Evolution and Integration of Classes in Object-Oriented Databases

Svein Erik Bratsberg

June, 1993
Abstract

For most database applications there is a need to let the schemas evolve and be integrated after the databases have been populated. Despite arguments in favour of using object-oriented databases in “evolutionary environments”, traditional object-oriented databases have limited support for schema evolution and integration. Due to this, many special-purpose techniques have been proposed: specialisation of classes, generalisation of classes, versioning of classes, class integration by views, and others.

This thesis has four main contributions. First, it introduces a class evolution framework that covers and extends the essential and common mechanisms of other proposed techniques for class evolution and integration. This gives the class designer considerable flexibility compared to the traditional, predefined “evolution operators”. Second, at the core of the framework is a separation of the intensional and extensional dimensions of classes into two hierarchies, which allows for class hierarchies to be evolved in “all directions”. Third, we use mathematical functions to specify dependencies in between objects residing in different classes. The specifications allow for reasoning about representation of objects, tuning the implementation of consistency maintenance, and to ensure termination of consistency maintenance in some cases of cyclic dependencies. Fourth, unlike existing class evolution techniques, all information regarding evolution is external to classes. This lets classes created as a result of evolution to become ordinary classes. Thus, they may be understood and further evolved like any other class.

The evolution framework is applied in two directions, both being accompanied by appropriate tools: Class derivation treats evolution where new classes are small adaptations of existing classes. Class integration covers the integration of independently developed classes. For class integration we have developed a methodology which exploits both the intensional and extensional dimensions of classes.
To my parents, Hildur and Svend.
Contents

1 Introduction ........................................... 1
   1.1 The emergence of object-oriented databases .............. 1
   1.2 Evolution and integration ............................. 3
   1.3 Context for our work .................................. 5
   1.4 Claimed contributions ................................ 5
   1.5 Overview of thesis .................................... 7

2 Background for Evolution and Integration ................. 9
   2.1 Object-oriented database concepts ...................... 9
   2.2 Database requirements from design environments ....... 11
      2.2.1 Software engineering databases .................. 12
      2.2.2 Product development databases .................. 15
      2.2.3 Miscellaneous .................................. 16
      2.2.4 Summary of characteristics ..................... 16
   2.3 Background for database integration ................... 17
      2.3.1 Multi-database systems and federated database systems ... 17
      2.3.2 Schema and object integration .................. 19
      2.3.3 Schema integration methodologies ............... 19

3 Evolution and Integration Techniques .................... 21
   3.1 Introduction ......................................... 21
   3.2 Schema extensions ................................... 21
      3.2.1 Subclassing ................................... 21
      3.2.2 Miscellaneous language mechanisms ............... 24
   3.3 Schema modification .................................. 27
3.3.1 Class modification ........................................... 27
3.3.2 Class reorganisation ........................................ 30
3.4 Schema versioning .............................................. 33
   3.4.1 Class versioning ........................................... 35
3.5 Schema integration ............................................ 38
   3.5.1 Class integration .......................................... 39
   3.5.2 Integration by views ...................................... 41
   3.5.3 Generalisation by upward inheritance .................. 45
   3.5.4 Miscellaneous ............................................ 46
3.6 Object-level evolution ....................................... 47
   3.6.1 Object versioning ......................................... 48
   3.6.2 Object conversion ........................................ 48
   3.6.3 Multiply classified objects ............................... 50
   3.6.4 Class learning ............................................ 51
3.7 Concluding the survey ...................................... 52

4 Framework for Class Evolution ................................. 55
   4.1 Basic object model ......................................... 55
      4.1.1 Objects, intents, extents, and classes ............... 56
      4.1.2 Properties: attributes and operations ............... 57
      4.1.3 Example ................................................. 59
      4.1.4 Rationale of the basic object model .................. 59
   4.2 Extent graphs and intent hierarchies ..................... 60
      4.2.1 Intent inheritance ...................................... 61
      4.2.2 Extent graphs .......................................... 62
      4.2.3 Object correspondence .................................. 65
      4.2.4 Binding of operations .................................. 67
   4.3 Specifying object consistency ............................. 69
      4.3.1 Attribute consistency relations (ACRs) ............... 70
      4.3.2 Interpretations of ACRs ................................ 71
      4.3.3 ACRs and mathematical functions ...................... 72
      4.3.4 Attribute consistency groups (ACGs) ................... 74
4.3.5 ACGs and null values ........................................ 76
4.3.6 Object consistency definition ............................... 77
4.3.7 Transitive ACRs ........................................... 78

4.4 Maintaining object consistency ............................... 79
4.4.1 Extent propagation ....................................... 80
4.4.2 Shared representation of attributes ....................... 80
4.4.3 Implementing shared representation ....................... 84
4.4.4 Operation consistency relations (OCRs) .................. 85
4.4.5 Separate representation of attributes ..................... 88
4.4.6 ACRs vs. OCRs .......................................... 92

4.5 Class evolution in the framework ........................... 95
4.5.1 Class evolution definition ................................ 95
4.5.2 Class evolution transparency ............................. 98
4.5.3 Class derivation and integration ......................... 99
4.5.4 Examples of typical class evolutions ..................... 100
4.5.5 Class evolution and abstract classes .................... 102

4.6 Comparisons ................................................. 103
4.6.1 General remarks ......................................... 103
4.6.2 Multiple hierarchies .................................... 104
4.6.3 Updatable views ......................................... 105
4.6.4 Explicit relations ........................................ 106

5 Class Derivation ............................................... 109

5.1 Introduction ................................................ 109

5.2 Applying the framework to class derivation ................ 110
5.2.1 Subclassing .............................................. 110
5.2.2 Incompatible, alternative representations ............... 112
5.2.3 Telescope evolution .................................... 114

5.3 Cost model .................................................. 114
5.3.1 Shared representation .................................. 115
5.3.2 Separate representation ................................ 116
5.3.3 Examples of cost analysis ............................... 117
5.4 Derivation and representation ........................................ 118
5.5 Class editor tool ....................................................... 121
5.6 Extensions ............................................................ 123
      5.6.1 Scaling up with ACRs ..................................... 123
      5.6.2 Dynamic binding and static typing ..................... 125
      5.6.3 Bridging to change-oriented versioning ............. 126
5.7 Comparison/discussion .............................................. 128
      5.7.1 Class versioning mechanisms ......................... 128
      5.7.2 Dependencies in relational databases ............... 129
      5.7.3 Subclassing ................................................ 130
      5.7.4 Consistency maintenance and encapsulation .......... 131

6 Class Integration ..................................................... 133
  6.1 Introduction ...................................................... 133
  6.2 Applying the framework to class integration .............. 135
      6.2.1 Various integration situations .................... 135
      6.2.2 Special integration situations .................. 138
  6.3 Object integration ............................................... 141
      6.3.1 Proxy and entity objects .............................. 141
      6.3.2 Facilitating object integration ................... 142
      6.3.3 Class integration and binding of operations .... 145
  6.4 Methodology for class integration ............................. 145
      6.4.1 Overview of the methodology ....................... 145
      6.4.2 The search for shared abstractions ............... 147
      6.4.3 Intensional comparison (Step 1) .................. 148
      6.4.4 Extensional comparison (Step 2) .................. 154
      6.4.5 Confirmation (Step 3) ................................ 155
      6.4.6 ACR and OCR generation (Step 4) ................. 156
  6.5 Integration interference ........................................ 159
      6.5.1 Extent propagation links ............................. 159
      6.5.2 Object consistency specifications ................. 160
  6.6 Tool support ..................................................... 161
7 Implementation Issues

7.1 Implementation issues for the framework .......................... 169
  7.1.1 Extent propagation ........................................ 169
  7.1.2 Implementing ACRs and OCRs ............................... 171
  7.1.3 Management of ACRs ....................................... 173
7.2 CEFP: Class evolution framework prototype ....................... 174
  7.2.1 The EPOS database ........................................ 174
  7.2.2 The object model of CEFP .................................. 174
  7.2.3 The interface of CEFP ..................................... 175
  7.2.4 The schema of the prototype ................................ 176
  7.2.5 Consistency maintenance ................................... 178
  7.2.6 Operation requests and OCR evaluation ...................... 178
  7.2.7 Evaluation of prototype ................................... 179
  7.2.8 Alternative consistency maintenance schemes ............... 180
7.3 Implementation by object-oriented views ........................... 181
  7.3.1 Query algebra .............................................. 182
  7.3.2 View implementation and class evolution .................... 184
  7.3.3 Examples .................................................. 187

8 Evaluation, Conclusions, and Further Work .......................... 191
  8.1 Evaluation .................................................... 191
    8.1.1 Integrating class evolution techniques ..................... 191
    8.1.2 Object model or mechanisms? ................................ 192
8.2 Conclusions ................................................. 192
8.3 Further work ............................................... 195

A The Components of the Framework .................. 197

B The Interface of the Prototype ....................... 199
List of Figures

2.1 System architecture for a federated database system. .................. 18

3.1 Missing classes example. .............................................. 27
3.2 Incremental reorganisation of a simple class hierarchy by Casais' algorithms. ................................................. 31
3.3 The initial shape class hierarchy........................................ 34
3.4 The multiple inheritance solution to the shape problem. .......... 34
3.5 The multiple views approach. ......................................... 36
3.6 The class integration example. ....................................... 39
3.7 Exclusionary and essential classes..................................... 49

4.1 The polyline/point example. .......................................... 60
4.2 Example of copy semantics of inheritance. .................................. 62
4.3 Example of an extent graph with its corresponding intent hierarchy. 64
4.4 The object correspondence situation 1. ............................. 67
4.5 The object correspondence situation 2. ............................. 67
4.6 The object correspondence situation 3. ............................. 68
4.7 The object correspondence situation 4. ............................. 68
4.8 The CartLoc/PolarLoc example. ..................................... 71
4.9 Illustration of the totally/partially combination. ..................... 82
4.10 Illustration of the partially/partially combination. ................ 82
4.11 Eager consistency maintenance using after timing mode. ........... 87
4.12 Lazy consistency maintenance using before timing mode. ......... 88
4.13 Eager consistency maintenance in the coordinate example. ........ 90
4.14 Lazy consistency maintenance in the coordinate example. ........ 91
4.15 Shared representation using OCRs. .................................. 92
4.16 The OCRs connected to Stack's operations. 95
4.17 The OCRs connected to Set's operations. 95
4.18 Example of level-0 class evolution transparency. 99
4.19 Some typical evolution situations. 101

5.1 Subclassing by class derivation. 111
5.2 Cell representation of 2D objects. 112
5.3 Boundary representation of 2D objects. 113
5.4 Eager consistency maintenance in the incompatible representations example. 113
5.5 Representing subclassing by vertical splitting. 119
5.6 Representing subclassing by horizontal splitting. 120
5.7 "Update-in-place" of a class using COV semantics. 128

6.1 Introductory integration example. 134
6.2 Some binary integration situations. 138
6.3 Aggregation integration. 139
6.4 Group integration. 140
6.5 The steps in the integration methodology. 146
6.6 Interference of EPLs. 159
6.7 An example term space. 162
6.8 Displaying nearness of classes. 163

7.1 The EPOS schema implementing the class level of CEFP. 176
7.2 The EPOS schema implementing CEFP at the object level. 177
7.3 The correspondence between the faculty and department domains. 186
List of Tables

4.1 Summary of the shared representation possibilities. ............... 83

5.1 Cost example 1. ...................................................... 117
5.2 Cost example 2. ...................................................... 118
5.3 Cost example 3. ...................................................... 118

6.1 Suggesting integration based on intensional comparison. .......... 148
6.2 Suggesting integration based on extensional comparison. .......... 155
6.3 Generalisation attribute set and new ACGs. ......................... 157
Acknowledgements

I’d like to thank my advisor, professor Reidar Conradi, for all help and guidance, and for many stimulating technical discussions.

This work has grown out from the work on FOOD, and from the work on versioned databases done in the EPOS project. The members of the project have been invaluable discussion partners. I’d like to thank them all, and mention in particular those who have been most involved with the database activity: Arne-Jørgen Berre, Bjørn Gulla, Even-André Karlsson, Jens-Otto Larsen, Anund Lie, Bjørn P. Munch, and Erik Odberg. I’d also like to thank Stewart M. Clamen at Carnegie Mellon University, who provided insight to his ideas on relationships between facets. This clearly influenced my work on attribute consistency relations.

During the period of my dr.ing. study, I have mainly been supported by a dr.ing. fellowship from the University of Trondheim, partly by the EPOS project, which itself was funded by the Norwegian Council for Scientific and Industrial Research (NTNF) under grant ED0224.18457, and partly by the REBOOT project, also funded by NTNF.

During all the years I spent at the department, my wife, Anne-Lise, gave me all support I needed, and constantly encouraged me to go on.
Chapter 1

Introduction

This thesis addresses the problems of evolution and integration at the schema level in object-oriented databases. This chapter introduces the thesis by describing the emergence of object-oriented databases, argues for evolution and integration being important topics, gives the context for the work, presents the claimed contributions, and finally gives an overview of the thesis.

1.1 The emergence of object-oriented databases

Object-oriented databases came into being in the mid-80’s, apparently as a merger between databases and programming languages. We think they appeared as a result of two different efforts, where the researchers became aware of each other:

- Programming languages were extended to support persistency of data.
- Database systems were extended to capture more of the semantics of the data, which previously were intervened in the applications.

Both of these branches of research were identified as being object-oriented. To clarify the terms Dittrich characterised the systems as being structurally or behaviourally object-oriented [Dit88], indicating the database approach to be structurally object-oriented, and the programming language approach to be behaviourally object-oriented. Approaches combining the two branches were characterised as fully object-oriented.

The database branch grew out from the work on semantic data models in the late 70’s and early 80’s. Of the major work done on semantic data models we will mention the entity-relationship model [Che76], the introduction of generalisation/specialisation in databases [SS77], the semantic data models SDM [HM81] and DAPLEX [Shi81], and extensions to the entity-relationship model, e.g., the entity-category-relationship model [EHW85]. These models tried to capture more of the semantics of the data by introducing semantic constructs like generalisation/specialisation, aggregation, and explicit relationships with attached constraints (cardinalities, existence dependencies.
etc.). These constructs were tailor-made to capture common semantics of data found in various applications. Many of these constructs were hard or expensive to implement, and have scarcely been found in commercial database systems.

Another trend was the introduction of database systems for software engineering applications: Adele [EGK84], Damokles [DGL86], PCTE OMS [GMT86] and ECLIPSE [CA87] were based on the entity-relationship model, but extended with concepts like object identity, complex objects, long fields and versioning.

An important subarea of structurally object-oriented databases was nested relational systems, e.g., AIM-P [DK+86] and DASDBS [PSS+87]. They extended relational systems by allowing fields in tables to be non-atomic. Hence, they are often named as non first normal form databases. These systems were built on the well-founded theory of relational databases, and thus they borrowed the declarative query approach.

Many commercial relational database systems, e.g., Ingres, ORACLE, Sybase, Informix\(^1\) have been influenced by object-oriented databases, and have been extended with “object-facilities” like stored procedures and binary large objects (BLOBs).

The other branch of research, which we may name persistent programming languages or behaviourally object-oriented databases, was concerned with adding persistency of data to existing programming languages. PS-Algol [Atk82] (Algol with persistent heap) is usually credited for conceiving object-oriented databases in this branch of research. It was shortly followed by Galileo [ACO85]. Since 1987 many of the approaches here are efforts to extend C++ [Str86] with persistency, e.g., ONTOS [AHS91] and ObjectStore [LLOW91]. However, GemStone [BOS91] was built on a Smalltalk-like model (named OPAL), and ITASCA [ita90] was built on LISP. O\(_2\) [D+91] is another major work within this branch, but this built on the “set-and-tuple” model FAD [BBKV87] and had user-defined behaviour (operations/methods) expressed in CO\(_2\), a C-like language.

The database branch of research appeared because the existing databases had limited expressiveness, which resulted in shared semantics to be scattered around in different applications. This branch basically focused on hard-wired declarative constructs to express more of the semantics of the data. The programming language branch was founded on languages being Turing-complete, and extended these with persistency. Despite the efforts to make object-oriented databases, most of the systems claimed to be object-oriented carried with them a taste of their background. Approaches originating from a database background often missed complete expressiveness, while approaches based on object-oriented programming languages often were rather primitive with respect to database concepts like sharing with concurrency control, transactions, recovery, and logging. While approaches originating from a database background often were multi-lingual by having interfaces to many programming languages, most of the persistent programming languages were tightly connected to one programming language.

---

\(^1\)Ingres, ORACLE, Sybase, and Informix are trademarks of Ingres corp., Oracle Inc., Sybase Inc., and Informix Software Inc., respectively.
On an application-driven level, we believe object-oriented databases appeared of several reasons:

1. As a consequence of integration of information systems, in which reliable data sharing is a key point.

2. Growing appetite for database support within unconventional database application areas, like user interfaces, software engineering, knowledge bases, office information systems etc.

3. As part of a natural trend to extend databases to share more than simple data, like semantics, “knowledge”, and the dynamics of applications (behaviour).

4. Due to the shortcomings of conventional databases [Kim90a], e.g., no complex, nested objects, too limited a set of attribute domains, the impedance mismatch [MP84], and limitations on the kinds of transactions, preserving the notion of serialisability.

1.2 Evolution and integration

Managing the consequences of evolution is a dominant activity in general in the software industry. The need for evolution in databases partly stems from the nature of the application domains. Typical examples of this are found where databases are used to model objects under development, where objects represent designs being used for experiments and rapid prototyping, and where objects and classes are long-lived, but their use and environment are changed.

Evolution may also result from bad schema design, requiring the structure of the database to be changed after the database has been populated. [Sjo93] reports from the study of the evolution of a large-scale database application running in several hospitals in the United Kingdom. The results reported were based on the use of a relational database system, and they confirmed evolution of database schemas to be an important topic – not only for typical “evolutionary environments”. It revealed that schema changes are significant both in the development period and after the system has become operational. Related to the two dimensions of consequences of schema evolution, [Sjo93] reports that consequences on application programs prove to be more complex than managing the impacts on the extensional data. In this thesis we will address this problem by supporting class evolution transparency (defined in Section 4.5.2).

Schema integration is less driven by special application areas than schema evolution, but is mainly a result of practicalities in database development. View integration is a planned integration by dividing the database design into subproblems, and later integrating the different designs into one [BLN86]. Database integration is more unplanned, where new applications (or external requirements) give the need to integrate databases being independently developed and populated. E.g., a change in an organisation’s structure may require different databases to be integrated.

3
Object-oriented technology has often been recognised for its ability to capture evolution:

**Extensibility and change management:** [OTK+91] puts *extensibility* and *change management* among the key characteristics of object models (object-oriented data models). Extensibility is defined as additions and modifications to existing classes or as adding new classes, possibly based on existing classes. *Change management* reflects the influence of design applications recording and managing the entire life-cycle of complex objects, e.g., software, chip designs, and documents.

**Principles of object-oriented computing:** [BGHS91] puts *evolution* among the four principles that capture the essence of object-oriented computing: “The third principle of object-oriented computing is that support should be provided for evolution. This is based on the observation that requirements change rapidly in computing environments.”

**The true value of object-oriented techniques:** “The true value of object-oriented techniques as opposed to conventional programming techniques is not that they can do things the conventional techniques can’t, but that they can often extend behavior by adding new code where conventional techniques would require editing existing code instead.” [SLU89]

**Additional quotes:** [CP89]: “Inheritance is a mechanism for differential, or incremental programming”. [Cox86]: “Object-oriented programming: an evolutionary approach”. [WZ88]: “Inheritance as an incremental modification mechanism”.

The strength of traditional object-oriented models to capture evolution comes from two constructs: *subclassing* and *dynamic binding* from operation requests to operations. Subclassing allows classes to be incrementally developed, while dynamic binding allows existing clients (applications) to transparently use objects of different classes. Together, these two constructs allow classes to be easily specialised (or changed) in subclasses. This is a “powerful” technique, but requires classes to be developed starting from the general superclasses and ending with the specialised subclasses. Unfortunately, many evolution situations are not captured by subclassing, and consequently a set of other tailor-made techniques have been developed to support evolution at the schema level:

- **Class modification:** Classes are modified directly, e.g., by changing domains of attributes, implementations of operations etc. Both the objects created from the classes and the clients of the classes must be converted to be compatible with the modified classes.

- **Class versioning:** A new version of an existing class is created to suit new requirements. The different versions of the class share extents, i.e. they have the same set of objects, but the properties of the objects will be (slightly) different. This allows existing clients to be unchanged.
• **Class hierarchy reorganisation:** Classes are badly organised in class hierarchies, and need to be reshuffled.

• **Class generalisation:** Existing classes are too specific, and need to have a generalised class, giving a common abstraction which is found in all the concrete subclasses. In addition to being an abstraction, the generalised class integrates the extents of the subclasses.

• **Integration by views:** Existing classes are integrated by creating views on top of them.

These different techniques will be surveyed in Chapter 3. In Chapter 4 we develop a framework which may be used to express most of these situations using a rather small set of concepts.

### 1.3 Context for our work

In the EPOS (Expert system for program and system development) project (1986–90) [C+89] a database system to support software engineering environments has been developed. The first version of the EPOS database was created in 1986-87. The experience with this concluded that an object-oriented database would be more suited [CDEH87]. In 1989–93 a structurally object-oriented database has been developed. In 1989–90 parallel work was done on FOOD (Fully object-oriented database) [Bra91, OB90] to experiment with even more object-oriented features. The first attempt with FOOD was basically a persistent C++ built on top of Damokles [ADG+88]. The second attempt included a richer data model, but was only a rudimentary implementation on top of C-ISAM\(^2\).

The work appearing in this thesis is connected to FOOD and the database prototypes built in EPOS in the following ways: First, the EPOS project has developed a software engineering environment, in which evolution both at the object and the schema level is found to be a prominent characteristic. It is from this experience that we started the work on schema evolution. Second, the basic part of the object-oriented data model used in this thesis (see Section 4.1) is a simplified version of the FOOD data model. Third, the evolution framework has been prototyped on top of the EPOS database (see Section 7.2). The prototype is a rather thin layer on top of the EPOS database, and consequently it “inherits” much of the platform’s strength and weakness. Related work on *type evolution* [Odb92] and *evolution in software process modelling* [JC92] has also been done in the context of the EPOS project.

### 1.4 Claimed contributions

We have identified the following main points, which we claim to be contributions to the state of the art in schema evolution and integration in the area of object-oriented

---

\(^2\)C-ISAM is a trademark of Informix Software Inc.
database technology:

**Unification of class evolution techniques:** We have factorised the common and underlying concepts in many of the existing class evolution and integration techniques, and provided these explicitly and “orthogonally” in a framework for class evolution and integration. This reduces the number of tailor-made, partly overlapping constructs. A main component of the framework is the separation of the intensional and extensional dimension of classes into distinct hierarchies. Separated hierarchies is not a new idea in object-oriented technology (see Section 4.6.2), but we have shown how valuable this is to evolution of classes. That is, when organised in hierarchies, the intensional and extensional dimensions of classes have different evolution characteristics. The framework may also be seen as a work on object-oriented data models: We allow for evolution and integration of classes by loosening up the rather strict semantics of class hierarchies.

**Externalised evolution information:** Unlike existing class evolution and integration techniques, the evolution constructs of the framework are external to the classes. This lets classes created as a result of evolution or integration, to be ordinary classes, which can be understood and be further evolved like any other class.

**Object correspondences and specification of object consistency:** When unifying several technologies one may choose to keep the best from the individual technologies. We claim that some examples of this may be found in the framework: We have unified the concepts of “proxy” object integration in class integration with multiple class membership for objects in traditional class hierarchies. This is used to exploit the same set of constructs to maintain the consistency of objects in those two cases. Concretely, this has allowed us to take advantage of specifications of dependencies by means of mathematical functions between attributes. Similar specifications have been used in database integration theory [LNE89], and when applying these to subclassing and class versioning, they allow for tuning the implementation of consistency maintenance. The specifications also make it easier to ensure termination of consistency maintenance in some situations where there are cyclic dependencies.

**Class derivation:** We have developed a cost model for deciding upon alternative implementations of consistency maintenance for objects in situations where classes are derived into new classes. Together with an analysis for when shared representation for objects is possible, and a set of recorded class derivations, this may be used in a “class editor” supporting the class designer when defining new classes.

**Class integration:** For class integration we have developed a methodology exploiting both the intensional and extensional dimensions of classes. This is used to have a broader comparison of the classes to be integrated. We suggest to enhance classes with abstract data type specification to help in understanding and integrating the behavioural parts of classes. We outline several ad hoc
techniques borrowed from software engineering which may be of help in class integration.

When concluding the thesis in Chapter 8, we give a more detailed list of the claimed contributions, and with references to places within the thesis.

1.5 Overview of thesis

The thesis is organised in the following chapters:

Chapter 1 is this introduction.

Chapter 2 presents the background for evolution and integration. We give an introduction to the concepts of object-oriented databases, and argue for the necessity for evolution by describing the characteristics of some application areas of object-oriented databases. We also give a brief introduction to schema and database integration.

Chapter 3 surveys evolution and integration techniques in the context of object-oriented databases. We concentrate on schema evolution techniques, but at the end we have included a section on evolution at the object level.

Chapter 4 is the main chapter of the thesis, presenting our framework for class evolution. After we have described the components of the framework, we define the meaning of class evolution and integration in the context of our framework.

Chapter 5 applies the framework to a set of class evolution situations labelled as class derivation. This is concerned with supporting small adaptations of classes without loss in efficiency.

Chapter 6 applies the framework to class integration, being evolution situations where independently developed classes and objects are integrated. We develop a methodology for class integration specially suited for the components of the framework.

Chapter 7 discusses some implementation issues for the framework, and it documents a small prototype which has been built to verify the design of the framework. At the end of this chapter we show an alternative design of the framework based on object-oriented views.

Chapter 8 evaluates and concludes the thesis, and points out further work.

Appendix A summarises the main components of the framework.

Appendix B is a presentation and discussion of the interface of the prototype being documented in Section 7.2.
Note that we make a change in the main terminology after the survey of existing techniques in Chapter 3. In Chapter 4 we present the framework for class evolution, which is exploited in two directions: class derivation and class integration, roughly corresponding to class evolution and class integration in the earlier chapters. The definitions of these concepts are delayed until Section 4.5, that is, after the introduction of the constructs of the framework.
Chapter 2

Background for Evolution and Integration

This chapter introduces the main concepts of object-oriented databases. We present typical application areas of object-oriented databases, and from these we argue for the need to handle evolution. At the end of this chapter we introduce database integration.

2.1 Object-oriented database concepts

The purpose of this section is to give a short introduction to object-oriented database systems\(^1\). Briefly, an object-oriented database system is a system with traditional database functionality and an object-oriented data model. Thus, object-oriented databases are mainly recognised by their data models, which we will name object-oriented data models or object models. The former term will be used in the context of database systems, while the latter term will be used when speaking more generally about object-oriented systems.

Object-oriented databases were conceived from several different research and development activities. Of this reason there has been no common agreement on what constitutes an object-oriented database system. We base our “feature list” on [ZM90] and on “The Object-Oriented Database System Manifesto” [ABD+89], which was an effort to make consensus on the issues of object-oriented databases.

Database functionality: This includes persistence of objects, which means that objects are accessible past the end of the process that created them. Objects must be stable, i.e. they are resilient against system crashes and media failures. This is typically done by recovery, relying on logging all changes done to the database on secondary storage. The database must also cope with sizes exceeding the main memory of the computer system. A very important aspect of databases is that objects are shared between different users. This means that

\(^1\)We will use the terms object-oriented databases and object-oriented database systems interchangingly.
concurrency control must be supported to prevent different users to perform inconsistent actions on the database. This is typically supported by locking and serialisable (atomic) transactions. An ad hoc query facility is often considered to be a part of the basic database functionality. Query languages are typically based on associative access, i.e. objects are returned from queries based on predicates on their properties.

**Object identity:** The most distinguishing feature of an object-oriented database is that objects have object identities, which are implemented by immutable and system-wide unique object identifiers. Object identifiers are the basis for an alternative to associative (or value-based) access, namely navigational access. Applications will navigate through an object structure, where objects are interconnected by references (or links).

**Encapsulation:** This means that objects have an interface and an implementation (or representation). Encapsulation represents a software engineering principle, where it is made a distinction between specification and implementation. This has several advantages: Clients (other objects, users) of objects do not have to understand all details about objects – only the relevant details are made public for the clients. It also gives the implementor of the object considerably freedom – the implementation of the objects may be changed without affecting clients.

**Complex objects:** These are built from simpler objects by applying constructors, e.g., special composite object references, PART-OF links, or similar. Such references or links have special built-in semantics for predefined operations like copying and deletion, and for management issues like locking and caching.

**Types and classes:** To create and interpret objects types or/and classes are used. A type is a denotation or description of the visible (public) properties of a set of objects, usually being the objects created from that type. Class is grossly used synonymously for type, but with a slight preference where the extensional dimension (of classes) is explicitly available for queries. The choice of words is also influenced by the origin of the specific model. E.g., several commercial object-oriented database systems have adopted C++’s classes for describing objects. In this thesis we will use class entirely in the context of object-oriented databases. A class typically contains the definitions of the attributes (instance variables) and the operations (methods) that the objects of the class possess. Attributes represent the state of objects, while operations implement the behaviour of objects.

**Class hierarchies:** According to [ZM90] class hierarchies have three different appearances: Classes may be organised in hierarchies according to specification, which means that objects of a subclass are compatible with (may type-safely be substituted for) objects of the superclass. Classes may be organised according to implementation, i.e. subclasses inherit the implementation of their superclasses. Finally, classes may be organised according to extension, i.e. the extension (or extent) of a subclass is a subset of the extension of the superclass.
Object-oriented databases usually bundle all these three aspects into one hierarchy. In Section 4.6.2 we discuss these three hierarchies and their relevance in object-oriented databases.

**Dynamic binding:** The binding from an operation call (or request) to a specific implementation of the operation is done based on the receiving object and not on the static context where the call takes place. This is an important feature which allows for *polymorphic programming*, i.e. the same client code may be used for objects of different classes. Dynamic binding is often combined with *redefinition* of operations in subclasses, i.e. a subclass provides a new implementation for an operation which is defined in one of its superclasses.

**Computational completeness:** The database system should provide a language for its operations, which can express any computable function. Usually the language of the operations are expressed in an existing programming language, possibly extended with special database facilitates, e.g. *pointer swizzling* and *associative queries*.

**Extensibility:** There should be no distinction between user-defined and system-defined classes.

Object-oriented databases may be viewed as structural databases extended with user-defined behaviour. An interesting question raised by this is where to put the line between the applications and the database? In our view the perhaps most important issue of a database is that it lets objects be shared between different clients. We think this is also the main criteria to use when deciding if some behaviour should be a part of the application, or a part of the user-specified behaviour in the database. To make behaviour a part of the shared database, the behaviour must be generally useful to other (possible) applications. An excellent example of such shared behaviour are structural constraints found in *semantic data models*. However, in practice it may be hard to see if a specific behaviour is an application-specific interpretation of the objects in the database, or if it is a behaviour that is to be shared by new applications.

## 2.2 Database requirements from design environments

This section presents some of the application areas being used as arguments for the introduction of object-oriented databases. We may characterise these as *design* or *engineering applications*, of which we will put the main focus on software engineering and product development databases. After the presentation of these two application areas, we briefly mention some of the other application areas, and summarise the characteristics found. Even if we use design environment as a rationale for the introduction of object-oriented databases, the applicability of object-oriented databases goes beyond this market niche [ZM90].
2.2.1 Software engineering databases

In the heart of a software engineering environment there should be a database [C+89]. The purpose of this database is to store and control all kinds of electronic information associated with software development and maintenance. An interesting characteristic is that the software is the product being developed and maintained. Thus, the database contains the reality, it does not just model the reality. Some of the main software engineering activities needing database support are the following:

**Software development:** A software engineering database is used to control the software products and their associated information during development. Typically, the data is copied out from the shared database into the user *workspaces*, i.e. data on the format the software tools expects to find it. This copying is feasible when the development is planned and controlled. This model of development holds for several of the steps in the software life-cycle: design, implementation, and maintenance.

**Software maintenance:** *Software maintenance* is used a common designation for enhancements, adaptations, perfections, and corrections. This activity treats incoming *change requests* or *bug reports*, and possibly make appropriate changes to the software products. The changes done to the products are typically more sporadic and less planned than in software development.

**Software reuse:** This may be seen as a part of software development, but it has different characteristics with respect to database support. In general, *reuse* means using an entity in a different context from where it initially was used. *Composition-based software reuse* [BP89] is characterised by existing software components being composed to form new components. This reuse is used for all kinds of software components, including documentation and test data. Software reuse demands descriptive information on software components and good retrieval support.

These three software engineering activities treat the same kind of information, but their access profile and requirement for tools vary slightly. Briefly summarised, software development impose massive and usually planned evolution, software maintenance involves more sporadic changes, while software reuse needs good retrieval facilities to find existing components.

**Fine vs. coarse granularity data**

Software engineering databases model the software either with *fine* or *coarse granularity*. A fine granularity approach represents the software possibly on the level of syntactic tokens, while a coarse granularity approach represents the software on the level of "components", being modules, procedures, header files etc. Fine-granularity approaches allow for more detailed dependencies to be represented, which may help in analysis of the impacts of changes, and they allow for a higher degree of concurrency without conflicts. The main problem with the fine-granularity approach is
that existing software tools expect to find the data stored on plain text files. Further
more, “modules” are often the natural units of work. E.g., due to the strong semantic
interconnections within code, it may be unnatural to let several users concurrently
edit the same ten-line procedure definition. The coarse-granularity approach is more
efficient and easier to manage by the database, and represents data on a level that
is close to the “units” offered by the language represented in the database. The
problem with this approach is that it does not allow for representing fine-granularity
dependencies, e.g., causing unnecessary recompilations etc. Due to the inheritance
from file systems, most existing software engineering databases use the coarse gran-
ularity model, where individual components are represented as uninterpreted long
fields, and dependencies between individual tokens are represented more coarsely
between the components containing the individual tokens.

Components, structures and inter-dependencies

There may be many kinds of components within and associated with software engi-
neering projects. This may be the software itself, which may exist in many formats,
both structured and unstructured: it may be modelled and stored as a character
sequence in a text file, it may be relocatable machine code, it may be a syntax tree,
it may be a symbol table, etc. Different tools use the same software components on
different representations. Associated with the software there may be many kinds of
information, like documentation, design, project or personnel information. This in-
formation may also be on different representations, like simple structured “records”,
bitmaps, attributed graphs, or text files.

Most software systems are composed of smaller subsystems, which again may be
compositions of even smaller subsystems or atomic components. E.g., a system may
be composed of modules, which have procedures, data structure definitions and local
modules, which again may have their own local definitions. We may have inclusion
of shared libraries in different software components. Similar data compositions and
inclusions may exist on auxiliary data created for software development, like docu-
mentation which may be a composition of heterogeneous objects. In object-oriented
databases such compositions are often directly supported by constructs for building
composite objects, e.g., the composite references of ORION [K+89] or the structure
clause of Damokles [DGL86].

There are various kinds of interdependencies between objects in software engineering
databases. The typical example is imports and exports between module interfaces
or bodies, which are semantic connections in the product structure. There may be
attached certain constraints or actions to such relationships. Upon changes to a com-
ponent, some actions are needed to make dependent components consistent. There
are several kinds of such semantic dependencies, for example a derivation cache (e.g.,
containing object code) should be updated when the primary object is changed. This
may be done automatically or manually by a human being notified by this change.
Note that such interdependencies also may exist within the associated information.
E.g., a user manual should be updated when the software itself is modified, a design
document should be updated when the requirement document is changed. These in-
terdependencies are more manifold for fine-granularity representations, e.g., def-use
Version and configuration management

Versioning is a prominent characteristic of software engineering databases. The reason for this is that software products undergo long-time considerable development effort. At specific stages the state of a component is defined as being a version. Versions may exist, with respect to each other, either as variants or as revisions. One version is a revision of another when the new version is intended to replace the other. A version is a variant of another when it is intended to coexist with the other version, i.e. they represent alternatives. In this scheme the versions of a software component together compose a version group [Tic88]. Relationships between versioned components may be dynamic or fixed version bound, meaning that they are between version groups or individual versions, respectively.

A system consisting of several versioned components is called a configuration [Tic88]. They exist in two categories: A generic configuration is an ambiguous configuration, i.e. at least one of its components exists in several versions within the configuration. A bound configuration is an unambiguous configuration, i.e. all its components exist in at most one version within the configuration. A bound configuration is created by selecting versions of components from a generic configuration. This selection is typically based on special version attributes of the software components. The selection is expanded by references with dynamic version binding, but constrained by references with fixed version binding.

Evolution and long-lived objects

Replacing existing software by new software may be costly. Therefore, existing systems are evolved (i.e. corrected, adapted, perfected, or enhanced) to keep up with new and existing requirements. This means that software components may be kept “alive” for long periods of time. However, their use, the methods applied to them, and the surrounding organisation and environment may evolve. A typical lifetime for a software component may be 10-15 years, while some may become 20-30 years.

Concurrency control and transactions

Large software products are developed and maintained by many persons simultaneously. This means that the database must support access and concurrency control. In software engineering databases versioning creates an extra dimension for concurrency control [Lie90]. Depending on the granularity of the database representation of the software, different kinds of concurrency control may be natural. On fine-granularity representations, traditional serialisable transactions may be appropriate, while on coarse-granularity representations, non-serialisable (“design”, “long”) transactions are more natural. The typical model for such long transactions is that several development groups get their own copy of a portion of the database, by
checking it out into “workspaces”. Upon commit of these design transactions, a
merge activity must take place. Such parallel work may be necessary of physical
reasons (distributed development), and is quite natural given existing software tools
working on plain files.

2.2.2 Product development databases

The purpose of product development databases is to integrate descriptions of prod-
ucts for diverse tools. They may be considered as a generalisation of CAD databases,
by being databases both for spatial and non-spatial data. In some sense product
development databases are a generalisation of software engineering databases as well.

We will present characteristics of product modelling for structural design [Dal91],
and use this as a representative for product development databases in general:

Multiple representations: A product model captures several aspects of the struc-
tures. In structural design three aspects need to be represented: A geometric
model defines the geometry and materials of a structure. A structural model
defines the idealised geometry and the loading conditions, which again is the
basis for describing a response model. A computational model is a set of data
derived from the structural model, and it may be input to special-purpose
analysing tools.

Design levels: Products are typically composed of smaller products. In structural
design four design levels are used: project, structure, assembly, and part. They
are used to describe products in several levels of detail, and to be able to share
products at various levels of detail. In structural design the design levels are
standardised to allow for exchange of products in the marketplace.

Data evolution: Due to design iterations evolution is needed. E.g., single parts
are evolved into detailed assemblies. This is typically “stable” and planned
evolution.

Division of work: There may be interactions of several people working on one de-
design. This gives needs for efficient communication and management of changes.

Variants: Alternative, complete solutions to a design problem, and variants of so-
lutions to subdesigns are usually present in product development databases.

Geometric abstractions: Several geometric abstractions are used: boundaries to
envelope abstract objects, reduced dimensionality to expose relevant charac-
teristics, and symbols to represent arbitrary abstract objects in the design.

Constraints: Three kinds of constraints are used in structural design: design con-
straints ensure functionality and usability of entities according to engineering
practice, interaction constraints are used to coordinate parallel work, while
consistency constraints are used to impose dependencies between arbitrary
data.
These characteristics are similar to those found in software engineering databases, but are harder to cope with due to the introduction of spatial data.

2.2.3 Miscellaneous

We will briefly mention some other application areas which have been used as arguments for the introduction of object-oriented databases. Knowledge-based systems need database support to get out of the prototype and toy state [Bro89]. Entity-based and term-based knowledge-representation systems [Myl91] have concepts similar to semantic data models [HK87] and object-oriented data models. Semantic constructs like generalisation/specialisation, aggregation, classification, and subsumption [BS89] may be found here. As knowledge is added, refined or corrected, the knowledge base will be exposed for rapid changes in both structure and content. Note that deductive databases [GMN84], another branch of database research, which due to its logical foundation, is a more natural choice than object-oriented databases for database support in knowledge-based systems.

Geographical information systems make heavy demands on database support for storing models of landscapes, usually augmented by property information. In this sense, they are similar to product development databases. The structures are complex, and the amount of data is enormous. However, most data here is static. Thus, evolution is a less prominent characteristics than in product development databases.

Office information systems are very dynamic and put high requirements on database support. Here, the problems often cannot be readily defined, they are goal-oriented, they expand in scope and evolve in time, and thus suffer from constantly changing requirements and assumptions [NT88]. An interesting observation here is that the first approach we have found to class versioning, [ABB+83], was done in the context of office information systems.

2.2.4 Summary of characteristics

The characteristics of these different application areas vary. As a summary we have coarsely grouped the characteristics into two:

Complex data: The different development environments quite uniformly have complex data, i.e. the data is highly interconnected with various semantics on the different interconnections. One example of an interconnection is the attribute lengthOf of a polyline object, which is defined to be equal to the sum of the lengths of all simple lines composing the polyline object. Another example is a Modula-2 module including a local module. A semantic information attached to this include dependency is that, once it is made a new version of the local module, a new version should be generated of the composite module. In object-oriented databases such complex interconnections may be modelled by built-in semantic constructs, e.g., objects being composed of other objects using part-of links. However, usually they will be modelled (or implemented) by the operations attached to the objects.
**Evolution:** An important characteristic of development environments is that data is not static – evolution both at the schema and the object level may be expected. Evolution may either be *anticipated* or *unanticipated*. Anticipated evolution may be handled well by having powerful constructs on the class level, while unanticipated evolution is tougher, but necessary to handle.

Evolution at the *object level* is normal: The values of objects and the relationships between objects are changed due to normal application evolution or due to external requirements. There may also be harder evolution cases at the object level, e.g., in office information systems objects may change behaviour. Some of the development environments require the evolution at the object level to be recorded by versioning.

