in] salary);
    META int averageSalary();
    int salary();
    void salary(int newSalary) RAISES(illegalSalary) Notifies(ChangedSalary);
int salary RAISES(illegalSalary) IN (set) NOTIFIES(ChangedSalary) IN (set);
void age(int [in] newAge) raises(illegalEmpAge) notifies(ChangedAge);

}; /* ObjInterface Employee */

OBJINTERFACE Department {
    Location loc;
    String name;
}; /* ObjInterface Department */

RELINTERFACE AssociatedWith {
    Person #N associates;
    Department #1 associated;
};

RELINTERFACE WorksFor < AssociatedWith {
    Employee #N employees;
    Department #1 works_in;
    Date startDate;
};

OBJINTERFACE Company {
    CONTRELINTERFACE TheDepts {
        #1;
        Department #N;
    };
    Location loc;
    String name;
}; /* Company ObjInterface */

}; /* PersonSchema */
Appendix B

COOM OML - Object Manipulation Language

This appendix contains ODL definitions for predefined object and relation interfaces, and support services, and an example of the use of some of these from C++.

- **Object interfaces**
  ObjInt, ObjImp, METAObjInt, METAObjImp, M(Type T), MI(Type T)

- **Relation interfaces**
  ROI, ROI, R(F, T, A), RI(F, T, A), RIT(F, T, A)

- **Interfaces for Support services**
  Collection(Type T), Set(Type T), List(Type T), Object Space, Transaction, Notification Management

- **C++ Example**
  Predefined framework-classes, interface-classes generated by the COOM ODL Compiler, implementation-classes generated by a component-integrator, and tool/application-classes created by a tool-builder.

B.1 Object Interfaces

B.1.1 ObjInt

```cpp
INTERFACE ObjInt
{
  Self copy();
  ObjInt oicopy();
  Self shallowCopy();
  Self deepCopy();
  void delete();

  METAObjInt objIntMan();
  METAObjImp objImpMan();

  InterfaceDef interfaceDef();
```
ImplDef implements();

OSpace oSpace();
String oidAsString();
Boolean isA(METAObjInt [in] m);

INTERFACE ObjImp: ObjInt
{String implemName();
};

B.1.2 METAObjInt

INTERFACE METAObjInt
{
    ObjInt createOId();
    Set<ObjInt> findOId(String [in] queryString);
    Set<ObjInt> extentOId();
    ObjInt tradeOId(String [in] tradeString);
};

INTERFACE METAObjImp: METAObjInt
{
    String implemName();
    OSpace ospaces();
};

B.1.3 M(OBJINTERFACE T)

INTERFACE M(OBJINTERFACE T): METAObjInt
{
    T create();
    T createInObjSpace( OSpace [in] os);

    Self getobjectFromOId( String [in] oid);
    Set<T> extent();
    Set<T> directExtent();

    Set<T> find(String [in] queryString);
    T trade(String [in] tradeString);

    addImplementation( MI<T> [in] aMan, OSpace [in] for);
    removeImplementation( MI<T> [in] aMan, OSpace [in] for);
    Set<MI<T> > implementations();

    addSubType( METAObjInt [in] om);
B.1.4 MI(Type T)

removeSubType(METAobjInt [in] om);
Set< METAobjInt> subTypes();

B.1.4 MI(Type T)

INTERFACE MI(Type T): METAobjImp, M<T>
{
T create(); /* Default ObjSpace */
T create(ObjSpace [in] os);
T getobjectFromOID(String [in] oid);
Set<T> extent();
Set<T> find(String [in] queryString);
T trade(String [in] tradeString);
}

B.2 Relations

B.2.1 Basic definitions

ENUM Cardinality {11, 1N, N1, MN};

Exception CardinalityViolation {
Cardinality valid;
};

Notification Inserted {
OIDString relIntMan;
OIDString From;
OIDString To;
};

Notification Removed {
OIDString relIntMan;
OIDString From;
OIDString To;
};

Struct RAtt {
};