Evolution at the *schema level* is also quite normal: Classes are defined pre-requisitely to objects, which means that all aspects of objects must be defined before it is seen how objects behave. Design environments have more frequent changes on the schema level than traditional database applications, because of the evolutionary and experimental nature of the objects being modelled.

In software engineering and product development databases the data may be quite long-lived, which means that the data outdates the databases that it is contained within. This means that *evolution of the environment* is also an issue.

We address the first of these characteristics by using object-oriented databases as the context of our work, because we think they represent an effort to capture more of the semantics of complex engineering data.

We think evolution on the object level is well-covered by existing techniques. Instead, we will concentrate on evolution at the schema level. This is a typical problem found in development environments, but according to [Sjø93], this is an important topic also for more traditional database applications. We do not address the problem of evolution of the environment. We believe this is a superset of the issue of reengineering data from existing databases into new databases. Unfortunately, this is a very important topic not having sufficient interest in the research community.

### 2.3 Background for database integration

#### 2.3.1 Multi-database systems and federated database systems

Database integration is usually done in the context of multi-database and federated database systems. These database systems extend traditional database system by dealing with *distribution*, *heterogeneity*, and *autonomy*. A *multi-database system* supports operations on multiple *component database systems* [SL90]. When the component database systems are different, the multi-database system is said to be *heterogeneous*. Otherwise it is said to *homogeneous*. A *federated database system* is
a multi-database system, where the component database systems are autonomous, but they take part in a federation to allow partial sharing of data [SL90].

![Diagram](image)

**Legend:**
- Process
- Data or schema
- Data flow

**Figure 2.1:** System architecture for a federated database system.

In Figure 2.1 we have shown the reference architecture for federated database system described in [SL90]. This figure shows the system components of a federated database system and the processes of making a federated database from a set of component databases. The ellipses of the figure illustrate the five levels of schemas, while the rectangles illustrate the processes translating between the different schemas. The *transforming processor* translates from the local schema expressed in the data model of the component DBS, into the component schema expressed in the canonical data model (CDM). The *filtering processor* removes the parts of the component schema which should not be made available in a federation, and adds access control information. The *constructing processor* composes federated schemas from a set of export schemas. There may be several federated schemas in a federated database system.
2.3.2 Schema and object integration

The intention with schema integration is to establish a non-redundant and unified representation of all data being managed in an organisation [BLN86]. [She92, SM92] distinguishes between integration and interdependencies in database systems. Integration means to create new objects as a merge of the source objects, and to establish mappings maintaining the consistency of the new, merged objects with respect to the source objects. Interdependency means establishing relationships and dependency description between the source objects, but does not necessarily result in integration [SM92].

There are two categories of schema integration for database systems: Bottom-up schema integration and top-down view integration. Bottom-up schema integration is applied in retrospect to allow for transparent access to multiple databases. The process of building a federated schema (from the bottom and up in Figure 2.1) is an example of such an integration. The main characteristic of this type of integration is that it is unforeseen or unplanned. Such integrations are typically done to avoid redundancy of data in organisations having several databases. It may also be a result of the incorporation of new applications having new requirements, where uniform access to (what used to be) different categories of data is needed.

The other category of schema integration, view integration, represents planned integration during database design. This may be seen as a work splitting technique, where different development groups are given responsibility for designing the conceptual schema for some of the applications. Later, the different conceptual schemas (or views) are integrated to form a whole. This is done because it is easier to develop an application without considering the requirements of all other possible applications, and that it might be physically or socially difficult to cooperate with other development groups.

In this section we have described integration of schemas, which give uniform access to different databases. However, integration may be equally relevant at the object level. This means that there are objects in different databases being semantically overlapping, i.e. they are modelling the same conceptual objects. Object integration means establishing relationships between different database objects being semantically overlapping, and possibly merging these into global or integrated objects at the federated level. Object integration is done after schema integration, because it is only relevant when different classes are merged or connected in the schema integration.

2.3.3 Schema integration methodologies

Schema integration involves many steps, and is rather complex because it involves the recognition of similarities expressed in different data models, and the establishment of appropriate relationships between heterogeneous data. Due to this complexity, schema integration is usually supported by methodologies, i.e. descriptions of the process of schema integration and its relationship to the data and schemas involved.
We will present the general model for schema integration from [BLN86], which is more on the conceptual level than the process indicated in Figure 2.1. The following steps of activities are present in most schema integration methodologies [BLN86]:

**Pre-integration:** This step involves choosing which schemas to integrate, in what order they should be integrated, giving priorities to the different parts of the schemas. Related to the process in Figure 2.1, this step includes all steps necessary to construct the export schemas. Thus, it includes the important problem of transforming schemas expressed in the data models of the component DBS into schemas expressed in the CDM. All the ensuing steps are part of the constructing processor of the federated database system reference architecture.

**Comparison of schemas:** This step compares the schemas, and possible conflicts are detected, e.g., *name, scale,* and *domain partitioning conflicts.* The categories of conflicts involved here are dependent on the data model in question (e.g., the CDM in federated databases). These conflicts cannot be detected solely relying on the structure of the schema, but information about the *semantics* of the schemas is needed (e.g., see [She91]).

**Conforming schemas:** This step resolves the conflicts that are found in the comparison step. E.g., name conflicts are solved by renaming one of the conflicting names. This step will often involve transformations between different concepts in the data model in question. E.g., an attribute may be transformed to a class, before it is ready for integration.

**Merging and restructuring:** In this step the schemas from the previous step are composed or merged into one, and some restructuring may have to be applied. E.g., two similar classes should be *generalised* to give uniform access to the extents of these classes.

In Section 3.5 we will survey various constructs which are used to integrate classes in the context of object-oriented databases.
Chapter 3

Evolution and Integration Techniques

3.1 Introduction

This chapter surveys different techniques to manage evolution and integration at the schema level in object-oriented databases. We will also treat some interesting constructs from object-oriented programming languages. At the end of this chapter we have included a section on evolution at the object level.

Throughout this chapter we will present some examples within each category of construct/technique. These examples are expressed using a C++-like data model. The basic part of this data model is the same as the basic object model, which will be presented in Section 4.1. These examples are intended to illustrate the different techniques and the problems they solve.

We have tried to keep a uniform vocabulary throughout this chapter. If the original vocabulary differs, we will add this in parentheses.

3.2 Schema extensions

This section treats constructs which extend or customise classes into new classes.

3.2.1 Subclassing

Subclassing is usually recognised as a modelling construct. We now present the capabilities of this construct that cater for evolution.

As part of a schema, we shown a class CSource below, which describes the properties of C modules. Note the startEditor operation which invokes an editor and, upon termination of an editing session, updates some of the other properties of the objects (e.g., author and lastUpdate):
class CSource {
    public:
        virtual startEditor ();
    setOf <string> keywords;
    setOf <link <CSource>> includes;
private:
    string author;
    date lastUpdate;
    longfield contents;
};

Later, there is a need to refine the CSource class to represent C headers and C implementations separately. A rationale for this extension is that some clients are only interested in querying the set of C headers, and are not interested in the implementations. This extension is done in object-oriented databases by defining two subclasses of the CSource class: CHeader and CImpl.

class CHeader : public CSource {
    public:
        startEditor ();
    link <CImpl> impl;
};

class CImpl : public CSource {
};

There are two main evolution capabilities in subclassing. First, it allows for inheriting and redefining the definitions of properties in the superclasses. Thus, subclassing may be seen as a reuse and tailoring (customisation) construct. This caters for certain limited evolution cases by adding, removing or redefining properties. This is seen in the two subclasses above, where CImpl and CHeader inherit properties from CSource. CHeader adds the impl attribute, and redefines the startEditor operation (e.g., to start a syntax-oriented editor for C header files). This redefinition means that a request for startEditor to objects of CHeader, will be bound to this operation, and not the one implemented in CSource.

The second evolution quality of subclassing is that the extent of a subclass is automatically included in the extents of its superclasses. E.g., the extents of CHeader and CImpl are included in the extent of CSource. All objects created of either of the two subclasses will be of the CSource class as well, i.e. when querying the CSource class, all the objects existing in the subclasses will be retrieved. Since CImpl adds no properties, the sole purpose of this class is to hold the specific extent. The two subclasses preserve the clients of the original class. All existing objects are still accessible through the class, and all new objects will be included in the extent of the original class.

The combination of these two evolution capabilities allows for the following: By iterating through all objects of CSource and requesting startEditor for each of them,
different editors may be started. For objects created in CHeader this request will be
dynamically bound to startEditor in CHeader, which starts a syntax-oriented editor
for C headers. For all other objects the request will be bound to startEditor in
CSource, which starts a traditional text editor.

The extent inclusion semantics is ensured by the creation and deletion operations:
When an object is created in a subclass, it automatically becomes a member of the
superclasses as well. Similarly, when it is deleted in the subclass, it is deleted from
the superclasses as well. We consider extent inclusion as an important evolution
facility, because it allows the extent of a class to be extended by new, possibly
customised objects. Generally, extents and extent inclusion are important for query
processing, because they represent predefined groups of objects which are likely to
be processed together.

The ability to remove properties in a subclass compared to its superclass may cause
problems when combined with dynamic binding. When requesting this property for
all objects of the superclass, a “property not found” error occurs for all objects of
the subclass having this property removed. This causes problems for clients of the
superclass.

Subclassing has also been used in semantic data models, e.g., SDM [HM81] and
Daplex [Shi81], prior to the introduction of object-oriented databases. We will com-
ment on the evolution capabilities of semantic database systems using the IFO model
[AH87a] as an example.

The IFO model is a rich structural data model. The model has three categories
of classes (types in the IFO model): printable, abstract1 and free. Printable classes
are basic classes that may be represented on the screen, like strings, integers, and
booleans. Abstract classes are used to represent real world objects by attributes of
the object, e.g., a Person class. A free class is connected to an abstract class through
an ISA-link as either a specialisation or a generalisation class. An object created in
an abstract class may obtain membership in connected free classes.

Specialisation links are used to define additional roles for members of abstract classes,
while generalisation links are used to represent situations where distinct, preexisting
classes are combined to form new classes. Abstract and free classes have different
semantics for creation and deletion of objects. Abstract classes are regarded as the
primary class of an object, in the sense that the object must be created there. If
an object is deleted in an abstract class, it is deleted from the whole database. Free
classes are more like roles: creation and deletion of objects in free classes are mere
adding and deleting roles of objects existing in the primary classes.

The ISA-links are primarily meant as a modelling construct. This capability may
be used for simple class evolutions, like adding new “roles” to objects, and making
generalised classes. But the IFO model has not the customisation capability of
object-oriented subclassing, where dynamic binding together with subclassing allows
for replacing the properties of the superclass.

1 Abstract classes in the IFO model are the opposite of abstract classes in object-oriented systems
with respect to the capability of creating objects.
3.2.2 Miscellaneous language mechanisms

We will now survey the evolution capability of various language constructs. These constructs primarily exist within programming languages, but we include them here due to the trend of adopting more programming language constructs into database systems. However, we will make some comments on how relevant these constructs are for databases. Some of these mechanisms are more generally compared in [Mey86] and [BGM89].

Subtyping

According to Blair [Bla91], subtyping\(^2\) expresses behaviour sharing between classes (types in [Bla91]). In contrast, Blair defines subclassing to be sharing of implementation. A class \(C_2\) is a subtype of a class \(C_1\), if \(C_2\) shares the behaviour of \(C_1\), i.e. \(C_2\) provides at least the behaviour of \(C_1\). An object of class \(C_2\) can thus be used as if it is of class \(C_1\). Usually, the “behaviour” of a class is defined as its interface, being a set of operation signatures.

Subtyping is usually used as definition of type compatibility, and is used for type-checking. Given a class \(C\) which is a client of \(C_1\). Any class \(C_2\) being a subtype of \(C_1\), may type-safely take the place of \(C_1\). This means that objects of \(C_2\) may type-safely be substituted for objects of \(C_1\).

An acknowledged example of subtyping is the conformance rules of Emerald [BHJL86, RTL+91]. A class in Emerald has a set of operations as its interface. A class \(C_2\) is a subtype of (conforms to in [BHJL86]) \(C_1\) if:

1. \(C_2\) provides at least the operations of \(C_1\),
2. for each operation in \(C_1\), the corresponding operation in \(C_2\) has the same number of parameters (arguments) and results,
3. the classes of the results of \(C_2\)’s operations conform to the classes of the results of \(C_1\)’s operations (i.e. “narrower” results), and
4. the classes of the parameters of \(C_1\)’s operations conform to the classes of the parameters of \(C_2\)’s operations (i.e. “wider” arguments).

The reason for this contravariance rule for operations, is to ensure that static type-checking is possible [MS82].

What role can subtyping play in evolution? Given a class \(C\) that is a client of class \(C_1\). Then, objects of any class \(C_2\), which is a subtype of \(C_1\), may type-safely substitute objects of \(C_1\). This means that we may use subtyping as a mechanism for checking if we can safely replace a class by a new class. Consider the class \texttt{Window} given below:

\(^2\)[Car84] is usually credited for defining subtyping in an object-oriented context.
class Window {
    public:
        move (int newX, int newY);
        draw ();
};

The class Widget given below is a subtype of Window:

class Widget {
    public:
        move (int newX, int newY);
        scale (float xs, float ys);
        draw ();
};

This means that any client using objects of Window can type-safely use objects of Widget instead.

The usefulness of subtyping as an evolution construct is limited, because few class "replacements" are behaviourally compatible. We would also like to mention that when a class C is a client of a class C₁, it is also a client of the set of objects of C₁. Thus, replacing C₁ by a new class being a subtype, means that we will (possibly) replace it by another set of objects being semantically different.

Parameterised classes

This is a technique that allows for parameterising classes with other classes [Bla91], e.g., the template construct in C++ [Str91]. A parameterised class is instantiated by another class. Parameterised classes are either unconstrained or constrained with respect to their parameters. Unconstrained means that any class may be used as the actual parameter to the parameterised class. This is typically parameterised classes only storing or managing objects without being dependent on their public properties. Below, we have shown an example of a parameterised class Stack having the parameter T, and an instantiated class aTerminalStack (in a C++-like syntax):

template <class T>
class Stack {
    private:
        link <T> s[MAXSTACK];
        int top;
    public:
        push (link<T> a);
        link <T> pop () ;
        int size () ;
};

stack <Terminal> aTerminalStack;
A constrained parameterised class means that it is put constraints to the actual class parameters, requiring the classes to be used as parameters to have the given properties. Typically, the actual class parameter should be a subtype of the formal class parameter.

Parameterised classes are useful when having anticipated evolution on the schema level. When creating a parameterised class that can be instantiated by a number of specific classes, it means that we have planned or foreseen the need for a group of similar classes. For every instantiated class of the parameterised class, we reuse definitions and code. In the Encore Data Model [SZ90] set classes (set types in [SZ90]) and tuple classes are instances of parameterised classes.

The incorporation of user-defined parameterised classes into the database greatly increases its potential for capturing specific classes that are needed in the future. However, parameterised classes are limited in the sense that what can be done with objects of parameterised classes is closed by the definitions in the parameterised classes. E.g., we cannot evolve the behaviour. Hence, we may say that parameterised classes are capturing a limited set of anticipated evolution, and we believe this to be the intended use of parameterised classes.

The traditional view of database systems is that the definitions made in the data definition language (e.g., a Person class) have extensions, which are sets of objects (e.g., the set of persons). It is not so obvious what is the extension of a parameterised class. The first possibility is to view the parameterised class as a superclass, making the extent the union of the extents of the instantiated classes. This means that the extent of the parameterised class type is a rather heterogeneous set, with elements having different characteristics. The second possibility is to view the parameterised class as a metaclass, i.e. the class of the class. Then the extent of the parameterised class is a set of instantiated classes. This resembles a data dictionary which is used for database administration purposes. For user purposes this might be applicative in reflective systems, e.g., in knowledge representation systems. However, we think parameterised class are more applicative in programming languages than in database and knowledge representation systems. This discussion is falling into the category of general discussion of merging programming language and database concepts [BZ87].

Excuses

[Bor88] shows an approach for handling abnormal objects with respect to a class. The approach is to create a subclass which “excuses” what was wrong in the superclass. This is similar to redefinitions of properties, but the excuses concept is incorporated into the system to avoid problems with type checking, to detect unsafe operations on objects, and to handle database queries correctly [Bor88]. The excuses are used to notify the system that it may retrieve objects not conforming to the superclass definition, and to let the system handle this.

Excuses may be used to handle groups of “anomalous” objects by tailoring the definitions of a class. The main advantage of it is that we may explicitly notify the system that it can expect non-conforming objects when querying a specific class.
3.3 Schema modification

3.3.1 Class modification

While schema extensions left the existing classes unchanged when evolving the schema, existing classes will here be modified to meet the new demands or requirements. The advantages of this are that it does not create new “artificial” classes to overcome problems with classes that never were right, and that the number of classes never is larger than what is necessary to model the current view of the world.

Class modification may be performed locally within a class, i.e. by removing, adding, or modifying attributes and operations of the class. Class modification may also be performed to the class hierarchy. Consider the following class hierarchy (taken from [Ped89, PS91b]): A class Device have three subclasses Terminal-1, Terminal-2, and Terminal-3. Later, it is discovered that there is a missing abstraction in this hierarchy. There is a need for a new class ANSI-terminal in between the other classes. This is shown in Figure 3.1. The purpose of this class is to serve as a useful abstraction for the three terminal classes, such that it becomes easier to extend the hierarchy by similar terminals later, and that it becomes easier for other people to understand the hierarchy. This modification to the class hierarchy may be viewed as a series of

![Diagram of class hierarchy](image)

Figure 3.1: Missing classes example.

atomic class modifications: The three inheritance links are removed. ANSI-terminal is defined as a subclass of the Device class, and Terminal-1 through Terminal-3 are defined as subclasses of ANSI-terminal.

Casais [Cas90] has made a generalised list of tasks performed when doing class
modifications:

1. Determine a set of integrity constraints that a collection of classes must satisfy. E.g., all attributes must have unique names; there should be no cycles in the inheritance graph.

2. Establish a taxonomy of all possible (atomic) updates to the classes. E.g., add an operation; delete a class; rename an attribute.

3. For each of the update categories in the taxonomy, a precise characterisation of its effects on the class hierarchy is given, and the conditions for its applications are analyzed. E.g., if an attribute of a class is deleted, all use of that attribute should be removed as well.

4. The effects of the schema changes are reflected to the persistent store by converting the objects to conform to the new class definitions.

We will now comment on some of the approaches to class modification which have been presented in the context of object-oriented database management systems.

- GemStone [CM84, MSOP86] is an early approach to object-oriented databases and to class modification [PS87]. It allows for adding, removing, renaming of attributes, and refining the class of attributes. It does not allow for doing any evolution with respect to the operations (methods), or to the class hierarchy, but it does allow for adding and dropping indices.

- ORION [BKKK87, Kim90a] shows a rather complete approach to class modification in presence of the ORION data model, which has focused on rich semantics for composing objects. It allows for adding, removing, and renaming attributes and operations. It also allows for changing the implementation of operations, and for adding and removing inheritance links between classes. Further on, it can generalise classes from existing classes, and manage evolution of shared values and composition links [Kim90a, Chapter 5].

- OTGen [LH90] is a tool which is applied to two versions of a class and automatically produces the programs and tables that are necessary to convert the existing objects to the new version of the class. The transformations are table-driven having default rules for how to transform on each type of class change. The default rule may be replaced by the database administrator. This is done in a structurally object-oriented database context, thus it may be categorised as a similar, but adaptable solution to the class modification problem. [LH90] also shows that their approach can be used to database reorganisations.

- $O_2$ Integrity Consistency Checker [DZ91] is similar to the ORION approach, but is mostly focused on ensuring structural consistency, and is generally simpler than the ORION approach, due to the $O_2$ data model [LRV90] being simpler.
For class modification there are generally two approaches for when to convert the objects from the old schema to the new schema. It is either done eagerly, or it is done lazily upon the first access of the objects. Related to performance and availability of the database, these two approaches have different advantages and disadvantages. Immediate conversion gives no overhead when accessing the database later, but may require a substantial down-time of the database when converting the objects. Lazy conversion gives a short down-time, but will give some overhead each time a new object is converted. GemStone converts the objects eagerly, while ORION and OTGen do it lazily. The approach presented in [DZ91] will not allow for a class modification to be done if the database is not consistent with the schema, thus it cannot do it lazily.

There are two issues that ensure that the modification primitives capture what is important to the user:

**Completeness:** Does the set of proposed changes actually cover all possibilities for schema modifications?

**Correctness:** Do these changes really generate classes that satisfy all integrity constraints?

ORION’s transformations are complete at the schema level, while GemStone’s transformations are not, but is restricted to cases that may be handled without too much loss in efficiency. The correctness property has been studied by [DZ91]. It focuses on *structural consistency*, referring to the structural part of the database. On the schema level of the $O_2$ data model this means that: the inheritance structure is a directed acyclic graph (DAG), attribute and operation scope rules are kept, and similar. On the object level it means that the values of objects must be consistent with the class it belongs to.

Recently, it has been focused on analysing the consequences of class modifications for the behaviour of the objects. [Zic91a, DZ91] introduce *behavioural consistency*, which is defined as follows: Each operation respects its signature, and its code does not result in run-time errors or unexpected results. To ensure behavioural consistency, it suggests to use data-flow techniques. [Wal91, AKRW92] introduces *method schemas* as a simple programming formalisms for object-oriented databases. Computations are modelled as graphs, and are used to check whether a given method schema can lead to inconsistencies, i.e., if it can lead to undefined operation requests. Schema evolution is formalised as *incremental consistency checking*, and a heuristic is given for such checks.

Compared to arbitrary modifications class modification provides a well-defined framework for schema modifications. It takes care of the existing objects, and will ensure that the schema and the database is kept consistent. But it does not consider the schema as a whole, only local constraints are taken under consideration. If attributes are removed, the information kept there will be lost. Further, it insists on keeping one canonical version of the schema that all clients and objects must comply to. That is, if two clients (users) disagree about how a class should look like, they have to make a compromise, or they will not be able to share the class and its objects.
3.3.2 Class reorganisation

Class reorganisation is a special case of schema modification, focusing on doing automatic modifications according to certain principles for organising classes into hierarchies.

Incremental class reorganisation

Casais [Cas91, Cas89] treats several aspects of evolution in object-oriented environments. It is argued that object-oriented concepts like specialisation, parameterisation and reusability are insufficient to cope with evolution. A set of algorithms for managing incremental class evolution is proposed, and is used when new subclasses are added to a class hierarchy. The algorithms are categorised either as decomposing or restructuring classes. The work is done on a synthesised object model capturing concepts from several object-oriented programming languages.

The algorithms are driven by different design criteria, e.g:

- Attributes should not be removed in subclasses.
- Operations should not be removed in subclasses.
- If an operation is redefined in a subclass, this operation should be virtual empty\(^3\) in the superclass, and implemented in the subclass.

We will now illustrate one of his algorithms by showing a tiny example. At the top of Figure 3.2 we show the initial situation. The class **MatrixPrinter** has an implemented operation **print**. This class is subclassed by a class **LaserPrinter** also having an implemented operation **print**. The incremental reorganisation driven by the virtual empty criterion does not allow for this situation, and will create a new abstract superclass (shown at the bottom of Figure 3.2) having a virtual empty operation **print**, and the two subclasses **LaserPrinter** and **MatrixPrinter**, both having **print** implemented. We have named this abstract superclass **Printer**, showing that it is somewhat simulating an interface definition for the two (implementation) subclasses. Casais’ algorithms do not create any names for the additionally created classes. Note also that this is a very simple example. Casais’ algorithms work on several levels in the class hierarchy, possibly reorganising all classes above the inserted class in the class hierarchy.

Casais shows the relevance of his algorithms by doing reorganisation on Version 2.1 of the Eiffel library [Mey88]. Many of the results found correspond to the results found (the hard way) by the implementors of library themselves, which is documented in [Mey90].

The algorithms of Casais are designed for class hierarchies in object-oriented design and somewhat for implemented class hierarchies in a programming language. They are not directly applicable to databases, because the algorithms do not consider

\(^3\)I.e. defined, but not implemented in the superclass. E.g., virtual int f() = 0 in C++.
Class refactorings

Opdyke [Opd92] describes class refactorings – an approach for automatic restructuring of object-oriented application frameworks, which are classes organised in hierarchies. The motivation for the refactorings is to be able to extract specific functionality out of a complete framework, to be able to improve the consistency among components of the framework, and to be able to support iterative design in the framework (a framework may be seen partly as design, and partly as implementation [WJ90]).

[Opd92] assumes a C++-like data model, and defines a set of low-level (atomic) refactorings on this data data model, e.g., create an empty class, change an attribute’s name, delete a parameter of an operation, change superclass, etc. From these low-level refactorings, several high-level refactorings are built:

Generalisation: This is a refactoring to create an abstract superclass from two classes. It is based on the assumption that a mature class hierarchy has concrete leaf classes, while the other classes are abstract.

The refactoring includes the following steps: The signatures of operations of the
subclasses are made compatible, which again means that clients of the classes must be updated to the new signatures. The compatible operation signatures are added to the superclass. The operation bodies (implementations) are made compatible in both subclasses. This is done by comparing the bodies as strings or syntax trees. Common code and attributes are migrated to the superclass.

**Specialisation:** This is a refactoring which is used when a class embodies a general abstraction and several concrete cases [Opd92]. To do this, the C++-like classes are extended with class invariants, which are predicates on the state of objects, i.e. attributes of the class are the free variables. The concrete subclasses are recognised by finding conditional statements in operations, which may be handled by letting the objects take care of the conditions by dynamic binding and being of different subclasses. When a condition indicating a subclass is found, a subclass is created with a class invariant matching the condition. The body of the conditional statement is moved to a corresponding operation in the subclass. This refactoring also requires to specialise the creation statements of objects of the classes involved.

**Aggregation and component:** These are refactorings to make components of composite objects explicitly available, and to convert from aggregation by inheritance to explicit aggregation.

A basic quality of the refactorings in [Opd92] is that they are behaviour preserving, i.e. the set of classes after the refactorings are legal, and perform operationally equivalent to the classes before the refactoring. But as seen in the generalisation refactoring, this may include conforming the clients of classes to the new interfaces.

The comments made to [Cas91] apply to [Opd92] as well. The refactorings are based on principles for good organisation of object-oriented programs. However, for object-oriented databases the extent dimension of classes and the existing objects must be taken care of as well.

**Object-preserving class transformations**

[Ber91a] shows the object-preserving class transformations of the Demeter System (partially described in [W.J90]). Object-preserving means that all objects will retain their original class and their original attributes. It is a small set of primitive transformations:

1. Deletion of not-used, empty abstract classes.
2. Addition of empty abstract classes.
3. Factorisation of common attributes.
4. Distribution of common attributes into subclasses.
5. Replacement of links to equal classes.
It is shown examples of composition of these primitive transformation into higher-
level, useful class transformations, like elimination of redundant attributes and elim-
ination of multiple inheritance.

These transformations are tightly connected to the Demeter object model [Lie88],
which only allows for leaf-classes to be concrete. That is, all non-leaf classes in the
hierarchy are abstract. The transformations do not preserve extents and their extent
inclusions.

### 3.4 Schema versioning

Versioning serves two main purposes [Lie90]:

- It represents a development history where each new version supersedes the
  older ones (sequential revisions).
- It represents co-existing versions, being used at the same time (parallel vari-
  ants).

For database schemas both of these purposes may be relevant. Variants represent
situations where different clients of the schema have different needs, and cannot
agree upon the schema. The different clients will then have their own co-existing
variants of the schema. Revisions are applicable when a schema is found to not be
“correct” or sufficient, and is “replaced” by a new and better one.

It may be questioned if schema versioning is a relevant issue for databases at all.
At the heart of database technology lies sharing – both at the schema and the data
level. During schema design, much work is put into getting a consolidated schema
for all clients. Schema versioning comes into play, when the initial schema design
was not correct for all clients, and when the domain being modelled is evolving. We
think schema versioning is especially relevant for object-oriented databases, because
they are often claimed to support engineering and other “evolutionary” application
areas. Compared to schema modification, schema versioning is chosen because it
allows different clients to use different versions of the schema. For corrective changes,
schema versioning may be necessary because a particular client cannot wait for all
other clients to be conformed to the new schema. Schema versioning may also be a
necessity due to other clients not being available for modification.

We will show an example of this based on one found in [SLU89]. Given a class hier-
archy for some simple geometrical objects as illustrated in Figure 3.3. Triangle and
Rectangle are subclasses of Shape. Initially, the geometric objects are colourless,
e.g., they are considered black on a white background. After a while some clients
would like to have coloured objects, and the following requirements are proposed:

- All existing (black-and-white) objects should be available as coloured objects
  as well, and all coloured objects should be available as uncoloured objects.
- The old black-and-white class hierarchy should be retained.
• A separate hierarchy for coloured objects should appear.

This means that we would like to have two separate class hierarchies, one for uncoloured objects and one for coloured objects. At the same time we would like that all objects may be seen both as coloured and uncoloured objects. The reason for retaining the old shape hierarchy may be for history tracking, for later (uncoloured) extensions, and to ensure that existing clients will not be corrupted by changes in the schema.

In a schema modification solution to this problem, the Shape class would have been modified directly by adding a colour attribute (and possibly some operations to make use of it). This requires all existing objects in the shape class to be converted to the new class. This solution will not keep two separate class hierarchies, and it must be ensured that existing clients are not damaged by the modification.

We will now try to solve this problem by the use of the built-in subclassing mechanism of object-oriented data models. We let each existing class in the shape hierarchy have a new subclass representing the coloured version, as illustrated in Figure 3.4. This solution retains the original class hierarchy, and has a separate colour
shape hierarchy. Objects of \texttt{ColRectangle} and \texttt{ColTriangle} become black-and-white and coloured shapes by the use of multiple inheritance. This solution retains all existing clients, since the black-and-white hierarchy is unmodified. The problem with this solution is that black-and-white shapes will not be automatically available as coloured shapes.

Thus, schema versioning requires other solutions than the built-in subclassing mechanism and class modification. We will now present some special-purpose mechanisms for class versioning.

3.4.1 Class versioning

Class versioning is similar to class modification, but instead of discarding the old classes, they are kept to prevent existing clients from becoming inconsistent with respect to the class definitions. The main property of class versioning is that it preserves \textit{class change transparency} [ABB+83, Zdo90]:

- Old clients will work with objects of new versions of a class.
- New clients will work with objects of an old version of a class.

Thus, the change to the class is transparent to clients. Old clients are allowed to exist without being changed, and new clients having slightly different requirements to the class, will be able to use existing objects. Another way of saying this is that the different versions of a class share extension. This is a tough goal to achieve when efficient retrieval and updates of objects are required.

Substitute read and write operations

[ABB+83] is the first approach that we have noticed to provide class change transparency. Each class is organised in linear chains of versions. An object is represented in the class version where it was created. When an object is accessed through another version than the one it is represented within, a conversion will take place. This is done by two categories of \textit{substitute operations}: When an object being represented under class version \(i\), is requested for the value of an attribute \(a\) of class version \(j\), a \textit{substitute read operation} will compute \(a\) from the values of attributes of version \(i\). Similarly, when \(a\) is updated, a \textit{substitute write operation} will update the attributes of the object in version \(i\).

This means that objects are only represented in the version of the class they were created from. If they are requested through other versions of the class, the object must be mapped back and forth. There are two main problems with this approach. First, when attributes are added in new versions of a class, these cannot be computed or stored for objects of old versions of the class. Such versions would not be regarded as \textit{consistent}, and are outside the scope of class versioning, according to [ABB+83]. Second, as classes may exist in long chains of versions, two different problems may appear: First, a large number of substitute read and write operations must be defined
back to every existing versions of the class. Second, this problem may be avoided by relying on the substitute read and write operations of the previous version. This means that when requesting an attribute of a version at the end of a chain, the request may possibly be mapped through all versions to the other end of the chain.

**Multiple views**

[Zdo90] presents the *multiple views approach* to class versioning. Each time a class changes, objects of this class acquire another layer that reflects those properties that are peculiar to the new version of the class. Thus, each version is a view on previous layers. A layer can introduce new attributes and operations, it can suppress the availability of an operation, and it can make use of operations in a previous layer.

![Figure 3.5: The multiple views approach.](image)

Figure 3.5 illustrates these concepts. C1 and C2 are versions of class C. O is an object of C, and op1 through op6 are operations that are defined on their respective class versions. C1 is the part of the object defined by the first version of class C. It has a representation used to store its attributes and it has four operations to access these attributes. C2 is the part of the object that is defined by the second version. It adds some additional attributes to support its new operation op6. Another new operation, op5, makes use of version C1. Operations op1 through op4 are not available as a part of the interface of C2. C2 can only access the attributes in C1 through C1’s interfaces. Thus, encapsulation is not broken. This may be an advantage to data abstraction and the integrity of each class version, but may a disadvantage with respect to efficiency. It is questionable if new class versions should have larger access permissions than ordinary clients or not (see Section 5.7.4 for more on this).

[Zdo90] extends [ABB+83] by allowing an object to be represented in several versions of a class. Thus, addition of attributes is possible. However, [Zdo90] only treats attributes which are either independent (added) or completely derivable. Another problem not solved in [Zdo90], is how to give values to independent attributes when the object is created in another version of the class. E.g., if an object is created in C1, how are the attributes used by op6 achieved for the same object?

[ Cla92] further extends [Zdo90] by introducing a richer apparatus for dependencies between attributes of different versions (facets in [Cla92]) of a class. Attributes of a version may be categorised as *shared*, *independent*, *derived*, or *dependent* with/
on another version of the class. Derived means that the value is computable from attributes of the other version, while dependent means they are dependent, but not derivable. A class may be seen as a disjoint union of all its versions. Unlike [Zdo90], [Cla92] does not respect encapsulation between class versions. All dependencies are described between attributes of objects.

[Cla92] shows two approaches for maintaining the dependencies: Propagate updates of attributes eagerly or lazily when the dependent attribute is read. A similar lazy strategy for creating representation of existing objects in new versions is also adopted. [Cla92] recognises that optimisations can be done to the disjoint union representation of versions, by letting shared and mutually derivable attributes have shared representation.

[MS92] introduces a prototype object environment named CLOSQL with versioning of classes. It allows for linear chains of versions of classes, but recognises the need for non-linear versioning as well. The approach is similar to [Zdo90], by representing objects “incrementally” in each version of the class. It uses backdate and update operations to convert an object temporarily upon request between different versions. All mappings between versions are done directly between adjacent versions in the chain. Thus, this approach suffers from the same problem with long mapping chains as [ABB⁺83]. When attributes are added in new versions, existing objects will not have values for these attributes. [MS92] handles this by exceptions, either trapped to the application program or to the user.

[Ber92] introduces updatable views for schema evolution. The idea is that the user should define views to experiment with schema changes prior to actually installing them. This approach is different from the other mechanisms by being an application of general database views defined in a query language. These views allow for augmenting the base classes (the classes queried from) with attributes and operations. This gives them the same capabilities as [Zdo90] with respect to viewing existing objects through new versions, but it does not solve the problem of viewing objects created in new versions (i.e. the views) through old versions.

**ENCORE: The exception handler approach**

[SZ86, SZ87] is similar to [ABB⁺83] with respect to representation of objects. An object is represented under the version of the class that it is created. When a new version of a class is added, the designer must also add exceptions to handle cases for which there is a conflict between old versions and the new version. Conflicts occur when a client attempts to access properties which are undefined for the class version the object is represented in, and when a client reads or updates a value that is unknown. The first category of errors occurs when a new class version adds or deletes properties, and the second occurs when the domain of an attribute is strengthened or relaxed (specific to the data model). The error handlers are classified as either prehandlers or posthandlers. A prehandler is executed when a property cannot be found for the version requested, while a posthandler is executed when an unknown result is about to be returned. These handlers are mainly used for mapping from one value to another.

37
Miscellaneous

[Odb92] considers schema versioning in context of a data model distinguishing between interface and implementation of classes. The advantage of this model is that changes which are not related to the interface, will be done by creating new implementations, and clients will not be affected at all. However, changes to the interface will be as cumbersome as in any of the other approaches.

A typical argument for distinguishing between interface and implementation of classes, is that an interface may have multiple implementations, each suited towards specific needs or implementation restrictions [ACK+91]. This can be used to change the implementation of objects. We doubt how reasonable this really is, because the interface of a class will very often contain an operation to create and initialise an object, e.g., new in Smalltalk-80 [GR83] and the constructors of C++ [Str86]. The implementation of an object will often shine through in these operations, because they usually have parameters to give initial values to the attributes representing the implementation. When the implementation of a class is to be changed, these operations may also have to be changed. Thus, the interface of the class will have to be changed, and all clients creating objects will have to be changed as well.

Comments to class versioning

Class versioning prevents information from being lost, and allows for different clients to have different views on how the class should look like. Class versioning is especially useful in design environments where changes are frequent and unpredictable, and in organisations having many different clients towards one schema, where maintaining all clients towards a canonical schema requires much work. When using the lazy allocation scheme of existing objects to new class versions, class versioning saves the clients for a long down-time in converting all objects to the new version of the class.

The main problem with class versioning is that it may be costly in time and space. If the conversion routines between different version of a class is written in a general programming languages, it might give too much overhead, especially upon a high number of class versions. Another disadvantage of class versioning is that it makes the schema more complex, since many versions of a class are available to new clients.

Class versioning and class modification do not exclude each other. An interesting combination could be to let all class modifications initially be implemented as class versions. When all clients are updated to the new version of the class, older versions of the class may be discarded.

3.5 Schema integration

In Section 2.3.3 we treated schema integration methodologies. In this section we will survey (language) constructs used for integrating classes in object-oriented database systems.
3.5.1 Class integration

It is not uncommon that databases created and developed individually, later have to be integrated. Typical reasons for this may be that it is easier to develop and maintain an application without considering the needs of other applications with respect to data sharing, and it might be physically or socially difficult to cooperate with other development groups.

To introduce some of the problems in class integration, we give an example adopted from [SN88]:

```c++
class OilPlant {
    public:
        fireOn ();
        powerOff ();
        fillOil (int barrels);
        string plantName;
        long produced;
        int oilFired;
};

class CoalPlant {
    public:
        start ();
        powerOff ();
        putCoal (int tons);
        string plantName;
        long produced;
        int consumed;
};
```

The two classes OilPlant and CoalPlant should be integrated such that it is possible to be clients of power plants without knowing if it is oil plants or coal plants which are used. At the same time coal and oil plants should be available as they are. This means that a generalised class PowerPlant should be defined, as shown in Figure 3.6.

![Figure 3.6: The class integration example.](image)

In this example the generalisation class PowerPlant should be *abstract*, i.e. it is not possible to create objects from it. This is because power plants will either be coal or
oil plants. `PowerPlant` should have the union of the extents of `OilPlant` and `CoalPlant`, and an “integrated” set of properties.

This “integrated” set of properties may be created based on semantic similarities between the properties of `OilPlant` and `CoalPlant`. `OilPlant`’s `fireOn` operation *semantically corresponds* to `CoalPlant`’s `start` operation, which means that they conceptually are the same operation. Thus, there is a *name difference* between those two operations. Another difference is that reloading oil plants is done in measure of barrels of oil, while reloading coal plants is done in tons of coal. This is called a *scale difference*. After integration, it should be possible to operate on oil or coal plants without needing to know if your plant needs oil or coal. One rationale for this requirement is that a client might be given a plant and an amount of “refill” that it might be issuing the plant, and not knowing what type of plant it actually is.

Traditional object-oriented data models do not support generalisation directly. Generalisations can only be created by modifying existing classes, objects and clients:

```cpp
class PowerPlant {
    public:
        virtual powerOn () = 0;
        virtual powerOff () = 0;
        string plantName;
        long produced;
        int consumed;
};

class OilPlant : public PowerPlant {
    public:
        powerOn () {...}
        powerOff () {...}
        fillOil (int barrels) {...}
};

class CoalPlant : public PowerPlant {
    public:
        powerOn () {...}
        powerOff () {...}
        putCoal (int tons) {...}
};
```

Here, we have removed the two classes and replaced them with three new classes: `OilPlant` and `CoalPlant` are made subclasses of the new generalised class. The interfaces of the classes are conformed: `fireOn` and `start` have become `powerOn`. `PowerPlant`’s `powerOn` is only an empty operation definition, which is implemented in the two subclasses. Dynamic binding assures that a request is bound to the right operation. We cannot do the same for `fillOil` and `putCoal`, because they have scale differences in their parameters. This means that this solution does not integrate the refill operations of those two classes.
We have now modified the existing classes, which means that all clients being dependent on the previous interface, have to be modified to conform to the new interface. E.g., all requests to `fireOn` and `oilFired` should be changed to `powerOn`. This process may be cumbersome and error-prone.

The next step in the process is to convert the existing objects of the two classes into the representation of the three new classes. This would probably be done by an off-line conversion process reading the old objects and creating new objects with the same values for attributes. If there are clients being dependent on the OIDs (e.g., in references), we either must have the ability to create objects with given OIDs, or scan the whole database and update references with the new OIDs.

As we have seen in this section, class integration is a tricky business when it is not supported by the object model itself. It required us to modify the existing classes, clients, and objects. Note that in some integration situations, due to semantically overlapping objects, we may have to integrate objects as well. In the ensuing sections we will see some approaches which preserve the existing classes, objects and clients, and which define integration classes on top of existing classes. This allows existing clients to keep their old classes, or to be conformed to the new integrated classes.

### 3.5.2 Integration by views

The relational model (originally presented in [Cod70]) gained popularity due to its simple model and powerful query languages. A strong point of the relational model is that the declarative nature and the simplicity of the relational query algebra made it possible to achieve deep insight into query optimisation. Another strong point was the simplicity of views, which are named, derived tables expressed as normal queries. These views may be utilised in integration. E.g., two tables may be “generalised” by defining a view using SQL’s `UNION` between the two tables.

[DH84] introduces views as means for database integration in multidatabase systems. Views are defined using a functional variant of QUEL against the functional data model (DAPLEX [Shi81]). The views are used to define generalisation classes (types in [DH84]), describing how the properties (functions) of the generalisation class relate the properties of the classes being generalised. [Mot87] describes a formal method for generating `superviews` for multidatabases. It defines a set of operators to integrate and modify database schemas expressed in a simple functional data model. Based on the integration operators applied to integrate two schemas, mappings from queries of the integrated schema into the original schemas can be generated.

We will now survey three approaches to integration by views, which are done within the context of object-oriented data models. Note that there is no common agreement on what “views” mean in object-oriented databases, but there are several proposals, e.g., [HZ90, Day90, SLT91].
ViewSystem

ViewSystem [KDN90] is an object-oriented programming environment with dedicated operators for integration of heterogeneous information. Integration in ViewSystem is based on recognising semantic relationships between the classes to be integrated, and from these build derived (view) classes on top of the already existing classes.

The following semantic relationships with their corresponding set relationships are recognised:

<table>
<thead>
<tr>
<th>Semantic relationship</th>
<th>Set relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialisation</td>
<td>Subset</td>
</tr>
<tr>
<td>Category generalisation</td>
<td>Disjoint union</td>
</tr>
<tr>
<td>Role generalisation</td>
<td>Overlapping union</td>
</tr>
<tr>
<td>Grouping</td>
<td>Power set</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Cartesian product</td>
</tr>
</tbody>
</table>

ViewSystem categorises classes as extensional (being stored) or intensional (non-materialised). Intensional classes are either external (imported from an external information base) or derived. Derived (view) classes are defined using class constructors, according to the five categories of semantic relationships. Derived classes may inherit properties from the fragment classes they integrate, and they may have their own local operations. Below, we have shown the category generalisation PowerPlant of OilPlant and CoalPlant using the syntax of [KDN90]:

```
DEFINE PowerPlant
    categoryGeneralizationOf:
        (OilPlant, CoalPlant);
    inheritsMethod: (powerOff);
    methods: [powerOn;
        refill (int mjoule)];

IMPLEMENTATION PowerPlant
    powerOn
        case
            OilPlant: fireOn()
            CoalPlant: start()
        esac
    refill (int mjoule)
        case
            OilPlant: fillOil (MJouleToOil*mjoule)
            CoalPlant: putCoal (MJouleToCoal*mjoule)
        esac
```

`PowerPlant` inherits the operation `powerOff` from both classes it generalises, while it defines two new operations `powerOn` and `refill`. These two operations are used to integrate `fireOn` and `start`, and `fillOil` and `putCoal`. Note the case-statement in the
implementation of these two operations, which is used to dispatch the operation request to the right class with appropriate mappings of parameters.

ViewSystem has a built-in set-oriented and object-oriented query language. It offers two ways of optimising queries: Materialisation of queries and transformation of queries into subqueries referring to external and extensional classes only.

ViewSystem represents an excellent approach to integration by views. It allows for considerable freedom by the “customisable forwarding mechanism” of operation request, as seen in the operations of PowerPlant. However, it only offers the integration possibilities provided by the built-in integration operators. The derived classes exist solely for the purpose of integration. They do not have the same behaviour as normal classes, e.g., it is not possible to create objects from derived classes. As this is not mentioned in [KDN90], we are curious whether the objects in the derived classes are created when they are queried, or if they are the same objects as the objects in the fragment classes, i.e. sharing OIDs.

Bertino et al.

[BNPS89] introduces an object-oriented interface which is used to integrate heterogeneous data repositories. The approach is based on operational mappings, which means that a client issues operation requests to “abstract” objects in the object-oriented interface level, and the system is responsible for translating these into operations in the local systems being integrated.

[Ber91c] builds on the previous work by introducing integrated views for integration of heterogeneous data repositories, which already have been homogenised by the object-oriented interfaces in [BNPS89]. The integrated views are expressed as queries towards a set of base views, representing the classes to be integrated. An integrated view consists of the following components:

- **Structural mapping**: This describes how attributes of the integrated view map to the attributes of the base views.

- **View-query**: This is expressed in an object-oriented query language, and defines the properties of the view implicitly.

- **View-operations**: (Methods in [Ber91c]). These are operations available to clients of the view. They may either be implemented in the view, or provided by behavioural mappings to the base views.

- **Behavioural mapping**: This is a mapping between operations of the integrated view and operations of the base views. A behavioural mapping has the following components: name of integrated view operation, name of local operation, mapping from result, and mapping to parameters.

- **Superviews**: A view definition may inherit properties from another (super-)view.

The objects provided through integrated views may almost be used as if they were
normal database objects, because they have only temporary OIDs, which are generated upon evaluation of the view-query.

The integrated views of Bertino are similar to the views provided by View-System, but they are not based on a predefined set of integration operators. The advantage of the integration operators of ViewSystem is that they have “integration skeletons” built-in, avoiding these to be recoded for every integration. A similar effect could probably be provided by using the superviews of [Ber91c]. However, the mappings of Bertino’s integrated views seem to be more high-level than the hand-coded mappings in the local operations of ViewSystem’s derived classes.

The COCOON project

The COCOON project [SS92] has focused on how to smoothly extend nested relational databases into object-oriented databases [SS91], and recently further into multidatabases (or multi-object bases) [SST93].

The basis for the integration of databases is an object-preserving query algebra (object algebra in their terminology) used for updatable views [SLT91]. The object algebra is set-oriented, i.e. the input and output of queries are sets of objects. By object-preserving it is meant that the output objects are the same objects as the input objects, but possibly having less or more properties. The object algebra consists of the following operators: selection, projection, extend, set operations, and generic update operations. Extend is the “opposite” operator of project, as it allows for extending objects with new properties. It can be used as a substitute for the join operator known from relational algebra, and in [SST93] it is extensively used for integration of databases.

Integration is done on two levels:

Level 1: On this level, parts of the component schemas are composed into a global schema, but no integration of objects is done. The global schema is composed by exporting and importing names of schema elements in between component databases. Classes\footnote{COCOON makes a distinction between types and classes, where a class represents the extent of a type, which itself is merely representing the intent of the objects.} from the different databases will be composed by defining some global classes. By assuming that basic classes (integer, boolean, string, ...) are common to the component databases, the extend operator allows for defining relationships among component databases by “extending” objects in views by queries to other databases.

Level 2: [SST93] adopts the notion of entity and proxy objects of [Ken91a], where proxy objects in the different component databases represent the same “real-world” entity object. On this level inter-database queries and updates are allowed by integrating proxy objects from the different component databases in global objects. A notion of global identity is introduced, which is defined as a boolean function that evaluates if two proxy objects are the “same object”. “Same” is an application-dependent operation (function) finding a corresponding object in another database. To achieve schema integration a global meta
schema is built, being a composition of the meta schemas of the component databases. Operations of the component databases are represented by meta objects, and they are integrated by defining same-operations on the meta-object level. [SST93] further argues that since all views are updatable, update propagation between different component databases comes for free.

COCOON’s views exceed the views of ViewSystem and Bertino in formality and “purity”. COCOON has an intuitive and expressive way of integrating proxy objects using normal queries. However, we think their schema integration by defining “same” functions on the meta-level seems rather complex and artificial, and not as practical as the integration done by ViewSystem and Bertino. While Bertino’s integrating views explicitly include the mappings (possibly resolving conflicts) to the base classes, schema integration in COCOON requires the meta-objects representing similar operations to be homogenised before they are integrated by “same” functions.

### 3.5.3 Generalisation by upward inheritance

Generalisation plays a central role in class integration [Ber91b], because it allows for combining existing objects and classes into a unified view. As we have seen in the previous section, generalisation is extensively used in integration by views. Generalisation also appears frequently in object-oriented design, mainly due to the human nature of identifying concrete examples before the abstractions are recognised [JF88, Ped89, WWW90].

The most comprehensive approach we have found to class generalisation is the generalisation by upward inheritance approach presented in [SN88, SN90]. The main contributions of [SN88] is that it revisits the handling of operation request (message handling) for generalisation classes, and that it provides for various kinds of generalisation dependent on how the classes to be integrated are semantically related (ViewSystem is clearly influenced by these).

To introduce the main concepts, we show how the power plant example from [SN88] is solved. Assume the OilPlant and CoalPlant classes as given in the start of this section, but with BarrelOfOil as the domain of oilFired and TonOfCoal as the domain of consumed. PowerPlant will be defined as follows:

```plaintext
CLASS PowerPlant
METACLASS: CategoryGeneralizationClass
GENERALIZATION OF: OilPlant, CoalPlant
ATTRIBUTES:
    consumed: MJoule
CORRESPONDING:
    OilPlant.oilFired
    CoalPlant.consumed
METHODS
    powerOn
```

45
CORRESPONDING:
OilPlant.fireOn
CoalPlant.start

refill (MJoule energy)
CORRESPONDING:
OilPlant.fillOil (BarrelOfOil barrels)
CoalPlant.putCoal (TonOfCoal tons)

PowerPlant is defined as a category generalisation of OilPlant and CoalPlant, having three “integrated properties”. consumed is defined as an integration of OilPlant’s oilFired and CoalPlant’s consumed. The scale difference between these two attributes is solved by defining MJoule as a generalisation of the classes BarrelOfOil and TonOfCoal of the two attributes oilFired and consumed. MJoule provides transformations between the different scales of the three attributes.

powerOn is an integration of OilPlant’s fireOn and CoalPlant’s start, which solves the name conflict between those two operations. When requesting powerOn for an object of either OilPlant or CoalPlant, powerOn will forward the request to fireOn or start, depending on which class the receiving object is a member of. PowerPlant’s refill integrates fillOil and putCoal by using the same mapping as the consumed attribute. This is the upward inheritance quality: properties requested in a generalised class, will be appropriately forwarded to the special class of the receiving object. In [SN90] a rich and complex apparatus for forwarding such requests is provided.

Different categories of generalisations are provided dependent on the semantic relationship between the classes to be generalised. Some examples of these are: do the classes model the same entity objects? Do they model different roles of the same entity objects? Do they model different history-versions of the same entity objects? Are the classes counterpart-related (modelling different entity objects having similarities)? Are they category-related (have common attributes)? These are used to make different generalisations having different characteristics and representations. For example, they will require different policies for forwarding operation requests [SN90]. The different kinds of generalisations are provided by corresponding metaclasses being tailored to the particular task, e.g., PowerPlant has CategoryGeneralizationClass as (meta-)class, while MJoule has DataConversionClasses as (meta-)class.

The upward inheritance approach seems to be a solid piece of work, and covers most of the generalisation situations that may appear. Like ViewSystem, this is perhaps its weakness as well. It provides a wide range of predefined generalisation semantics, but does not provide the underlying mechanisms in their pure form, like Bertino’s views. Furthermore, [SN88, SN90] treats generalisations in only one level. Thus, a generalised class behaves like a special class not suited for any further evolution.

3.5.4 Miscellaneous

[HD92] adopts the operational mapping approach of Bertino [BNPS89, Ber91c], but provides this differently than the view approach of Bertino. It uses an object-
orientated framework for integration of component database systems into a heterogeneous database system. A predefined set of classes are provided, which are tailored to the specific needs using subclasses redefining the operations of the predefined classes. Thus, heterogeneous database systems may be efficiently built using the powerful techniques of object-oriented programming. Similar experiences with building other types of object-oriented frameworks are reported in [WJ90].

Underlying the upward inheritance approach and ViewSystem is a work on metaclasses [KNS90, KNS89]. [KNS89] shows how metaclasses may be used in class integration by defining different metaclasses for the different class integration situations as described in [SN88] and [KDN90]. Seen in perspective, [KDN90, BNPS89, SST93] provides class integration by views, [SN88] by metaclasses, and [HD92] by object-oriented frameworks. The metaclass approach is perhaps the most tailorable and exotic, the framework approach is most appealing from an object-oriented programming point of view, while the view approaches use well-known techniques from databases.

[SSG+91] adopts the concepts of subsumption and classification from KL-ONE [BS89], and applies this for schema integration purposes in the CANDIDE semantic model. Based on these two concepts, it defines a set of operators to determine class relationships (e.g., equivalent, disjoint, is-included-in etc.). These operators are used to automatically merge schemas which are “well-defined”. Wherever there are conflicts in the merge, the user must interact.

[CHS91, FN92] represent knowledge-based approaches to integration of databases. [CHS91] uses a large knowledge-base as a help in schema integration, which unlike traditional schema integration approaches, exploits additional information, e.g., allowable operations on the schemas, comments from the schema designer, integration guidance, and organisational rules. [FN92] proposes to use fuzzy real-world knowledge as help in integration in heterogeneous databases. Classes are enriched semantically by finding the best spanning tree for the classes and their properties. The semantic resemblance of different classes are found by unifying their spanning trees. The unified tree is used as a skeleton for integration of the classes.

[Ped89] and [PW89, PW90] recognise the need for generalisation of classes in object-oriented programming. [Ped89] treats this on one level, and focuses on generalisation of (virtual and non-virtual) operations, where factorisation of common code is important. [PW89, PW90] shows the inheritance factoring process, which is a method for automatically constructing an “optimal class hierarchy” (i.e. with maximal sharing of attributes) from a set of flattened classes.

3.6 Object-level evolution

This section will step through some of the approaches to how evolution is managed at the object level.

47
3.6.1 Object versioning

A prominent characteristic of design applications is the need for versioning of objects. Since design applications have been used as one of the motivations for object-oriented databases, some of them support versioning [DGL86, K+89, ABGS91].

Versioning is typically done at the granularity of objects, where objects exist in many version “slices” having their own values for attributes. The object identification mechanisms are extended to capture for versions, typically by extending the OID with a version number for each version. In ORION [K+89] this gives rise to two ways of referencing versions: Dynamic version binding, which means that a client has a reference to the object, and the system resolves which version to bind to, typically by having a default version. Fixed version binding means that the client has a reference to a specific version of an object.

The versions of an object are often organised according to how they are created. E.g., Damokles lets versions be organised in linear revision-chains, in variants trees, or directed acyclic graphs (DAGs) [DGL86]. These structures appear by letting new versions be created based on existing versions. This organisation means that the history of creation of versions are recorded, and may be traversed by applications. This recording may be used for traceability purposes, and is the basis for 3-way (“intelligent”) merging of variants [Bra89].

Versioning of objects is a necessity in design environments, where it is used to record the history of objects, and for alternative variants of objects. Thus, any object-oriented database system claiming to support CAD or design environments, should include this feature.

3.6.2 Object conversion

By object conversion we mean the ability to let an object achieve or lose membership in classes. Consider the example in Section 3.2.1, where CSource is extended into the two subclasses CHeader and CImpl. CHeader adds one attribute, and CImpl none with respect to CSource. When creating these two subclasses, CSource may already be populated. Some of the existing objects might represent C header modules and some C implementation modules, while others might represent C modules being both. This means that some of the objects existing in CSource must be converted “down” to the subclasses. To be able to efficiently perform the conversion onto the subclasses, we need to have the extent of CSource represented.

Object-oriented databases scarcely allow for existing objects to be converted in the class hierarchy. However, even if the data model supports conversion of objects, there may still be some problems. A common implementation technique in object-oriented databases is to store a class identifier together with the OID in references [Kim90b]. This means that if an object is converted, all references to the object must be updated. To avoid this problem an object table is used. This table is indexed by the OID, and for each entry it has the following: A class identifier, physical location for the object, reference counts, access rights, locks etc. However, the object table
introduces an extra indirection level.

Object conversions may neatly be used to evolve objects along class hierarchies. But
the class hierarchy represents a limitation as well, since an object cannot become a
member of an unforeseen class not connected to the creation class.

Constraining object conversion

[Zdo90] introduces an additional semantic construct on classes to control an object’s
(foreseen) evolution path in a class hierarchy. An essential class (type in [Zdo90])
is a class that an object never can loose, the object must always be a member of
that class or a subclass. An exclusionary class is a class that an object never can
become a member of once the object has been created. These two concepts are used
to control the evolution path of objects created in specific classes. Figure 3.7 shows

![Diagram of class hierarchy]

Figure 3.7: Exclusionary and essential classes.

an example of these features. In this example an object is created in Child and this
object may evolve to be become successively a Student, a Professor, and a Retiree.
As can be seen from the example, two “artificial” classes (C1 and C2) had to be
created to manage this.

The essential and exclusionary constraints are useful for constraining evolution of
objects in a class hierarchy, but it can only model limited kinds of anticipated evo-
lutions.

[Su91] describes how we can control the conversion (migration in [Su91]) of objects
being members of multiple classes (viewed as roles). A role set is a set of classes
where an object can reside simultaneously, an evolution pattern is a sequence of role
sets, and an evolution inventory is a set of evolution patterns. Evolution inventories
are used as dynamic constraints on where objects may evolve in the class hierarchy.
Three different update languages are studied for soundness (if the update language
generates only patterns in an inventory) and completeness (if the update language generates all evolution patterns in the inventory).

[Su91] shows an interesting approach to how to constrain the roles that an object may achieve in a class hierarchy, but in case of evolution of inventories and class hierarchies, several new problems are introduced. Upon adding or changing a class, we may possibly update all evolution patterns being external to the class hierarchy. It may happen that even the sequence in all patterns have to be reordered when doing small changes. The main problem with this is that the evolution patterns are external to the class hierarchy, i.e. any set of unrelated classes may be connected in an evolution pattern, making it unmodular. Thus, this approach seems good for object evolution in a static class hierarchy, but is a bit unclear if the class hierarchy itself evolves.

[Ara90] presents a similar approach to control the evolution of objects, but by means of constraints specified in propositional temporal logic. Temporal constraints are associated with classes (roles in [Ara90]), and describe the set of legal sequences of state changes of objects, as well as the set of legal sequences of operation and attribute requests which can be sent to an object of the class. A context defines the set of classes which an object may acquire through its lifetime, and a set of temporal constraints on legal sequences of object conversions, temporal relationships between operation requests and receiving classes, and legal sequences of operation requests to objects being member of different classes within the same context. [Ara90] gives an algorithm for monitoring the evolution of objects within a context. This algorithm assures that objects do not violate the context’s specification.

In this approach an object is allowed to be converted between classes, where the evolution is constrained by temporal predicates. This gives strictly anticipated evolution, and like [Su91], it does not consider how the object evolution works when the set of classes and their interrelationships evolve.

### 3.6.3 Multiply classified objects

In the previous section objects achieved and lost membership in classes as a part of object evolution. To freely do this, it is assumed that objects may be members of multiple classes. In traditional object-oriented data models, an object cannot have more than one most specific class. This means that an object will be a member of its most specific class, and all the superclasses (transitively) of this. However, some objects tend to play many “roles” in a system, possibly with some kind of organisation between the roles. We have observed two typical examples of role-playing objects. First, long-lived objects, like representations of people and organisations, tend to evolve and achieve many roles as time goes by. Second, we have objects carrying “long” information not interpreted directly by the database, i.e. objects with long fields. These long fields tend to be interpreted in various ways, which give rise to multiple classifications. Imagine objects representing C modules in the example in Section 3.2.1, which are both C headers and implementations. If we had operations in the CHeader and CImpl, which were specific for analysing the long field contents in their own ways, it would be wise to let these objects be members of all three classes.
An example of such an operation could be an operation of \texttt{CHeader}, which produced an interface description of the header in a standard format.

A solution to this problem could be to use multiple inheritance, and define a class being a subclass of both \texttt{CHeader} and \texttt{CImpl}. The problem, as pointed out in \cite{Ste91}, comes when we have several possible roles an object may play. This means that we would have to create a separate subclass for every cross-product of the "role" classes, i.e. creating a forest of classes just for allowing multiple most specific classes. This modelling may also be overkill, because often an object is used only in one of the most specific classes at a time.

Allowing for multiple most specific classes may seem like a simple extension to object-oriented data models. However, multiple most specific classes introduce ambiguities for dynamic binding between property requests and properties. In traditional object-oriented data models, the search for the requested property starts in the most specific class of the receiving object, and proceeds in the superclasses. When requesting a property for an object being member of many classes having this property, it is not clear which property to bind to.

\cite{RS91} allows for both object conversions and multiple classified objects, by introducing \textit{aspects}. An aspect is an extension of a class, onto which attributes and operations can be added, and on which it can be decided how much of the original class that "shines through". This may be used to extend existing objects with new state and behaviour, while preserving the OIDs.

Even if an object with multiple "aspects" has one OID, all references to the object are qualified to specific classes. This allows an object to have multiple instances of the same "aspect class". Aspects are much like multiple objects having shared parts, but aspects may share the "overall" OID. This allows for an object to be treated as a whole (with all its aspects) in some contexts, e.g., for copying and locking.

\cite{Sci89} shows a similar approach, but here an extension of an object is another object. This gives less opportunities to view an object as a whole. Clovers \cite{Ste88, SZ89} resembles the Aspect approach, by allowing an object to be member of several leaves in a class hierarchy, thus looking like a clover. References to a clover object may be converted to different parts of the clover by using coercion operators. In Clovers a leaf class must be a subtype of the original class, while an aspect class is an arbitrary extension.

### 3.6.4 Class learning

Another category of object evolution problems is adoption of external objects into a database. This problem may appear when objects live longer than the environment they are contained within, or when objects migrate between databases having different schemas.

This problem may be formulated as learning classes from examples. \cite{BL91, LBS91} shows such an approach, where classes are constructed solely based on object examples. \cite{BL91, LBS91} argues that this is especially useful in the design phase, because
humans are better at recognising object examples than classes. They show a series of algorithms for the following class learning problems:

1. Minimum class hierarchy learning gives a class hierarchy with a minimum number of edges.

2. Class hierarchy learning gives an arbitrary class hierarchy that fits the objects (do not have to be minimal).

3. Common normal form class hierarchy learning gives a class hierarchy with maximum factorisation of attributes.

4. Consolidation of alternative class hierarchies being learned.

5. Class hierarchy minimisation, i.e. with a minimal number of edges.

6. Object example generation, i.e. finding a set of object examples from which a particular class hierarchy can be reconstructed.

The approach is interesting, at least for "design by example". Class learning may be used to consider existing class hierarchies by generating object examples from the existing classes, and use these examples as the first objects to be used in the learning. Our main objection to the approach is that it is based on the Demeter data model, which is too simple: Only leaf classes are allowed to be populated, and properties are restricted to be structural.

3.7 Concluding the survey

This chapter has presented various constructs used for evolution and integration of classes in object-oriented databases. To conclude this survey we will now make a small summary of these constructs where we will focus on the following characteristics:

- Is the construct aimed at handling anticipated or unanticipated evolution?

- Is the construct destructive (modifying classes) or additive (leaves existing classes unmodified)?

- What is the set relationship between the extents of new and old classes?

- Is the scope of the construct the entire class hierarchy, or only local to one existing class?

We have extracted the following seven constructs for evaluation:

Subclassing is an additive evolution (and modelling) construct mainly aimed at anticipated evolution, but it may also handle unanticipated evolution by severe redefinition of properties in the subclass with respect to the superclass. The scope of the evolution is local. The extent of the subclass is usually a subset of the extent of the superclass.
Parameterised classes are the archetypal example of anticipated evolution. It is additive and local. The parameterised class has no clear concept of extent.

Class modification is a destructive evolution construct aimed at unanticipated situations. This is a local evolution, where the extent of the modified class is retained, i.e. the extent of the class prior to the modification is equal to the extent of the class after the modification.

Class reorganisation is a destructive evolution technique for unanticipated evolution. It is global to a class hierarchy, but it does not consider how to retain the extents of the modified classes.

Class versioning is a local, additive evolution technique for unanticipated situations. The extents of the different versions of the class are equal.

Generalisation is an additive evolution construct, where unanticipated classes are defined on top of existing classes. A generalisation class has an extent which is the union of the extents of the existing classes.

Views are additive and aimed at unanticipated evolution by freely combining information from existing classes and other views. This may be seen as a global evolution, where the new set of objects are subsets (possibly combinations) of the existing objects.

These constructs have many similarities, yet still, they are different. If we consider a class evolution as a process of creating a new class based on one or several existing classes, we can better compare them. Subclassing, class modification, class versioning, generalisation, and views may be seen as a filter to or modification of existing classes. The ways of expressing the “modification” of the properties are different. E.g., in most object models a superclass shares its attributes with the subclasses, while a generalised class typically forwards its incoming requests to the appropriate subclasses. Additionally, the set relationships between the extents of the classes are different in the various techniques.

These constructs cover for many of the evolution situations possible in an object-oriented data model. To support these kinds of evolution, a system must include all of these constructs. It is our belief that having a high number of partly overlapping, but yet different constructs, has disadvantages. It means many constructs to learn for the users, both for creation and understanding. It also means many constructs to implement for the developers of the database system.

The main goal of the framework for evolution that is going to be presented in the next chapter, is to unify many of these techniques. The framework is based on recognising the similarities between the set relationships for the extents in the evolution, and the similarities between the different ways properties of the classes are “modified” to form a new class.
Chapter 4

Framework for Class Evolution

This chapter will present a framework for class evolution, i.e. a set of constructs being used to evolve classes in various ways. These constructs will be used to replace the constructs from many of the evolution techniques surveyed in Chapter 3. Chapter 5 applies the framework to class derivation, mainly being small adaptations of existing class hierarchies. Chapter 6 shows how the framework is used in class integration, where we will concentrate on integration of classes being developed and populated independently.

This chapter starts by introducing a basic object model, which includes classes to describe objects and their properties. In Section 4.2 we introduce intent hierarchies and extent graphs, which are the organisation of the intensional and extensional dimensions of classes in two separate hierarchies. In connection with the extent graph we introduce a construct named $\exists$ function, which is used to relate corresponding objects in different extents. In Section 4.3 and 4.4 we show how we specify and maintain the consistency of corresponding objects and objects being members of multiple classes. In Section 4.5 we define class evolution in the context of the framework. We also introduce class evolution transparency, a basic quality of our class evolution, which generalises the concept of class change transparency [ABB+83]. Section 4.6 makes some general remarks to the framework, and compares it to other work done on object models. The comparison of the framework’s evolution capabilities to existing class evolution constructs is delayed until we have shown how we may cover for them, i.e. at the end of Chapter 5 and 6.

The definitions of the essential components of the framework will be emphasised by being put into special Definition paragraphs ended with a $\Box$. Parts of this chapter are published in [Bra92] and [Bra93].

4.1 Basic object model

We will now present the basic object model, which may be characterised as an object-oriented data model without subclassing. The basic object model will be used as an example model for the framework. It is meant as a “flattened” representative for the object models found in many object-oriented databases, like $O_2$ [D+91], ObjectStore
[
LLOW91], Orion [K+89], and many others. The “flattening” is in the class hierarchy
dimension, i.e. classes do not have subclasses or superclasses. However, the basic
object model allows for composite objects and for links between objects. In the
successive sections we will introduce the other components of the framework, which
“unflatten” the class structure by interrelating classes and their objects. The basic
object model serves also as an example of the object-oriented database constructs
briefly surveyed in Section 2.1. Although federated database systems is not a central
topic in this thesis, we do not deny that this basic model together with the rest of
the components of the framework, may be a suitable canonical model [SCG91] (this
is treated further in Section 6.7.1).

4.1.1 Objects, intents, extents, and classes

An intent is a named set of properties. It is used to describe the implementation and
the interface of objects, e.g., it is used to create and interpret objects. The intent is
used to describe the common “form” and “behaviour” shared by the objects created
from that intent. An object created from an intent “is of that intent”.

An object possesses a system-assigned object identifier (OID), which is used to repre-
sent the identity of the object. An object also possesses the properties described
by the intent: The attributes of the object represent the state, while the operations
represent the behaviour. Within a database all objects have distinct OIDs, thus an
OID is unique. If two objects possess the same OID, they are identical. When an
object is created, it is automatically given an OID. For illustrative purposes we will
use the notation oN for the identity of an object, where N is a natural number.

An extent is a set of objects. To every intent we will connect an extent with the
same name, and being the set of objects created from the intent. An object created
from the intent will be a member of the extent. We assume a one-to-one and onto
function from extents to intents. We call an intent–extent pair a class. Due to
this correspondence, we will use class when we do not address the extensional or
intensional quality specifically. Thus, the term “member of a class” is often used. The
main reason for us to separate the intensional and extensional dimensions of classes,
is that we organise these constructs into separate hierarchies (see Section 4.2). A
class that has objects, is said to be populated.

The extent of a class is automatically maintained and is the basis for making queries.
The maintenance of extents is considered important, because they constitute relevant
groups of objects. As will be further treated in Section 4.2, an object is created in
one class, but may be (manually or automatically) added to other classes. Adding
an object to a class is similar to creating an object in that class, but adding does not result in creating new OIDs. Automatic adding of objects to classes will be an
important mechanism in our class evolution framework.

A client of a class C is another class having objects that possibly make requests to
(i.e. use) objects of class C. We will also use client for any application which is
requesting objects of the class.
4.1.2 Properties: attributes and operations

An intent will have a set of properties. A property is either an attribute or an operation. An attribute is a named description of the state of an object. The state of an attribute is a value of a specific attribute domain. All attributes are “typed”, in the sense that they can only hold values of the associated attribute domain. An intent defines which attributes each object created from that intent should possess, but each object will have their own values for the attributes. All attributes in an intent must have unique names. An attribute may have a default value, i.e. a value it will automatically achieve if no other value is specified upon creation and addition of the object to a class.

We divide attribute domains into basic and composite. Basic attribute domains are predefined, like char, int, float, links, and user-defined enumerations. To connect objects we use links. For every class, there is a link attribute domain. This is described by the link keyword and the name of the class. This means that a link attribute is “typed” to reference objects of that class. Link attributes take values which, like other values, belong to the state of objects. In the basic object model, a link value is an OID. For the purpose of dynamic binding we will in Section 5.6.2 extend a link value to also include the identifier of an intent.

A link is one-way only, meaning that with a link it is only possible to go from the object having the link attribute to the connected object. However, it is possible to constrain a pair of links to be inverse, meaning that the inverse connection is always maintained, like in Vbase [AH87b] and ObjectStore [LLOW91]. Our intention with supporting unidirectional links instead of symmetric, bidirectional relationships, is to have a more light-weight construct. By supporting the inverse link construct, we have a simple substitute for binary relationships. The framework may, as argued for in Section 4.6.4, be seen as an alternative to “semantic-rich” explicit relations [DG91].

Containment links are special link attributes having domains described by the cont-Link keyword and the name of a class. Containment links are special links for defining composite (or complex) objects. An object having a containment link will “contain” the object referenced by the link. A composite object is an object composed of part objects, which are ordinary objects themselves. The reason for a separate link construct for composite objects is that we would like to express special semantics as done for composite objects in Orion [KBC+87, KBG89]. This includes rules for existence dependency, copying, deletion, caching, and locking. We will not treat these issues further here, but we refer to [Bra91, BO92] for further discussions. Containment links are clearly unidirectional in their semantics: one object contains another object. It is possible to have recursive references with both ordinary and containment links, meaning that a link attribute definition may be defined to the same class that contains it. For containment links we add the constraint that an object cannot contain itself, neither directly nor indirectly. Thus, we allow class-level recursive containment, but not object-level.

To make composite attributes we use attribute domain constructors: record, setOf, and listOf. record is used for aggregating attributes, setOf is used for defining sets of
attribute values, while listOf is a constructor for lists of values.

We have pointed out that attributes are values and not objects. Values are immutable and have no separate identification from their values, while objects are mutable and have separate OIDs for identification. E.g., it does not make sense to “update” the value 4 to 5, while it is ok to update the age attribute of a Person object from 4 to 5. Other differences are that values only have predefined behaviour and no encapsulation, i.e. an attribute may be updated to any value of its domain.

We will introduce null values as special attribute values. Null values are typically used for denoting incomplete information [Gra91], and have been given multiple meanings in literature. Our use of null values will be in the context of non-derivable dependencies between attributes. In this context, a null value will mean that the value of the attribute is not known, but there exist a set of possible values. See Section 4.3.5 and 7.3.2 for more on that.

A class may have operations, which means that all objects of that class will share these operations. An operation is a description of behaviour. An operation is requested as a property of an object (of the class it is defined within), and it is executed in the context of that object. The operation may request (read or update) the state of the object and may request other operations belonging to the object; it may also request properties belonging to other objects. When requesting a property of an object referenced by a link attribute value, the binding from the operation request is to an operation of the class in the link attribute definition. This is static binding, but in Section 4.2.4 and 5.6.2 we show possible extensions to dynamic binding.

An operation may be a mutator, which changes the state of an object. An operation may also be an observer\(^1\), which reports on the state of an object. We will also speak of mutator requests (updates) and observer requests (reads) to attributes. A basic assumption is that every attribute may be exposed for mutator and observer requests. We do not introduce constant attributes here, because they are not particularly interesting to class evolution and consistency maintenance. Later in this thesis we assume that attributes are requested through attribute request operations, which are primitive operations for requesting attributes. E.g., the attribute a will be requested through an observer operation, a(), and a mutator operation, a(new_a).

An operation will have a signature, which is the name of the operation, the domains of the parameters, and the domain of the return value. Two signatures are equal if they have equal names and equal lists of parameter domains and equal result domain. All operations of an intent must have distinct signatures. This constraint is used to have one matching operation for an operation request. An operation has an implementation, which is executed upon receipt of a request for the operation. We do not define a language for the implementation of operations, but we assume that it allows for requests to properties of given objects and for queries for objects in given extents. For the prototype described in Section 7.2, Prolog is used for implementing operations.

We make no assumption regarding whether classes are represented as objects or not, but we will assume that for every class there exist predefined operations for creating

\(^1\) The mutator and observer terminology is borrowed from [LG86].
objects in the class and for adding objects to the class. Create makes new objects from the corresponding intent, it lets the new object achieve initial values, and it gets a system-defined OID for the object. Add does the same, but instead of creating a new OID, it will create an object with a given, existing OID.

Every property of an intent will either be public or private. A public property is requestable (accessible) from all objects, while a private property is only requestable from the object itself. These keywords are intended to have similar meanings as the ones in C++ [Str91], but our private is somewhere in between C++’s private and protected, as will be seen in Section 5.7.4. We use visibility mode as a common designation for private and public.

4.1.3 Example

In Figure 4.1 we show a simple example described in this object model. The syntax is similar to the syntax of C++ class definitions [Str91]. lineDmn is an example of a user-defined enumeration domain. An attribute of this domain will have one of the three values. date is an example of a record definition. Note that date could have been defined as a class, but in this case every date would be a separate object and had to be explicitly created.

We show two class definitions, PolyLine and Point. PolyLine defines a public operation to add a new point to the polyline, and a public operation to draw the polyline. PolyLine defines four private attributes. The points attribute is a list of containment links to objects of class Point. The closed attribute is given the default value false. Point defines two public attributes: x and y.

4.1.4 Rationale of the basic object model

An intention with the basic object model is to be a representative for the object models of object-oriented databases. There have been many different views of what constitutes an object-oriented database, and with respect to this we have tried to make a compromise. Our object model resembles the basics of a C++-based object-oriented database, but we have made some extensions by introducing some new attribute domain constructors, containment links, and extents.

Another intention with our basic object model is to follow the “Scandinavian” approach to object-orientation [Mø91]. The main idea here is that objects are intended to represent application specific phenomena and that classes of objects are intended to represent the corresponding application specific concepts [Mø91]. This choice has resulted in allowing for non-encapsulated objects and for a rich set of attribute domain constructors.

The choice of allowing all kinds of properties to be private or public is intentional. We think attributes and operations may equally well be parts of the interface of an object, as they may implementational details. By this we allow for traditional structural databases, purely encapsulated objects, and combinations of these. En-
domain lineDmn enum {dotted, dashed, filled};
domain date record {int day; month; year;};

class PolyLine {
  public:
    addPoint (link <Point>);
    draw ();
  private:
    listOf <contLink <Point>> points;
    boolean closed = false;
    lineDmn lined;
    date creDate;
};

class Point {
  public:
    float x,y;
};

Figure 4.1: The polyline/point example.

capsulation provides for some of the power of object-oriented computing, but many database application areas are such that encapsulation is strictly not necessary. With respect to query processing, the database community has not yet agreed upon how to manage encapsulation. Thus, there are good reasons for providing for both possibilities.

Unlike many object models, which regard all “entities” as objects, our model has support for values by a rich set of attribute domain constructors. Both values and objects are useful, and the difference is worth keeping [Mac82]. When doing integration of independently developed classes, this is particularly useful, because the values in an attribute domain are easier to predict than the objects in the extent of a class. Slightly paraphrased: Attribute domains are a way of limiting the set of possible “entities”. The clue with respect to integration is that the process of defining relationships between corresponding classes becomes easier. To promote sharing of attribute domains, they are defined outside classes.

4.2 Extent graphs and intent hierarchies

In this section we will “unflatten” the class structure by letting intents and extents be organised in different hierarchies. In Section 4.5 we will argue for this separation by showing that the two hierarchies have different qualities with respect to evolution.
4.2.1 Intent inheritance

Intents may be organised in inheritance hierarchies, named *intent hierarchies*. Here, an intent can be defined to inherit properties through *inheritance links* from several other intents, i.e. multiple inheritance is supported.

**Definition 4.1 (Inheritance link)** An intent $I_2$ may inherit a set of properties $P$ from an intent $I_1$, by an *inheritance link* from $I_2$ to $I_1$. The semantics of the inheritance link is *copying*, i.e. intent $I_2$ behaves the same whether $P$ is defined locally or inherited from another intent. □

The term “set of properties $P$” means that an intent may inherit all or some of the properties from another intent. “An intent behaves the same” means that clients will not notice the difference.

**Definition 4.2 (Intent hierarchy)** An *intent hierarchy* is a directed acyclic graph $D = (V, E)$ such that

1. $V$ is a set of intents.
2. $E$ is a set of inheritance links.

The graph may be connected or unconnected. □

The copy semantics has severe consequences for property binding. Properties being requested by an inherited operation are the properties of *this* intent (illustrated in Figure 4.2, and explained below). As the intent hierarchy is a DAG, we allow for multiple inheritance. Due to the copy semantics of inheritance, name conflicts are not allowed to appear. In case of name conflicts, one of the properties must be chosen at the time of the definition of the intent.

Each intent has a set of properties, where each property is either inherited or locally defined. An object created from one intent has the properties defined in that intent and the inherited ones. There is no encapsulation of inherited properties, meaning all inherited properties are visible to the operations of this intent. Inherited properties retain their visibility mode, unless the visibility mode is redefined in the inheriting intent. E.g., a private property may be inherited and made public. This definition of intent inheritance gives the implementor of the intent full control of the bindings, because inherited and locally defined properties behave the same. Intent hierarchies are meant as a *reuse mechanism*, and thus we mean that “it is not the business of a class to decide how it will be extended in the future” (slightly paraphrased from [Mey88]). The question of encapsulation will also be applicable to the organisation of extents in graphs, and this will be discussed in Section 5.7.4.

Consider the example in Figure 4.2. Class C2 inherits operation op1 from C1 (the C1::op1 statement in the heading of C2). When op1 is executed in the context of an object of class C2, it is the local a1 definition that is referenced.
class C1 {
    public:
        op1 () {
            a1 = ...;
        }
    private:
        dom1 a1;
};

class C2 : C1::op1 {
    private:
        dom1 a1;
};

Figure 4.2: Example of copy semantics of inheritance.

The copy semantics of inheritance covers somewhat the use dynamic self binding in traditional inheritance. If the a1 attributes in Figure 4.2 were operations, the request to a1 in op1 would be similar to "self a1" in Smalltalk-80 [GR83]. If "self" was an object of intent C2, the a1 of C2 would be used. If "self" was an object of intent C1, the a1 of C1 would be used. This is a simple result of the copy semantics of our inheritance, while Smalltalk-80's dynamic binding is a result of search at runtime. A difference is that the copy semantics holds for both operations and attributes, while Smalltalk's dynamic binding is only applicable to operations (methods). The introduction of copy semantics for inheritance may seem strange. However, it is a consequence of separating the intent and extent of classes into separate hierarchies, and the wish to give more control of the binding to the schema designer.

The main reason for the introduction of intent inheritance is to have it as a reuse mechanism, since an intent definition may reuse the definition of another intent. The choice to give the schema designer full control of the bindings and visibility modes further underlines this. Another reason is that the intent hierarchy may coincide with a conceptual, design hierarchy. If this is true, we believe that the local properties in an intent constitute a unit of abstraction. They are not an accidentally composed set of properties. E.g., a Student intent may add the tightly related properties studentID and courses to the properties inherited from the intent Person.

4.2.2 Extent graphs

Recapitulating from Section 4.1: a class consists of an intent and an extent. For simplicity we will assume a one-to-one and onto function between intents and extents. This may be relaxed without any large impact on the rest of the thesis. An extent is a named set of objects, which may be retrieved based on this name. All objects being created from an intent will be a member of the corresponding extent, unless
they have been explicitly deleted.

Until now, we have said that an object will be a member of the class that it is created in, and that it may be added to other classes. We will now introduce a mechanism to let an object automatically be added to classes.

**Definition 4.3 (Extent propagation link (EPL))** An extent propagation link (EPL) is a directed edge from an extent $C_2$ to another extent $C_1$. This will be written as $C_2 \Rightarrow C_1$. The extent propagation link means that for an object which is a member of $C_2$, there is a corresponding member in $C_1$. □

The notation $C_2 \Leftrightarrow C_1$ is a shorthand notation for $C_2 \Rightarrow C_1$ and $C_2 \Leftarrow C_1$. In most practical situations we will treat this as one bidirectional link.

In most cases we will understand the EPL $C_2 \Rightarrow C_1$ as “an object being a member of $C_2$ will also be a member of $C_1.”$ In other situations “has a corresponding object in” is more appropriate. In the former situation, an EPL may be understood as a flow of copies of OIDs between extents. E.g., given an object with OID o101 being a member of $C_2$, $C_2 \Rightarrow C_1$ means that the same OID o101 will also exist in $C_1$. In the latter situation, an EPL will be understood as a set of pairs of OIDs, which is used to relate objects which are “semantically overlapping”.