ENUM Cardinality (C11, C1N, CN1, CMN);

B.2.2 ROI

INTERFACE ROI
{

insertOI(ObjInt [in] from, ObjInt [in] to, AttO [in] a);
removeOI(ObjInt [in] from, ObjInt [in] to, Att [in] a);

Set<ObjInt> getFromObjOI(ObjInt [in] from);
Set<ObjInt> getToObjectOI(ObjInt [in] to);

RTOI getIteratorOI();
RTOI getIteratorFromOI(ObjInt [in] fromObject);
RTOI getIteratorToObjectOI(ObjInt [in] toObject);
RTOI getIteratorToObjectOI(ObjInt [in] toObject);

addImplementation( RI<F,T,A> [in] rImpMan, OSpace [in] for);
removeImplementation( RI<F,T,A> [in] rImpMan, OSpace [in] for);
Set< RI<F,T,A> > implementations();
Set< RI<F,T,A> > rImpManagers();

/* The individual parameters might be converted from generic
OI to the same as this
rel interface manager type */

addSubRelation( ROI [in] rMan);
removeSubRelation( ROI [in] rMan);
Set< ROI<F,T,A> > subRelations();
};

B.2.3 RI(F, T, A)

INTERFACE RIOI: ROI
{ /* For implementation - same interface as ROI */
};

INTERFACE RI(OBJINTERFACE From, OBJINTERFACE To, BASE Att): R<F, T, A>
{
};

B.2.4 R(F, T, A)

INTERFACE R (OBJINTERFACE From, OBJINTERFACE To, BASE Att): ROI
{
changeCardinalityTo(Cardinality [in] card);
Cardinality getCardinality();
int nrelships();

insert(From [in] from, To [in] to, Att [in] a)
    Raises(CardinalityViolation) Notifies(Inserted);
remove(From [in] from, To [in] to, Att [in] a)
    Raises(NotExist) Notifies(Removed);

Set<To> getFromObj(From [in] from);
Set<From> getToObj(To [in] to);
RIT<From, To, Att> getIterator();
RIT<From, To, Att> getIteratorFrom(From [in] fromObject);
RIT<From, To, Att> getIteratorFromTo(From [in] fromObject, To [in] toObject);
RIT<From, To, Att> getIteratorTo (To [in] toObject);
addImplementation( RIT<From, To, Att> [in] rImpMan, OSpace [in] for);
removeImplementation( RIT<From, To, Att> [in] rImpMan, OSpace [in] for);
Set< RIT<From, To, Att> > implementations();

/* Subrelations might be converted from generic OI to the correct super-type !! */
addSubRelation( ROI [in] rMan);
removeSubRelation( ROI [in] rMan);
Set< ROI<From, To, Att> > subRelations();
}

/* Notifications On/Off */
BOOL activateNotifications();
BOOL passivateNotifications();
BOOL NotificationsActive();

B.2.5 RIT(F, T, A)

INTERFACE RIT (OBJINTERFACE From, OBJINTERFACE To, BASE RAtt)
{
    Boolean atEnd();
    first();
    next();
    From from();
    To to();
    RAtt att();
};

INTERFACE RITOI
{
    Boolean atEnd();
    first();
    next();
    ObjInt from();
    ObjInt to();
    Att att();
};
B.3 Interfaces for Support services

B.3.1 Set(Type T)

INTERFACE Collection (Type T)
{
    Boolean insert(T [in] t); // add
    Boolean remove(T [in] t);
    Boolean in(T [in] t);
    Boolean makeEmpty();
    Boolean isEmpty();
    int cardinality();
    T getFirst();
    T getNext();
    T getLast();
    T getPrevious();
    Iterator<T> getIterator();
};

INTERFACE Set (Type T): Collection<T>
{
    Set<T> intersection(Set<T> [in] s);
    Set<T> difference(Set<T> [in] s);
    Set<T> union(Set<T> [in] s);
    Boolean equal(Set<T> [in] s);
    Boolean subSetOf(Set<T> [in] s);
    Boolean addSet(Set<T> [in] s);
};

B.3.2 List(Type T)

INTERFACE List(Type T): Collection<T>
{
    Boolean addAt(T [in] t, int [in] index);
    Boolean deleteAt(int [in] index);
    T get(int [in] index);
    Boolean insertBefore(T [in] before, T [in] new);
    Boolean insertAfter(T [in] after, T [in] new);
    Boolean subListOf(List<T> [in] s);
    Boolean addList(List<T> [in] s);
};

B.3.3 Object Space

Interface OSpace
{
B.3.4 Notification Management Services

OBJINTERFACE LocalNotificationManager {
    notify(Interface [in] from, Notification [in] not);
    register(Interface [in] interested_object,
        Notification [in] not, Interface [in] from);
    registerCB(Interface [in] interested_object, OperationDef [in] op,
        Notification [in] not, Interface [in] from);
    unregister(Interface [in] interested_object,
        Notification [in] event_type, Interface [in] from);
    check_for_Notification()
    wait_for_notification()
}

OBJINTERFACE NotificationReceiver {
    posted_notification(Notification [in] not, Interface [in] from)
}

INTERFACE Notification {
    String name();
    String writeAttToString();
    readAttFromString(String [in] str);
}

ObjInterface GlobalNotificationManager {
    notify(ODIString [in] from, String [in] notificName,
        String [in] notificString);
    register(ODIString [in] from, String [in] notificName,
        PortID [in] processPort);
    unregister(ODIString [in] from, String [in] notificName,
        PortID [in] processPort);
}

Interface LocalNotificationManagerPort {
    eventReport(ObjIntString [in] from, String [in] notificName,
        String [in] notificString);
}

B.4 OML C++ Example

The following contains some selected class-definitions in C++. The example is taken from the Person-Department scenario described in section 9.8 and section 15.9.

The following 4 categories of classes are presented:

- Predefined framework-classes.
- Interface-classes generated by the COOM ODL Compiler.
- Implementation-classes generated by a component-integrator.
- Tool/application-classes created by a tool-builder.

B.4.1 Interface Classes - Framework predefined

The predefined framework-classes have the functionality described through the ODL definitions in the previous section, and their C++ equivalent is not repeated here.

class ObjInt { ... }; // For ..., see ODL specifications
class METAObjInt { ... };

template<class T>
class M: public METAObjInt
{
  public:
  virtual T* create();
  virtual Set<T>** find(String*);
  ...
};

class ROI { ... };

template<class F, class T, class A>
class R : public ROI {
  public:
  virtual void insert(F* f, T* t, A* a);
  virtual Set<T>** getFromObj(F*);
  virtual RIT<F,T,A>** getIterator(F* f, T* t, A* a);
  ...
};

template<class F, class T, class A >
class RIT : public RITOI {
  public:
  virtual void next();
  virtual F* from();
  virtual T* to();
  virtual A* att1();
  ...
};

B.4.2 Interface Classes - Generated by ODL Compiler

/*** PERSON */

class Person: public ObjInt {
  private:
virtual void personNumber(int newPersonNumber); // readonly

public:
// Operations of the interface manager is made available through
// static definitions, so that a tool-builder do not have to use
// the manager objects.
static void* setDefaultSpace();
static void* cacheHint(String* hint);
static Set<Person*>* extent();
static Set<Person*>* find(String* queryString);
static Person* create(String* name, int personNumber, int age);
...  
  virtual String* name() = 0;
  virtual void name(String* newName);
  virtual int age();
  virtual void age(int newAge); // throws(IllegalAge);
  virtual int personNumber();
/* Relationship-access AssocWith */
  virtual Department* assocDept();
  virtual void assocDept(Department* newDepartment);
...  
};

class METAPerson: public M<Person> {
public:
  virtual Person* create(String* name, int personNumber,
                        int age, OSpace* os);
  virtual int averageAge();
  virtual int nrOfPersons();
  ...  
};