An EPL has two important semantics. First, it ensures that an object has a corresponding object in another class. This is the extent propagation quality of an EPL. Related to the terminology introduced in Section 4.1, $C_2 \Rightarrow C_1$ means that creating or adding an object $O$ in $C_2$ implies adding $O$ to $C_1$, or it ensures that $O$ has a corresponding object in $C_1$. If $O$ already has a corresponding member of $C_1$, it is not made duplicate member in $C_1$. The second quality of EPLs becomes apparent after introducing object consistency in Section 4.4, where we will see that EPLs are used to relate objects for the purpose of maintaining the consistency of their “overlapping properties”.

An EPL may be understood and used in two different ways. Firstly, it may be used as an IS-A relationship in semantic data models [HK87]. Secondly, an EPL may be understood as a “mechanical” relationship to relate objects where a “consistency relation” exist in between them. In Section 4.5.4 we will see two different evolution situations, class generalisation and class versioning, which belong to these two categories, respectively.

We will now introduce extent graphs to describe the total structure of EPLs.

**Definition 4.4 (Extent graph)** An extent graph is a directed graph $D = (V, E)$ such that

1. $V$ is a set of extents.
2. $E$ is a set of extent propagation links. □

We use the term class hierarchy when referring to an extent graph together with its intent hierarchy.
An extent graph may be connected or unconnected. If there is a need to easily retrieve all objects, we may create an extent (with name “thing”, “object”, “root” etc.) which all extents have an extent propagation link to (either directly or indirectly). Then, the graph will be connected. Note that an extent graph may have cycles, which will be necessary when defining class versions.

For a given class \( C \), we will define the extent propagation set of \( C \), \( EPS(C) \), to be the set of extents reachable by transitively following the EPLs from \( C \) (the class \( C \) will not be a member of this set itself). An object in class \( C \) will be propagated to or have corresponding objects in all classes in \( EPS(C) \).

In Figure 4.3 we see an example of an extent graph with its corresponding intent hierarchy. There are four classes Person, FacultyStudent, UniversityStudent, and Teacher. An object created in FacultyStudent will be propagated to UniversityStudent and Person, while an object created in Teacher will be propagated to Person. \( EPS(\text{FacultyStudent}) \) is \{ UniversityStudent, Person \}, and \( EPS(\text{Teacher}) \) is \{ Person \}. As we see in the figure, we have made use of cycles in the extent graph. The cycle between FacultyStudent and UniversityStudent does not indicate wrong modelling, but it is a way of letting the two classes share extents. The intent hierarchy has appeared as follows: FacultyStudent and Teacher are defined from scratch (without the use of inheritance), while the intents UniversityStudent and Person inherit properties from FacultyStudent.

The framework defines the following existential operations to manipulate objects with respect to extent graphs:

\textbf{create}: This operation creates an object in a class \( C \) and will propagate the new object to all classes in \( EPS(C) \), or ensure that the new object has corresponding
objects in all classes in \( EPS(C) \).

**delete:** This operation deletes an object and its corresponding objects from the entire extent graph.

**add:** This operation allows an object to be added to (become a member of) a given class \( C \). We restrict this to classes in the same connected extent graph as the creation class of the object. The object will be added to all classes in \( EPS(C) \) (or it will be ensured that it has corresponding objects in these classes).

**remove:** This operation removes an object from a given class \( C \). The object (or its corresponding objects) will also be removed from all classes having \( C \) in their EPSs.

An object may be a member of all classes of a connected subgraph of an extent graph. We will not allow to add an object to a class that is not in the connected subgraph the object was created from. The rationale for this is that all classes that an object can be a member of are connected by EPLs, either directly or transitively. Note that given two classes connected by an EPL, \( C_2 \Rightarrow C_1 \), and an object \( O \) being a member of \( C_1 \), we allow for adding \( O \) to \( C_2 \). But this has to be done by requesting the *add operation* explicitly, in contrast to the automatic propagation done by EPLs.

In Figure 4.3 it may look like there are some superfluous EPLs. Objects in Faculty-Student are propagated to Person both directly and through UniversityStudent. Regarding the flow of OIDs, these multiple extent propagation paths are superfluous. But it may happen that the “consistency relations” are different between those two paths, such that an object in Person is dependent on an object being members of both FacultyStudent and UniversityStudent. An object in FacultyStudent may have the same OID as an object in UniversityStudent, but may have different properties and values.

We think the extent graph is a better conceptual organisation of objects than the intent hierarchy, because it tells which classes describe aspects of the “same” set of objects. Thus there is a conceptual link between classes connected by EPLs, while classes might be accidentally connected by inheritance links. Thus, for “conceptual” class browsing, the extent graph will be used. Browsing the intent hierarchy may also be useful for developers. Similar arguments are used by others for separating specification and implementation hierarchies (further discussions of separating hierarchies are found in Section 4.5 and 4.6.2).

### 4.2.3 Object correspondence

In Section 4.2.2 we read an EPL \( C_1 \Leftarrow C_2 \) in two different ways. Firstly, “every object that is a member of \( C_2 \), will also be a member of \( C_1 \)”, where the correspondence between objects in different extents connected by EPLs is described operationally: creating or adding an object in \( C_2 \) implies adding the same object to \( C_1 \). This means that the correspondence between the OIDs of objects in \( C_2 \) and \( C_1 \) is given automatically, i.e. the object will have the same system-assigned OID in both classes.
Secondly, if the classes \( C_1 \) and \( C_2 \) are connected by \( C_1 \Leftrightarrow C_2 \) after both classes have been populated, we will read the EPL differently. In this case, there may be objects in \( C_2 \) which already “are present” in \( C_1 \). This means that there are different objects which are in “semantic overlap”, i.e. they model the same “real-world” (entity) object. In these situations we will use the other way of reading the EPL: “every object that is member of \( C_2 \), will have a corresponding object in \( C_1 \).” The process of finding and connecting corresponding objects is called object integration, which is the topic of Section 6.3.

This means that the framework requires a way of relating the OIDs of objects in extents connected by EPLs. For each EPL, we will require a function \( \mathcal{S} \) to exist, which relates the OIDs of the objects being members of the two extents.

**Definition 4.5 (\( \mathcal{S} \) function)** Let \( C_1 \) and \( C_2 \) be two classes and \( \text{id}_1 \) and \( \text{id}_2 \) be the sets of OIDs in the extents of \( C_1 \) and \( C_2 \), respectively. We define \( \mathcal{S} \) to be a function

\[
\mathcal{S} : \text{id}_1 \rightarrow \text{id}_2
\]

We will name this function the \( \mathcal{S} \) (im) function. For the different situations of connecting \( C_1 \) and \( C_2 \), the following symbols and constraints will be given to the \( \mathcal{S} \) function:

<table>
<thead>
<tr>
<th>EPL</th>
<th>( \mathcal{S} ) Symbol</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( C_1 \Leftrightarrow C_2 )</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
<td>Total bijection (one-to-one and onto)</td>
</tr>
<tr>
<td>2. ( C_1 \leftarrow C_2 )</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
<td>Partial bijection (one-to-one and onto)</td>
</tr>
<tr>
<td>3. ( C_1 \rightarrow C_2 )</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
<td>Total injection (one-to-one)</td>
</tr>
<tr>
<td>4. ( C_1 \Rightarrow C_2 )</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
<td>Total function (one-to-one not required)</td>
</tr>
</tbody>
</table>

If no \( \mathcal{S} \) symbol is explicitly provided for an EPL specification, the \( \mathcal{S} \) function is assumed to be one-to-one and to be implicitly given by equal OIDs. I.e. depending on the EPL, one of the three situations 1, 2, or 3 will be assumed.

Situations 1, 2, and 3 are the “classical” object correspondences. These are illustrated in Figures 4.4, 4.5 and 4.6, respectively. We have illustrated classes as shaded ellipses, objects are shown below their classes as filled ellipses, and object correspondences given by broken lines. The definition essentially says that given an EPL \( C_1 \Leftrightarrow C_2 \), all objects in \( C_2 \) must have a corresponding object in \( C_1 \), but only a subset of the objects in \( C_1 \) have a corresponding object in \( C_2 \). When combining \( C_1 \Rightarrow C_2 \) and \( C_1 \Leftrightarrow C_2 \), it means that every object in \( C_1 \) must have a corresponding object in \( C_2 \), and vice versa.

Situation 4 appears in integration situations where a group of objects in \( C_1 \) corresponds to one object in \( C_2 \). Here, \( \mathcal{S} \) is relaxed to a general function not required to be one-to-one. It means that there may be \( N \) objects in \( C_1 \) per corresponding object in \( C_2 \). In this situation an EPL \( C_1 \Rightarrow C_2 \) may be read as “every object in \( C_1 \) has a corresponding object in \( C_2 \).” This situation is illustrated in Figure 4.7.

66
4.2.4 Binding of operations

One consequence of our extent graphs is that an object may be a member of many “most specific classes”, i.e. classes which correspond to leaf nodes in the extent graph. E.g., an object may be a member of both FacultyStudent and UniversityStudent in Figure 4.3. When assuming dynamic binding between an operation request and an operation, using the normal inheritance hierarchy search mechanism will not be sufficient. Since the object is member of multiple classes, of which several are “most specific”, it is not clear where to start the search. One way of solving this problem would be to disallow dynamic binding, restricting all requests to be done statically to a specific class. This would steal one of the greatest strengths of object-oriented programming, so we propose the following extensions:

Explicit qualification: Wherever there are binding conflicts, the user has to qualify explicitly. The problem of binding to multiple most specific classes is essentially equivalent to the problem of binding in multiple inheritance [Ste91]. Instead of letting the system guess what the client wants, the client must tell it himself, whenever there is a conflict. This is the solution taken in C++ [Str91] upon name conflicts in multiple inheritance. E.g., consider an object which is
a member of both $C_1$ and $C_2$, and which is referenced by $\text{Obj}$. If both classes have an operation with signature $\text{op}$, the following request is an example of explicit qualification to the $C_1$ class (illustrated in C++ syntax):

\[
\text{Obj} \rightarrow \text{C1::op};
\]

**Usage context:** This is partly motivated from the versioning mechanism of the EPOS database [Lie90], where we may restrict each transaction working against the database to have only a subset of the classes visible. As class versioning evolves a class “sideways”, it may result in multiple “most specific” classes. This solution is different from the previous, since it lets bindings be resolved by the system and appear unambiguous to clients, while the previous solution “passed the buck” back to clients. This solution will be further treated in Section 5.6.3.
Binding for EPL-connected classes residing in different component databases may be seen as a special case of usage context. In this case the binding is dependent on which schema the application accesses. E.g., given the EPL University → Student, the binding is dependent on if the application is connected to the component database for the faculty or to the component database for the university.

**Tagged link values:** This solution is motivated by role models à la Aspects [RS91], where each class that an object is a member of corresponds to another role of the object. By tagging each link value by an identification of the class where it is was retrieved from, we will get another form of dynamic binding. This approach will be pursued in Section 5.6.2.

**Multiple operation requests:** We may let the operation request be bound to all matching operations in the object. The motivation for this is from the semantics of updates of union views in [SLT91]. Given a class LegalEntity which is a union view of Company and Person. If an object is created in LegalEntity, it will be created in both “base” classes Company and Person. This situation resembles one where a generalised class represents an “overlapping union” of several other classes. In our framework we will support such multiple operation requests by operation consistency relations, which are introduced in Section 4.4.4. Here, the request of an operation in LegalEntity may trigger corresponding operations for the same object in Company and Person (though, this example is a bit obscure).

Note that these are not alternative techniques, but may be used within the same object model. We will not advocate any of these possible solution, but leave this open to the particular object model and direction of usage of the framework. However, Section 5.6.2 and 5.6.3 will elaborate on the tagged link values approach and on an example of the usage context approach, respectively.

### 4.3 Specifying object consistency

Extent propagation links ensure that objects may be members of several classes, or that an object has corresponding objects in other classes. The properties associated with an object or corresponding objects in two classes connected by an extent propagation link, are often partly “overlapping”. Informally, object consistency means that the classes agree upon the overlapping properties. We will now show how we specify the “agreement” between two classes that are connected by an EPL. This is done by attribute consistency relations, which rely on mathematical functions to specify dependencies between values of attributes. In Section 4.4 we will show how the consistency is maintained. To do this we introduce a trigger mechanism, which we name as operation consistency relations. Note that these may be used as an alternative to attribute consistency maintenance, by directly implementing the “consistency agreement”.

Attribute consistency relations are mainly applicable to structural objects, where
there is a clear relation between the attributes of the classes connected by an EPL. We will focus on using attribute consistency relations, and rely on operation consistency relations where there is heavy use of encapsulation and where there may be a large discrepancy between the attributes. We may say that the two alternative ways of describing object consistency are suited for two different ways of utilising the basic object model: relying solely on structure or exploiting encapsulation. In Section 4.4.2 and 4.4.6 we will see that attribute consistency relations are more flexible for management purposes. But as we will also see in Section 4.4.6 that sometimes it is more convenient to use operation consistency relations.

4.3.1 Attribute consistency relations (ACRs)

When we have two classes \( C_1 \) and \( C_2 \) connected by an EPL, there may be some attributes of \( C_1 \) and \( C_2 \) that are dependent on each other. \( C_1 \) and \( C_2 \) must agree upon the values of the dependent attributes of all objects that are members of both classes, and for corresponding objects in those two classes. In this section, we introduce attribute consistency relations to describe the agreement between classes. The attribute consistency relations will be used to define object consistency, and further on they will be used as a basis for maintaining object consistency.

A starting point for the ensuing sections is that attributes are local to the class they belong to (after "expanding" the inherited properties), i.e. attributes are not shared between classes. However, there may exist dependencies between attributes in different classes. E.g., the value of an attribute may be defined to be equal to the value of another attribute. By analysing these dependencies and the request profile of the attributes, the system may actually decide to represent attributes shared between classes (see Section 4.4.2).

Consider class \( C_1 \) having attributes \( a_1 \) through \( a_n \), and class \( C_2 \) having attributes \( b_1 \) through \( b_m \). Class \( C_1 \) and class \( C_2 \) are connected by an EPL (the direction of the link is not important here). An attribute \( a_i \) of \( C_1 \), may either be independent of or dependent on the attributes \( b_1 \) through \( b_m \). In connection with EPLs, we will describe all attributes that are dependent on attributes of the other class. Attributes not taking part in such dependencies will be considered as independent. The dependencies will be described by attribute consistency relations (ACRs). The form of an attribute consistency relation is that a set of attributes of one class is dependent on a set of attributes from the other class.

**Definition 4.6 (Attribute consistency relations (ACRs))** Given two classes \( C_1 \) and \( C_2 \) having attributes \( a_1 \) through \( a_n \) and \( b_1 \) through \( b_m \), respectively, and which are connected by an extent propagation link. Given a subset \( x_1..x_k \) of the attributes \( a_1..a_n \) \( (k \leq n) \) and a subset \( y_1..y_p \) of the attributes \( b_1..b_m \) \( (p \leq m) \), an attribute consistency relation (ACR) is a relation of one of the following three categories:

- **Totally derivable relation:**

\[
(x_1, ..., x_k) \leftarrow (y_1, ..., y_p)
\]
The attributes \( x_1, \ldots, x_k \) are totally derivable from the attributes \( y_1, \ldots, y_p \).

- **Partially derivable relation:**

  \[
  (x_1, \ldots, x_k) \leftrightarrow (y_1, \ldots, y_p)
  \]

  The attributes \( x_1, \ldots, x_k \) are partially derivable from the attributes \( y_1, \ldots, y_p \).

- **Non-derivable relation:**

  \[
  (x_1, \ldots, x_k) \not\leftrightarrow (y_1, \ldots, y_p)
  \]

  The attributes \( x_1, \ldots, x_k \) are dependent on all the attributes \( y_1, \ldots, y_p \), but it is not possible to derive them.

\[ \square \]

It is important to notice that in all these three relations, *all* attributes on the left side of the relation are dependent on all attributes on the right side of the relation. In Section 4.3.2 we show how we operationally interpret the different dependencies, and in Section 4.3.3 we will define the semantics of these dependencies by interpreting them as mathematical functions.

Attribute consistency relations will be specified in connection with EPLs, which will have a (possibly empty) set of ACRs. Given two classes \texttt{CartLoc} and \texttt{PolarLoc} with two attributes each, representing the coordinates in the plane by cartesian and polar coordinates, respectively. In Figure 4.8 we show the EPL \texttt{CartLoc} \leftrightarrow \texttt{PolarLoc}, and two accompanying ACRs. The two ACRs constitute what is called an *attribute consistency group* (defined in Section 4.3.4), and they are written on the same line. The specification says that objects will be members of both classes, and that an object’s attributes \( x \) and \( y \) are totally derivable from \texttt{rho} and \texttt{theta}, and vice versa.

\[
\begin{array}{c|c}
\text{CartLoc} & \leftrightarrow & \text{PolarLoc} \\
\hline
(x, y) & \leftrightarrow & (\text{rho}, \text{theta})
\end{array}
\]

Figure 4.8: The CartLoc/PolarLoc example.

### 4.3.2 Interpretations of ACRs

We will now describe the meaning of attribute consistency relations. A totally derivable attribute consistency relation

\[
(x_1, \ldots, x_k) \leftarrow (y_1, \ldots, y_p)
\]

means that if *one* of the \( y \)'s is mutated, the values for *all* \( x \)'s must be derived from the \( y \)'s.
Given a non-derivable attribute consistency relation,

\[(x_1, \ldots, x_k) \leftrightarrow (y_1, \ldots, y_p)\]

it is not possible to map from the \(b\)'s to the \(a\)'s. If a client attempts to mutate (update) one of the attributes \(y_1\) through \(y_p\), the following different meanings may be given:

**Null value:** Insert *null* values in the non-derivable attributes \(x\)'s, meaning that the values of the \(x\)'s are not derivable. The correct values for the \(x\)'s may later be inserted manually, if they are known.

**Mutation denial:** Deny mutator requests to \(y_1\) through \(y_p\), because we do not have an “updatable dependency”, similar to the policy of the weak attribute equivalences of [LNE89].

We will refer to these as *non-derivability mutation policies*. We may also allow for inconsistencies by omitting to specify dependencies, but this gives the full responsibility for maintaining dependencies to the clients. Operation consistency relations may also be applied. Since these allow for operationally defined consistency maintenance, virtually any consistency maintenance policy may be defined.

Which policy to use is of course dependent on the intended semantics. If the user wants one specific non-derivability mutation policy, he might specify this with a *non-derivability mutation symbol*, which appears together with the ACR itself:

\[
\begin{align*}
\text{Null value:} & \quad (x_1, \ldots, x_k) \xleftrightarrow{NV} (y_1, \ldots, y_p) \\
\text{Mutation denial:} & \quad (x_1, \ldots, x_k) \xleftrightarrow{MD} (y_1, \ldots, y_p)
\end{align*}
\]

Note that the mutation denial symbol \((MD)\) indicates that the attributes on the tail side of the arrow, *as a result of the ACR*, may not be mutated.

A partially derivable relation

\[(x_1, \ldots, x_k) \xleftarrow{P} (y_1, \ldots, y_p)\]

will for some of the values of \(y_1\) through \(y_p\) be interpreted as a totally derivable relation, and for the rest of the values of \(y_1\) through \(y_p\) be interpreted as a non-derivable relation. This will be explained using mathematical functions in Section 4.3.3. A non-derivability mutation symbol may also be specified together with a partially derivable ACR.

### 4.3.3 ACRs and mathematical functions

We will now define the dependencies in ACRs using mathematical functions. To make this possible, we will treat attribute domains as mathematical sets. For a (basic or composite) attribute \(a\), \(\text{dom}(a)\) denotes the set of possible values \(a\) may
hold. Let $X$ be the set of attributes $x_1$ to $x_k$; $\text{dom}(X)$ is then a shorthand for $\text{dom}(x_1) \times \ldots \times \text{dom}(x_k)$.

Given two classes, $C_1$ with attributes $a_1$ to $a_n$ and $C_2$ with attributes $b_1$ to $b_m$, an EPL $C_1 \subseteq C_2$, a non-null subset $A$ of the attributes $a_1$ to $a_n$, and a non-null subset $B$ of the attributes $b_1$ to $b_m$. The different categories of ACRs will be given the following mathematical interpretations:

- **Totally derivable relation:** $A \rightarrow B$ means that there exists a total function:
  \[
  \mu : \text{dom}(A) \rightarrow \text{dom}(B)
  \]

- **Partially derivable relation:** Let $D$ be a non-null proper subset of $\text{dom}(A)$. Then, $A \rightarrow B$ means that there exist a function $\mu : D \rightarrow \text{dom}(B)$. We will introduce the extended function $\mu'$,
  \[
  \mu' : \text{dom}(A) \rightarrow \text{dom}(B)
  \]
  defined as
  \[
  \mu' = \begin{cases} 
  \mu & \text{for all elements in } D \\
  (\text{null}, \ldots, \text{null}) & \text{else}
  \end{cases}
  \]

- **Non-derivable relation:** $A \leftrightarrow B$ will be read as a null function
  \[
  \phi : \text{dom}(A) \rightarrow (\text{null}, \ldots, \text{null})
  \]
  which is a function resulting in $(\text{null}, \ldots, \text{null})$ for all elements in the domain.

This means that all ACRs involve functions, which fall into one of these three categories:

1. **Totally derivable relations** involve total functions, which can never be $(\text{null}, \ldots, \text{null})$.

2. **Partially derivable relations** involve extended functions, which are $(\text{null}, \ldots, \text{null})$
   for some elements in the domain.

3. **Non-derivable relations** involve null functions, which are $(\text{null}, \ldots, \text{null})$ for all elements in the domain.

Total and null functions may be seen as special cases of extended functions, where $(\text{null}, \ldots, \text{null})$ never and always appears, respectively. A partially derivable relation is thus handled as a totally derivable relation for the derivable part of the domain ($D$), and as a non-derivable relation for the rest of the domain ($\text{dom}(A) - D$).

ACRs are specifications of the dependencies between attributes, but to be useful each ACR must have an accompanying implementation of the function, which conforms to the specification.
4.3.4 Attribute consistency groups (ACGs)

Attribute consistency relations will appear in pairs. Such a pair is named an attribute consistency group (ACG), i.e. a group of attributes which are related by ACRs. An attribute consistency group will often correspond to “one conceptual equation”, e.g., the mathematical equation \( x = y_1 + y_2 \) will be described by the following two ACRs:

\[
\begin{align*}
(x) & \rightarrow (y_1, y_2) \\
(x) & \leftrightarrow (y_1, y_2)
\end{align*}
\]

This means that \( x \) may be derived from \( y_1 \) and \( y_2 \), but not vice versa. Those two ACRs constitute an ACG, which will be written as:

\[
(x) \leftrightarrow (y_1, y_2)
\]

Such a group, where a derivable relation has a non-derivable relation as “inverse”, may be a sign of wrong modelling. E.g., \( x \) is defined to be the sum of \( y_1 \) and \( y_2 \), and thus should rather be modelled as an operation that computes this sum, instead of being a stored and directly mutable attribute.

**Definition 4.7 (Attribute consistency group (ACG))** An attribute consistency group \( (x_1, \ldots, x_k) \oplus (y_1, \ldots, y_p) \) (\( \oplus \) is the generic ACG relation symbol) is a pair of ACRs defined using the same set of attributes, but in opposite directions. We will allow ACGs of the following forms:

\[
\begin{align*}
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p) \\
(x_1, \ldots, x_k) & \leftrightarrow (y_1, \ldots, y_p)
\end{align*}
\]

\( \square \)

We will refer to the set of attributes on one side of an ACG symbol as an attribute set.

Note that all \( x \)’s belong to one class, and all \( y \)’s to the other class. We assume that the optional non-derivability mutation policy is specified per ACG, e.g., \( A \leftarrow \rightarrow B \). For ACGs having no totally derivable ACR, this means that the two ACRs will share the policy. It would not be problematic to allow the two ACRs to have different non-derivability mutation policies, but for simplicity we have avoided this here.

Two examples of ACGs are the following: The equation example \( x = y_1 + y_2 \) has the ACG \( (x) \leftarrow \rightarrow (y_1, y_2) \). The coordinate example in Figure 4.8 has the ACG \( (x, y) \leftarrow \rightarrow (\text{rho}, \text{theta}) \).
We will disallow ACGs having two ACRs of the form \((x_1, \ldots, x_k) \leftrightarrow (y_1, \ldots, y_p)\) and \((x_1, \ldots, x_k) \not\rightarrow (y_1, \ldots, y_p)\). This means that for example \(x_1 + x_2 = y_1 + y_2\) will be disallowed as an ACG. The reason for this design decision is that such dependencies ensure that either the value of all involved attributes are \textit{null}, or that none of attributes can be mutated. Thus, this kind of dependencies are either blocking mutation of attributes completely, or ensuring attributes to always be \textit{null}.

We have limited the ACRs to be of the form: \((x_1, \ldots, x_k) \leftarrow (y_1, \ldots, y_p)\), where the \(x\)'s belong to one class and the \(y\)'s to another class, because we do not intend to let ACRs model constraints \textit{within} a class. ACRs are intended to capture inter-class constraints resulting from class evolution. The previously treated equation \(x_1 + x_2 = y_1 + y_2\) does not go well with this scheme. This equation would have been described as two ACRs

\[
\begin{align*}
(x_1, x_2) & \leftrightarrow (y_1, y_2) \\
(x_1, x_2) & \leftrightarrow (y_1, y_2)
\end{align*}
\]

Thus, when updating one of the attributes, it is not possible to derive any of the other attributes. This is an example of a constraint that holds both within a class and between classes. In addition to being a constraint connected to an EPL, the equation \(x_1 + x_2 = y_1 + y_2\) tells something about the sum of two attributes in each class. We could have described this dependency by allowing attributes to move to the other side of the relation, and achieve four ACRs similar to \((x_1) \leftarrow (y_1, y_2, x_2)\). We do not address general consistency constraints, and thus, we disallow relations where attributes have changed side.

We will now relate ACGs to mathematical functions. For each of the following ACGs between the two attribute sets \(A\) and \(B\), a function with domain \(\text{dom}(A)\) and range \(\text{dom}(B)\) would have the given characteristics:

<table>
<thead>
<tr>
<th>ACG</th>
<th>(\text{dom}(A) \rightarrow \text{dom}(B)) characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A \leftarrow \rightarrow B)</td>
<td>Bijection</td>
</tr>
<tr>
<td>(A \leftrightarrow \rightarrow B)</td>
<td>Total injection</td>
</tr>
<tr>
<td>(A \leftrightarrow \leftrightarrow B)</td>
<td>Total surjection</td>
</tr>
<tr>
<td>(A \leftarrow \rightarrow B)</td>
<td>Partial injection</td>
</tr>
<tr>
<td>(A \leftrightarrow \rightarrow B)</td>
<td>Partial surjection</td>
</tr>
</tbody>
</table>

The three lasting ACG combinations may be given a functional interpretation by defining the function in the opposite direction:

<table>
<thead>
<tr>
<th>ACG</th>
<th>(\text{dom}(B) \rightarrow \text{dom}(A)) characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A \leftarrow \rightarrow B)</td>
<td>Total injection</td>
</tr>
<tr>
<td>(A \leftrightarrow \leftrightarrow B)</td>
<td>Total surjection</td>
</tr>
<tr>
<td>(A \leftarrow \leftrightarrow B)</td>
<td>Partial surjection</td>
</tr>
</tbody>
</table>

ACGs will be used as units of mutation, i.e. all attributes in either \(A\) or \(B\) must be mutated as a unit. The reason for this lies in the management of object consistency. Given an ACG \(A \oplus B\), a mutation to one attribute in \(A\) implies that all attributes
in $B$ should get a new value, which again implies that all attributes in $A$ should have a new value. To avoid this trivial cyclic dependency situation, we define that all attributes in $A$ or $B$ should be mutated at once. This does not necessarily mean that a client has to explicitly mutate all attributes — this may be done implicitly by “accepting” the values of the other attributes as “consistent”. To explain the rationale for this, we will use the equation $x = y_1 + y_2$ as an example. This equation has the ACG $(x) \leftarrow x \leftrightarrow (y_1, y_2)$. When mutating $y_1$, $x$ should get a new, non-null value. This again implies that $(y_1, y_2)$ should get a $(null, null)$ value (in the null value interpretation of non-derivable relations). To avoid this, the mutation of $y_1$ accepts the value of $y_2$ as “consistent” when maintaining the consistency of the dependent attributes.

There is a more conceptual reason for introducing ACGs, since they often represent a “semantic unit”, i.e. attributes which naturally belong together. In the example of Figure 4.8 the $x$ and $y$ attributes belong naturally together, because they are equally important components of a cartesian coordinate pair. An attribute representing a graphical symbol at the point would not naturally belong to the same ACG. E.g., mutating $x$ would not imply that a new type of symbol should appear.

We do not allow ACRs to exist on their own: they must exist within ACGs. One-way dependencies are not modelled by an ACR, rather by operations. E.g., $x = y_1 + y_2$ could have been modelled by a single ACR $(x) \leftarrow (y_1, y_2)$. Due to the restriction above, such dependencies must be modelled by operations. E.g., by providing an observer operation $x()$ computing the sum of $y_1$ and $y_2$.

### 4.3.5 ACGs and null values

As said in Section 4.3.1, a null value means that the value could not be derived. When a non-derivable ACR is introduced and the null value approach to non-derivability mutation policy is chosen, a client must expect to get null values when observing objects. The simplest approach to manage null values is to accept them as they are: “the value is not known”. A more advanced solution is to treat them as a set of possible values. Given a consistency relation between faculties and departments at a university, where a faculty is understood in the European tradition, i.e., as an organisational unit of departments. An ACG modelling this dependency would be: $(department) \leftrightarrow (faculty)$. If faculty is mutated, department would be set to null. In this case we could interpret the null value of department as the set of possible departments (i.e. the set of departments at the faculty). This approach is referred to as the marked null value approach. We will not follow this line any further here, but refer to Section 7.3.2, where we outline such an approach in a query algebra for object-oriented views, and to [Gra91] as an in-depth study of incomplete information in databases, and finally to [Zic91b] for introducing null values into object-oriented databases.
4.3.6 Object consistency definition

When allowing objects or corresponding objects to be members of multiple classes having overlapping properties, it is necessary to keep the objects consistent. We have now introduced ACRs and ACGs as a way of specifying dependencies between attributes of corresponding objects. We will now define more precisely what we mean by object consistency in our framework.

We start by introducing satisfaction of ACGs. Informally, an ACG is satisfied if the two classes participating in that ACG agree upon the values of attributes.

Definition 4.8 (ACR and ACG satisfaction)

- An object $O$ (or two corresponding objects $O_1$ and $O_2$) satisfies an attribute consistency relation:

  - $(x_1, \ldots, x_k) \leftarrow (y_1, \ldots, y_p)$
    
    if the value of $(x_1, \ldots, x_k)$ equals $\mu(y_1, \ldots, y_p)$, for $O$ (or: $O_1$ and $O_2$), where $
    
    \mu$ is the corresponding total function.

  - $(x_1, \ldots, x_k) \leftarrow (y_1, \ldots, y_p)$
    
    if the value of $(x_1, \ldots, x_k)$ equals $\mu'(y_1, \ldots, y_p)$, for $O$ (or: $O_1$ and $O_2$), where $\mu'$ is the corresponding extended function.

  - $(x_1, \ldots, x_k) \leftrightarrow (y_1, \ldots, y_p)$
    
    if the value of $(x_1, \ldots, x_k)$ equals $\phi(y_1, \ldots, y_p)$, for $O$ (or: $O_1$ and $O_2$), where $\phi$ is the corresponding null function.

- An object $O$ (or two corresponding objects $O_1$ and $O_2$) satisfies an attribute consistency group, if one or both of the ACRs are satisfied for $O$ (or: $O_1$ and $O_2$).

For an ACG $A \leftarrow B$ to be satisfied, both ACRs must be satisfied, because the two corresponding functions are total and inverses of each other. For any of the other ACG combinations for $A \oplus B$, the satisfaction might hold for only one of the ACRs. This is due to the satisfaction of non-derivability (either in a $\rightarrow$ or a $\leftrightarrow$ relation), where either or both A and B hold null values. For an ACG $A \leftarrow \leftrightarrow B$, at one particular moment, only one relation can be satisfied.

An alternative way of defining ACG satisfaction could be to only require one of the two derivable relations in an ACG to be satisfied. This would have given an asymmetrical consistency mechanism, and would not require that the two functions were inverses of each other. We exclude this possibility because the ACG concept is intended to be used for reasoning about shared representation of attributes, and any asymmetry would have prevented this. If such a dependency is wanted, it may easily be implemented using operations.

We are now ready to define object consistency by the use of attribute consistency groups. Informally, an object is consistent when it is propagated properly, the classes
agree upon the values of attributes, and the object is consistent in each class, which means that it does not hold illegal values.

**Definition 4.9 (Object consistency)**

- An object \( O \) is a valid member of a class \( C \), if it for all attributes \( a_i \) of \( C \) holds a value of \( \text{dom}(a_i) \) or a null value.

- An object \( O \) is consistent if
  - for every class \( C \) that \( O \) is member of, \( O \) is member of all classes in the extent propagation set of \( C \) (\( EPS(C) \)), or has corresponding objects in \( EPS(C) \), and
  - \( O \) (and its corresponding objects) is/are valid members of all its/their classes, and
  - for all extent propagation links \( C_i \rightarrow C_j \), where \( O \) (and its corresponding objects) is/are member(s) of \( C_i \) and \( C_j \), all attribute consistency groups are satisfied. □

Note that the definition includes satisfaction of ACGs, which again means that the consistency of an object is dependent on the functions implementing the ACRs in an ACG. Thus, we cannot tell if an object is consistent without knowing the functions implementing the relevant ACRs.

We give a simple example of an object being consistent: Given the class \( C_1 \) having one float attribute \( x \) and the class \( C_2 \) having the two float attributes \( y_1 \) and \( y_2 \). There is an extent propagation link (EPL) \( C_1 \Leftrightarrow C_2 \) with the “consistency relation” \( x = y_1 + y_2 \) specified by the ACG \( (x) \leftarrow \rightarrow (y_1, y_2) \). In this case the object, being illustrated by tuples \( \text{Class,OID,Values}) \): \( (C_1, 0100, 7.0) \) and \( (C_2, 0100, 3.0, 4.0) \), is consistent, because the object is consistent in both classes, it is a member of all classes in \( EPS(C_1) \) and \( EPS(C_2) \), and the ACG is satisfied, because the derivable ACR is satisfied \((7.0 = 3.0 + 4.0)\).

### 4.3.7 Transitive ACRs

When several ACRs involve the same attribute set, they will constitute a transitive ACR. Transitive ACRs appear straightforward according to the following rules:

<table>
<thead>
<tr>
<th>1st ACR</th>
<th>2nd ACR</th>
<th>Transitive ACR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \rightarrow B )</td>
<td>( B \rightarrow C )</td>
<td>( A \rightarrow C )</td>
</tr>
<tr>
<td>( A \rightarrow B )</td>
<td>( B \rightarrow C )</td>
<td>( A \rightarrow C )</td>
</tr>
<tr>
<td>( A \leftrightarrow B )</td>
<td>( B \rightarrow C )</td>
<td>( A \leftrightarrow C )</td>
</tr>
<tr>
<td>( A \rightarrow B )</td>
<td>( B \leftrightarrow C )</td>
<td>( A \rightarrow C )</td>
</tr>
<tr>
<td>( A \leftrightarrow B )</td>
<td>( B \leftrightarrow C )</td>
<td>( A \leftrightarrow C )</td>
</tr>
<tr>
<td>( A \leftrightarrow B )</td>
<td>( B \leftrightarrow C )</td>
<td>( A \leftrightarrow C )</td>
</tr>
</tbody>
</table>
Note that depending on the domains and ranges of the accompanying functions in the fourth rule, the transitive ACR may actually be a non-derivable relation. Given the three classes \( C_1, C_2, \) and \( C_3 \) having attribute sets \( A, B, \) and \( C, \) respectively. The two ACGs \( A \leftarrow\rightarrow B \) and \( A \leftarrow\rightarrow C \) result in two transitive ACRs to appear in the ACG \( B \leftarrow\rightarrow C. \) Note that \( A \) is dependent and derivable from both \( B \) and \( C. \) These two ACRs \( (A \leftarrow B \) and \( A \leftarrow C) \) are independent in the sense that \( A \) may be derived from \( B, \) and \( A \) may be derived from \( C. \) With ACRs it is not possible to express dependencies such that \( A \) will be derived from the combination of \( B \) and \( C \) (unless \( B \) and \( C \) are attributes of the same class).

We introduce another class \( C_4 \) having the attribute set \( D, \) and two new ACGs \( B \leftarrow\rightarrow D \) and \( C \leftarrow\rightarrow D. \) In this situation all four attribute sets \( A, B, C \) and \( D \) will depend on each other along two paths. E.g., we will deduce \( A \leftarrow\rightarrow D \) “through” both \( B \) and \( C. \) For an object being member of these four classes to be consistent, all four ACGs must be satisfied. If the value of \( A \) is derived from \( D, \) the derivation along both paths must give the same value. This means that to provide consistent objects according to the definition, such duplicate paths must be internally “consistent”.

Generally, we see it as the schema designer’s responsibility to detect and solve such conflicts. However, it is possible to find possible conflicts by simply detecting cyclic dependencies. This may be done by constructing a dependency graph among attribute sets, where the vertices are attribute sets, and the edges are dependencies between attributes. E.g., the ACR \( A \rightarrow B \) gives rise to two vertices \( A \) and \( B, \) and with a directed edge from \( A \) to \( B. \) A single ACG \( A \oplus B \) introduces a trivial cycle in the dependency graph, i.e. a cycle of length 2. The semantics of trivial cycles are handled by the definition of satisfaction of ACGs. The problem with conflicts between ACRs may appear when there are non-trivial cycles in ACRs. When such a cycle is found in a dependency graph, the database system may notify the schema designer that a cycle has been found, and that this may include a conflict. The schema designer then has to carefully examine the accompanying functions of the ACRs to see if there is any disagreement.

### 4.4 Maintaining object consistency

The previous section defined what we mean by object consistency. In this section we will be concerned about maintaining object consistency. Here we will focus on the two important points in the definition of object consistency:

1. Extent propagation
2. Satisfaction of ACGs

The former point builds on Section 4.2.2, where EPLs were introduced to describe how objects should be propagated between extents. We will briefly present some timing policies for extent propagation.
The latter point builds on Section 4.3, where we described dependencies between attributes of objects (or corresponding objects) that are members of multiple classes. The main bulk of this section presents how to maintain these dependencies. We start by describing how we can infer from ACGs that shared representation of attributes is possible. When using shared representation, consistency will be “automatically” maintained, since there is no room for representing any inconsistencies. We present operation consistency relations, which will be an important tool in the consistency maintenance for separate representation of attributes. Operation consistency relations may also be used directly to implement the dependencies between properties. In Section 4.4.6 operation consistency relations and attribute consistency relations are compared as alternative mechanisms for describing consistency of objects.

4.4.1 Extent propagation

Extent propagation is the process of ensuring that all objects being member of the class $C$, have a corresponding object in all classes in $EPS(C)$. This means propagating objects between extents, or checking whether corresponding objects exist in the extents connected by EPLs. We see two main timing models for when to propagate objects between extents:

Eager propagation: Objects are propagated as soon as they are created or added to a class.

Lazy propagation: Objects are propagated when they are retrieved from a class.

In Section 7.1.1 we treat implementation issues for extent propagation. We will see that extent propagation may be implemented by a construct similar to operation consistency relations. In Section 7.3 we will show how views may be used to implement lazy extent propagation. In Section 6.3.2 we show three different ways of registering corresponding objects.

4.4.2 Shared representation of attributes

This section will continue the work described in Section 4.3 by showing how to reason about the representation of attributes using attribute consistency relations. By analysing the ACRs in ACGs, we can tell if the attributes may be represented shared, or if separate representation is the only alternative. This is an important topic, because shared representation is favourable with respect to the cost of consistency maintenance (see Section 5.3). As we stressed in Section 4.3, all attributes are local to a class, and it is not possible for the schema designer to define that an attribute should be physically shared between two classes. However, based on the ACRs, the system can decide if shared representation is feasible.

In the previous section we showed how ACGs were used as a unit for specifying object consistency. We will now use them as a unit for shared representation (storage) of attributes. The basic idea is simple: Given an ACG $A \leftarrow \rightarrow B$, the attributes in
A are derivable from the attributes in B, and the attributes in B are derivable from the attributes in A. This facilitates storing one of the attribute sets, either A or B, and using the function implementation to map back and forth when requesting the other set. The simplest example of this is attribute equivalence \( x = y \), which gives rise to the ACG \( (x) \xrightarrow{\mu} (y) \). In this case we may choose to represent \( x \) and \( y \) shared in either \( x \) or \( y \).

For an ACG \( A \oplus B \), the choice of representation depends on \( \oplus \). We will now look at some of the different combinations of ACG characteristics, and illustrate why and how we can use shared representation. In Table 4.1 we summarise the representation possibilities for the different ACG combinations.

For the ACG \( A \xrightarrow{\mu} B \), we may represent \( A \) and \( B \) shared in \( A \) or in \( B \). The reason for this is that all values in \( A \) have a corresponding value in \( B \), and vice versa. As in the shared/non-shared decision, we may use request profile to decide which one to choose (\( A \) or \( B \)). In this discussion it is important to notice that accompanying the total functions are inverses of each other. If they were not, \( A \) and \( B \) could not have been represented shared without consequences for the semantics of attributes. If this was the case, mutation of an attribute with one specific value would have given another value upon observation. In the terminology of [LNE89], \( A \) and \( B \) would not have been in attribute equivalence. In such situations, it would not have been clear what the meaning of object consistency is. Consequently, such ACGs are disallowed.

Consider the coordinate example in Figure 4.8. This has the ACG \( (x, y) \xrightarrow{\mu} (\rho, \theta) \). We may choose to represent \( (x, y) \) and \( (\rho, \theta) \) either separately or shared. For shared representation we may choose to represent either \( (x, y) \) or \( (\rho, \theta) \). If we choose to represent \( (x, y) \), when requesting \( \rho \) or \( \theta \), we must map back and forth to \( x \) and \( y \) both on observer and mutator requests.

Given \( A \xrightarrow{\mu} B \), we have two different representation choices, depending on the choice of non-derivability mutation policy. In Figure 4.9 we have illustrated the corresponding functions for an example of \( A \xrightarrow{\mu} B \). The domains of \( A \) and \( B \) are illustrated as tables, and the functions as arrows between elements of these domains. \( \text{dom}(A) \) represents four levels of academic credits, while \( \text{dom}(B) \) represents six levels. Given the ACG \( A \xrightarrow{\mu} B \), i.e. having the policy for non-derivability mutation where null values will be inserted in \( A \) for the “extended part” of the function (illustrated with nulls outside the table in Figure 4.9), we may let \( A \) and \( B \) be represented shared in \( B \). The reason for this is that the attributes in \( A \) may always be derived from the attributes in \( B \), and in the cases where \( A \) may be null, the extended function of \( \xrightarrow{\mu} \) will take care of this. In the example this can be seen by that \( \text{dom}(A) \) is actually a subset of \( \text{dom}(B) \). If the other non-derivability mutation policy is chosen, \( A \xrightarrow{\mu} B \), i.e. denying the mutator requests to \( B \) that set the value of \( B \) outside \( R \), then we may actually let the two attribute sets be represented shared either in \( A \) or \( B \). The reason for this last approach is that \( B \) is not allowed to take values outside those that are mappable to values of \( A \), thus we have a bijection. With respect to Figure 4.9, this means that \( B \) is not allowed to hold those values, where the extended function of \( \xrightarrow{\mu} \) maps to null values. In the example, this means
that a student being represented in both classes, is not allowed to hold a **MS** or **PhD** degree in this database.

In Figure 4.10 we have illustrated an example of $A \longleftrightarrow B$. In this case, we have to represent both sets of attributes in the *null* value approach, because there are cases where both sets of attributes are known, but the function to the other attribute set is *null*. If we had chosen to represent them shared in one of them, mutating the other to a non-*null* value may have made both $A$ and $B$ *null*. In the mutation denial approach, we may choose to represent them shared in either $A$ or $B$. This situation requires $A$ and $B$ to hold values within the “common” area.

Given the ACG $A \longleftrightarrow \nrightarrow B$, we have two possible ways of utilising shared representation, depending on which policy to non-derivability mutation is taken:

1. If the mutation denial approach is taken ($A \overset{MD}{\longleftrightarrow} B$), we may choose to represent $A$ and $B$ shared in $B$, because $A$ may be derived from $B$ ($A \leftarrow B$).
This is possible as long as no one mutates $A$, or if all mutator requests to $A$ are denied.

2. If the null value approach to non-derivability is taken ($A \xleftarrow{\text{NV}} \xrightarrow{} B$), shared representation is not realistic. Shared representation in $A$ would mean that $B$ would always be null, while shared representation in $B$ would mean that all information would be lost as soon as a client mutated $A$.

As an example of this, we will use the equation $x = y_1 + y_2$, which gives the ACG $(x) \xleftarrow{} \xrightarrow{} (y_1, y_2)$. In this case we may choose to represent the attributes in $y_1$ and $y_2$, and map to $x$ each time $x$ is observer-requested. As long as $x$ is never mutated, this is quite ok, but if $x$ is frequently mutated and observed, separate representation is more reasonable.

In Table 4.1 we have summarised the shared representation possibilities for all combinations of ACG situations and non-derivability mutation policies.

<table>
<thead>
<tr>
<th>ACG</th>
<th>Shared representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$A$ or $B$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$A$ or $B$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>No</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$A$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>$A$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$A$ or $B$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>No</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$A$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>No</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{NV}} \xrightarrow{} B$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A \xleftarrow{\text{MD}} \xrightarrow{} B$</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of the shared representation possibilities.

There are several issues to consider when choosing the representation scheme. First, shared representation gives only one choice with respect to timing policy for consistency maintenance (this will be further discussed in Section 4.4.4 and 4.4.6). De-
pending on the request profile, this might be an important question. E.g., if a new
version \( C_2 \) of a class \( C_1 \) is created, and we choose to represent attributes of \( C_2 \) shared
in \( C_1 \), we might end up with a situation where all clients of \( C_1 \) have disappeared and
all requests to these attributes in \( C_2 \) must be mapped back and forth to \( C_1 \). Second,
there may be a question of storage space reduction. Third, the representation scheme
influences the clustering of objects. The architecture of the database system will also
influence the choice of representation. Shared representation provides for “automatic
consistency maintenance”, since there is no way of representing any disagreement.
In a distributed setting some level of inconsistency may have to be accepted, thus
shared representation may not be desirable.

4.4.3 Implementing shared representation

We will now outline how consistency maintenance may be implemented when shared
representation is chosen.

Every derivable ACR must have an accompanying implementation of the mathemat-
ical function. For a totally derivable ACR,

\[
(x_1, ..., x_k) \mapsto (y_1, ..., y_p)
\]

the function (named \( \text{mu} \)) will have the following signature:

\[
domain \text{mu}_\text{dmn} \ record \{ \dom(y_1) \ y_1; \ldots \ \dom(y_p) \ y_p; \};
\]

\[
\text{mu}_\text{dmn} \ \text{mu} \ (\dom(x_1), \ldots, \ \dom(x_k));
\]

In the coordinate example there are two functions, which we name \text{polarToCart} and
\text{cartToPolar}. If we choose to represent \((x, y)\), we can implement the consist-
cy maintenance by embedding the two functions into the attribute request operations
of \( \rho \) and \( \theta \). When observing \( \rho \) or \( \theta \), the observer operation \((\rho()\) or
\(\theta()\) first requests the values of \(x\) and \(y\), and from these it calculates the values of
\(\rho\) and \(\theta\) using \text{cartToPolar}. Mutation becomes a bit harder due to the mutation
semantics enforced by the ACG concept: All attributes in an attribute set must be
mutated together. When mutating \(\rho\) and \(\theta\) as one unit, we calculate the \(x\)
and \(y\) values to be stored using the \text{polarToCart} function. The situation becomes
more tricky if only one of \(\rho\) and \(\theta\) is mutated. In this situation (e.g., \(\rho\) is
mutated), we assume that the value of the other attribute (\(\theta\)) is to be kept as it is.
Since this attribute is not stored, we first have to calculate the existing value (of
\(\theta\)), and then mutate the attribute \(\rho\), and at the end \(x\) and \(y\) are mutated by
the new values calculated using \text{polarToCart}. The unit of mutation used here (per
attribute set) may be implemented either by an atomic, generic mutation operation,
or implicitly by accepting the values of the other attributes in the attribute set.

For a partially derivable ACR,

\[
(x_1, ..., x_k) \mapsto (y_1, ..., y_p)
\]
an implementation of the partial function must be provided to the system. Given
the ACG \( A \xrightarrow{\text{NV}} B \), where we choose to represent them shared in \( A \), this can
be implemented similar to the coordinate example. The observer operation for \( B \)
must return \textit{null} for those values where the partial function is not specified. In the
mutation denial approach, the mutator operation for \( A \) must deny mutation of \( A \)
for those values of \( A \) where the partial function is not defined.

For an ACG containing a non-derivable ACR, shared representation is only possible
in the mutation denial approach. An implementation of this is similar to ACGs
containing partially derivable relations, but here all mutations will be denied by the
mutation operation of the attributes on the non-represented side of the ACG. This
can be done by not providing any attribute mutation operations for the relevant
attributes.

### 4.4.4 Operation consistency relations (OCRs)

We will now describe a special-purpose trigger facility, named \textit{operation consistency
relations}, which will be used for two different purposes. In Section 4.4.5 we will show
how to use them for implementing ACGs when attributes are separately represented.
In Section 4.4.6 we will compare operation consistency relations with attribute consis-
tency relations, and show how they without the use of ACRs may implement
object consistency.

Active databases and triggers are parts of current trends in database research. While
the research focuses on languages for describing triggers, execution models, and
optimisations [VK93], we merely describe an application of an Event Action trigger
facility, e.g., Ariel [Han89], POSTGRES [SR86], and Hipac [MD89]. An Event
Action trigger is an operation that is automatically invoked upon a given event in
the database. An operation consistency relation is an Event Action trigger where
the event is the invocation of a given operation in the database. See Section 7.1.2
for more on the relationship between operation consistency relations and triggers.

**Definition 4.10 (Operation consistency relation (OCR))** An \textit{operation consis-
tency relation (OCR)} is an operation that is automatically requested when an-
other given operation is requested. It cannot be requested explicitly, only implicitly
upon a normal operation request. An operation consistency relation consists of a
timing mode, an operation signature, and a body.\(\square\)

Like an attribute consistency relation, an operation consistency relation is specified
in connection with an EPL between two classes \((C_1 \Rightarrow C_2)\). Similar to the definition
of object consistency in Section 4.3.6, for an EPL \( C_1 \Rightarrow C_2 \), we require an object \( O \)
in \( C_2 \) to have a corresponding object in \( C_1 \), for an associated OCR to be allowed to
be executed. I.e. if an object is only member of \( C_2 \) there is no need to execute the
OCR.

We will use the following syntax to illustrate OCRs:
extprop C1 \Rightarrow C2 
\{ 
  ocrs: 
    timing op (i1, \ldots, ik) body 
\};

The \textit{timing mode} of an OCR is either \textit{before} or \textit{after} (explained below). The operation signature, \( op(i_1, \ldots, i_k) \), corresponds to a signature of an operation \( op' \) from one of the classes \( C_1 \) or \( C_2 \), i.e. \( op \) is the name of \( op' \), and \( i_1 \) through \( i_k \) correspond to the formal parameters of \( op' \). If both classes have operations with the same name, the OCR must qualify the signature with the correct class, e.g.

\begin{verbatim}
extprop C1 \Rightarrow C2 
\{ 
  ocrs: 
    before C1::x (newX) \{ \ldots \} 
\};
\end{verbatim}

When \( op' \) is requested, the actual parameters of the operation request will be bound to the formal parameters, \( i_1 \) through \( i_k \), of the OCR as well. Depending on the timing mode, the body of the OCR will be executed either before or after the execution of the body of \( op' \). The body is a piece of code that is specified in the same language as the language of the ordinary operations. We allow the body of an OCR to request all properties of objects in both classes connected by an EPL. One consequence of this “freedom” is that encapsulation may be broken by OCRs. In Section 5.7.4 we will discuss the encapsulation issue further.

We will use the two different timing modes of OCRs to implement two different \textit{timing models} for consistency maintenance:

1. \textit{Eager consistency maintenance}: This means that consistency maintenance is done busily upon mutation of objects. Eager consistency maintenance is done through \textit{after timing mode} OCRs.

2. \textit{Lazy consistency maintenance}: This means that the consistency maintenance is delayed until the properties are observed. Lazy consistency maintenance is done through \textit{before timing mode} OCRs.

\textit{After timing mode} for an OCR is such that when \( op' \) is requested in \( C_1 \) for an object \( O \), the body of the OCR is executed \textit{after} \( op' \) has executed. The utilisation of this construct is when \( op' \) is a mutator. The body of the OCR will then ensure that \( O \) in \( C_2 \) is consistent with respect to \( O \) in \( C_1 \). This means that once an object has been mutated in one class, it will be mutated in the other classes. In Figure 4.11 we have illustrated eager consistency maintenance in the coordinate example from Figure 4.8. We have illustrated an object \( o200 \) as tuples (\textit{Class}, \textit{OID}, \textit{AttrValues}) in the two classes. To the left, the object is represented as \((\text{CartLoc},o200,x,y)\) in the \textit{CartLoc} class, and to the right as \((\text{PolarLoc},o200,\text{rho},\theta)\) in the \textit{PolarLoc} class. On the top of the figure the two “tuples” are consistent. Then, \( o200 \) is requested with a mutator operation \textit{rho}(7.0) mutating the \textit{rho} attribute to the new value 7.0. Connected to the \textit{rho(newRho)} operation it is an \textit{after} OCR: once \textit{rho}(7.0) is executed, the body
of the OCR ensures that the same object will become consistent in CartLoc, i.e. the o200 “tuple” in CartLoc is mutated. The time axis illustrates the execution of the body in “parallel” to the execution of the requested operation. This means that these two operations are executed as an atomic unit.

![Diagram](image)

**Figure 4.11:** Eager consistency maintenance using *after* timing mode.

*Before timing mode* for an OCR means that when op₁ is requested in C₁ for an object O, the body of the OCR will be executed *before* op₁. With this timing mode, the body of the OCR will ensure that O in C₁ is made consistent just before it is accessed. The utilisation of this construct is mainly when op₁ is an observer operation which has related mutator operations in other classes connected by EPLs. In Figure 4.12 we have illustrated lazy consistency maintenance by the same example as for eager consistency maintenance. Here, o200 is mutated by some operation in PolarLoc, which means that o200 becomes inconsistent in CartLoc with respect to PolarLoc. At this point in time a client issues an observer request to the x attribute of o200. Since the x attribute is not consistent for o200, an associated *before* OCR will mutate x (and y) to their new values. After the execution of the body of the OCR, the operation itself can be executed. In Section 4.4.5 we will explain this example in more detail.

Operation consistency relations are implemented in the context of two classes connected by an EPL (C₂ ⇒ C₁). Every operation signature specified in OCRs has a corresponding operation in either C₁ or C₂. Since an OCR may request operations of an object in both classes, in case of ambiguities a qualification of class must be given. Connected to the EPL, each operation of the classes may have any number of OCRs. If there are several OCRs with the *same* timing mode connected to the same operation, the sequencing of the execution of the OCRs will be unspecified.

We think it is appropriate to give some guidelines on the usage of the OCR construct. Given an EPL, C₂ ⇒ C₁, and an OCR, Topbody, where T is a timing mode and op is a signature of an operation in C₂. If T is *after* timing mode, it is most natural for the body to observe the properties of C₂ and to observe and mutate the properties
of $C_1$. If $T$ is before timing mode, it is most natural for the body of the OCR to observe and mutate the properties of $C_2$ and to observe the properties of $C_1$. These guidelines are given to conform to the intention with our consistency maintenance to only cover inter-class dependencies.

Until now we have said that an OCR connected to the EPL $C_1 \Rightarrow C_2$, maintains the consistency of an object $O$ in the two classes. E.g., if $O$ is mutated in $C_1$, a similar mutation may have to be done to $O$ in $C_2$. If $S : \mathbf{id}_1 \rightarrow \mathbf{id}_2$ is only required to be a total function, an object in $C_2$ may have $N$ corresponding objects in $C_1$ (cf. Figure 4.7). Let $o101..o110$ in $C_1$ have $o200$ as the corresponding object in $C_2$. If $o200$ is mutated, consistency will have to be maintained for all objects $o101..o110$ in $C_1$. For eager consistency maintenance this means that a mutation to $o200$ will start an associated OCR for each corresponding object $o101..o110$. When one of the objects in $o101..o110$ is mutated, e.g., $o101$, the consistency has to be maintained to $o200$ and to all objects in $o102..o110$. The reason for this semantics is that $o200$ and each of the objects in $o101..o110$ have a mutual dependency.

OCR s are important for several reasons. First, when using separate representation for attribute consistency groups, the consistency maintenance will be implemented using OCRs. Second, as will be seen, operation consistency relations may be used as an alternative to attribute consistency relations. Third, operation consistency relations may be used in addition to attribute consistency relations.

4.4.5 Separate representation of attributes

This section will show how consistency of objects may be maintained when attributes are separately represented. We will connect OCRs to attribute request operations to
maintain the consistency as specified by ACRs. This is a straightforward and simple way of applying OCRs for consistency maintenance. Note that realistic systems could do many optimisations compared to what is presented in this section. In Section 5.6.1 we will outline some of these when scaling up the dependency graphs.

**Eager consistency maintenance** means that once an attribute is mutated, all dependent attributes are mutated according to the specifications in the ACRs (for non-derivable ACRs, denial of mutation is another possibility). Upon such a mutator request to an attribute, an operation consistency relation may be used to maintain object consistency.

In Figure 4.11 we illustrated an example of eager consistency maintenance in the coordinate example from Figure 4.8. In Figure 4.13 we show two of the OCRs implementing this consistency maintenance. We have shown the OCRs connected to the mutator operations for \( x \) and \( \rho \). The principle here is that as soon as an attribute is mutated (e.g., \( x \)), the dependent attributes (\( \rho \) and \( \theta \)) are mutated with the new values. Since there is a trivial cyclic dependency here (as always in ACGs), we have to avoid non-terminating consistency maintenance. This is done by introducing “visited flags” for each attribute. When \( x \) is mutated, \( (x,y) \) is marked as visited, and the OCR mutation of \( \rho \) and \( \theta \) will not result in a new mutation of \( x \) and \( y \). This implementation uses the mutation semantics mentioned in Section 4.3.4, i.e. when \( x \) is mutated, both \( x \) and \( y \) are marked as consistent (by the visited flag). The OCRs for the other two mutation operations are similar. Note that the set of OCRs in Figure 4.13 may be generated from the ACRs and the accompanying implementations of the mathematical functions (the code within the if statements).

**Lazy consistency maintenance** means that the consistency maintenance is delayed until the time of observer requests. This situation occurs when an attribute is observer requested for an object in a class, and the object in this class is inconsistent with respect to other classes. In this case, we will use a “before OCR” type of relation to make the object consistent before the observation is actually done.

In the coordinate example lazy consistency maintenance (previously sketched in Figure 4.12) means that \( (x,y) \) and \( (\rho,\theta) \) may be mutated without maintaining the consistency with respect to each other before the other attribute set is observed. The last one to be mutated holds the “correct” value, and the dependent attributes are then “incorrect”. This means that when observing an attribute (e.g., \( x \)), it is checked whether it is consistent with respect to the dependent attributes (\( \rho \) and \( \theta \)). If it is not, it is made consistent by applying the corresponding function of the ACG.

Lazy consistency maintenance requires us to register if an attribute holds the consistent (i.e. the most recent) value. This may be done by using “dirty flags”. Then upon each observation, the flag is checked. If it is set, the value will be mutated by the correct value, and the flag will be unset. Upon mutation we have to set the dirty flags of the dependent attributes. To make this approach reasonable, the cost of setting flags must be much lower than the cost of making an attribute consistent. Results from distributed file systems indicate that this approach may be reasonable in a distributed setting, because explicit notification of the fact that “you are dirty” is favourable over checking for inconsistencies upon each observer request [HKM+88].

89
extprop CartLoc <=> PolarLoc {
    ocrs:
    after rho(float newRho) {
        mark (rho,theta) as visited;
        if (x,y) is unvisited
            x(newRho*cos(theta()));
            y(newRho*sin(theta()));
        mark (rho,theta) as unvisited;
    }
    after x(float newX) {
        mark (x,y) as visited;
        if (rho,theta) is unvisited
            rho(sqrt(newX**2 + y()**2));
            if newX /= 0.0
                theta(arctan(y()/newX));
            else
                theta(signum(y())*pi/2.0);
        mark (x,y) as unvisited;
    }
    ...
};

Figure 4.13: Eager consistency maintenance in the coordinate example.

The set of OCRs in Figure 4.14 outline parts of a realisation of this approach. Note that each attribute will have both a before and an after OCR. The after OCR marks the dependent attributes `rho` and `theta` as dirty. The marking presented in this figure is illustrative only. All dependent attributes of `x` and `y` should be marked as dirty. This means that if there are transitive dependent attributes (e.g., depending on `rho` and `theta`), these should be marked as well. This can be done by either putting the marking of an attribute set into an operation that can trigger new OCRs, or by having a generic marking operation knowing the set of dependent attributes. In the latter solution, the after `x` OCR would replace the single mark statement, with a request to this generic operation: `mark.dep.of (x, y)`.

Again we will emphasise that the OCRs are implementation of the ACRs specified in Figure 4.13. The code and representation to manage the dirty flags are generated from the user-specified ACRs and their accompanying functions. Also in this case the mutation is done per attribute set. When mutating `x` this means that either must `y` be mutated as well, or the current value of `y` must be accepted as consistent. If only `x` was marked as consistent upon mutation of `x`, all the three other attributes `y`, `rho`, and `theta` would have been marked dirty. Thus, we would not have “enough” consistent attributes to derive a consistent coordinate pair.

The different timing policies for object consistency maintenance have different characteristics. Eager consistency maintenance ensures immediate mutation of all de-
extprop CartLoc <=< PolarLoc {
  ocrs:
  before float x () {
    if (x,y) is dirty
      x(rho)*cos(theta()));
      y(rho)*sin(theta()));
      mark (x,y) as not dirty;
  }
  after x (float newX) {
    mark (rho,theta) as dirty;
  }
};

Figure 4.14: Lazy consistency maintenance in the coordinate example.

dependent properties in other classes, while lazy consistency maintenance will wait until an inconsistent property is requested before the maintenance is done. The utilisation of these two timing modes should be dependent on the planned profile of requests to the database. Consider the coordinate example. If the CartLoc class is far more exposed to mutator requests than PolarLoc is exposed to observer request, lazy consistency maintenance of PolarLoc is feasible. If CartLoc is less exposed for mutator requests than PolarLoc is exposed for observer requests, eager consistency maintenance of PolarLoc is appropriate.

In this discussion we may see shared representation as the end of the scale where one side of the ACG is far more requested than the other side. This can be realised by combining before and after timing mode for OCRs, similar to the example in Figure 4.15. Here the attributes rho and theta will not be stored at all. When observing rho or theta, the values will be calculated by before OCRs, and when mutator requesting rho or theta, x and y will be mutated with the new values.

We will now make some comments to consistency maintenance when there are non-trivial cycles in the dependencies between attributes. Recapitulating from Section 4.3.7, in case of cycles between ACRs, i.e. an attribute set is dependent on another attribute set through several dependency paths, these paths have to “agree” for an object to be consistent. Assume the example from Section 4.3.7, where A is dependent on D along two different paths. For an object to be consistent, the value derived for A from D must be equal for these two paths. When the consistency maintenance is implemented by OCRs as described in this section, a conflict between these two paths will lead to unspecified results for A. The reason for this is that A will be derived from D along two different paths by means of multiple OCRs. In a lazy consistency maintenance situation, there will be two OCRs connected to the observer operation(s) of A. Assume that the non-dirty value is represented in D. When observing A, the two OCRs will request B and C, respectively, and both will compute the new value for A. The execution order between these OCR is
extprop CartLoc <=> PolarLoc {
  ocrs:
  before float rho () {
    rho(sqrt(x()**2 + y()**2));
  }
  before float theta () {
    if x() /= 0.0
      theta(arctan(y()/x()));
    else
      theta(signum(y()*pi/2.0));
  }
  after rho (float newRho) {
    x(newRho*cos(theta()));
    y(newRho)*sin(theta());
  }
  after theta (float newTheta) {
    x(rho())*cos(newTheta));
    y(rho())*sin(newTheta));
  }
};

Figure 4.15: Shared representation using OCRs.

not specified, which means that the first OCR to mutate A will “win”, because it unmarks the dirty flag of A. This is not a problem as long as both paths give the same value. We consider it as the schema designer’s responsibility to resolve such conflicts.

4.4.6 ACRs vs. OCRs

We will now discuss and compare the use of attribute consistency relations and operation consistency relations. In some sense this is like comparing specification and implementation, turning the discussion into “what are advantages of specification?” Generally in software development, a specification is used as an agreement between providers and users of a service [LG86]. Compared to an implementation, a specification tells only what is necessary for the user of a service; it avoids overspecification. This allows for easier understanding of the properties of the service, for multiple implementation strategies, and often for formal reasoning about certain aspects of the specification.

When using ACRs for describing dependencies, for each ACR we will have:

1. A specification of the dependency by a mathematical function.

2. An optional non-derivability mutation symbol.
3. An *implementation* of the mathematical function.

We will exploit the specifications for multiple implementation strategies and partially for formal reasoning. In addition they serve as documentation of the dependencies.

OCRs are *implementation* of dependencies that are due to class evolution, where each OCR consists of:

1. A timing mode.
2. An operation signature.
3. An operation body.

The operation body consists of general code, and can thus implement any kind of dependencies expressible in the language of the implementation. Note that a function implementation of an ACR is not a complete OCR, but it may be embedded in bodies of OCRs, or it may be embedded in the attribute request operations (shown in Section 4.4.2).

We will show *four* main differences between ACRs and OCRs, which are due to ACRs being declarative, while OCRs are implementation. The *first* main difference lies in ability to reason about the representation. ACRs are description of the dependency between the state of objects in different classes, while OCRs are description of actions to be executed when operations are requested. As seen in Section 4.4.2, ACRs may be used to reason about the representation of attributes. In Section 5.3 we will see that this is important, because shared representation is favourable with respect to the cost of consistency maintenance. For OCRs this is not feasible, because it requires us to analyse the body of OCRs to understand the intention of the dependency, and infer that shared representation is possible.

The *second* main difference lies in that the timing model must be *explicitly* expressed when using OCRs, because the signature and the body of the OCRs must be specified according to the chosen timing model. This may be seen by comparing the OCRs in the Figures 4.13 and 4.14. For ACRs the timing model is not expressed or implicitly given, and the implementation may be tuned to the request profile.

The *third* main difference is that OCR must have the non-derivability mutation policy hard-coded (using the standard ones, null values and mutation denial, or some user-specified ones), while ACRs may be implemented according to a general policy. We are not sure how valuable this really is, because we believe that very often the choice of which policy to choose is dependent on the intended semantics of the attributes. E.g., given the equation \( x = y_1 + y_2 \), which has the ACG \( (x) \leftarrow \leftrightarrow (y_1, y_2) \), and we know that \( x \) is probably never going to be mutated. \( x \) merely represents the sum of the other two attributes, and thus denial of mutation of \( x \) is the most reasonable policy. For ACRs this may be specified using a non-derivability mutation symbol, and the system will take care of generating the appropriate code for this.

The *fourth* main difference is the control of termination of consistency maintenance. As seen in Section 4.3.7 ACRs introduce cycles between attributes. This cyclicity
gives rise to non-terminating consistency maintenance. For ACRs we avoid the cycle problem by introducing “visited” and “dirty” flags like those shown in Section 4.4.5. For OCRs this is not so easy. The reason for this is the freedom of the OCRs, by being arbitrary pieces of code specified by the user. This means that the user himself must specify in each OCR a method of ensuring the termination of the consistency maintenance. This problem is equivalent to the problem of database triggers which through interaction do not terminate.

We think that maintaining object consistency should not degrade the efficiency of the database so much that the user chooses to live with inconsistencies and manual consistency maintenance. Our intention is that ACRs should mainly be used, because they are flexible with respect to optimisations and management. OCRs do not provide for many optimisation opportunities. As another corroboration that the ACRs are reasonable for specifying dependencies, we will compare our class evolution framework to subclassing in Section 5.7.3. There, we find that the relationships between attributes of subclasses and superclasses are special cases of ACRs.

There are situations where OCRs are favourable to ACRs. One case is when the schema designer does not want to understand the representation of objects of a class, and the class has easy-to-understand operations, e.g., the class represents a well-known abstraction. The choice of using ACRs vs. OCRs is dependent on the nature of the objects being modelled. ACRs are applicable for fairly static attribute domains. In a more dynamic setting, where attributes are links to objects appearing and disappearing from the system, there may not exist a clear function between the attributes. The idea is that the body of an OCR may uncover this – what cannot easily be described by a mathematical function between the attributes.

To illustrate this we will include an example from [HKP+88]. This example shows a class Stack and a class Set which propagate objects to each other, i.e. Stack ∆ Set. One possible “consistency relation” is this: “A set p and a stack q are consistent if they contain the same objects as elements” [HKP+88]. Assuming the typical insert, delete, push, and pop operations of these classes, when using OCRs, Figure 4.16 and 4.17 outline a realisation of the “consistency relation”. Figure 4.16 shows the OCRs connected to the operations of the stack, while Figure 4.17 includes the OCRs corresponding to the operations on the set. In this example we assume that each object has a dirty flag in each class it is a member of. In a more complex example we might have the need for dirty flags for each attribute of an object.

There are several reasons why this example may be hard to express using ACRs. The implementations of the stack and the queue probably make use of linked lists, arrays, or other composite data structures. If the set is implemented by a fixed size array of elements, and the stack is implemented using a linked list where there is a list head attribute in each stack object, there may not be a clear function. However, as seen, the object consistency is rather easily expressed by OCRs.

In this section we have given favourable comments to the use of ACRs compared to OCRs. In Section 5.7.4 we discuss how object consistency maintenance relates to the important concept of encapsulation in object-oriented systems. As encapsulation usually means hiding the attributes and exposing the operations, OCRs have a larger potential than ACRs for respecting encapsulation.
after push (elmDmn e) {
    mark stack as visited;
    if set is unvisited
        insert(e);
    mark stack as unvisited;
}

after pop (elmDmn e) {
    mark stack as visited;
    if set is unvisited
        If last on the stack
            delete it from the set;
    mark stack as unvisited;
}

Figure 4.16: The OCRs connected to Stack's operations.

after insert (elmDmn e) {
    mark set as visited;
    if stack is unvisited
        push(e);
    mark set as unvisited;
}

after delete (elmDmn e) {
    mark set as visited;
    if stack is unvisited
        delete all occurrences of e from the stack;
    mark set as unvisited;
}

Figure 4.17: The OCRs connected to Set's operations.

4.5 Class evolution in the framework

4.5.1 Class evolution definition

The previous sections introduced the different components of the framework. Sum-
marised, the class evolution framework has the following components:

1. Basic object model
2. Intent hierarchy
3. Extent graph
4. $\exists$ functions
5. Attribute consistency relations
6. Operation consistency relations

We have named this set a framework and not a data model. The reasons for this is that we try to cover the core of many of the existing object-oriented data models. The main contribution of the framework is to show some extensions/changes of the interclass dependencies of the existing data models so as to make them suited for a wider range of class evolution situations.

We will now define what class evolution means in this framework. We will start by defining intent and extent evolution, which are important components of class evolution. At the end we will argue for the separation of intent hierarchies and extent graphs by discussing their characteristics with respect to evolution.

**Definition 4.11 (Extent and intent evolution)** Given an extent graph $D_1$ and an intent hierarchy $I_1$

- An *evolution of an extent graph* $D_1$ means addition of extents and EPLs, thus creating a new extent graph $D_2$, such that $D_1$ is a directed subgraph of $D_2$.

- An *evolution of an intent hierarchy* $I_1$ means addition of intents and inheritance links, creating a new intent hierarchy $I_2$, such that $I_1$ is a directed subgraph of $I_2$. $\square$

These two definitions essentially say that evolution means *adding* intents/inheritance links and extents/EPLs to an existing intent hierarchy and extent graph, respectively. Note that the “added” extents may already be populated upon evolution, such that an extent evolution becomes connecting existing extent graphs by appropriate EPLs.

**Definition 4.12 (Class evolution)** A *class evolution* is an evolution of an extent graph and its corresponding intent hierarchy, such that there is

- A one-to-one and onto function (bijection) from the added extents to the added intents.

- An $\exists$ function for each added EPL.

- A (possibly empty) set of ACGs with accompanying implementations of the mathematical functions for each added EPL.

- A (possibly empty) set of OCRs for each added EPL. $\square$

This means that class evolution in our framework is only *additive*, i.e. classes and EPLs are never removed. Thus, clients will never have problems with classes that are modified or removed. However, if an existing class has no clients, it could be possible to remove it. Additive evolutions being transparent to clients will be of
special interest to us. The definition of class evolution transparency is given in Section 4.5.2.

As classes may already be populated upon class evolution, we may have to propagate existing objects between extents. E.g., if the class hierarchy consisting of the single, populated class $C_1$ is evolved creating the class $C_2$ and the EPL $C_1 \Rightarrow C_2$, all objects existing in $C_1$ must be manually added to $C_2$. Objects created after the class evolution will be automatically propagated by letting the create operation in $C_1$ automatically trigger an add operation of $C_2$.

For a given extent graph and its corresponding intent hierarchy, there is a one-to-one and onto function from extents to intents, but there does not have to be correspondence between the EPLs and inheritance links. Intent hierarchies are directed acyclic graphs (DAGs), while extent graphs are general directed graphs. Thus, an extent graph and its corresponding intent graph do not have to be isomorphic. If we introduced cycles in the extent graph, they clearly cannot be isomorphic.

In Figure 4.3 we saw that the extent graph and the intent hierarchy are not isomorphic: e.g., the inheritance link between Person and FacultyStudent is in the opposite direction to the EPL. The intent hierarchy indicates partially the creation order of the classes, while the extent graph does not. The intent hierarchy indicates the creation order, since when viewing an inheritance link as a relation between two intents, the intent hierarchy is a partial order of classes.

Class evolution is related to the evolution of object-oriented design. One of the main issues in object-oriented modelling is that the “distance” between design and implementation of a system should be small; e.g., like in responsibility-driven design [WWW90]. This means that class hierarchies in object-oriented databases are meant to be close to the conceptual view of the system being modelled (often named the real world). Thus, the evolution of class hierarchies is very dependent on the evolution of the design of the schema. If the design of the schema is comprehensive and the domain being modelled is fairly static, the resulting class hierarchy is expected to be rather static. But even if the schema is “complete” before objects and applications are created, there may be substantial evolution of the class hierarchy [Sjø93].

The reason to split the intent and extent dimensions from the traditional class hierarchy is that these two concepts have different change characteristics. Intent inheritance and intent hierarchies are practical mechanisms for reusing and organising intensional descriptions of objects; they inherently evolve “downwards”. Due to this, intent hierarchies cannot have cycles. Extent graphs do not have the non-cyclic constraint and are much more independent with respect to when abstractions are discovered. The EPL $C_2 \Rightarrow C_1$ does not tell which class ($C_1$ or $C_2$) was created first. Conversely, if we have an inheritance link from $C_2$ to $C_1$, it means that $C_2$ has been created based on and after $C_1$. Extent hierarchies may, however, evolve in all directions. Literature [SN88, Ped89, WWW90, Cas91, Mey92] indicates that classes and their abstraction are likely to be discovered in the opposite direction of the “subclass” direction. This results from humans’ way of making generalisation after they have seen several concrete examples. If the class hierarchy is discovered (and developed) in a traditional “top-down” fashion, the extent graph and the intent hierarchy might of course be isomorphic.
Note that in terms of the framework, we use class evolution, and not schema evolution. In a particular basic object model, a schema will typically be a collection of classes, where all referenced classes (e.g., through links, inheritance etc) are within the same schema. When applying the framework in this context, a schema evolution will be a set of class evolutions, where all classes involved are in the same schema. This means that in schema evolution an object cannot be propagated to classes outside the schema. In Section 5.6.3 we will see an application of the framework where we constrain the context of the propagation of objects by the use of a versioning mechanism.

Our definition of class evolution does not cover for modifications of the general class structure, i.e. modifications to the extent graph. The input extent graph(s) to an evolution is/are always a subgraph of the resulting extent graph. Modifications are not treated in this thesis, because they introduce some additional questions to be answered. E.g., given the three classes CoastLine, Line, and Border, and the EPL CoastLine ⇒ Line. If the EPL is found to be wrong, and the correct EPL should be CoastLine ⇒ Border, a modification of the extent graph is required. Adding an EPL between CoastLine and Border is a normal class evolution, while removing the old EPL is a class hierarchy modification. An interesting question is: What should be done with the objects which have been propagated from CoastLine to Line? Most probably these should be removed from Line. These objects may be found because they are the intersection of objects in those two extents. Another possibility is to let them be as they are, i.e. to retain them in both classes. In this situation it may happen that they will not be kept “up-to-date” in Line, because the client who used to keep them “up-to-date” is connected to CoastLine.

### 4.5.2 Class evolution transparency

One of the main aims with our class evolution constructs is to let class evolution be transparent to clients of classes. We will now define what class evolution transparency means in our framework.

**Definition 4.13 (Class evolution transparency)** A class evolution resulting in an extent graph \( D \), is transparent if all objects (both the objects created before and after the evolution) in the extents of \( D \) are consistent upon all observer requests.

We consider our definition of class evolution transparency to be a generalisation of Ahlsén et al’s class change transparency [ABB+83], because it did not consider transparency of general evolution of classes; they only considered how to make “changes” to one class transparent by creating versions of the class. By generalising their “change” of a single class to all kinds of additive evolution that can described by EPLs, 3 functions, ACGs, and OCRs, we can generalise the transparency definition accordingly.

If we assume that all attributes are mutated regularly, we can identify two levels of class evolution transparency: Level-0 and level-1. **Level-0 class evolution transparency**, which is the weakest form, means that the added EPLs may result in null
values to appear or in denial of mutation. I.e., there are partially derivable or non-
derivable ACRs involved in the class evolution. In these cases, we may discuss if the
evolution is really transparent. The reason for this is that clients that used to work
may have to stop mutating attributes, or to handle null values. The introduction
of null values or denial of mutation must be seen as a “compromise” between the
department of the different classes involved in the EPL. As long as null values can be
handled in a meaningful way [Gra91], the evolution is transparent.

<table>
<thead>
<tr>
<th>FacultyStudent</th>
<th>⇔</th>
<th>UniversityStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td>←→</td>
<td>(name)</td>
</tr>
<tr>
<td>(startYear)</td>
<td>←→</td>
<td>(enrolmYear)</td>
</tr>
<tr>
<td>(department)</td>
<td>←→</td>
<td>(faculty)</td>
</tr>
</tbody>
</table>

Figure 4.18: Example of level-0 class evolution transparency.

In Figure 4.18 we have illustrated Level-0 class evolution transparency\(^2\). To the
extent graph consisting of the class FacultyStudent, a class evolution has resulted in
the added classes UniversityStudent and the EPL FacultyStudent ⇔ UniversityStudent.
The problem introduced here is the non-derivable ACR from faculty to department.
If the faculty attribute is mutated, we either have to deny the mutation or to insert
a null value in the department attribute.

*Level-1 class evolution transparency* is when a class evolution does not result in null
values or in the denial of mutation. This means that all involved ACRs are totally
derivable. The evolutions are really transparent since existing clients will continue
to work, and new clients will work. This is the strongest form of class evolution
transparency. Figure 4.8 may be read as an example of level-1 class evolution trans-
parency. If the PolarLoc class and the EPL CartLoc ⇔ PolarLoc are added to the
extent graph consisting of CartLoc as a result of a class evolution, the class derivation
is level-1 transparent.

Note that the definition of class evolution transparency relied on the definition
of object consistency, which again is defined by satisfaction of ACRs. We rely on ACRs
and not on OCRs for this definition, because ACRs are specifications of consistency,
while OCRs are implementation which may be used for any purposes, and may have
unpredictable behaviour (cf. Section 4.4.6).

4.5.3 Class derivation and integration

We see two main ways of applying our framework to class evolution, which are
roughly categorised as follows:

\(^2\)Note that the names of the classes used in this example are indicating a Student class used
by the faculty level and a Student class used by the university, where a faculty is understood in
the European tradition, i.e., as an organisational unit of departments. In a more realistic example
both classes would have the name Student, but in two different databases.
Definition 4.14 (Class derivation and integration)

- *Class derivation* is a class evolution which adds new classes and extent propagation links to an existing connected extent graph, resulting in another connected extent graph. The added classes are created based on the classes of the existing extent graph.

- *Class integration* is a class evolution which by extent propagation links connects two independently created, connected extent graphs, resulting in one connected extent graph. □

Chapter 5 applies our framework to class derivation, while Chapter 6 applies it to class integration. Class derivations are typically small, incremental changes to class hierarchies. This covers adaptive and corrective class evolutions. Chapter 5 will be concerned with how small, incremental changes are supported practically. In class derivation we assume the 3 functions to be implicitly given by the creation and addition operations.

Class integration covers those situations where the classes of the two extent graphs are created independently. In Chapter 6 we will focus on several issues connected to class integration. Firstly, objects being created independently, may be modelling the same conceptual object. Thus, upon class integration we may have to integrate objects as well. This means that we have to provide the 3 functions explicitly. Secondly, as class integration is more complex than class derivation, we will develop a methodology for class integration specially suited for our framework. In this methodology we will be concerned about giving the schema designer help in finding commonalities between the classes to be integrated. Thirdly, we will treat “interference problems” resulting from multiple EPLs and multiple ACRs to the same attributes.

### 4.5.4 Examples of typical class evolutions

We will now show four typical examples of class evolutions, being illustrated by extent graphs in Figure 4.19.

1. **Generalisation**: Adding a new class $C_G$ with one or several EPLs going into it. Given the classes $C_i$, $i = 1..N$, we create a new class $C_G$ and the EPLs $C_i \Rightarrow C_G$, $i = 1..N$. We say $C_G$ is a *generalisation* of the classes $C_i$, $i = 1..N$.

   There are two different semantics of generalisation. In some cases it may be necessary to “join” objects from the different classes. After the terminology of [KDN90] we name this *role generalisation*. This means that an object in $C_G$ may have corresponding objects in all classes $C_i$, $i = 1..N$. The other situation is where objects are not “joined”, but the extent $C_G$ is the disjoint union of the extents of $C_1$ and $C_2$. After [KDN90] we name this *category generalisation*.

2. **Specialisation**: Adding a new class with one or several EPLs going out from it. Given the classes $C_i$, $i = 1..N$, we create a new class $C_S$, and the EPLs $C_i \Leftarrow C_S$, $i = 1..N$. We say $C_S$ is a *specialisation* of the classes $C_i$, $i = 1..N$. 

100
3. **Class versioning:** Adding a new class $C_V$ with an EPL $C_1 \Leftrightarrow C_V$ to an existing class $C_1$. We say $C_V$ is a **version** of $C_1$.