/*** EMPLOYEE ***/
class Employee: public Person {
public:
  static Set<Employee*>* find(char* queryString);
  virtual int salary();
  virtual int salary(int newSalary);
  // role-access
  virtual Department* department();
  virtual void department(Department* newDepartment);
  ...  
};

class METAEmployee: public M<Employee> {
public:
  virtual Employee* create(String* name, int age, OSpace* os);
  virtual int averageSalary();
  ...  
};
/*** DEPARTMENT ***/
class Department{
    public:
    virtual Set<Employee*>* employees();
    ...
};

class METADepartment: public M<Department> {
    public:
    Department* create(String* name, OSpace* os);
    ...
};

// The following interface managers are made available through
// the persistent variable facility of the object-oriented database used for
// managing the distributed object space.
// OD is the object dictionary space
// A Tool Builder do not have to use these, and can access the necessary
// functionality through static operations for the class.

persistent<od> METAPerson* metaPerson;
persistent<od> METADepartment* metaDepartment;
persistent<od> METAEmployee* metaEmployee;

/*** WORKSFOR relation interface ***/
persistent<od> R<Person, Department, AWorksFor>* WorksFor;

B.4.3 Implementation Classes - Generated by a Component Integrator

These classes contain the necessary mapping code.

Class PersonIngres: public Person {
    ...
}

Class METAPersonIngres: public MI<Person>{
    ...
};

Class EmployeeIngres: public Employee {
    ...
}

Class DepartmentIngres: public Department {
    ...
}

// The following implementation managers are associated
// with the interface manager for Person and WorksFor
persistent<Ingres1> MAPersonIngres* metaPersonIngres1;
persistent<OS1> MAPersonOS* metaPersonOS1;

persistent<OS1> RIOS<Employee, Department, AWorksFor>* WorksForOS1;
persistent<Ingres1> RIngres<Employee, Department, AWorksFor>* WorksForIngres1;

B.4.4 Tool/Application - Created by a Tool Builder

Class DeptEmpModel shows operations that support the required functionality of the tool. The implementation of these operations will utilize the classes made available by the component-integrators. Some of these implementations are shown to illustrate how a tool builder interacts with the interface-classes. The tool-builder will only indirectly use the implementation-classes, through the interface managers.

/*** DEPTEMPMODEL ***/

class DeptEmpModel: public Model {
  public:
    virtual Set<Department>* getDepartments();
    virtual Set<Employee>* getEmployeesInDept(Department* d);
    virtual int getSalaryForEmp(Employee* e);
    virtual int setSalaryForEmp(Employee* e, int newSalary);
    virtual Department* createDepartment(String* name);
    virtual Employee* createEmployeeInDept(Department* d,
                                           String* name, int pnr, Date* startDate);
    virtual Set<Employee>* findEmployeeInRangeInDept(int min,
                                                      int max, Department* d);

    DeptEmpModel(); // Initiate Object-Spaces
    ~DeptEmpModel();
    ...
};

// Initiate Object-Spaces in ContextFile:
"os1 Ingres1
os2 ObjStore1
DefaultOS Ingres1
Person Ingres1"

DeptEmpModel::DeptEmpModel()
{
  // Initiate object-spaces from context
  Dictionary<String, OSpace>* OSpace::activateFromContext();
  ...
}

="/*********************************************************************************/
Set<Department>* DeptEmpModel::getDepartments()
{

return Department::extent();
// return metaDepartment->extent(); /* direct use of interface manager */
}

//***************************************************************/
Set<Employee*>* DeptEmpModel::getEmployeesInDept(Department* d)
{
    return d->employees(); /* direct role access */
    // return WorksFor->getObjTo(d); /* use of relation interface manager */
}

//***********************************************************************/
int DeptEmpModel::getSalaryForEmp(Employee* e)
{
    return e->salary();
}

//***********************************************************************/
int DeptEmpModel::setSalaryForEmp(Employee* e, int newSalary)
{
    try{
        return e->salary(newSalary);
    } catch(IllegalSalary is) {
        cerr << is.printString << ia.value << '
';
    }
}