4. **Class connection:** Two existing classes will be connected by adding suitable EPLs between them. This may be in either or both directions. Given two classes $C_1$ and $C_2$ which are populated. When creating EPLs, either $C_1 \Rightarrow C_2$, $C_1 \Leftarrow C_2$, or $C_1 \Leftrightarrow C_2$, we name this situation **class connection**.

Note that we allow for all kinds of class evolution possible to describe in our framework, but these are four typical ways of utilising the class evolution framework. Other approaches provide for some of these evolution situations by built-in evolution “operators”, e.g. [SN88, KDN90].

Class versioning is an example of class derivation, while class connection is an example of class integration. Generalisation and specialisation may either be class derivation or integration, depending on the actual situation. E.g., creating a generalised class of another class is class derivation, while creating a generalised “view” class to many independently created classes is class integration.

Given the generalisation situation, where $C_G$ represents the generalisation of $C_1$ and $C_2$, i.e. $C_1 \Rightarrow C_G$ and $C_2 \Rightarrow C_G$. The distinction between role and category generalisation becomes apparent by considering the two $\exists$ functions:

$$\exists_1 : \text{id}_1 \rightarrow R_A, R_A \subseteq \text{id}_G$$

$$\exists_2 : \text{id}_2 \rightarrow R_B, R_B \subseteq \text{id}_G$$

If $R_A$ and $R_B$ are disjoint, we have a **category generalisation**. If they are overlapping, we have a **role generalisation**. Anyway, the semantics of EPLs is still the same: if
an object is a member of $C_1$, it is also a member of $C_G$, or it has a corresponding object in $C_G$.

### 4.5.5 Class evolution and abstract classes

A common semantic extension to classes is to distinguish between abstract and concrete classes. We will now describe how such an extension can be incorporated in our framework, and how it interferes with class evolution.

An *abstract class* is a class which cannot be used to *create* objects, while a *concrete class* may be used to create objects. The concept of abstract classes is meant as a semantic construct to force clients to “tell enough” about an object when creating it. E.g., to prevent clients to create objects like “thing”, but force them to create the object of a class representing concrete objects. In our framework, this means that an object may be *created* and *added* in a concrete class, but only *added* in an abstract class.

Following are the typical restrictions to which classes in a hierarchy that may be abstract: an abstract class must have at least one incoming EPL, i.e. it must have at least one class (abstract or concrete) propagating to it. Also required is that an EPL cannot go from an abstract class to a concrete class, i.e. an abstract class cannot propagate to a concrete class. One immediate implication of this is: If we have a cycle in the extent graph, $C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_k \Rightarrow C_1$, all $C_i$ are abstract or all are concrete. Given a cycle of only abstract classes, there must be a concrete class propagation into to this chain, which means that one of the classes in the cycle must have an incoming EPL, coming (directly or indirectly) from a concrete class. In the example in Figure 4.3, these restrictions would mean that only class *Person* is allowed to be abstract. These restrictions means that “leaf classes” are concrete, and classes “higher” in extent graph should be abstract.

As an abstract class is not allowed to exist as a “leaf node” in the graph, we will show how to allow for creating class hierarchies containing abstract classes. To prevent this we must always introduce an abstract class together with one of its concrete “subclasses”. When introducing the abstract class $C_A$, it must be introduced together with $C_S$ and the EPL $C_A \leftarrow C_S$. Later, several other concrete specialised classes of $C_A$ may be created. However, a common situation is that an abstract class $C_A$ is found after defining a set of concrete classes $C_1$ through $C_N$, such that adding the abstract class becomes a generalisation. The opposite situation (i.e. specialisation) may require that a concrete class be made abstract (after creating a set of concrete specialisations). This interferes with class evolution transparency: There might be clients that are dependent on creating objects of the class that used to be concrete. If these dependencies are registered, and no clients dependent on the creation capability have been found, the class may be allowed to be made abstract. If it is allowed for making a class $C_1$ abstract in connection with adding the specialisation $C_S$, all objects existing in $C_1$ prior to the evolution need to be added to $C_S$.

Thus, abstract and concrete classes may be added to the framework provided that
4.6 Comparisons

This section makes some general remarks to the framework, and compares it with other suggestions for separate hierarchies in object-oriented data models. Further on, we compare it with views in database systems and with object models facilitating explicit relations. The two following chapters (5 and 6) compare the framework to other specific class derivation and integration approaches.

4.6.1 General remarks

The main intention with the framework is that it should support various forms of transparent class evolution. This is done by factoring out the essential and common components of various evolution techniques, and make them explicitly available. In Section 4.5.4 we saw some examples of how we apply the framework on the extent level to cover for various evolution situations. An important quality of the framework is that 3 functions and ACRs/OCR may be applied to all of these evolution situations. We think this generalisation is a considerable enhancement compared to the various existing evolution techniques. E.g., a class specialisation may be done after both the general and special classes are created and populated. Typically the subclass has all properties of its superclass. In our framework the schema designer may describe the connection between the subclass and superclass in more detail using ACRs and OCRs. This provides the schema designer with flexibility, instead of the rather strict predefined subclassing mechanism. In Chapter 5 and 6 we will show several examples of how the framework may be used to cover for various evolution situations.

Another important quality of our class evolution framework is that it is additive, in the sense that classes are never removed or edited as they are. This is a typical property of versioning systems, where previous versions of the entities in focus are always retained. The framework is not intended to be a pure versioning system. Classes are retained because it is the obvious approach to class evolution transparency. The preservation of classes works fine when an existing class is found not to be adequate, and should be replaced by a new better class. This means that a new class is created with proper “connections” (EPLs, ACRs, and OCRs) to the old class. As an extension to the evolution in the framework, we could allow for classes to be removed from the system when there are no clients left. This can be seen as a combination of transparent class evolution and class modification.

We have not addressed the class modification and class hierarchy modification capabilities in the framework. However, as will be further discussed in Chapter 8, in a complete set of tools for class evolution and integration, several of these evolution techniques could be provided together with transparent class evolution.
4.6.2 Multiple hierarchies

According to [ZM90], an object-oriented database system may potentially have three different “class” hierarchies. These stem from three different aspects of classes:

**Specification:** Based on the properties of the interfaces of classes (the specifications), classes may be organised in specification hierarchies. Such hierarchies describe which objects may substitute other objects without type conflicts, and are mainly used for static type checking.

**Implementation:** The implementation of classes (representation and bodies of operations) may be defined by inheriting such properties from other implementations. This results in implementation hierarchies.

**Extension:** The extension of a class may be defined to be a subset of the extension of another class. This results in extent hierarchies.

In our framework we have merged the first two hierarchies, such that we have two separate hierarchies. It was not necessary to separate the first two hierarchies in order to meet our goal of unifying the different transparent class evolution techniques. Separating specification from implementation is interesting in contexts where several alternative implementations may exist for a each specification. An example of this is from federated database systems, where a specification on the federated level may have different implementations in the different component database systems.

In object-oriented database systems the support of these three qualities is varying from system to system, and they are usually merged into one hierarchy. Most object-oriented database systems support a notion of extents of classes. However, some require the extent to be explicitly defined, e.g., ObjectStore [LL91] and O₂ [D+91]. Extent inclusion usually follows the class hierarchy, yet still, there are approaches where extent inclusion has to be explicitly requested in queries, e.g., POSTGRES [RS87] and [BNPS92]. According to Maier [Mai91], the extent notion of classes should be more flexible: classes should be allowed to exist without explicit extents, classes should be allowed to be partly computed and partly represented collections of objects. In this thesis we assume that all classes have their own extent, which may be partly represented and partly computed. However, we do not deny that there exist classes without the need for explicit extents. However, if explicit extents are not supported, it may be expensive to propagate the extents eagerly upon class evolution. In this situation it is probably most wise to delay the propagation until objects (individually) are requested.

It is widely recognised that class hierarchies in object-oriented programming languages are used for multiple purposes, and would take advantage of separated hierarchies [Sny86, Lun89, CHC90, Por92]. The typical example of this is the separation of specification and implementation into separate hierarchies. This development has an interesting historical background. From the introduction of classes in Simula-67 [DMN70], it grew out two branches of programming language tradition. The first tradition introduced abstract data types (ADTs), where it was focused on separating between specification and implementation. Typical languages here were Mesa
[GMS77], CLU [LSAS77], and Alphard [WLS81]. The other branch focused on
organising classes into inheritance hierarchies and the use of dynamic binding. Typical
languages here were Smalltalk-80 [GR83], C++ [Str86], and Eiffel [Mey88]. The in-
troduction of both specification and implementation hierarchies into programming
languages may be seen as a merger between these two branches. Examples of such
languages are Trellis/Owl [SCB+86] and Emerald [BHJL86]. From this separation
they gain the ability to have both the substitutability property of specification hi-
erarchies, and the code reuse ability of implementation hierarchies. Further on, it
avoids conflicts between the aspects of code reuse and substitutability.

In the context of database integration, it has been recognised that separating the
structural and semantic dimensions of classes (in the Dual Model) allows for integ-
rating classes that are structurally similar, but semantically different [GPN91b].
Classes are considered semantically similar if they model the same objects in the
application domain [GPN91a]. The structural integration is seen as an alternative
to integration by generalisation, and is of considerable interest to (top-down) view
integration, according to [GPN91a].

The main problem with multiple hierarchies is that they introduce more complexity,
resulting in more aspects to understand for developers and users. The main advan-
tage is that the separation makes the model more flexible – it can be suited towards
many needs. However, if the schema designer cannot handle the flexibility, it may
result in unmanageable complexity.

### 4.6.3 Updatable views

The classes of our framework may be understood as updatable views. We will now
relate the classes of our framework to the general model of consistent (updatable)
views in [GPZ88]. In Section 7.3 we present a design of the framework by updatable
views, which are based on an “objectified” version of a relational algebra

[GPZ88] defines a static view as a triple \((A, B, f)\), where \(A\) is a base data abstraction,
\(B\) is a view data abstraction, and \(f\) is an abstraction function between the states
of \(A\) and \(B\). A (dynamic) view is defined as a quadruple \((A, B, f, \tau)\), where \(\tau\) is
a translator between the update operations of \(B\) and the set of update programs
of \(A\). The view \(B\) is consistent if knowing the effect of an update program on \(B,
implies that the effect of the corresponding update on \(A\) is known. This relates to
our framework as follows: \(A\) and \(B\) correspond to two classes \(C_1\) and \(C_2\) having the
attribute sets \(X\) and \(Y\), respectively. \(C_1\) and \(C_2\) are connected by an EPL having
the ACG \(X \oplus Y\). \(f\) corresponds to the ACR from \(X\) to \(Y\). View consistency implies
that given a mutation operation on \(Y\) of \(C_2\), we know the corresponding mutation
to \(X\) of \(C_1\). Thus, maintaining object consistency is essentially the same problem
as maintaining consistent views. Shared representation in our ACGs corresponds to
the normal view situation, where the view is represented as a query that is evaluated
upon request. Separate representation in our ACGs corresponds to materialisation
of views [BLT86]. In this case, our OCRs maintain object consistency similar to how
the production rules of [CW91] maintain the consistency of materialised views.
The are some differences between the ways typical views and our classes are utilised. As an example of this we use the Model-View-Controller (MVC) architecture for user interfaces (presented in [Cox86]). This architecture divides a user interface into three “boxes”: model, view, and controller. A model is a fixed description and representation of the objects being modelled, while views represent different ways of “looking at” the objects. MVC assumes that all entities that will be modelled in the system can be anticipated. In our framework, classes may be both models and views at the same time: Classes represent and present the objects created locally, and they may represent objects created in other classes. This allows us to evolve the model and the views at any time.

4.6.4 Explicit relations

There have been several suggestions for the introduction of explicit relations\(^3\) into object-oriented databases [Bra91, DG91, AGO91]. Typical arguments for introducing explicit relations are [BO92]:

- Relations are higher level constructs than links. They are on the same level as classes.
- Semantics belonging to connections between objects is kept explicitly, and is not intermixed with the objects themselves.
- Relations are symmetric, thus the inverse-link property is automatically maintained.
- If there are properties belonging to the relationships between objects, deciding where to put them is not a problem.
- It becomes possible to add and delete relations without modifying the participating objects.
- The implementation of relations may be tuned to the access profile.
- It avoids an incorrect implementation of relationships.

We think that explicit relations and our model of classes cover much the same needs. Explicit relations are used to model the roles objects play in connection with other objects. The framework allows for objects to be members of multiple “most specific” classes. Often these multiple most specific classes represent the different “roles” the objects play. As an example of this we present a role generalisation example (presented in more detail in Section 6.2.1): The two classes FacultyStudent and UniversityStudent are role generalised into Student. Thus, they are considered as different roles of Student. In our framework this will be modelled by letting the two “role classes” propagate objects to Student. With explicit relations this could be modelled by having two relations: at_faculty between the two classes Student and Faculty, and at_university between the two classes Student and University. Thus, what

---

\(^3\)We use relation and relationship similarly as we use class and object.
is modelled by explicit relations, will in the framework be modelled by “leaf” classes in the class hierarchy.

We have chosen to avoid explicit relations in the framework of the following reasons:

- Explicit relations are not common in object-oriented models. Thus, it would be in conflict with our intention of having a basic object model resembling the core of most object models (cf. Section 4.1.4).

- Explicit relations and multiple “most specific” classes are often alternative modelling constructs.

- To support symmetric links, we have introduced the inverse constraint for link attributes.

However, an important influence from our previous work on explicit relations [Bra91, BO92] may be seen in the framework. Unlike existing class evolution approaches [KDN90, SN88, Ber91c], which embed the evolution information into the classes (or views) themselves, our framework externalises all evolution information into EPLs and their associated information (3 function, ACR, and OCRs). This means that our framework has much of the same advantages compared to the existing approaches, as explicit relations have to embedded references. E.g., evolution information is not intermixed in the classes themselves. We think this is an important contribution, because all classes become equally complete classes, and not special-purpose classes with semantics depending on when they were created. This makes the model uniform, e.g., all classes may equally well be evolved, used to create objects from, etc.
Chapter 5

Class Derivation

This chapter will treat incremental “changes” to classes. The basic situation is that a client is not satisfied with a class, and wants to make changes to it. This is done by letting the user “edit” the class, which in our framework means creating a new class and specifying the appropriate EPLs and ACRs/OCRs.

After a brief introduction, we will show a couple of examples of class derivations. We develop a cost model for deciding upon the implementation of consistency maintenance of objects. Together with an analysis for representation of objects upon derivation, the cost model is used in a tool for “editing” classes. The two last sections present various extensions and comparisons with other class derivation approaches and related concepts.

5.1 Introduction

A class derivation extends an existing extent graph and intent hierarchy with a new class, such that a new extent graph and intent hierarchy appear. Thus, a class derivation introducing a new class $C$, consists of the following steps:

1. Add a new extent for $C$ to an existing extent graph by creating appropriate EPLs.
2. Create a new intent for $C$, possibly by inheriting properties from existing intents.
3. Define the necessary ACRs and OCRs.

A class derivation focuses on the creation of a new class based on an existing class, named the input class. This differs from a general class evolution by not considering user-specified $Ω$ functions, and that there are only one input class. This means that we in this chapter neither consider problems of integration of properties of classes, nor integration of existing objects. Due to the single input class, we consider ACRs to be most relevant for maintenance of object consistency. This is because the properties
of the new class are developed based on the properties of a single, existing class. However, in Section 5.2.2 we will show an example involving OCRs directly.

A class derivation falls into one of these three categories:

**Class versioning:** This is the archetypical example of class derivation, where an existing class (or a version of class) \( C_1 \) is derived into \( C_2 \), a new version of the class, giving the EPL: \( C_1 \leftarrow \rightarrow C_2 \). \( C_2 \) is intended to replace \( C_1 \) for the clients of the class which requested the changes. If there are other clients of \( C_1 \), it must be retained such that conversion of these clients is prevented. If there are no clients left of \( C_1 \), e.g., when the only client is the one requesting the changes, \( C_1 \) may be discarded with respect to consistency maintenance of objects. I.e. there is no use in propagating objects and changes to a class having no clients.

Note that in class versioning the different versions of the class will probably share name. This means that only one of the versions will be visible for each client of the class. In Section 5.6.3 this will be shown in the context of the EPOS database [Lie90], where each version of a class will be tagged with a *visibility* (logical expression), such that at most one version will be visible in each transaction working towards the database.

**Specialisation:** A new class \( C_2 \) is developed as a specialisation of an existing class \( C_1 \), resulting in the EPL \( C_1 \leftarrow \rightarrow C_2 \). This is typically done by subclassing in traditional object-oriented data models.

**Generalisation:** A new class \( C_2 \) is developed as a generalisation of an existing class \( C_1 \), resulting in the EPL \( C_1 \Rightarrow C_2 \). Note that a class derivation being a generalisation, only involves generalisation of a single class. Generalisations of several existing classes are more troublesome, and are treated in Chapter 6.

Note that although a class derivation will fall into one of these categories, it does not necessarily mean that it is intended as a generalisation, specialisation, or class version. An EPL only tells something about the set relationship between the extents of the classes, while the names we have put on the different derivation categories indicate the semantics of the evolutions.

### 5.2 Applying the framework to class derivation

#### 5.2.1 Subclassing

We will now show how the subclassing example from Section 3.2.1 is expressed using the framework. In Figure 5.1 we have used the syntax of Figure 4.2 to express that the intents of \texttt{CHeader} and \texttt{CImpl} both inherit the intent of \texttt{CSource}. \texttt{CHeader} inherits all properties except \texttt{startEditor}, which is (re-)defined in \texttt{CHeader} itself. \texttt{CImpl} inherits all properties of \texttt{CSource}. The following two EPLs and their associated ACGs "connect" the extents and attributes of \texttt{CHeader} and \texttt{CImpl} to \texttt{CSource} as in Section 3.2.1:
class CSource {  
public:  
    startEditor ();  
    setOf <string>  
    setOf <link <CSource>>  
private:  
    string  
    date  
    longfield  
author;  
lastUpdate;  
contents;
};

class CHeader : CSource {  
public:  
    startEditor ();  
private:  
    link <CImpl>  
impl;
};

class CImpl : CSource {
};

Figure 5.1: Subclassing by class derivation.

When simulating subclassing with the constructs of the framework, we get totally derivable ACGs between the attributes of the superclass and the corresponding attributes of the subclasses, which should be clearly illustrated in the example above.

The example above is slightly different from the example in Section 3.2.1, because the operations above are statically bound, while the operations of the subclassing example in Section 3.2.1 are dynamically bound (cf. the virtual keyword). In Section 5.6.2 we will show how dynamic binding may be introduced into the framework.

This example includes what is called a soft change in [JC92], which means that only the behaviour of the source class is “modified”. When viewing CHeader as a modification of CSource, all attributes are kept, but the operation startEditor is changed. This evolution is “soft” in the sense that objects of CSource need not have their representation changed due to the new class. This is contrasted with hard changes, which are changes done to the representation of objects [JC92]. In some application areas, e.g., software process modelling, much of the information attached
to classes will be description of behaviour. Hopefully, this means that there will be less “changes” to the structural part of objects. However, when the major part of classes are pure behaviour, class derivation may give large impacts on the process of ensuring that “changes” are behaviourally consistent [Zic91a].

5.2.2 Incompatible, alternative representations

We will now show an example from the domain of modelling geometric objects. This domain is characterised by many different representations which suit different needs. We show two different ways of representing a two-dimensional area. The first representation is a “raw” area-based method, where the objects are represented as two-dimensional arrays of filled/empty information. This representation is shown by the 2dCell class in Figure 5.2. After a while new clients want to have a boundary representation of the same objects, e.g., represented using a list of control points. Such a representation is shown by the 2dBoundary class in Figure 5.3. This class has two operations to view the object: view creates a view with straight edges, while bSplineView creates a spline-based view.

The 2dBoundary class is created as a version of 2dCell, as shown in Figure 5.4. These two representations are both approximations of the “real-world” objects, and they are incompatible in the sense that there is no one-to-one mapping between the representations. It is possible to update the geometric object in both representations using the associated mutation operations.

Because both representations are mutable, these representations are quite problematic. Each time a cell is filled or unfilled in the cell representation, a new line representation has to be created to approximate the cell representation. Each time a control point is moved, added, or removed in the boundary representation, the cell representation will be recomputed. There are possible improvements to this scheme, e.g., incrementally updating the cell representation from a move of a control point, and vice versa.

```cpp
domain planDmn enum {filled, empty};

class 2dCell {
    public:
        view ();
        fillCell (int x, int y);
        emptyCell (int x, int y);
    private:
        planDmn cells[NX][NY];
};
```

Figure 5.2: Cell representation of 2D objects.

112
class 2dBoundary {
    public:
    view ();
bSplineView ();
addPoint (link <Point> newPoint);
removePoint (link <Point> point);
    listOf <contLink <Point>> points;
};

class Point {
    public:
    float x,y;
};

Figure 5.3: Boundary representation of 2D objects.

The OCRs will not request the mutator operations on the other representation, but work directly on the attributes containing the geometric representation. This can be done because OCRs are allowed to break the encapsulation of the classes they are associated in between.

extprop 2dCell <= 2dBoundary {
    ocrs:
    after fillCell (int x, int y) {
        // Compute line boundary from cell representation
    }
    after emptyCell (int x, int y) {
        // Compute line boundary from cell representation
    }
    after addPoint (link <Point> newPoint) {
        // Compute cell representation from line boundary
    }
    after removePoint (link <Point> point) {
        // Compute cell representation from line boundary
    }
};

Figure 5.4: Eager consistency maintenance in the incompatible representations example.
5.2.3 Telescope evolution

To illustrate a known weakness of our framework, we will show an example referred to as telescope evolution in [Mot87, Ber92, Cla92]. This is concerned with information about an object being stored in another object. A telescope evolution is to move this information to the object itself. Consider a class Employee defining addr, which is a link attribute to an object of the class Address. If the attribute, country, is “moved” from Address to Employee (creating the new versions Employee’ and Address’), we would get the following EPL and ACG (in addition to some others, which are not shown):

\[
\begin{align*}
\text{Employee} & \quad \Leftrightarrow \quad \text{Employee’} \\
\text{(addr} \rightarrow \text{country)} & \quad \leftrightarrow \quad \text{(country)}
\end{align*}
\]

This would be an extension to our ACRs. The problem is that objects of Address may be referenced from other places, and thus the dependency could be broken by others. If it was possible to guarantee that the address object would only be used through addr, there would be no problem. Unfortunately, in the general situation such a guarantee cannot be given.

A solution to this problem is to instead of ACRs use OCRs being connected to the operations requesting these attributes. In this solution, an object of Address would have to be connected to a unique Employee object. Another solution is to model address as a record attribute domain, which would have resulted in the following EPL and ACG:

\[
\begin{align*}
\text{Employee} & \quad \Leftrightarrow \quad \text{Employee’} \\
\text{(addr.country)} & \quad \leftrightarrow \quad \text{(country)}
\end{align*}
\]

In this case addr would be fully contained within Employee, and would not have the problem of being mutated from other places.

5.3 Cost model

In Section 4.4.2 we saw that attributes in ACGs could be represented shared, and in Section 4.4.5 we saw how lazy and eager consistency maintenance could be utilised when attributes in ACGs were separately represented. In this section we will develop a cost model for evaluating which representation and consistency maintenance policy to use. The development of the cost model has two purposes:

- Through some examples we will see that shared representation is most efficient. This weighs in with the previous approaches to class versioning utilising shared representation, e.g. [ABB+83, Zdo90].
- When shared representation is not possible, the cost model may be used to decide upon timing policy of consistency maintenance. This may be built into a tool, e.g., in the class editor presented in Section 5.5 or in a schema compiler.
The analysis is dependent on having ACRs as means for describing dependencies. The cost model is put in this chapter, because it is specifically developed for class derivation. A cost model used in class integration would be more complex, e.g., it should probably model the cost of applying ACRs, and it must consider the cost of using \( \exists \) functions. Additionally, the choice of representation is more constrained because of the many input classes already having representations.

The inputs to this analysis are \( N \) attribute sets, which all are related to each other by ACGs, either directly or transitively. For each attribute set we assume the frequency of observer and mutator requests to be given (possible ways of achieving these will be shown in Section 5.4). The following parameters are used in the cost analysis:

\[
\begin{align*}
O_i & \quad \text{Frequency of observation of attribute set } i \\
M_i & \quad \text{Frequency of mutation of attribute set } i \\
C_M & \quad \text{Cost of mutating an attribute set} \\
C_O & \quad \text{Cost of observing an attribute set} \\
C_\mu & \quad \text{Cost of mapping in an ACG} \\
C_F & \quad \text{Cost of checking or setting a \text{``dirty flag''}}
\end{align*}
\]

We assume all attributes in an attribute set to be mutated simultaneously (cf. Section 4.4.3). All the four basic costs given above are assumed to be constants, i.e. independent of attribute domains, number of attributes in the attribute set, algorithm of the mapping etc. The cost of consistency maintenance is dependent on the implementation. In Section 4.4.5 we illustrated the implementation of ACRs by using database triggers. In this section we will assume an implementation where the consistency maintenance is embedded into the attribute request operations. Related to the problem of cyclic dependencies (cf. 4.4.6), we will assume no cost for setting and checking “visited flags”. However, we will include the cost of setting and checking “dirty flags”. The reason for this separation is that “dirty flags” are permanent information, while “visited flags” are only used temporarily during consistency maintenance caused by one operation request.

### 5.3.1 Shared representation

For shared representation, one attribute set must be chosen for representation. The other attribute sets will be mapped back and forth upon requests, using the functions of the ACRs. We choose to represent the attribute set which is most frequently requested, but weighed for the cost of mutation vs. observation, i.e. the attribute set having the highest value for \( C \):

\[
C_i = M_i C_M + O_i C_O
\]

To be able to choose between shared and separate representation, we need the measure of the total cost. If we choose to use attribute set \( i \) for representation, the total cost is:

\[
C_{i \text{\: shared}} = M_i C_M + O_i C_O + \sum_{j \neq i}^N (M_j (C_\mu + C_M) + O_j (C_\mu + C_O))
\]
Here, we have made the assumption that transitive mappings have the same cost as a single mapping. This can be justified by optimising transitive mappings by composing them into one.

As a special case, when \( A_i = A_j \) for all \( i, j \in 1..N \), there is no mapping, and the cost reduces to:

\[
C_i^{\text{shared}} = \sum_{j=1}^{N} (M_j C_M + O_j C_O)
\]

### 5.3.2 Separate representation

For separate representation we have two possibilities, using either eager or lazy consistency maintenance. Eager consistency maintenance means that every mutator request will ensure that the dependent attribute sets are made consistent immediately. Upon observer requests only the “local” attribute set is observed. The total cost for separate representation using eager consistency maintenance is:

\[
C_i^{\text{eager}} = \sum_{i=1}^{N} \left( O_i C_O + M_i (C_M + (N - 1)(C_M + C_n)) \right)
\]

Lazy consistency maintenance delays the consistency maintenance until “dirty” (inconsistent) properties are requested. The most recently mutated attribute set is up-to-date (“correct”). Upon mutator requests the other attribute sets will be marked as dirty. The cost of mutation for each attribute set \( i \) is \( M_i (C_M + NC_F) \), i.e. the attribute set is mutated and marked as up-to-date, and all dependent attribute sets are marked dirty.

When a dirty attribute set is observed, the correct values will be derived from attribute sets being up-to-date. We introduce \( p_i \) as the probability that attribute set \( i \) holds an up-to-date value. This occurs when attribute set \( i \) was the last to be mutated, or when it has already been observed after any of the other attribute sets has been mutated. This means that an increase in the frequency of observation and mutation of \( i \) will increase \( p_i \), while an increase in the frequency of mutation of the other attribute sets will decrease \( p_i \). We use the following expression as an estimate for \( p_i \):

\[
p_i = \frac{O_i + M_i}{O_i + \sum_{j=1}^{N} M_j}
\]

These probabilities are not independent, because there will always be (at least) one attribute set which is up-to-date.

The main problem now is to find the length in the dependency graph that must be searched to find a value being up-to-date. This is dependent on the implementation of derivation and on the connectivity of the dependency graph. We will simplify this situation by finding the average length to search when all attribute sets have equal probabilities for being up-to-date: \( p = (\sum_{i=1}^{N} p_i)/N \). With this assumption, the average length to search will be \( 1/p \). When \( p \) is a small number, the fact that one attribute set always will be up-to-date comes into play. In this situation the average length will be \( N/2 \). This estimate does not consider the individual attribute
sets’ probabilities together with their observer frequency. To make the estimate a bit better, we consider separately the case of observing an up-to-date attribute set. The cost of this is given by $O_i p_i (C_F + C_O)$. The cost of observing the attribute set when it is dirty, is given by the estimate $O_i (1 - p_i) (L_i (C_F + C_\mu + C_M) + C_O)$, where $L_i$ is defined to be the minimum of $N/2$ and $1/p_i'$, where $p_i' = (\sum_{i \neq j}^N p_j)/(N - 1)$.

Thus, the total cost for separate representation using lazy consistency maintenance is:

$$C^{\text{laz}} = \sum_{i=1}^N \left( M_i (C_M + NC_F) + O_i p_i (C_F + C_O) + O_i (1 - p_i) (C_O + L_i (C_F + C_\mu + C_M)) \right)$$

### 5.3.3 Examples of cost analysis

<table>
<thead>
<tr>
<th>$AG_i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_i$</td>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$M_i$</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

$C_1 = C_M + 10C_O$

$C_2 = 5C_O$

$C_3 = 2C_M + 2C_O$

$C_1^{\text{shared}} = 3C_M + 17C_O + 9C_\mu$

$C_1^{\text{eager}} = 9C_M + 17C_O + 6C_\mu$

$C_1^{\text{laz}} = 7.98C_M + 17C_O + 4.98C_\mu + 27.17C_F$

Table 5.1: Cost example 1.

We will now show some examples of cost analysis using these formulas. The figures of the first example are given in Table 5.1. This example includes three attribute sets which are related by ACGs. To the top right we have shown the results of a shared representation analysis. The attribute set having the highest individual cost should be the one to be represented. Depending on the ratio $C_M/C_O$, either group 1 or 3 could be successfully used for shared representation. In this situation we choose to represent attribute group 1. Below in the figure we have shown the total cost of the three different policies of maintaining the consistency. If $C_\mu$ is comparable to the other basic costs, shared representation is the best choice.

In the next two examples we will focus on the differences between eager and lazy consistency maintenance. The figures of the second example is given in Table 5.2. Here, we have illustrated a situation where 5 attribute sets have equal request frequencies, but being far more observed than mutated. In this situation eager consistency maintenance is preferable to lazy consistency maintenance. The reason for this is that mutations are infrequent, and thus the “eagerness” they impose gives little contribution to the total cost.

In Table 5.3 we have shown a third example, where there are 5 attribute sets also having equal request frequencies. In this situation, lazy consistency maintenance is preferable to eager consistency maintenance, because the overhead imposed by the “eagerness” becomes measurable.
<table>
<thead>
<tr>
<th>$AG_i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_i$</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>$M_i$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
C^{\text{shared}} = 5C_M + 500C_O + 404C_\mu
\]
\[
C^{\text{eager}} = 25C_M + 500C_O + 20C_\mu
\]
\[
C^{\text{lazy}} = 24.8C_M + 500C_O + 19.8C_\mu + 525.80C_F
\]

Table 5.2: Cost example 2.

<table>
<thead>
<tr>
<th>$AG_i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_i$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>$M_i$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

\[
C^{\text{shared}} = 50C_M + 50C_O + 80C_\mu
\]
\[
C^{\text{eager}} = 250C_M + 50C_O + 200C_\mu
\]
\[
C^{\text{lazy}} = 133.3C_M + 50C_O + 83.3C_\mu + 350C_F
\]

Table 5.3: Cost example 3.

In all three examples shared representation is most favourable. This is much a result of the assumption that the mappings are embedded into the attribute request operations, and that the cost of such a mapping is constant and independent on the “length” between the represented attribute set and the attribute set being requested. In the two other situations the mappings must be applied for each step in the propagation. An interesting aspect not considered in this cost analysis is the effect of data fragmentation, which may be a result for objects having attributes represented shared in other classes. However, our conclusion is that whenever shared representation is applicable, it should be used.

### 5.4 Derivation and representation

In Section 4.4.2 we showed that by careful analysis of ACRs, shared representation of attributes was possible. That analysis considered objects individually. In this section we will extend this to consider extents and extent propagation, and how this relates to evolution. We only consider tabular representations of objects in extents.

To introduce some of the interesting issues, we show two alternative representations of a simple subclassing situation (adopted from [NM90]). There are two classes: Student having an attribute age, and ScienceStudent having two attributes age and
labDesk. These two classes are connected as follows:

\[
\begin{array}{c}
\text{Student} \quad \Leftarrow \quad \text{ScienceStudent} \\
\text{(age)} \quad \rightarrow \quad \text{(age)}
\end{array}
\]

The first representation scheme, *vertical splitting*, is illustrated in Figure 5.5. It represents all objects being members of a class together in the same table, but it uses shared representation of attributes in the superclass. This approach is quite reasonable when requesting many of the objects in the extent of a class (e.g., associative access). For creation and deletions of objects, all tables may have to be involved.

<table>
<thead>
<tr>
<th>Student</th>
<th>ScienceStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID</td>
<td>age</td>
</tr>
<tr>
<td>34</td>
<td>19</td>
</tr>
<tr>
<td>33</td>
<td>23</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 5.5: Representing subclassing by vertical splitting.

The second representation scheme, *horizontal splitting*, is illustrated in Figure 5.6. It represents objects fully in their creation class, and uses lazy extent propagation. This scheme utilises shared representation of attributes, but the representation is done individually for objects according to their creation class, and not uniformly for all objects in an extent. This approach is reasonable when objects are operated on individually, and not requested through associative queries. Creation and deletion of objects operate only on one table, because objects are not fragmented or replicated. Unfortunately, for associative queries many tables will be involved.

There are two interesting aspects to notice in this simple example. First, the choice of shared representation of attributes is dependent on the EPL between the classes involved. Second, the decision on shared representation of attributes may be done individually for different objects in the same extent.

Given a class \(C_1\), which is derived into the class \(C_2\), where there is an ACG \(A \oplus B\), the following factors must be considered when deciding on the representation of the objects:

**ACG characteristic:** As seen in Table 4.1, the characteristics of the ACG \(A \oplus B\) decide if shared representation is possible at all, and is important for which attribute set to choose for shared representation.

**EPL:** The direction of the EPL between \(C_1\) and \(C_2\) is also of interest. Given that the characteristics of \(A \oplus B\) is such that shared representation is possible in both \(A\) and \(B\), the direction of the EPL constrains the choice as follows:
<table>
<thead>
<tr>
<th>OID</th>
<th>age</th>
<th>labDesk</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OID</th>
<th>age</th>
<th>labDesk</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>19</td>
<td>43</td>
</tr>
<tr>
<td>33</td>
<td>23</td>
<td>97</td>
</tr>
</tbody>
</table>

**Student**

**ScienceStudent**

Figure 5.6: Representing subclassing by horizontal splitting.

<table>
<thead>
<tr>
<th>EPL</th>
<th>Possible sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 \leftrightarrow C_2$</td>
<td>Shared uniformly in either $A$ or $B$</td>
</tr>
<tr>
<td>$C_1 \leftarrow C_2$</td>
<td>Shared uniformly in $A$, or individually in the object's creation class</td>
</tr>
<tr>
<td>$C_1 \Rightarrow C_2$</td>
<td>Shared uniformly in $B$, or individually in the object's creation class</td>
</tr>
</tbody>
</table>

When $C_1$ is derived into $C_2$, there may be existing objects in $C_1$. It may be wise to let the existing objects retain their representation. If $B$ is chosen for uniform shared representation in the first and third situation, $A$ must be converted into $B$ for the existing objects in $C_1$. If shared representation is still possible, according to Section 5.3 it should be chosen.

**Cost analysis for choice of timing policy:** In Section 5.3 we presented a cost model for deciding upon timing policy for separate representation of attribute sets. This was based on the request profile for the attribute sets ($A$ and $B$) as input. An interesting question is then how to obtain the request profile? Generally, this may be obtained by run-time statistics (registering requests, which may be costly itself), by predicting the request profile, or by a combination of these two. A simple prediction approach in our framework is to let the frequencies be proportional to the number of possible clients, being defined as the number of objects in the extents of client classes. The ratio between mutation and observation is predicted by static analysis of the client code. Note that the choice of non-derivability mutation policy influences the request profile. E.g., if the ACG is $A A \xrightarrow{MD} B$, this means that some of the mutation requests to $B$ will be denied, which again means that the frequency of mutation requests for attribute set $B$ will be decreased.

If separate representation is possible after the two first of these factors, it should be chosen. If not, the cost analysis evaluates which timing policy to use.
5.5 Class editor tool

We will now introduce an editing tool which may be of help in class derivation. The class editor takes an existing class as input, and allows the schema designer to interactively edit a copy of the intent of this class. Based on the existing class and the editing instructions given by the schema designer, the class editor will as far as possible generate appropriate ACRs to maintain the consistency of objects between the output and input classes. The class editor only considers situations with one input class. In Section 6.6 we will show some tools which help the integration of several classes, i.e. there are more than one input class.

Based on the input class the class editor performs the following:

- Requests the schema designer for the EPL-connection between the input and output class.
- Recognises the editing operations performed by the schema designer, and from this it establishes ACRs between the attributes of the input and the output classes.
- Based on request profile and the ACRs generated in the previous step, it generates extent propagation implementation, and implementation of the consistency maintenance between attributes.

The first and important step in the editing process is to decide the EPL-connection between the extents of the classes. With one input class $C_I$, we consider the four alternative ways to connect to the new class $C_O$:

<table>
<thead>
<tr>
<th>Semantics</th>
<th>EPL notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>$C_I \leftrightarrow C_O$</td>
</tr>
<tr>
<td>Specialisation</td>
<td>$C_I \leftarrow C_O$</td>
</tr>
<tr>
<td>Generalisation</td>
<td>$C_I \rightarrow C_O$</td>
</tr>
<tr>
<td>Intensional reuse</td>
<td>None</td>
</tr>
</tbody>
</table>

With the basic object model presented in Section 4.1, we identify the following atomic editing operations of interest:

**Add an attribute definition:** If an attribute is added with respect to the input class, this attribute will be independent of the attributes of the input class, and no ACRs are needed. For such an independent attribute, it is wise to give a default value, because objects may be added to this class without any initialisation values.

**Remove an attribute definition:** If an attribute is removed compared to the input class, this attribute will be independent of the output class, and no ACRs are needed.

**Modify an attribute definition:** If only the name of an attribute is modified, e.g., from $a$ to $b$, the following ACR will be generated: $(a) \leftrightarrow (b)$. When
the domain of an attribute is modified, we consider this to be a sign that there
is a semantic connection between the old and the new attribute. Let \( \text{dom}(a_I) \)
be the domain of the attribute of the input class, and \( \text{dom}(a_O) \) be the domain
of the attribute of the output class. By having a global catalogue of all ACRs
with their associated attribute domains introduced in the system, we can give
some support in suggesting appropriate ACRs between the attributes. If there
are any ACRs between \( \text{dom}(a_I) \) and \( \text{dom}(a_O) \), these are good candidates as
ACRs between \( a_I \) and \( a_O \). When the editor recognises the modification of the
attribute domain, it displays the alternative ACRs and the accompanying
implementation of the functions. The schema designer approves or disapproves
the suggested ACRs. In case of the latter, the schema designer must provide
the appropriate ACRs and their implementation functions. Assume that an
attribute \( a \) of \( \text{dom}(A) \) in Figure 4.9 is modified to be of \( \text{dom}(B) \) (shown in the
same figure). As there already exist an ACG between \( \text{dom}(A) \) and \( \text{dom}(B) \),
the class editor suggest this as an ACG for the attributes:

\[
(a) \leftarrow \longrightarrow (a)
\]

If there are several ACGs between the same attribute domains, the number of
occurrences of attributes using the different ACGs could be registered as well,
such that the possible ACGs could be ranked according to previous usage. In
Section 6.4.3 we will show how we may suggest possible ACRs for attributes
being links to objects, in case there are any EPLs either directly or transitively
between the classes of the objects.

**Operation definitions:** Operations being removed, added, or modified, do not
have any impacts on the consistency maintenance expressed by ACRs. If OCRs
are used directly for consistency maintenance, it would also be interesting to
keep track of operation definitions being modified.

All attribute definitions \( A \), which are not “touched” in the editing, will take part in
an ACG \( A \leftarrow \longrightarrow A \). Depending on the richness of the basic object model, there
may be other editing operations as well. E.g., in the basic object model of Section
4.1, another editing operation is to change the visibility mode of an attribute or
an operation. This change has no impacts with respect to ACRs, because they are
allowed to break encapsulation. However, it is of interest for the existing clients
which are going to be modified to become clients of a new version of the class\(^1\).

After the schema designer has committed the editing, the class editor decides upon
the representation of objects, and generates the implementation of the consistency
maintenance. By using the successive steps in the analysis developed in Section
5.4, the class editor is able to decide on the representation of attributes. From this
decision an implementation of the consistency maintenance is generated as shown in
Section 4.4. At this phase class evolution transparency can be skipped by discarding
the input class with respect to consistency maintenance. When the request frequency
to a class is zero, e.g., there are no clients left, it is no reason to maintain the
consistency of the objects of this class.

\(^1\) However, the purpose of class evolution transparency is that existing clients should be allowed
to exist as they are.
A nice way of implementing the editor is to use a generator for language-oriented (structure) editors, e.g., The synthesizer generator [RT88]. The grammar of the schema syntax is given as an input to this generator, and a structure-oriented editor is automatically generated. To the different productions in the grammar, special actions are attached. Thus, according to what is being edited (e.g., attribute domain definitions, operation definitions), the editor invokes actions according to what has been described above.

5.6 Extensions

This section presents various extensions to the framework presented in Chapter 4. First, we will consider extensions to make consistency maintenance more efficient for “transitive dependencies”. Then, we will show how dynamic binding may be introduced in the framework by the tagged link values approach. At the end we will outline class versioning in the EPOS database.

5.6.1 Scaling up with ACRs

When classes are derived in many steps, an object may be exposed to consistency maintenance in equally many steps. The performance of the database system as viewed by a client, would be degraded for each step of derivation. In this section we will outline some ways of preventing such degradation, and how to optimise consistency maintenance when many steps of dependencies are present.

Some class derivations are done to replace classes which were incorrect or did not provide the clients with the right functionality. Such a derivation may be triggered by a single, existing client, or perhaps by a new client. In this situation there may be several other existing clients of the old class, which are not ready to adopt the newly derived class. However, after some time has gone, these clients may be converted to the new class, or they may simply be outdated and “thrown away”. When there are no clients of the old class left, it is no reason to keep on maintaining the consistency of objects in that class. To save the overhead present by this useless consistency maintenance, the EPLs and associated OCRs (or whatever the implementation mechanism is used) may be “switched off”\(^2\). The old class, the EPLs, the OCRs, and the objects may be kept for history tracking reasons, but the existing objects will not be affected by the consistency maintenance. To make this technique practicable, a reliable way of registering active clients and a service to notify the responsible humans must be supported.

In Section 4.3.7 we saw that several ACRs could compose transitive ACRs. The shared representation analysis presented in Section 4.4.2 holds for transitive ACRs as well. This means that many attribute sets may be represented together. E.g., \(A \overleftarrow{\rightarrow} B \overrightarrow{\rightarrow} C\) means that we have the transitive ACG \(A \overleftarrow{\rightarrow} C\).

Thus, if we have \(A \overleftarrow{\rightarrow} B\), any of these three attribute sets may be used as the

\(^2\)E.g., in FOOD [Bra91] triggers may be switched on and off.
basis for representation. If we have $A \leftarrow \cdots \rightarrow B$, $A$ should preferably be used for shared representation. If we choose to use $A$ for shared representation, the idea here is that the mappings between $A$ and $C$ could be as efficient as the mappings between $A$ and $B$. This could be done by either coding the transitive mapping explicitly, or by putting the existing mappings together to compose a new mapping. This means that the cost of having a transitive ACR is essentially the same as the cost of having a directly connected ACR. This implementation technique is assumed in the cost model for shared representation presented in Section 5.3.1. As seen in Section 4.4.2, not all ACR characteristics are suited for shared representation of attribute sets. However, if an ACR which does not allow for shared representation is introduced into a chain of transitive ACRs, all attributes on each side of this ACR may be represented shared.

When having large dependency graphs for attribute sets and separate representation, the efficiency of the consistency maintenance becomes noticeably dependent on the implementation. The dependencies imposed by ACRs are similar to the dependencies imposed in attribute grammars [Knu68]. An attribute grammar is a context-free grammar extended by attaching attributes to the symbols of the grammar. Associated with the productions of the grammar are semantic equations for attributes, which express how an attribute in the production is defined by other attributes. Thus, a semantic equation is essentially the same as a derivable ACR. This observation is interesting, because there is a large volume of work done in optimising the evaluation in attribute grammars.

The main bulk of this work has been done for non-cyclic attribute grammars. ACGs impose trivial cyclic dependencies between attribute sets. According to the definition of object consistency in Section 4.3.6, trivial cyclic dependencies cause no semantic problems, and may be avoided during consistency maintenance by not evaluating ACRs going “backwards” in the consistency maintenance. Attributes may appear in non-trivial cycles as well, but as explained in Section 4.3.7, these cycles do not cause any semantic conflicts as long as the schema designer has done his work correctly. This means that two different paths sharing start and end node in the dependency graph, will give the same result. Furthermore, this means that we may remove an ACR in the cycle, and still get the same semantics.

The straight-forward approach to attribute evaluation is change propagation [RTD83]. The optimisation done here is that the reevaluation of attributes is stopped when the derived value is found to be equal to the old value. For eager consistency maintenance this technique can be adopted directly given that the order of the derivation of the dependent attributes for a given attribute remains the same. For lazy consistency maintenance this technique can also be used, but the attribute sets “optimised away” must be marked as up-to-date before the consistency maintenance terminates. [RTD83] contains a set of other optimisations as well: E.g., removing “transitive checks” on equality of old values and new values: If the first check in a chain of dependent attributes fails to be equal, it is not necessary to check on the others, and a plain rederivation can be done. The Cactis DBMS project done at University of Colorado [HK86, DKH90] has adopted techniques from attribute grammars, and does similar optimisations as the ones that are done in [RTD83].
5.6.2 Dynamic binding and static typing

Dynamic binding between operation requests and operations constitutes much of the power of object-oriented programming [CP89]. In general, the power of dynamic binding comes from its polymorphic nature. Given a link attribute \( a \), which is allowed to hold link values to objects of any class. An operation request to the object referenced by this link attribute, \( a->\text{op}() \) (C++ syntax), will work for any object which is capable of handling the \( \text{op} \) request. This capability is especially important for class derivation, because it allows for incremental development of classes and their objects. Consider the example in Section 5.2.1, where the \texttt{CSource} class having the operation \texttt{startEditor} is given initially. Assume there are some clients requesting \texttt{startEditor} for objects being members of \texttt{CSource}. The new class \texttt{CHeader} having a refined \texttt{startEditor} operation, is connected to \texttt{CSource} by the EPL \texttt{CSource \Leftarrow CHeader}. The intention with this specialisation is that requests to \texttt{startEditor} for objects being members of \texttt{CHeader}, should be bound to \texttt{CHeader}'s \texttt{startEditor}. We will now show how this can be incorporated into the framework. Later, we will show how dynamic binding may be combined with static type checking.

Dynamic binding means that when requesting a property of an object referenced by a link attribute, the binding from the property request to the property is based on the actual object, and not statically on the domain definition of the link attribute. To do this, we extend every link value to also hold an identifier to the intent any request should be bound to, i.e. every link value is tagged with an intent. Thus, a link value \( v \) will be a pair \((OID, intent)\). When requesting a property \( p \) for the object referenced by the link value of \( v \), the lookup for the property will be done in \textit{intent} for the receiving object \textit{OID}. For a link attribute, the \textit{intent} identifier is acquired in one of three ways: when creating an object of \textit{intent}, when retrieving an object by associative access to the class having \textit{intent}, and when copying the value of another link attribute (by assignment or by parameter passing).

In the \texttt{CSource} example above, this means that every link attribute will either be \((OID, CSource)\) or \((OID, CHeader)\). Due to the copy semantics of intent inheritance, there will be no further search than \textit{intent} for a property requested for \((OID, intent)\).

In Section 4.1 it was enough to represent the link value as an \textit{OID}, and the intent used upon property request was the statically defined intent of the domain of the link attribute.

The main drawback of storing the class identifier together with the OIDs in the attribute, is that object conversions become more expensive, because all link values referencing the object must be replaced upon conversion of the object. In [Kim90b] storing the class identifier together with the object references is referred to as an implementation technique to speed up dynamic binding, and the alternative is to represent the class identifier in the object itself. In our object model this tagging is necessary in order to achieve dynamic binding. Thus, it is more than an optimisation trick.

As presented here, dynamic binding is applicable for both operations and attributes. Most object models only allow for operations to be dynamically bound, while attributes are statically bound and are often only available internally to the operations.
of the class.

Dynamic binding allows for a form of polymorphism which is fundamental to object-oriented programming. But it also includes the possible danger that an object does not have a corresponding operation to the operation request, or that the link attribute is null. In both these situation, an error will occur at run-time. Static typing means to detect such type errors at compile-time. In our object model this means that these type errors are detected when new classes with operations are introduced (compiled) into the system.

We define a type to be an unnamed set of properties. The type of an intent is the set of public properties of the intent. Thus, a type is a mere characterisation of the public properties of objects, and is not usable for creation of objects. Each link attribute definition will have a required type. The type of the intent of the object referenced by the link value will be named the actual type. The following type-rule for assignment\(^3\) will be imposed: To allow a link value \(V\) with actual type \(A\) to be assigned to the link attribute \(a\) with required type \(R\), \(A\) must conform to \(R\) (defined in Section 3.2.2).

Dynamic type-checking means that the check is done at run-time upon assignment of new values. Static type-checking tries to detect errors at compile time. For this to be possible the language of the operations must be statically typed, which means that the type of every expression in the language can be determined at compile time [CW85]. Instead of giving an explicit required type, we could also apply type inference to the code of operations to infer the required type. This means finding all properties that objects (referenced by link values) will be requested for, and use these sets as the required types. [PS91a] presents a type inference algorithm for object-oriented programs with inheritance, assignments, and dynamic binding.

### 5.6.3 Bridging to change-oriented versioning

This section has two purposes. First, it shows how “updates” of classes may be done using change-oriented versioning (COV) [Hol88, LCD+89]. Second, it shows an example of the usage context approach for dynamic binding.

Change-oriented versioning focuses on functional changes to a product structure. A functional change is described by a named, boolean global option. In the EPOS database [Lie90] a functional change is performed in the context of a long transaction. When a transaction is started, two parameters are given:

- A version-choice is a binding of boolean values to all options, and is used to decide which fragments (versions of objects) will be visible for reading\(^4\) in the transaction.

- An ambition is a binding of values to some of the options, and decides which fragments are to be affected by writes.

\(^3\)The same rule may be used for parameter passing.

\(^4\)In this section we try to be faithful to the vocabulary of the EPOS database.
The version choice is constrained to be within the ambition, i.e. the version choice implies the ambition. A database consists of fragments tagged by visibilities, which are evaluated to false or true under a given version-choice. The granularity of fragments are dependent on the granularity of the update primitive in the database. In the EPOS database this is object, i.e. an object will exist in many versions with disjoint visibilities. A version of a database may be understood as a subdatabase, consisting of versions of objects with visibilities evaluating to true under the given version-choice. The COV model is different from version-oriented versioning [LCD+89], where clients navigate between versions of objects. In COV a version-choice is given as a parameter to the start transaction operation, resulting in clients not seeing the versioning because it is given by the context.

In the EPOS database each class is represented by a class descriptor (type descriptor), being an object of a predefined class. An update to a class will be accomplished as follows: The old class will be retained with its definition, and a new version of the class having the modified definition will be created. Let \( S \) be the visibility of a class, and \( A \) the ambition of a transaction updating the class. Then, adopted from the COV-semantics of update-in-place [Lie90, page 68], the visibility of the old version of the class will become:

\[
S' = S \land \neg A
\]

While the visibility of the new version will become:

\[
S'' = S \land A
\]

The new visibilities are non-intersecting, and their union is precisely the old visibility. These two versions share name, but clients are bound to the correct version of the class by the complete option setting in the version-choice. The two versions of the class cannot be visible within the same complete version-choice.

At the extent level a two-way EPL will be created between the two versions of the class. This means that the two versions of the class will share extents. Like the class, the existing objects will have their visibilities updated with \( \neg A \) for the extent of the old version, while the objects will exist with visibility \( S'' \) in the new version.

The extent propagation will be limited (with respect to the complete extent graph) by the ambition of the transaction and the visibilities of the classes. A new object will only be propagated to the classes which have visibilities within the ambition of the transaction. When creating an object \( O \) of a class \( C \), all classes \( C_i \in EPC(C) \) where the visibility of \( C_i \) logically implies the ambition, will have \( O \) in their extents. This means that it is necessary to find the whole \( EPC \) of the creation class, and then to evaluate which classes the object should be propagated to based on the ambition and the visibilities of the classes.

In Figure 5.7 we have illustrated a simple example of a COV-update. Initially there are two classes File and BinaryFile, which both have visibility true. A transaction with ambition VMS updates File. This is accomplished by duplicating the File class into two versions having visibility \( \neg VMS \) (the old class) and VMS (the new class). There is a two-way EPL between the File classes. The BinaryFile class will still propagate its objects to the original File class (i.e. the one with visibility \( \neg VMS \)). Assume another transaction with ambition VMS, creating an object of the BinaryFile
class. This object will be propagated to File with visibility VMS, but not to File with visibility ¬VMS. If the transaction had ambition true, the object would have been propagated to both File classes.

In the EPOS database an object is constrained to have one most specific class within each transaction. Thus, dynamic binding is accomplished for an object by finding its most specific class, where the search for the requested property is started. Still an object may have multiple most specific classes, as long as the classes have disjoint visibilities.

5.7 Comparison/discussion

5.7.1 Class versioning mechanisms

We will now briefly compare the framework’s capability of doing class versioning with the approaches surveyed in Section 3.4.1. The surveyed approaches (except [Ber92]) deal solely with class versioning, and do not consider this in connection with other kinds of additive class evolution. Our framework uses the same dependency descriptions (EPLs, ACRs, OCRs) for class versioning as for specialisation and integration of classes. [Ber92] applies general database views for the purpose of schema evolution, and they may be used for several kinds of class evolution (cf. the comparison in Section 4.6.3).
Like [Zdo90, Cla92, MS92], the framework extends [ABB+83] by allowing new versions to add representation with respect to previous versions of a class. The added attributes of [Zdo90] are allowed to be independent or completely derivable compared to attributes of the previous versions. [Cla92] enriches this by providing for shared (logical sharing), independent, derivable, and dependent attributes. This is similar to our ACRs, but we collapse the shared and derivable category into derivable ACRs.

Common to the other surveyed approaches is that they always use shared representation of attributes. According to the examples in Section 5.3 this is advantageous with respect to the cost. However, in some situations the semantics of the dependencies requires separate representation, e.g., when attributes are mutually partially derivable. In distributed settings separate representation may also be advantageous.

A problem common to [ABB+83] and [MS92] is that successive class versions may create long chains of propagation for making objects consistent. In Section 5.6.1 we showed several techniques for optimising transitive dependencies. The most prominent and “cheap” of these was utilising shared representation for attributes in transitive ACGs, and having optimised mappings inside the attribute request operations. None of the surveyed approaches provide for this.

The surveyed approaches provide for implementation of dependencies between attributes, by implementing operations that convert between the attributes. An exception to this is [Cla92], which recognises the possibility of having shared attributes. In our work this is extended by introducing specifications, which allow us to have shared representation for attributes not being “shared”, but which are derivable from each other. See Section 4.4.6 for additional advantages of having ACRs as specifications.

[Zdo90] is the only one to use something similar to OCRs, by allowing operations of new versions to request operations of previous versions. However, this is only in the direction from new versions to existing versions. OCRs may be added in between any classes (new or existing). In [Zdo90] this can only be done by modifying the operations of existing versions of classes. A disadvantage of OCRs is that they are more expensive to implement than pure operations.

5.7.2 Dependencies in relational databases

ACRs provide a way to describe dependencies between attributes of objects residing in several classes. We will now compare these to dependencies used in relational database theory (e.g., [TL82, pages 276–282]). Functional dependencies of the form \( x_1 \ldots x_n \rightarrow y \) correspond to ACRs of the form \( (x_1, \ldots, x_n) \rightarrow (y) \), while multi-valued dependencies of the form \( x_1 \ldots x_n \rightarrow\rightarrow y \) correspond to ACR of the form \( (x_1, \ldots, x_n) \rightarrow\rightarrow (y) \). ACRs and relational dependencies are similar in that they describe dependencies between attributes. However, dependencies in relational database theory are used to prevent redundancy (which may result in operational anomalies) during design of relational schemes, while ACRs are used to specify consistency of objects which are members of multiple classes.

An ACR describes that there exists a mathematical function from one attribute do-
main to another attribute domain, while dependencies are not mathematical functions because they vary over time. E.g., the functional dependency \( x \rightarrow y \) is true for both pairs of the following tuples of the relation \( xy: (4, 5), (4, 5) \) and \( (4, 2), (4, 2) \). For an ACR which is accompanied by a particular function \( \mu : \text{dom}(x) \rightarrow \text{dom}(y) \), \( \mu(4) \) would have given the same value all the time. It is possible to read an ACR as “it exists a mathematical function at a particular point in time”, and that the actual function implementing the specification may be changed (or replaced) as time goes by. With this understanding of ACRs, technically they become very similar to dependencies in relational database theory.

### 5.7.3 Subclassing

As seen in Section 3.2.1 and 4.6.2 subclassing may serve many purposes in object-oriented databases. In this section we will briefly compare the evolution capability of subclassing with the framework.

At the extent level subclassing provides subsets of the extent of existing classes, while the EPLs of the framework allow for new classes having extents being supersets, equal sets, or subsets of the extents of existing classes. Subclasses will share attributes with its superclasses. Given class \( C_1 \) having attributes \( A \), a subclass \( C_2 \) of \( C_1 \) will share all attributes \( A \) with \( C_1 \). Expressed in our framework, this means that the only possibility is the following:

\[
\begin{array}{c}
C_1 \\
A \\
\end{array} \iff \begin{array}{c}
C_2 \\
A \\
\end{array}
\]

Our framework provides a much richer repertoire of dependencies between attributes of the classes, allowing for partially derivable, totally derivable, and non-derivable dependent attributes. Additional semantics may be put on non-derivable ACRs by the use of non-derivability mutation symbols.

In object-oriented data models the behavioural part of a class is equally important as the structural part. Subclasses inherit the behaviour of their superclasses as well, but it may be replaced by dynamic binding and redefined operations. The intent inheritance of the framework does not provide any construct to lookup operations dynamically along the inheritance hierarchy, but dynamic binding itself may be supported, as seen in Section 5.6.2. We view dynamic binding as a very important concept in object-oriented programming, but we do not agree that the lookup search for operations along the inheritance hierarchy (as in Smalltalk [GR83]) is an equally fundamental and important concept. The reason is that the search is automatically performed by the system according to a specific search strategy, and when combined with multiple inheritance, it may do too much “magic”, possibly giving unexpected results to clients.

Viewed in perspective of the expressiveness of the framework, subclassing becomes a rather strict and limited construct. Still, expressiveness is often traded for efficiency; It can be argued that subclassing provides for the most common evolution technique, and that it may be efficiently implemented. However, as seen in Section 5.4 and
5.6.2, the framework may take advantage of the same representation schemes as in subclassing, and it may support dynamic binding.

5.7.4 Consistency maintenance and encapsulation

In Section 5.6.2 we said that dynamic binding makes object-oriented programming powerful. In this section we address another strong point of object-oriented data models: encapsulation. We use encapsulation to express the ability to separate between the interface and the implementation of objects, such that clients of the encapsulated object are able to see the interface, but not the implementation. The essential gain of encapsulation is that clients will only see the relevant information of the objects, and the implementor of the objects may change the implementation as long as it does not impose semantic conflicts to the existing clients.

In the basic object model presented in Section 4.1, encapsulation is present by the use of the visibility modes public and private. Private properties are not accessible for ordinary clients of the class, i.e. other classes having link attributes to this class are only allowed to request public properties. However, classes connected by EPLs may have their encapsulation broken. This “breaking” can only be done through ACRs and OCRs connected to the EPL between the classes. Operations of one class are not allowed to break encapsulation of any other class.

There has been a debate on encapsulation and inheritance in object-oriented models [Sny86, Sny87, Sak89, Wol92]. This debate has resulted in the question of what do we want to achieve by inheritance and subclasses in object-oriented models. One branch of the debate has been whether subclassing should be used for pure modelling, or as a more general software engineering (reuse) mechanism.

On a more detailed level, the discussion has been whether subclasses should be allowed to access the internals of its superclasses. In C++ [Str91] clients can access public properties (members), subclasses can access public and protected properties, while private properties can only be accessed by the class itself. This means that subclasses are granted special rights when compared to clients, but have less rights than the class itself. In Smalltalk-80 [GR83], Objective-C [Cox86], and Eiffel [Mey88] subclasses may access the attributes of their superclasses, while normal clients cannot. Thus, the attributes are protected (in C++ terminology). Our framework does not allow classes to break the encapsulation of other classes, but ACRs/OCRs are allowed to. This means that private in the framework corresponds to private in C++ for other clients, but to public in C++ for directly related ACRs/OCRs.

Snyder [Sny87] argues that granting subclasses special rights, means exposing implementation to subclasses. Thus, this substantively reduces the possibility for the implementor of the superclass to change the representation/implementation. Sakkinen [Sak89] argues similarly that the integrity of an object (through encapsulation) is more important than subclassing in object-oriented programming. Wolczko [Wol92] argues that subclasses should have access to operations that are explicitly made accessible to them, but that attributes should be fully encapsulated, such that only

---

5In addition, friends can access private properties.
locally defined operations may access attributes directly. According to Wolczko, this

The extent and intent dimensions of our approach may be seen as mechanisms to

“loosen” the rather strict mechanisms of subclassing, and to unify these with similar

mechanisms like generalisation and class versioning. In this perspective, whether
classes connected by EPLs should be allowed to break encapsulation, is essentially
the same question as whether subclasses should be allowed to break encapsulation
of superclasses. As said previously, classes connected by EPLs are not allowed to
break the encapsulation of each other. However, ACRs and OCRs connected to the
EPLs are allowed to break encapsulation. ACRs and OCRs are used to let different
classes have overlapping views on the same or corresponding objects. Encapsulation
usually means that attributes are private while operations are public. When this
is the case, ACRs break encapsulation, while OCR do not. Due to the advantages
of ACRs with respect to OCRs (cf. Section 4.4.6), we have chosen to allow ACRs
between private attributes.

The main disadvantage of breaking encapsulation is that the integrity of objects may
be broken. A class may be viewed as having invariants, which are satisfied for an
object whenever none of the operations of the object is executing. Usually, a special
constructor or initialisation operation (possibly mingled with the object creation
operation) establishes the invariant. All public operations of the class preserve this
invariant. If private properties are exposed to code outside the class (subclasses,
ACRs, and OCRs), invariants may become unsatisfied, and the integrity is lost.
The important point here is that the control of the integrity no longer resides only
in the class itself, but possibly in all code associated with connected EPLs. A
typical example of such invariants is when a class appears as an implementation of
an abstract data type (ADT). One basic concept here is the representation invariant
[LG86, Mey88], which expresses the consistency of a representation (in our case given
by a class) vis-à-vis an underlying abstract data type.

Another disadvantage of exposing private properties is that it may prevent the im-
plementor of the class to change the representation. The inheritance/encapsulation
discussion above is mainly relevant in the context of object-oriented programming
languages. As recognised by Bloom and Zdonik [BZ87], there are some differences
between the database and programming language “worlds”, despite all efforts to make
“persistent programming languages” and “database programming languages”. We
think structural objects are equally important as encapsulated objects in databases.
Thus, we think the disadvantages of breaking encapsulation are more relevant in a
programming language context, than in a database context. We would also like to
mention that encapsulation is often broken in query languages for object-oriented
databases [DMZ91].
Chapter 6

Class Integration

This chapter will show how we apply our framework to integration of classes being independently developed. We focus on how the different components of the framework are used in the integration: Integration of extents by EPLs, integration of objects by $\exists$ functions, integration of properties by ACRs and OCRs. Two of the main qualities of our class integrations are: we do not require global schemas or global OIDs, and we support class evolution transparency. However, for our class integration to be successful, it is important to find and understand commonalities in the classes to be integrated.

We start this chapter by a brief introduction to the process of class integration. We will then show how we apply the framework to various integration situations by giving some examples. Furthermore, we treat integration of objects. Then we will introduce a methodology for integrating classes, and present some tools to support this. The chapter is concluded by some comparisons.

The main contribution of this chapter is how we exploit all components of the framework to do class integration. E.g., unlike many existing approaches our class integration methodology uses both the intensional and extensional dimensions of classes as guidelines in the integration. Parts of this chapter are published in [Bra93].

6.1 Introduction

A class integration in our framework consists of the following:

1. Connect two extent graphs by proper EPLs. New classes may have to be created to do this connection.

2. For each added EPL, an $\exists$ function must be defined.

3. A set of ACRs and OCRs should be defined for each added EPL.

The focus for our class integration is to let objects developed independently be seen and used as if they have been developed together. The main construct for allowing
this is to connect extents by the use of EPLs, which allow classes to share extents. In addition we have to integrate objects which are “semantically overlapping” by describing object correspondences, and to integrate the properties of objects by the use of ACRs and OCRs.

An example of this is illustrated in Figure 6.1, where two classes ColourShape and Shape and their clients are shown. By integrating the two classes by the EPL ColourShape $\Rightarrow$ Shape, an $\exists$ function, and some ACR/OCR, the clients of the two classes will share extents. Note that in this figure we have illustrated clients by boxes being different from the classes. Clients will either be other objects (statically: classes) or external applications.

The main quality of the framework, class evolution transparency, may be present for class integration as well. However, we think that class integration is more exposed to direct use of OCRs, because the differences between the classes will be larger than in class derivation. An additional quality of our framework that becomes visible in integration, is the transparency of object integration. Objects will be integrated by explicitly or implicitly relating objects from the two classes connected by an EPL. This means that none of the OIDs of the integrated objects will be changed to conform to each other or to a global OID. If there are clients being dependent on OIDs of specific objects (e.g., through link values), this is a very important quality.

We will address both schema integration and object integration, as defined in Section 2.3. Related to the issues of multi-database systems, we do not address the distribution issue at all. Neither do we consider whether the classes and objects to be integrated are within the same database or in different databases. For object integration we will somewhat discuss the issue of autonomy.

Heterogeneity may be an issue for our class integration through federated database systems [SL90]. Related to the reference architecture for federated database systems (shown in Figure 2.1), we believe that our framework may successfully serve as a
canonical data model, which is the data model used to express the component, export, and federated schemas. Our approach to class integration may be utilised in the constructing processor by integrating classes from different export schemas.

6.2 Applying the framework to class integration

6.2.1 Various integration situations

The generic examples to be shown are binary integration situations, i.e. there are two classes $C_1$ and $C_2$ to be integrated. The generic integration situations are illustrated by extent graphs in Figure 6.2. In this figure extents created as result of integration are filled with horizontal lines.

**Category generalisation:** This means creating a generalisation class $C_G$ of the two classes $C_1$ and $C_2$. The extent of $C_G$ is the union of the extents of $C_1$ and $C_2$. If $C_G$ is a concrete class, the extent of $C_G$ may have new members which are not members of either $C_1$ or $C_2$.

As an example of a category generalisation we use the power plant example from Section 3.5 (which again is based on an example from [SN88]), where we define a generalised class `PowerPlant` of the two classes `OilPlant` and `CoalPlant`. The generalisation results in the following new class `PowerPlant`:

```java
class PowerPlant {
    public:
        powerOn ();
        refill (mJoule energy);
        int consumed;
};
```

The integration is done by two EPLs `PowerPlant <= OilPlant` and `PowerPlant <= CoalPlant` and their associated ACRs and OCRs. In this case we have both ACRs and OCRs. Below we have shown the specification of an ACG in connection to one of the EPLs:

\[
\begin{array}{c}
\text{PowerPlant}\
\text{(consumed)}
\end{array} \quad \in \quad \begin{array}{c}
\text{OilPlant}\
\text{(oilFired)}
\end{array}
\]

Below we have specified two OCRs in connection with the same EPL:

```java
extprop PowerPlant <= OilPlant {
    ocrs:
        after powerOn () {
            mark PowerPlant as visited;
            fireOn();
            mark PowerPlant as unvisited;
        }
}
```
after fireOn() {
    mark OilPlant as visited;
powerOn();
    mark OilPlant as unvisited;
}
}

Similar ACRs and OCRs exist for the other EPL. Upon this class integration, all existing objects in OilPlant and CoalPlant must be manually added to PowerPlant. We do not specify any explicit 3 functions here, because the corresponding objects in PowerPlant will have the same OIDs as the objects in the two existing classes. In this example there will be two OCRs connected to both fireOn and powerOn. However, only one of these will be triggered for each object, because it is a requirement that the object must have a corresponding object in the other class for the OCRs to be triggered (cf. Section 4.4.4).

This example demonstrates a combination of the use of ACRs and OCRs. The objects probably have some state registering if they are active or not. The use of OCRs between powerOn and fireOn could be removed by the use of ACRs between those encapsulated attributes. On the other hand, we could also have removed the ACR between consumed and oilFired by having OCRs between the operations operating on these attributes, and thus being similar to the integrations done in Section 3.5. The following two OCRs provide an alternative to the previous ACR:

after refill (mJoule energy) {
    mark PowerPlant as visited;
    fillOil(mJouleToOil*energy);
    mark PowerPlant as unvisited;
}

after fillOil (barrels oil) {
    mark OilPlant as visited;
    refill(oilToMjoule*oil);
    mark OilPlant as unvisited;
}

A general discussion on the use of ACRs vs. OCRs was presented in Section 4.4.6.

Role generalisation: This creates a role generalisation class \( C_R \) of the two classes \( C_1 \) and \( C_2 \). This means that corresponding objects in \( C_1 \) and \( C_2 \) will have a generalised object in \( C_R \). In Section 7.3 we will show how class evolution may be implemented using object-oriented views. There, a category generalisation class will be defined as a union query of \( C_1 \) and \( C_2 \), while a role generalisation class will be defined as join query between \( C_1 \) and \( C_2 \).

We will use the two classes FacultyStudent and UniversityStudent introduced in Figure 4.18 as an example. We introduce Student as a role generalisation class of these two classes being independently developed in this situation. The
generalised class Student has the “union” of the attributes of FacultyStudent and UniversityStudent.

class Student {
    public:
        string name;
        int startYear;
        depDmn department;
        facDmn faculty;
};

We introduce two EPLs to accomplish this generalisation. In these three classes we will use the name attribute to register object correspondences, i.e. ACGs involving the name attributes will be used as definitions of  functions. If there are several students with the same name, we must use additional information for this. This is treated in Section 6.3.2. The first EPL has four ACGs specifying the consistency between name, start year, and department and faculty. The last two ACGs interfere, because department of FacultyStudent is dependent both on the department and the faculty of Student.

<table>
<thead>
<tr>
<th>Student</th>
<th>⇐</th>
<th>FacultyStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td>→ </td>
<td>(name)</td>
</tr>
<tr>
<td>(startYear)</td>
<td>→</td>
<td>(startYear)</td>
</tr>
<tr>
<td>(department)</td>
<td>→</td>
<td>(department)</td>
</tr>
<tr>
<td>(faculty)</td>
<td>→</td>
<td>(department)</td>
</tr>
</tbody>
</table>

The other EPL has similar interference between ACGs. This situation will be further explained and discussed in Section 6.5.

<table>
<thead>
<tr>
<th>Student</th>
<th>⇐</th>
<th>UniversityStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td>→ </td>
<td>(name)</td>
</tr>
<tr>
<td>(startYear)</td>
<td>→</td>
<td>(enrolmYear)</td>
</tr>
<tr>
<td>(faculty)</td>
<td>→</td>
<td>(faculty)</td>
</tr>
<tr>
<td>(department)</td>
<td>→</td>
<td>(faculty)</td>
</tr>
</tbody>
</table>

In these two EPLs we have used ACGs as definitions of  functions (indicated by  as subscript on the ACG symbol). The semantics of the EPLs in this situation is that an object that is a member of Student may have corresponding objects in both UniversityStudent and FacultyStudent.

**Specialisation:** This is a situation where the integration of the two classes \( C_1 \) and \( C_2 \) gives need for creating a specialisation class \( C_s \). In this case existing objects in \( C_1 \) and \( C_2 \) may have to be integrated in \( C_s \). If \( C_1 \) and \( C_2 \) have common generalisation classes, the object integration may be done by equal OIDs, otherwise it has to be done manually by specifying  functions.

**Connecting variants:** The two independently developed classes \( C_1 \) and \( C_2 \) are variants of each other, and need to connect their extents. As an example of this we can consider the classes UniversityStudent and FacultyStudent from Figure 4.18, and again as independently developed classes. Here, they should be connected by an EPL and its associated ACGs:
In this situation we have specified both the name and the start year in the ACG used as a definition of the \( \mathcal{R} \) function. This gives us less of a chance for "clashes" in the object correspondences.

**Connecting specialisation/generalisation**: The two classes \( C_1 \) and \( C_2 \) are generalisation/specialisation of each other, and need to be connected to each other upon integration.

### 6.2.2 Special integration situations

We will now show how some special class integration situations known from [KDN90] and [Ken91b] relate to our sense of integration. Afterwards we will discuss the use of EPLs for these integration situations.

**Aggregation integration**: In our framework we will interpret this as connecting a composite object to its components. Given an object integration where an
object (of class $C_A$) being a “composite” of attribute values, and its corresponding “components” (of classes $C_{P_i}$ through $C_{P_N}$) appear as objects. This means that we have $N$ “part objects” corresponding to $N$ attribute values of another object. The integration may be done by creating $N$ EPLs $C_{P_i} \Leftrightarrow C_A$, $i = 1..N$, where each $\exists$ function is a bijection $\text{id}_A \rightarrow \text{id}_{P_i}$. This situation is illustrated in Figure 6.3, where objects are shown as “bullets” below the class they are members of. An example of this is the following. In one database an

![Figure 6.3: Aggregation integration.](image)

automobile is modelled by a class **Automobile** having three attributes, none of them being links to other objects.

```cpp
domain EngineDmn = enum { B18, B21, B25 };  
domain SteerDmn = enum { power, normal };  
class Automobile {
    public:
        EngineDmn engine;
        float milesPerGallon;
        SteerDmn steering;
};
```

In another database engines and steerings are modelled using the classes **Engine** and **Steering**. This means that two of the attributes of an automobile object correspond to objects of the **Engine** and **Steering** classes in the other database.

```cpp
class Engine {
    public:
        string type;
        int nCylinders;
        float effect;
};
class Steering {
    public:
```
string name;
    boolean power;
}

These could have been integrated by the following two EPLs with one ACR each:

\[
\begin{align*}
\text{Steering} & \leftarrow \text{Automobile} & \text{Engine} & \leftarrow \text{Automobile} \\
\text{(power)} & \longleftrightarrow \text{(steering)} & \text{(type)} & \longleftrightarrow \text{(engine)}
\end{align*}
\]

Here we have chosen to make one-way EPLs. This ensures that all automobiles will have an engine and a steering, but not every object of Steering and Engine need to have a corresponding Automobile. The accompanying \(\exists\) functions should be one-to-one, such that each automobile may have at most one engine and one steering. This example is quite problematic, because the values of steering and engine attributes of Automobile correspond to objects of the Steering and Engine classes. If the steering attribute changes, the corresponding object of class Steering should be replaced as well. This is not handled by the ACR given above, but may be handled by OCRs.

**Group integration:** This is similar to the previous situation, but an object in a class \(C_G\) has \(N\) corresponding objects in another class \(C_M\). Here, we will use the EPL \(C_M \Rightarrow C_G\), requiring each object in the member class \(C_M\) to have a corresponding object in the group class \(C_G\). We do not require the \(\exists\) function defined on \(\text{id}_M \rightarrow \text{id}_G\) to be one-to-one, which means that many objects in \(C_M\) may have the same corresponding object in \(C_G\). This situation is illustrated in Figure 6.4.

![Figure 6.4: Group integration.](image)

**Meta-object/normal object integration:** This is a situation where a meta object corresponds to a normal object. If the system supports schema entities as “normal” objects, e.g. as done in the EPOS database, this is in principle equal to a normal integration. However, schema entities are often not mutable, which means that the consistency maintenance is only done upon creation of such a normal object or schema entity.
We would like to give some comments to the use of EPLs in the two first situations mentioned here. In Section 4.2.2 we outlined two different ways of utilising EPLs: either as IS-A relations, or as mechanical constructs exploiting consistency maintenance. As part objects and group members clearly are not in IS-A relations to their containing or group object, these are examples of the use of EPLs for mechanical consistency maintenance. We prefer EPLs used for IS-A situations, and we could be similarly negative to the “mechanical use” of EPLs as [Op92] is to the related problem of using inheritance to model part objects.

6.3 Object integration

6.3.1 Proxy and entity objects

A basic assumption in object-oriented databases is that OIDs are achieved by requesting object creation operations. This means that each creation event will give a new object. The operation to add an object to a class loosened this restriction, by allowing an object to become a member of a class without being created in the class. However, as noted in Section 4.2.3, this is not enough.

When classes developed and populated independently are to be integrated, the problem of semantically overlapping objects might appear. As in [Ken91a, KAA+92] we will introduce (real-world) entity objects, which are the real-world objects\(^1\) we think about, and their (multiple) representations in databases, named proxy objects. Entity objects are the conceptual objects being modelled in the database by proxy objects.

If two proxy objects model the same entity object, they are said to be entity identical. If these proxy objects also are identical database objects, they are said to be proxy identical. We will often say that two entity identical objects are in semantic overlap. Usually when proxy objects are entity identical, they have properties which are semantically overlapping as well. However, if two proxy objects model disjoint roles of an entity object, they may happen to have no properties in common.

We assume that (proxy) OIDs are system-assigned, arbitrary, immutable, and unique within the context where they were created, usually within a database. Object integration means finding proxy objects being entity identical. The framework provides \(\exists\) functions to relate corresponding objects, and this will be used to integrate proxy objects which are entity identical. Note that the framework can register corresponding objects, and use this for consistency maintenance, but it is only in the domain of the application it can be understood if two proxy objects really are entity identical. Thus, in the context of the framework we will speak of corresponding objects, while in the context of the application we will speak in terms of entity identical objects.

In the framework, object integration means finding an \(\exists\) function for each EPL introduced as a result of class integration. In basic object integration situations, one

---

\(^1\)We will not go into philosophical questions about the existence of one true real world, but we assume a conceptual real world, because this makes it possible to reason about integration.
object in one class corresponds to one object in another class. The group integration example given in Section 6.2.2 shows that an object in one class may also correspond to a group of objects in another class.

Object integration does not result in any direct representation of entity objects. The entity identical relation is only present through the object correspondences using the Θ functions. Thus, we do not introduce any notion of global objects or global OIDs. However, if a class integration results in two existing classes being connected through a new class, the integration of two corresponding objects will result in a new proxy object to be created as an “integrating object”. In the situation where two classes $C_1$ and $C_2$ are role generalised in the new class $C_G$, each object in $C_G$ will be an “integrating object” of objects in $C_1$ and $C_2$. This object is a proxy object like any other object, but in this context it represents the integration of two other proxy objects. In principle there are no differences between “integrating objects” and normal objects. E.g., they may be further integrated, and have their own attributes. Note that an “integrating object” is not an entity object, it is an object created to “connect” two proxy objects found to be entity identical.

When doing object integration, existing OIDs are preserved. This approach is justified by the difficulty in conforming (changing) OIDs, since they may be spread around in link values everywhere in the database. If users are allowed to see OIDs, they may also be dependent on the preservation of OIDs. Immutability of OIDs is fundamental to object-oriented databases, and our object integration does not compromise this.

The purpose of integrating objects is two-fold. First, it registers proxy objects being entity identical. If desired, this information can be utilised by clients by providing for a relation-like construct to traverse proxy objects being entity identical. The second purpose is to allow for consistency maintenance. If two proxy objects being integrated have overlapping properties, the system will maintain the consistency of these properties.

### 6.3.2 Facilitating object integration

We see three ways of providing Θ functions, i.e. registering object correspondences:

**Automatic**: The Θ function is defined using properties of objects, like employee number or social security number [Ken91a]. We will specify this with special ACGs. E.g. the specification

\[
\begin{array}{c|c|c}
\text{FacultyStudent} & \Leftrightarrow & \text{UniversityStudent} \\
\text{(name)} & \longrightarrow & \text{(name)}
\end{array}
\]

means that student objects having equal name in those two classes are entity identical. Since the EPL is two-way, $\Theta : \text{id}_{FS} \rightarrow \text{id}_{US}$ has to be a bijection (see Section 4.2.3). The introduction of this automatic Θ function could have constrained the name attribute of those two classes to be unique within their respective classes. In this example this would have meant that two objects are
not allowed to have the same name in the same class. However, we will allow such ambiguous correspondences to appear, by allowing for additional manual registration (see the combined approach below).

For the different characteristics of \( \mathcal{S} : \text{id}_1 \rightarrow \text{id}_2 \) (cf. Section 4.2.3) we will use the following ACG notations:

<table>
<thead>
<tr>
<th>Sit.</th>
<th>EPL</th>
<th>( \mathcal{S} ) characteristic</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( C_1 \leftrightarrow C_2 )</td>
<td>Total bijection (one-to-one and onto)</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
</tr>
<tr>
<td>2.</td>
<td>( C_1 \leftrightarrow C_2 )</td>
<td>Partial bijection (one-to-one and onto)</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
</tr>
<tr>
<td>3.</td>
<td>( C_1 \Rightarrow C_2 )</td>
<td>Total injection (one-to-one)</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
</tr>
<tr>
<td>4.</td>
<td>( C_1 \Rightarrow C_2 )</td>
<td>Total function (one-to-one not required)</td>
<td>( \leftarrow \rightarrow \mathcal{S} )</td>
</tr>
</tbody>
</table>

An \( \mathcal{S} \) function definition has additional semantics over a normal ACG definition, because it decides which objects are related with respect to consistency maintenance. However, it still serves as a dependency description. E.g., if a student changes the name in the FacultyStudent class, the same student will have its name changed in the UniversityStudent class.

**Manual:** This is the tedious approach of manually registering pairs of proxy objects. The characteristics of the \( \mathcal{S} \) function should also be specified here. Below we have shown the specification of a manual registration of the \( \mathcal{S} \) function from objects in the FacultyStudent class to objects in the UniversityStudent class.

\[
\text{FacultyStudent} \leftrightarrow \text{UniversityStudent} \quad \leftarrow \rightarrow \mathcal{S}
\]

The manual registration requires information to be explicitly stored. This can be done either by registering references from objects to objects, or by explicit relations, i.e. either an object is told which object is the corresponding object in the other class, or “correspondence tuples” are inserted into a relation.

We consider this object correspondence registration mainly as a meta-activity. Often it may go across databases. The correspondences are intended to be static, i.e. corresponding objects will remain corresponding objects. However, for management reasons, deleting “tuples” and adding new ones may be allowed.