//***********************************************************************/
Department* DeptEmpModel::createDepartment(String* name)
{
    Department::creationSpace(os1);
    Department* e1 = Department::create(name);
    return d1;
}

//***********************************************************************/
Employee* DeptEmpModel::createEmployeeInDept(Department* d, String* name, int pnr, int age, Date* startDate)
{
    ...
    Employee::creationSpace(os1);
    Employee::cacheHint("name, age, salary");

    Employee* e1 = Employee::create(name, pnr, age, salary);
    // Employee* e1 = metaEmployee->create(name, pnr, age, salary);

    AWorksFor* awf = new AWorksFor();
    awf->startDate = Date::today();

    WorksFor->insert(e1, d, awf);
    return e1;
}
Set<Employee*>* DeptEmpModel::findEmployeeInRangeAge(int minAge, int maxAge, Department* d1)
{
    return Employee::find(" age >= %i && age <= %i && worksFor == %o ", d1);
}

The user-interface part of the tool is created by the use of the object-oriented application framework, CommonView. The browser-tool is created as an instance of class UIWin, as a subclass of the generic application window class. An object of UIWIN is initiated with a model object of class DeptEmpModel.

class UIWin : public TopAppWindow
{
    DeptEmpModel* model;
    ListBox* LeftPane,* RightPane;
    ...
};

UIWin::UIWin(DeptEmpModel* m)
{
    model = m;
    menu.AppendItem(1,"New Department");
    menu.AppendItem(2,"New Person in Department");
    LeftPane = new ListBox(this, 1, Point(0,0), Dimension(0,0));
    RightPane = new ListBox(this, 2, Point(0,0), Dimension(0,0));
    LeftPane->AddModel(m->getDepartments());
    ...
};
Appendix C

OQL Grammar

This appendix gives a more detailed description of COOM OQL (Object Query Language). The syntax of OQL is an extension to the REBOOT AQL as described in [OH91].

The following is based on the current version of the COOM OQL compiler, as described in [Ste92], with additions for object-values, rolenames, arguments, operations, nested queries.

A more comprehensive and alternative grammar based on an inclusion of concepts from SQL is described in [Low92].

C.1 OQL Grammar

\[
\begin{align*}
\text{<predicate> } &::= \text{<atomic Predicate>} \\
&\quad | \text{<predicate> } '&&' \text{<predicate>} \\
&\quad | \text{<predicate> } '||' \text{<predicate>} \\
&\quad | '((' \text{<predicate>} ')')' \\
&\quad | '(!)' \text{<predicate>} \\
\text{<atomic Predicate> } &::= \text{<comparison> } | \text{<quantifier>} \\
\text{<comparison> } &::= \text{<lvalue> } \text{<operator> } \text{<rvalue>} \\
\text{<lvalue> } &::= \text{<dot list>} \\
\text{<rvalue> } &::= \text{<dot list>} \\
&\quad | \text{<const exp>} \\
&\quad | \text{<nestedQuery>} \\
&\quad | \text{<argument>} \\
\text{<dot list> } &::= \text{<dot element>} \\
&\quad | \text{<dot list> } '.' \text{<dot element>} \\
\text{<dot element> } &::= \text{<relInterfaceID>} \\
&\quad | \text{<relInterfaceID>}_\text{INV} \\
&\quad | \text{<roleName>} \\
\end{align*}
\]

349
The following has been defined in the ODL grammar in Appendix A: <relInterfaceID>, <roleName>, <objInterfaceID>, <const_exp>, <operationName>.

C.2 OQL Examples

Set<Person> * teenAgers = metaPerson->find("age >= 12 & age <= 19");

Set<Employee> * s = metaEmployee->find("age >= 20 & age <= 30
 & worksFor.name == 'Computer Science'");

Set<Department> * s1 =
    metaDepartment->find("worksFor_INV[age >= 20 & age <= 30] ");

Set<Department> * s2 = /* Uses role-access through employees */
    metaDepartment->find("employees[age >= 20 & age <= 30] ");

Set<Employee> * s =
    metaEmployee->find(" department IN Company{TheDepts.name== 'SALE'} ");
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


[Tel90] TeleSOFT. *Architecture experience from ARISE and Museion*, October 1990.


Index

Abstract attribute, 210, 260
  definition, 211
  unique combination, 213
AD/Cycle, see SAA
ANSA, 310
Apple IAC, 82
Application Framework, 97, 194
ARISE, 83
ATIS, 83
Autonomy, 273

Basic Types
  definition, 160
Behavioural Model, see COOM_B
Behaviourally object-oriented model, 117
BMS, 68
Broadcast Message Server, see BMS

Carnot, 82
Class, 93, 100
Coarse-grain functional service, see Large-grain functional service
CODE, 264, 303
CODEDISC, 109, 136, 306
Common Object Request Broker Architecture, see CORBA
CommonView, 138
Component integration, 270
  fine-grain functional service, 288
  large-grain functional service, 285
OODBMS, 281
RDBMS, 275
Component-builder, 256
Component-integrator, 256, 262, 265, 270
Conformance, 100, 165
Containment relation
  definition, 226
Context, 202
  definition, 202
  example, 203
  management, 203
  syntax, 203
CONTRELINTERFACE, see ODL
Control integration, 24, 34, 155, 173, 297
  notification-oriented, 38, 107, 298

Coom evaluation, 299
  definition, 38
  requirement areas, 38
  request-oriented, 34, 106, 298
COOM evaluation, 298
  definition, 34
  requirement areas, 35
solution-evaluation, 86
CODEDISC, 113
SITE, 113
solutions
  COOP/COOM, 173, 207
distributed computing, 55
ESF, 79
HP/Softbench - Field, 68
Object-Oriented Architectures, 95
OMG, 63
PCTE, 73
Pegasus, 75
SITE, 106
COOM, 150
COOM_B, 150, 173
  C++, 190
  definition, 174
  Smalltalk, 193
COOM_F, 150, 157
  definition, 158
COOM_S, 150, 209
  C++, 230
  definition, 210
  Smalltalk, 231
COOP, 143
  architecture, 148, 204, 264, 302
  logical, 204
  physical, 206
  prototypes, 306
  vision, 3, 145
COOP Development Environment, see CODE
COOP Methodology, see COORASS
COOP Object Model, see COOM
Cooperative Work Environment, 15
Cooperative Work Support, 18
solutions, 21
COORASS, 154, 255, 259, 302
  Tool-builder, 262, 263
INDEX

CORBA, 64, 207

Data integration, 20, 24, 39, 155, 209, 299
  multi-model, 301
  COOM, 232
  COOM evaluation, 301
  definition, 44
  requirement areas, 45
object-oriented, 125
single-model, 39, 125, 300
  COOM evaluation, 300
  definition, 39
  requirement areas, 40
solution-evaluation, 86, 126
solutions
  COOP/COOM, 209, 232
  databases, 57, 125
  ESF, 80
  HP/Softbench - Field, 70
Object-Oriented Databases, 95
OMG, 66
PCTE, 73
Pegasus, 76
Disjoint objects, 233
Display-oriented Presentation integration,
  see Presentation integration

DOM, 189, 310
Dynamic Binding, 94

ECMA/NIST Toaster, 24
EIS, 310
Encapsulation, 93
End-user, 256
Equivalent objects, 233
ESF, 78, 84, 135, 309
  CoRe, 26
Eureka Software Factory, see ESF

Exception, 199
  definition, 199
  example, 200
  management, 201
  syntax, 200
Extent, 101, 151
  definition, 170

Federated Database, 58
Fine-grain functional service, 288
Foundation Model, see COOM
Framework-builder, 256
Fully object-oriented, 150
Functional service, 285
  fine-grain, 288
  large-grain, 286

Generalization, 236

HP/Softbench - Field, 67, 83
HyperModel Benchmark, 121
  application, 122
  description, 121
  experiences, 124
  operations, 122

Implementation, 100, 151
  definition, 169
Implementation Manager Object
  definition, 180
Inheritance, 94
Instance, 93
Instantiation, 94
Integration
  Control integration, 24
  Data integration, 24
  definition, 17
    Data integration, 20
    Location integration, 20
    Presentation integration, 20
    Platform integration, 25
    Presentation integration, 24
    Process integration, 24
    requirements, 33
    Tool Building integration, 25
    Tool Insertion integration, 25
Integration Space
  definition, 219
Intent, 101
Interaction, 26, 27, 29
Interconnection, 27, 29
INTERFACE, see ODI.
Interface, 100, 151
  definition, 162
  parameterized, 168

Interface Manager Object
  definition, 175
Interfunction, 27, 29
Interoperability
  definition, 17
Interoperation, 27, 29
Interworking, 26, 27, 29

Large-grain functional service, 285
Late Binding, see Dynamic Binding
Location integration, 20
Manager object, 174
Mapping, see Schema reconstruction
Message, 93
META, 175
INDEX

Method body, 93
Model-oriented Presentation integration, see Presentation integration
Model-View-Controller, see MVC
Multi-model Data integration, see Data integration
Multidatabase, 58
Multiple inheritance, 166
Musieon, see ARISE
MVC, 134
N-ary relations, 232
Notification, 195
definition, 195
definition, 195
definition, 195
definition, 195
definition, 195
example, 196
example, 196
example, 196
example, 196
management, 197
management, 197
management, 197
management, 197
syntax, 196
syntax, 196
syntax, 196
syntax, 196
Notification-oriented Control integration, see Control integration
NOTIFIES, see ODL

Object, 93
definition, 160
Object Definition Language, see ODL
Object Dictionary, 205, 272
Object identity, 273
fine-grain functional service, 289
large-grain functional service, 286
OODBMS, 282
RDBMS, 275
Object Implementation
definition, 179
fine-grain functional service, 290
large-grain functional service, 288
management
fine-grain functional service, 289
large-grain functional service, 287
OODBMS, 282
RDBMS, 277
OODBMS, 284
RDBMS, 279
Object Interface, 174
definition, 174
management, 272
Object Management Group, see OMG
Object Manager, 174
Object Manipulation Language, see OMG
Object Query Language, see OQL
Object Request Broker, see CORBA
Object Space, 187, 272
definition, 170
Integration Space, 290
interface, 187

Object-Oriented Analysis and Design, 98
Object-Oriented Application Frameworks, 97
Object-Oriented Architectures, 95
Object-Oriented Databases, 95, 124, 130
Client-oriented, 97, 128, 206
Server-oriented, 97, 129, 206
Object-Oriented Programming
definition, 92-94
Object-Oriented Programming Languages, 98, 190
Object-Oriented User Interfaces, 97
ObjectSpace
Integration Space, 205
Local Object Space, 205
OBJINTERFACE, see ODL
ODL, 152
CONTEXT, 203
CTRLINTERFACE, 227
EXCEPTION, 200
IN, 212
META, 176
NOTIFICATION, 196
NOTIFIES, 197, 212
OBJINTERFACE, 174
RAISES, 201, 212
REAONLY, 212
RELINTERFACE, 217
SCHEMA, 177
UNIQUE, 214

ODL Schema, 161, 177, 216
definition, 160
OID, 179, 273
OMG, 62, 83, 309
OML, 153

Object Interface Operations, 178
copy, 179
delete, 179
information, 179
Object Manager Operations, 182
addImplementation, 184
addSubType, 184
cacheHint, 186
create, 184, 278, 284, 287, 290
creationSpace, 186
extent, 186, 277, 282, 287, 290
find, 186, 283, 290
getFromOIDString, 185
implementations, 184
removeImplementation, 184
removeSubType, 184
subTypes, 184
trade, 186
INDEX

Object Space operations, 187
Relation Iterator operations, 224
Relation management operations
  addImplementation, 228
  addSubRelation, 229
  changeCardinality, 229
  getCardinality, 229
  implementations, 228
  removeImplementation, 228
  removeSubRelation, 229
  subRelations, 229
Relation operations
  getFromObj, 222
  getIterator, 224
  getIteratorFrom, 224
  getIteratorFromTo, 224
  getIteratorTo, 224, 280, 285
  getToObj, 222
  insert, 222, 280, 284
  remove, 222
Transaction operations, 188
OORAM, 255
OORASS, 99, 255, 256
Operation
  definition, 163
  semantics, 167
Operation-signature
  conform, 165
OQL, 154, 241
  comparisons, 245
  dot notation, 246
  motivation, 241
  nested queries, 247
  object-generating, 243
  object-preserving, 243
  query arguments, 248
  query processing, 249, 311
  relations, 246
  SQL inspiration, 251
  syntax, 244
Ordered relations, 232
PCTE, 71, 83
Pegasus, 75, 84
PIONA, 138
Platform integration, 25
Polymorphism, 94
Portable Common Tool Environment, see PCTE
Presentation integration, 20, 24, 49, 50, 135, 136, 155, 290, 303
display-oriented
  COOM evaluation, 303
definition, 50
requirement areas, 50
model-oriented
  COOM evaluation, 304
definition, 50
requirement areas, 50
object-oriented, 133
solution-evaluation, 86, 138
solutions
  CODEDISC, 136
  COOP/COOM, 269
  ESF, 81
  HP/Softbench - Field, 70
  object-oriented, 133
  Object-Oriented User Interfaces, 97
  OMG, 67
  Pegasus, 77
  SITE, 135
  user interface tools, 59
Process integration, 24, 31, 52, 81, 312
  architectural support, 52, 61
  interaction-oriented, 53, 302
  interworking-oriented, 53, 302
  process interaction engine, 28, 53, 60, 302
  process interworking engine, 28, 53, 60, 302
solutions, 60
  BMS, 70
  COOP/COOM, 302
  ESF, 81
  HP/Softbench - Field, 70
  OMG, 67
  PCTE, 75
  Pegasus, 77
RAISES, see ODL
REBOOT, 309
Relation, 102, 260
  definition, 214
  example, 215
Relation Implementation
  definition, 218
  fine-grain functional service, 290
  large-grain functional service, 288
OODBMS, 284
RDBMS, 280
Relation Implementation Manager
  definition, 219
Relation Interface
  definition, 217
  management, 273
  syntax, 217
Relation Interface Manager
  definition, 218
Relation iterator
  example, 224
  operations, 223
Relationship, 102
  definition, 214
RELINTERFACE, see ODL
Request-oriented Control integration, see
  Control integration
Role model, 259
SAA, 82
SC, 106, 135
Schema enhancement, 290
Schema integration, 233, 266, 290, 292, 311
Schema reconstruction, 265, 270, 311
  fine-grain functional service, 289
  large-grain functional service, 286
  OODBMS, 282
  RDBMS, 276
Self
  definition, 167
Semantic interoperability, 233
Semantic Object Oriented Model, see SOOM
Service, see Functional service
Service Component, see SC
SI Team Environment, see SITE
Single-model Data integration, see Data integration
SITE, 105, 135
Software Bus, 79
SOOM, 115
Structural Model, see COOMs
Structurally object-oriented model, 117
Sub-relation
  definition, 227
Substituteability, 102, 165
Synthesis, 258
Systems Application Architecture, see SAA
Systems integration, 155, 194, 269, 295
Tool Building integration, 25
Tool Insertion integration, 25
Tool-builder, 256, 263, 267, 290
ToolTalk, 82
Tornado*, 118
Transaction, 120, 187, 190
  interface, 187
Type, 100
UIC, 106, 135
UNIQUE, see ODL
User Interaction Component, see UIC
User Task Support, 18, 19
  solutions, 19
Value
  definition, 159
Work Process Support, 18, 22, 105
  solutions, 23
Work-context, 106
ZOOIFI, 310