**Combined:** This appears in situations where automatic registration fails to work because there are some ambiguous object correspondence “tuples”. Thus, both manual and automatic registration will be used in what we name the combined approach. E.g., if there are two students having the same name in the student example, the object correspondence “tuples” for these students will be ambiguous since we do not know which of the objects to relate to. These ambiguous tuples will instead be registered using (additional) manual registration. This means that upon consistency maintenance, the manual registration will be looked up first, before the automatic correspondence is used.

The combined approach appears where the automatic approach fails to work for all objects. This means that an automatic registration will become combined when ambiguous tuples appear. Upon every creation of new objects, it must
Object correspondence registration introduces several questions of management. Consider the classes $C_1$ and $C_2$, integrated by the EPI $E$. To avoid the objects in the extent of $C_1$ being integrated using automatic registration, an important issue is what to do upon creation of objects. Normally, when creating an object in $C_1$, we would add the same object to $C_2$. Since this is an integration of an object and a corresponding object must be added to $C_2$. In some situations it may be reasonable to delay this propagation of objects until a client of the other class creates an entity. We could have supported such an approach by having additional semantics over EPIs to describe extent propagation policies. However, in this thesis we will not consider this.

When integrating classes residing in different component databases, we must consider the autonomy issue. EPIs ensure that an object may be added to several other classes when creating an object in a given class. The situation is when the two classes are within the same component database, and the object is added to the other class with a given (and still unique) OID. If the class that the object should be added to is part of a different component database, this may not be possible since the OIDs themselves are unique. It would also mean that the component databases must give up some of their autonomy, because they cannot assign the OIDs themselves. The appropriate approach here is to let the system use manual registration of the object's correspondences. When manual registration is the only option, OIDs are the prime identification mechanism, external identification must be used as an additional mechanism for specific criteria. If there are any ambiguities, a notification must be given to the client or to an integration manager telling that manual registration is required.
6.3.3 Class integration and binding of operations

In Section 4.2.4 we discussed different kinds of dynamic binding in the presence of extent graphs. All of these techniques can be used within the context of class integration. We assume that each client is connected to a specific component database, which means that for classes being integrated and residing in different component databases, all requests are to the objects of the class residing in the same component database. *Explicit qualification* and *tagged link values* may be used for proxy objects residing within the same component database. However, *static binding* is often more appropriate for class integration. Integration is often intended for resource sharing, and not so much for replacing an object by another object. This is different from class derivation, where classes (and their objects) may be replaced by more specialised representatives.

6.4 Methodology for class integration

Due to its complexity, schema integration is often supported by *methodologies* describing the necessary steps in such a process [BLN86]. We will now outline a *binary class integration methodology* for our class evolution framework.

Class integration is a result of different developments with different views of the real-world. The input to our class integration methodology is two classes having objects which in some way are entity identical. The purpose of the methodology is to give the class integrator a guideline in how to proceed to find how objects of the two classes are entity identical, and to establish EPLs, Σ functions and ACRs/OCRs between those classes. Important for the methodology is finding the *semantic set relationship* between the extents of the classes to be integrated. The semantic set relationship is defined to be the set relationship between the entity objects of the proxy objects in the two extents. This will be found by letting the user introduce the Σ function.

6.4.1 Overview of the methodology

Given two populated classes with their intents and extents: $C_1 : (I_1, E_1)$ and $C_2 : (I_2, E_2)$, the following steps should be performed:

1. **Intensional comparison**: Compare $I_1$ and $I_2$, and suggest how $E_1$ and $E_2$ should be related by EPLs.

2. **Extensional comparison**: Find the *semantic set relationship* between $E_1$ and $E_2$, and (again) suggest appropriate EPLs.

3. **Confirmation**: If there are any conflicts between the results of step 1 and 2, let the user confirm how the extents should be connected by EPLs, possibly creating new classes.
4. **ACR/OCR generation**: Suggest appropriate ACGs and OCRs. This is based on the decision from step 3 and on the comparison from step 1.

---

**Figure 6.5**: The steps in the integration methodology.

The steps in the methodology are illustrated in Figure 6.5. The methodology facilitates two kinds of comparisons between classes: comparing the intents and comparing the extents, where the latter is based on results from the former. The results of these two comparisons are given to the user for approval in step 3, where the final decision on the EPL connection is done. In step 4 appropriate ACRs/OCRs are suggested based on the decision from step 3. After step 4 is performed, the actual integration may start by creating new classes, EPLs, ACRs, OCRs, and by propagating the existing objects.

The methodology is based on a traditional principle of suggesting integration based on the relationships between the intents, e.g. see [SLCN88]. We have extended this in two ways. First, we include integration of operations, which is necessary when using object models including specification of behaviour. Second, we base the integration methodology on comparison of extents in addition to comparison on intents. The extensional comparison requires an $\exists$ function between the two classes. We think the extensional comparison is a very helpful extension for defining the EPLs. Intensional comparison is most helpful when defining the $\exists$ function used in the extensional comparison, and for the generation of ACRs/OCRs. The weakness with purely relying on intensional comparison is that overlapping intents may occur: two classes may happen to have the same properties, while there is no semantic relationship between them, i.e. the set of entity objects modelled by the two corresponding extents are disjoint. Extensional comparison works with objects which are found to be entity identical, and thus should be connected in some way in an extent graph.
6.4.2 The search for shared abstractions

[GLN92] says that the main goal of schema integration is to interconnect separately defined structures via their shared concepts. In [SLU89] the essence of object-oriented programming is said to be sharing in various forms, where traditional object-oriented systems let classes be shared among objects, and superclasses shared among subclasses. In our framework sharing is accomplished among classes and objects by the use of EPLs, Σ functions, and ACRs/OCR. E.g., the EPL \( C_1 \leftrightarrow C_2 \) lets \( C_1 \) and \( C_2 \) share extents, and the ACG \( A \leftarrow \rightarrow B \) lets the attributes in \( A \) and \( B \) be shared. The sharing imposed in both examples is in the sense of mutual dependencies. The framework also allows for sharing between intents using inheritance links. However, this type of sharing is not particularly interesting to class integration, because it can only be exploited at the time of creation of intents (since intents are immutable).

The main problem of class integration is to find the shared abstractions hidden in independently developed classes. [Opd92] says that the difficulty in finding common abstractions of classes come from

1. differing vocabulary, and

2. incompatibility between the implementations of the classes.

The first problem is well-known from schema integration, appearing as name differences (homonyms and synonyms), scale differences, domain partitioning differences etc. We will address this by suggesting a tool for uncovering the differing vocabulary, and by using the components of the framework for the actual integration. For the second problem we will introduce abstract data types (ADTs) as additional specifications to understand the semantics of operations.

Despite our support for resolving name differences (i.e. synonyms and homonyms), class evolution transparency requires us to retain all properties of classes as they are. This means that we have no visible conformation phase [BLN86]. However, we will use internal conforming (name changes etc.) in step 1 to be able to compare properties of intents. With all classes retained, ACRs and OCRs make properties “shared” between two classes. ACRs and OCRs can be specified between properties with any names or scales, the properties do not have to be conformed to participate in the consistency description. E.g., given the attributes \texttt{height} measured in feet, and \texttt{length} measured in centimeters. The ACG \((\texttt{height}) \leftarrow \rightarrow (\texttt{length})\) represents an integration of these two attributes.

Object-oriented design is the construction of software systems as structured collections of abstract data type implementations [Mey88]. This means that object-oriented classes may be seen as implementations of abstract data types (ADTs). Therefore we would like to suggest ADTs as additional specifications to help the integration of operations. We will use ADTs in the intensional comparison as an alternative to comparing the bodies of operations. In Section 6.4.3 we will also outline a scheme for comparing bodies of operations, but this is a special case of reverse engineering, which is a research area outside the scope of this thesis. The advantage of ADTs is that they capture the essential properties of objects without overspecifying
them. A typical slogan is: “ADTs state what a system does rather than how it does it.” For our comparison of classes this means that ADTs provide some means for comparing what classes do, rather than comparing how they are implemented. Even if a mechanical comparison cannot find shared abstractions between two ADT specifications, they give the integrator an understanding of the semantics of the classes. ADTs are frequently used in canonical data models [ADK91, EK91b, Zdo91]. E.g., [EK91b] incorporates ADTs in a canonical data model integrating functional and object-oriented concepts.

### 6.4.3 Intensional comparison (Step 1)

*Intensional comparison* means comparing two intents and suggesting how to integrate the two classes by creating new classes, appropriate EPLs, and ACRs/OCR. By viewing intents as sets of properties, a comparison of two intents will give a set relationship.

An intensional comparison of two classes, $C_1$ and $C_2$ with intents $I_1$ and $I_2$, consists of two steps:

1. Create two conformed intents $I'_1$ and $I'_2$ to be used temporarily in the comparison.

2. Compare $I'_1$ and $I'_2$ and suggest EPLs and ACRs/OCR involving $C_1$ and $C_2$.

Based on the set relationships between $I'_1$ and $I'_2$, we suggest the extent integrations (including alternatives) given in Table 6.1. The rationale for the two first sug-

<table>
<thead>
<tr>
<th>$I'_1$ is-subset-of $I'_2$</th>
<th>$C_1 \subseteq C_2$</th>
</tr>
</thead>
</table>
| $I'_1$ equals $I'_2$       | $C_1 \Leftrightarrow C_2$
|                             | $C_G \Leftrightarrow C_1$ and $C_G \Leftrightarrow C_2$ |
| $I'_1$ overlaps $I'_2$     | $C_G \Leftrightarrow C_1$ and $C_G \Leftrightarrow C_2$ |
| $I'_1$ is-disjoint-with $I'_2$ | No integration
|                             | $C_G \Leftrightarrow C_1$ and $C_G \Leftrightarrow C_2$ |

Table 6.1: Suggesting integration based on intensional comparison.

gestions is taken from a modelling point of view: a general class should have fewer properties than a special class. Thus, if the two classes have the same properties (the *equals* situation), they are on the same level. This may be accomplished either by class connection ($\Leftrightarrow$) or a generalisation. In the *overlaps* and *is-disjoint-with* situations, we suggest to create a new class $C_G$ generalising the two existing classes. If the two classes have something or nothing in common, they might be partly overlapping or disjoint roles of the same objects. Thus, a role generalisation might be appropriate.
In the comparison we will use all properties of classes, both public and private. If the class integration results in a new class, it is important to record if the compared properties were private or public. This will be used to decide the visibility mode of the properties of the new class.

This methodology measures how much two intents have in common. If two intents happen to have much in common, it does not necessarily mean that their extents should be related in any way. We assume that the classes to be integrated by the binary integration methodology are given by the user, and thus they are considered to be semantically related (having entity identical objects).

Conforming intents

By conforming intents we mean resolving differences between intents by introducing common names and domains. This is often referred to as name and scale differences. Most of these problems would have disappeared if the developers of the intents had used the same vocabulary and attribute domains. A guideline for well-designed classes is to have easy-to-understand names for properties. Slightly paraphrased from [WWW90, pages 162-163], the following naming rules for properties are given:

- Use a single name for each conceptual property.
- Associate a single conceptual property with each property name.
- If classes have the same specific property, make this explicit in the class hierarchy.

The two first rules impose a one-to-one mapping between a property name and a conceptual property. A classical example of the contrary is the “Cowboy-Windows” example:

```cpp
class Cowboy {
public:
    move (int newX, int newY);  
    draw ();                   
    shoot ();                  
};

class Window {
public:
    move (int newX, int newY);  
    draw ();                   
};
```

These two classes have two names of operations in common: move and draw. The move operations are conceptually the same, since both operations are moving an object to a new location. The draw operations are clearly different, since one is
used to visualise the window on the screen, while the other is used to draw the gun from the holster. Thus, if a shared, unambiguous vocabulary were used, the draw operation of the Window class would be renamed, e.g. to display. The draw operation is an example of a name difference.

Experiences from developing the Eiffel library indicate that naming should be chosen according to external instead of internal criteria [Mey90]. As the Eiffel library contains standard classes with typical data structures used in program development, this made the developers consolidate the naming of operations throughout the entire library. For the typical container classes: Stack, Array, Queue, and H-Table (hash table), resulting in the following change of the naming of operations:

<table>
<thead>
<tr>
<th>Old version</th>
<th>Stack</th>
<th>Array</th>
<th>Queue</th>
<th>H-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>push</td>
<td>enter</td>
<td>add remove oldest</td>
<td>insert</td>
</tr>
<tr>
<td></td>
<td>pop</td>
<td>entry</td>
<td>oldest</td>
<td>delete</td>
</tr>
<tr>
<td></td>
<td>top</td>
<td>entry</td>
<td>put</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>put</td>
<td>put remove item</td>
<td>put</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>item</td>
<td>remove item</td>
<td>item</td>
</tr>
</tbody>
</table>

Such a streamlining of naming is hard to achieve when there are different persons developing the classes. We think this is more an of organisational than a technical issue. One way of addressing this is a organisation-wide (or the scope that the class may be used within) approval of naming.

In the rest of this section we assume the conformation of naming to be done. In Section 6.6 we will introduce a tool which uses a thesaurus to automatically conform names. Similar to the common naming issue, using common attribute domains would help much in finding common attributes.

Comparison of attributes

Attributes are compared by finding whether they relate by ACRs. If two attribute sets A and B exist in an ACG $A \oplus B$, we say $A$ and $B$ are equal with respect to intensional comparison. E.g., the connecting variants example from Section 6.2.1 has two ACGs:

$$
\begin{align*}
(name, startYear) & \leftrightarrow (name, enrolmYear) \\
(department) & \leftrightarrow (faculty)
\end{align*}
$$

If we assume the two classes to only have these three attributes, we conclude that \texttt{FacultyStudent equals UniversityStudent}, and the EPL \texttt{FacultyStudent \Leftrightarrow UniversityStudent} is suggested.

We will assume that the ACGs are specified by the integrator, but that the system may give some help in suggesting ACGs. If classes are developed using common
attribute domains and by following the naming rules listed above, we might be able to suggest ACGs automatically. E.g., if both name attributes are of the domain personNameDmn, it is very probable that these two attributes would take part in an ACG (name) \(\leftarrow \rightarrow\) (name).

By having a catalogue of ACGs between attribute domains, we can also suggest ACGs for attributes of different attribute domains. E.g., assume (depDmn) \(\leftarrow \rightarrow\) (facDmn) has been recorded, meaning that the ACG may be valid for all attributes of these two attribute domains. Thus, When integrating two classes having attributes of facDmn and depDmn, we may suggest such an ACG even if the attributes happen to have different domains and names. For attributes of link domains we may use EPLs as an appropriate attribute comparison mechanism. Consider the two attributes:

\[
\text{link <UniversityStudent> stud;} \\
\text{link <FacultyStudent> student;}
\]

If it already exists an EPL between UniversityStudent and FacultyStudent, an ACG may be suggested between stud and student. Generally, we will use the following guidelines:

<table>
<thead>
<tr>
<th>Existing EPL</th>
<th>Suggested ACG</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_2&gt;)</td>
</tr>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_2&gt;)</td>
</tr>
<tr>
<td>(C_1 \Rightarrow C_2)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_2&gt;)</td>
</tr>
</tbody>
</table>

To implement the functions in the ACRs, the integrator may use the \(\exists\) function of the EPL between \(C_1\) and \(C_2\). E.g., the ACRs between the two attributes stud and student may use the existing ACG (name) \(\leftarrow \rightarrow\) (name) to implement the ACG (stud) \(\leftarrow \rightarrow\) (student). This technique may be extended to classes transitively connected by EPLs according to the following scheme:

<table>
<thead>
<tr>
<th>1st EPL</th>
<th>2nd EPL</th>
<th>Suggested ACG</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(C_2 \leftrightarrow C_3)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_3&gt;)</td>
</tr>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(C_2 \Rightarrow C_3)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_3&gt;)</td>
</tr>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(C_2 \leftrightarrow C_3)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_3&gt;)</td>
</tr>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(C_2 \Rightarrow C_3)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_3&gt;)</td>
</tr>
<tr>
<td>(C_1 \leftrightarrow C_2)</td>
<td>(C_2 \Rightarrow C_3)</td>
<td>(\text{link}&lt;C_1&gt; \leftarrow \rightarrow \text{link}&lt;C_3&gt;)</td>
</tr>
</tbody>
</table>

The other combinations of EPLs may be utilised similarly. In the transitive EPL situation, the ACRs between the attributes may be implemented by composing the \(\exists\) functions of the EPLs.

Thus, some techniques may be used to make the comparison of attributes semi-automatic, but any decision based on the comparison should be confirmed by the integrator.

**Comparison of operations**

Comparison of operations can be done on three levels:
Signatures: Comparing signatures means comparing their names, parameters (names and domains), and return domains.

Bodies: Two operation bodies may be compared by string and tree comparisons [Opd92].

Constraints: The semantics of two operations may be compared by comparing the constraints (axioms) in the (optional) ADT specifications.

Signatures are the interface of operations, they are meant to tell something about the operations to clients. Two signatures are equal if they have the same name, the same number of parameters, the same names and domains of parameters, and the same return domain. By introducing a similarity measure, we will be able to tell when two signatures are similar. Again, assuming all conceptually equal properties to have the same name (also for parameters), the following measure might be used:

\[
sigSim(Op_1, Op_2) = \frac{\#(\text{parms}(Op_1) \cap \text{parms}(Op_2))}{\#(\text{parms}(Op_1) \cup \text{parms}(Op_2))}
\]

This gives a value between 0.0 and 1.0, where 1.0 means that the signatures are equal, while 0.0 means that they have nothing in common. \text{parms}(Op) is the set of the following pairs \((\text{retDm}, \text{opName})\), \((d_1, p_1)\), \(\ldots\), \((d_N, p_N)\) of the operation \(Op\), where \text{retDm} is the return domain of \(Op\), \text{opName} is the name of \(Op\), and \(d_1\) through \(d_N\) are the domains of the parameters, and finally \(p_1\) through \(p_N\) are the names of the parameters.

An interesting property of our integration is that the orders and names of the parameters do not have to match. This is due to the use of operation consistency relations to relate operations of different classes in consistency maintenance. E.g., if two operations having two parameters in opposite orders, \text{op1} \((d_1 \ p_1, \ d_2 \ p_2)\) and \text{op2} \((d_2 \ p_2, \ d_1 \ p_1)\), are found to be similar, we can integrate the operations by the following OCRs:

```plaintext
after op1 (d1 p1, d2 p2) { 
  op2 (p2, p1);
}

after op2 (d2 p2, d1 p1) { 
  op1 (p1, p2);
}
```

This is a result of class evolution transparency, which preserves classes and describe the consistency between the properties of the two classes explicitly and externally. Assume the classes \(C_1\) with the operation \text{op1} and \(C_2\) with the operation \text{op2}, to be integrated with the EPL \(C_1 \Rightarrow C_2\). Clients of \(C_1\) will not request the operation \text{op2} in the objects of \(C_2\) directly, but indirectly through the OCR connected to \text{op1}. If we had an integration technique such that the clients of \(C_1\) had to request the \text{op2} directly, we had to require \text{op2} to conform to \text{op1}, e.g., using the conformance rules of Emerald [BHJL86], which require the number and order of the parameters to be the same.
Operation **bodies** may be compared using either *string* or *tree comparison* [Opd92]. *String comparison* means comparing the bodies of operations by viewing them as strings of characters, e.g., like the Unix `diff` program finding the different lines between two text files. *Tree comparison* compares two operation bodies by representing them as *syntax trees*. As we have not defined any concrete language for the operation bodies in the framework, we will not go into detail about these comparison, but refer to [Opd92]. These comparisons will be used to evaluate the similarity of two operation bodies, and as in the previous case this may defined as an expression giving a value between 0.0 and 1.0. E.g., in the string comparison this may be defined as the number of equal lines divided with the average number of lines of code in the two bodies. If an integration results in a new class to be generated, the comparison of the bodies may also be used as help in defining a corresponding operation in the generated class.

The third level of operation comparison is to compare **constraints** (axioms) specifying the semantics of operations. An important question here is where do the constraints come from? We see two ways of achieving the ADT specification including the constraints. Either they are there as a part of the design of the classes, as argued for in Section 6.4.2, or they may be achieved through a *semantic enrichment* phase. For federated database systems involving local schemas specified in data models with limited expressiveness, schema integration is based on *semantic enrichment* of the local schemas [CS91]. Related to the reference architecture for federated databases illustrated in Figure 2.1, the semantic enrichment is a necessary step to apply the *transforming processor*. Thus, prior to class integration, if possible, ADT specifications have to be created. We extend classes with specification because they describe the essential properties of operations, and may be a better basis for comparing operations than the comparison of operation bodies.

We will demonstrate this by a small comparison using a syntax (created for illustrative purposes) similar to the one of the constraints section of the Larch shared language [Win87]. Given the constraints section of an ADT specification of a **Stack** class and a **Set** class:

**Stack constraints:**

```
isEmpty(empty())  == true
isEmpty(push(x,s)) == false
top(push(x,s))    == x
pop(push(x,s))    == s
```

**Set constraints:**

```
isEmpty(empty())   == true
isEmpty(insert(x,s)) == false
elem(x,empty())   == false
elem(x,insert(y,s)) == if y = x then true
                        else elem(x,s)
remove(s,empty())  == empty()
remove(y,insert(x,s)) == if y = x then remove(y,s)
                        else insert(x,remove(y,s))
```

153
After substituting names according to the streamlining of naming from the Eiffel library [Mey90], we achieve the following set of constraints:

Stack constraints:
- isEmpty(empty()) == true
- isEmpty(put(x,s)) == false
- item(put(x,s)) == x
- remove(put(x,s)) == s

Set constraints:
- isEmpty(empty()) == true
- isEmpty(put(x,s)) == false
- item(x,empty()) == false
- item(x,put(y,s)) == if y = x then true else item(x,s)
- remove(x,empty()) == empty()
- remove(y,put(x,s)) == if y = x then remove(y,s) else put(x,remove(y,s))

The `isEmpty` operations of the two classes have the same behaviour. The `empty` operations have similar behaviour since they appear in two equivalent constraints. The `item` operations are different in the two classes: For `Stack` it is an operation taking a stack and returning the last element `put` on the stack, while for `Set` it takes an element and a set, and tests if the element is a member of the set. `remove` removes the last element `put` on the stack, while for `Set` it is an operation removing a specific element from a given set. This comparison reveals the two specifications to be similar, but having some differences. The stack is operated on with a given current element in the stack, while the set operations always have the current element as a parameter. Concretely, we have revealed the `isEmpty` operations to be the same, while the `empty`, `remove` and `put` are similar. They can probably be integrated. An example of an integration of these two classes is shown in Section 4.4.6.

Comparison of the ADT specifications may tell more about the similarity of two classes than a comparison of the implementation of operations, because the implementations are full of details irrelevant to the comparison. If the `Set` and `Stack` classes are implemented using different data structures (e.g., linked lists and arrays), it may be hard to find similarities in the bodies of the operations.

Thus, ADT specification is an alternative technique to comparing implementations of operations. Another possible technique could be to use function abstraction [HPLH90], which is a technique to find and describe the essential properties of operations by mathematical functions.

### 6.4.4 Extensional comparison (Step 2)

Given two classes $C_1$ and $C_2$ having extents $E_1$ and $E_2$. The extensional comparison is based on finding the semantic set relationship between two classes. The semantic
The set relationship between $C_1$ and $C_2$ is defined to be the set relationship between the sets of entity objects of $E_1$ and $E_2$.

To find the semantic set relationship between two classes $C_1$ and $C_2$, we must have an $\exists$ function to relate which proxy objects are entity identical. Thus, the semantic set relationship is defined as the set relationship between $E_1$ and $E_1$ using the $\exists$ function as a definition of equality between the set elements. If it exists an automatic $\exists$ function, it should probably be among the ACGs found in step 1. If it is not there, it must be manually defined. This may seem as a rather large work to do for the sake of guidance of integration of classes, but it has to be done in the integration any way.

We will use the notation $\text{Ent}(E)$ to denote the set of entity objects of the proxy objects in extent $E$. For the different semantic set relationships between $C_1$ and $C_2$, we will suggest appropriate EPLs according to the scheme (including alternatives) given in Table 6.2. There are some differences from the suggestions given by the

| $\text{Ent}(E_1)$ is-subset-of $\text{Ent}(E_2)$ | $C_1 \Rightarrow C_2$ |
| $\text{Ent}(E_1)$ equals $\text{Ent}(E_2)$ | $C_1 \Leftrightarrow C_2$  \hspace{1cm} $C_G \Leftrightarrow C_1$ and $C_G \Leftrightarrow C_2$ |
| $\text{Ent}(E_1)$ overlaps $\text{Ent}(E_2)$ | $C_1 \Leftarrow C_S$ and $C_2 \Leftarrow C_S$ |
| $\text{Ent}(E_1)$ is-disjoint-with $\text{Ent}(E_2)$ | No integration  \hspace{1cm} $C_G \Leftarrow C_1$ and $C_G \Leftarrow C_2$ |

Table 6.2: Suggesting integration based on extensional comparison.

intensional comparison in Table 6.1. While $I'_1$ is-subset-of $I'_2$ gave the suggestion $C_1 \Leftarrow C_2$, $\text{Ent}(E_1)$ is-subset-of $\text{Ent}(E_2)$ give the opposite suggestion $C_1 \Rightarrow C_1$. $I'_1$ overlaps $I'_2$ suggests a role generalisation, while $\text{Ent}(E_1)$ overlaps $\text{Ent}(E_2)$ suggests a specialisation class to cater for the “common” objects. For equals and is-disjoint-with the suggestions are the same.

Our semantic set relationships correspond to the assertions of Sheth et al. [SLCN88], which are specified prior to schema integration. This is similar to our relationships, but we build our “assertions” on $\exists$ functions which are used in the integrations themselves. This difference stems from our inclusion of object integration as a part of class integration, while [SLCN88] only considers integration on the schema level. Our use of semantic relationships is thus extensional, meaning that the semantic relationship between classes are depending on the extents. [SK92] uses semantic proximity, which is an intensional semantic relationship.

### 6.4.5 Confirmation (Step 3)

Based on the suggestions from step 1 and 2, a final decision on how to connect the extents is made. If there are any conflicts between these two suggestions, we will give the extensional comparison higher confidence because it tells something about
the relationships between the entity objects of the classes being compared. The intensional comparison is more mechanical by comparing the properties of classes. It finds out whether two classes are modelled similarly, and does not consider whether they model the same entity objects.

If the extent integration requires new classes to be created, it should be done now. Which properties to be included in the new classes, are the choice the integrator. In step 4 we will give some advice based on the results from the intensional comparison in step 1.

We might see the intensional comparison as a help in the extensional comparison, by possibly giving an $\mathcal{S}$ function to the comparison. In the conformation phase it gives extent integration guidance, and finally in the generation of ACRs/OCRs it suggests which attributes and operations to integrate.

### 6.4.6 ACR and OCR generation (Step 4)

In this phase $\mathcal{S}$ functions will finally be set, and ACRs and OCRs will be defined. This means that the definition will be made of how to integrate objects, attributes, and operations from the two classes. This phase makes heavy use of the results from the intensional comparison phase. In the two following integration situations:

$$C_1 \implies C_2$$

$$C_1 \iff C_2$$

we will use the results from the attribute and operation comparisons directly. Let $A$ be the attributes in $I_1$ and $B$ the attributes in $I_2$ corresponding to the common attributes in $I'_1 \cap I'_2$. For both integration situations we will suggest an $\text{ACG } A \oplus B$. Practically, this may be specified as many $\text{ACGs } A_i \oplus_i B_i \ (i = 1..N)$, where $A_i$ and $B_i$ are subsets of $A$ and $B$, respectively. In case of non-derivability in the ACGs, non-derivability mutation symbols may also be given.

We get similar situations for operations. Let $m$ be an operation in $I_1$ and $p$ an operation in $I_2$ corresponding to an operation in $I'_1 \cap I'_2$. In both integration situations we will suggest two OCRs:

```plaintext
after m { p() }
after p { m() }
```

As shown in Section 4.4.5, due to mutual dependencies, both OCRs need to have code to ensure termination of execution.

In the generalisation case:

$$C_G \leftarrow C_1, \ C_G \leftarrow C_2$$

we will be concerned about which properties the new class $C_G$ should have. For attributes we will decide this based on the ACGs found in between $C_1$ and $C_2$. Let $G$ be attributes in the new class $C_G$ corresponding to the attributes $A$ in $C_1$ and $B$ in $C_2$, for which the intensional comparison has found the comparison $\text{ACG } A \oplus B$. 

156
In Table 6.3 we have shown the choice of $G$ and new ACGs based on the comparison ACG. $G$ is chosen such that it can include both the values of $A$ and $B$. If we choose among $A$ and $B$, the choice is the same as the choice for shared representation of ACGs given in Section 4.4.2 and summarised in Table 4.1. In Table 6.3 we have only shown those situations where shared representation for $A \oplus B$ was possible. In those situations where $A$ and $B$ may equally well have been chosen, we have illustrated the new ACGs by choosing $A$. For the other situations, i.e. where shared representation was not possible:

<table>
<thead>
<tr>
<th>Comparison ACG</th>
<th>$G$</th>
<th>ACG for $C_G \Leftarrow C_1$</th>
<th>ACG for $C_G \Leftarrow C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow \rightarrow B$</td>
<td>$A$ or $B$</td>
<td>$G \rightarrow \rightarrow A$</td>
<td>$G \rightarrow \rightarrow B$</td>
</tr>
<tr>
<td>$A \rightarrow \rightarrow B$</td>
<td>$B$</td>
<td>$G \rightarrow \rightarrow A$</td>
<td>$G \rightarrow \rightarrow B$</td>
</tr>
<tr>
<td>$A \rightarrow \rightarrow B$</td>
<td>$A$</td>
<td>$G \rightarrow \rightarrow A$</td>
<td>$G \rightarrow \rightarrow B$</td>
</tr>
<tr>
<td>$A \rightarrow \rightarrow B$</td>
<td>$A$</td>
<td>$G \rightarrow \rightarrow A$</td>
<td>$G \rightarrow \rightarrow B$</td>
</tr>
<tr>
<td>$A \rightarrow \rightarrow B$</td>
<td>$A$ or $B$</td>
<td>$G \rightarrow \rightarrow A$</td>
<td>$G \rightarrow \rightarrow B$</td>
</tr>
</tbody>
</table>

Table 6.3: Generalisation attribute set and new ACGs.

we will not give any advice to which attribute to choose, but leave it to the integrator. Here, it is very likely that a choice different from $A$ and $B$ must be made. In the $A \rightarrow \rightarrow \rightarrow B$ situation (illustrated in Figure 4.10), $G$ should be an attribute capable of representing both the extended part of $A \rightarrow B$ and $A \leftrightarrow B$. E.g., if $A$ is an attribute of the subrange 1..100 and $B$ of the subrange 50..200, and there exist a comparison ACG $A \leftrightarrow \rightarrow B$, $G$ could be defined as a subrange 1..200.

The new ACGs shown in Table 6.3 are suggested according to the following principle: If $G$ is chosen to have the same domain as $A$, $G \rightarrow \rightarrow \rightarrow A$ will be suggested. As $A$
and $G$ are totally derivable from each other, the other new ACG will then be the same as the comparison ACG (possibly reversed in the table).

In case there are operations in $I_1 \cap I_2$, we will let $r$ be an operation in $C_G$ corresponding to $m$ (of $C_1$) and $p$ (of $C_2$), which are found to be similar. For $C_G \subseteq C_1$ we will suggest the following two OCRs:

$$\text{after } r \{ m() \}$$
$$\text{after } m \{ r() \}$$

For $C_G \subseteq C_2$ we similarly suggest the following two OCRs:

$$\text{after } r \{ p() \}$$
$$\text{after } p \{ r() \}$$

As skeleton for the definition of the $r$, we may use the results from the operation comparison in step 1. We will suggest that $r$ is initially defined as the “intersection” of the comparison of $m$ and $p$, and is edited simultaneously with the display of both $m$ and $p$. This holds both for the definition of the signature and the body. Similar techniques may be used for providing ADT specification for the generalised class $C_G$.

The intensional comparison in step 1 disregarded whether properties were private or public. If properties in $C_1$ and $C_2$ are found to be similar in the intensional comparison, and they have different visibility modes, we get a dilemma of the choice of visibility mode in $C_G$. For example, should an attribute being public in $C_1$ and found to be similar to an attribute being private in $C_2$, result in a public or private attribute in $C_G$? This choice must be made by the integrator, and it must go along with the choice of the property to be included in $C_G$.

For the specialisation situation:

$$C_1 \subseteq C_S, C_2 \subseteq C_S$$

we will suggest $C_S$ to have the “union” of the properties in $I_1$ and $I_2$, and do an equivalent decision as in the generalisation case for the properties in the “intersection” of $I_1$ and $I_2$.

We have presented the integration of attributes and operations (by ACRs and OCRs, respectively) of two classes as two tasks that must be done in a class integration. However, often either the operations or the attributes must be integrated. We tend to think of classes as developed in two different categories: Either they are structural (having public attributes), or they are encapsulated with public operations and private attributes. In the structural situation, it is necessary to integrate attributes, while in the encapsulated situation, it is necessary to integrate either the operations or the attributes. Since the operations are the interface and attributes the representations, integrating both is the same thing twice. The issue of whether we should integrate operations (using OCRs) or attributes (using ACRs) was also discussed in Section 4.4.6.
6.5 Integration interference

Section 6.4 presented a binary integration methodology. When integrating classes existing in extent graphs, e.g., by integrating two schemas, we might get interference between different class integrations.

6.5.1 Extent propagation links

Consider Figure 6.6, where we at the extent level have illustrated two component schemas which are integrated in a federated schema. In the component schema Sch1 we have two classes Person1 and FacultyStudent, and in the component schema Sch2 we have the classes Person2 and UniversityStudent. Class Person in FedSch is created as a generalisation of the person classes in the two component schemas. Student is similarly created as a generalisation of FacultyStudent and UniversityStudent. These two generalisations are done without considering each other. If Student is concrete, we will have some problems in the federated schema: objects created in Student will not be propagated to Person.

There are several alternative integrations to cater for this problem. One alternative is to use two-way EPLs between Student and the student classes in the two component schemas. But this solution does not visualise the relationship between Student and Person in the federated schema. In Figure 6.6 we have illustrated an extra EPL with a broken line arrow. This extra EPL “integrates” the two generalisations. The
conclusion of this small example is that different classes created as results from binary integrations of classes in the same extent graphs, should be reviewed for possible missing internal connections.

### 6.5.2 Object consistency specifications

In Section 4.4.5 we saw that several ACRs interfered during consistency maintenance by being mutual dependent, i.e. OCRs triggered each other recursively. This was solved by introducing marking flags to never “touch” an attribute more than once for one consistency maintenance. Ensuring termination is not the only problem of mutually dependent properties; we may also get semantic conflicts, especially in extent graphs being highly interconnected. E.g., given three classes $C_i$ ($i = 1..3$) having one attribute $a_i$ each, and the three EPLs below:

$$
\begin{align*}
C_3 & \leftarrow C_2 & a_3 &= a_2 + 1 \\
C_3 & \leftarrow C_1 & a_3 &= a_1 + 1 \\
C_2 & \leftarrow C_1 & a_2 &= a_1 + 1
\end{align*}
$$

For simplicity we have described the ACGs as mathematical equations. The problem that have appeared here is that each attribute $a_i$ is dependent on the other two $a$’s along two different paths. Along one path, $a_3 = a_1 + 1$, and along another it is $a_3 = a_1 + 2$. Assuming the “visited flags” presented in Section 4.4.5, mutating $a_1$ to the value 3 might give the value 4 or 5 in $a_3$ and 4 or 3 in $a_2$, all depending on the execution order of the OCRs implementing the consistency maintenance. As noted in Section 4.3.7 these ACGs make it impossible to provide object consistency, because all ACGs cannot be satisfied at the same time. We considers such situations to be in *semantical conflict*, and by analysing the ACRs we can notify the class integrator about possible conflicts.

This situation might appear for example when two different persons are defining the two EPLs going into $C_3$. We will not try to deduct what the reasonable semantics of such a situation should be. Instead, we will let database system check for cyclic dependencies, and whenever such cycles are found, it tells the class designer that a cycle has been found, and that it may possibly result in a conflict. The class designer, who introduced the ACRs resulting in that conflict, is then responsible for taking proper actions to solve the conflict, or to approve that it is correct.

When an attribute takes part in several ACGs within one EPL, we may also get more indirect interference problems. Consider one of the EPLs and its connected ACGs in the role generalisation example from Section 6.2.1:

<table>
<thead>
<tr>
<th>Student</th>
<th>⇐</th>
<th>FacultyStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td></td>
<td>(name)</td>
</tr>
<tr>
<td>(startYear)</td>
<td></td>
<td>(startYear)</td>
</tr>
<tr>
<td>(department)</td>
<td></td>
<td>(department)</td>
</tr>
<tr>
<td>(faculty)</td>
<td></td>
<td>(department)</td>
</tr>
</tbody>
</table>

Here, department of FacultyStudent is dependent on both department and faculty of Student. If faculty is mutated in Student, the department attribute of Student will be
affected by consistency maintenance indirectly through department of FacultyStudent. This simple problem could have been solved by omitting the second ACG, and by having an integrity constraint in Student involving department and faculty. Generally, we see operations as the means for integrity constraints in object-oriented databases, but this example could also easily be expressed by ACRs if they were allowed “inside classes”, i.e. (faculty) $\leftrightarrow$ (department).

6.6 Tool support

The section outlines some tools which may be of help in integration. These tools are generally useful for schema integration, but here we will connect them to the specific methodology for class integration presented in Section 6.4.

6.6.1 Comparing intents

To compare intents in step 1 we need to have conformed intents. Instead of conforming the intents explicitly, we will adopt a technique used for search in libraries, e.g., in software reuse libraries [KSST92]. During comparison we will find the “semantic nearness” between names by using term spaces, which are data structures representing the semantic nearness between different names. A term space will be represented as a weighted undirected graph, where each node represents a term (name), and each edge the semantic nearness between the two terms represented in the associated nodes. The weight will be a value between 0.0 and 1.0. 0.0 means there is no connection (i.e. there will be no edge between the terms), while 1.0 means the terms are complete synonyms.

We will have one such graph for each of the following categories of terms:

- operation names
- attribute names
- class names
- domain names

In Figure 6.7 we have illustrated an example term space for operations. In the graph we have made use of two different edges: full lines have the additional semantics that they represent terms being specialisation/generalisation of each other, while the broken lines represent mere synonyms (with no additional semantics).

The utilisation of the term space is during comparison of intents. Two properties will be considered equal if they have a semantic nearness above a certain limit, e.g. above 0.2. The semantic nearness between two terms directly connected in the graph is the weight on the edge. For terms connected transitively, the semantic nearness is defined as the product of the weights along the path between those two terms. If there are several paths, the path with the highest value will be chosen.
We can also use this to conform intents explicitly by creating new intents. If the names are found in the term space, the technique may also be used for other properties of classes than operations. E.g., attribute domains may be related in the term space. An example: the domain `feet` may be connected to `meter` by the weight 0.9. This can be used to uncover scale differences.

It is the class integrator who has to do the actual integration. The term space tool can only `suggest` what to do based on comparisons. A partial conclusion of this tool is that the foundation for integration is set at the time of design of the classes, i.e. by choosing appropriate names.
6.6.2 Finding candidate classes for integration

If there is a large set of classes to be integrated (e.g., what is referred to as *industry size problems* in [She92]), it may be a large work to manually find pairs of classes to be integrated. We will address this problem by the use of two techniques from software engineering and software reuse.

The first approach is to use an affinity browser [PT91], which is a tool to interactively display the nearness of a set of classes. The nearness may be defined using several different criterias, and the affinity browser may have several windows displaying the nearness according to the different criterias.

We will show two possible *affinity criterias* defined on two intents. Given two conformed intents $I'_1$ and $I'_2$, where $opr(I)$ denotes the set of operations in intent $I$, the *operation affinity* is defined as:

$$OA(I'_1, I'_2) = \frac{\#(opr(I'_1) \cap opr(I'_2))}{\#(opr(I'_1) \cup opr(I'_2))}$$

A corresponding *attribute affinity* is shown below:

$$AA(I'_1, I'_2) = \frac{\#(attr(I'_1) \cap attr(I'_2))}{\#(attr(I'_1) \cup attr(I'_2))}$$

These two different affinities could for example be used for encapsulated and structural classes, respectively.

![Diagram](image)

Figure 6.8: Displaying nearness of classes.

In Figure 6.8 we have shown a possible display of the affinity browser, where one class Cx is displayed together with a set of classes C1 through C7, which it has been compared with. Here, class C5 is most similar to Cx. Other capabilities of the affinity browser could be to inspect the intents and extents of classes, e.g., by “clicking” at the classes.

Another possible technique for finding candidate classes for integration, is to classify schema entities, which again is a technique borrowed from software reuse. E.g., the
**faceted classification** of Diaz et al. [Pri89, OHPB92]. This can be adopted in our class integration methodology by defining a set of **facets** (attributes) that each class should be classified according to, which means that each class should have a specific value (term) for each facet required. The value domains for each facet should be organised as term spaces. Two components are then compared for similarity by comparing the values of the facets, and not by comparing the intents themselves. This requires each class to be classified according to standard terms before they are integrated. According to experiences from software reuse, it is advantageous to do this classification *during design* of components, and not in retrospect [HKS+91].

### 6.6.3 Customised integration guidance

After the differences between intents have been uncovered, as seen in Section 6.4.3 and 6.4.6, we gave some general rules for how to integrate classes based on comparing the intents and the extents. This guidance can be customised by letting the user insert rules telling how classes should be integrated for different intensional and extensional comparisons.

Below, we show an example of a rule `right_epl` expressed in Prolog, saying that there should be an EPL $C_1 \Rightarrow C_2$ if the intent of $C_1$ **conforms** to the intent of $C_2$ and the visibility of $C_1$ **implies** the visibility of $C_2$:

```
right_epl(C1,C2) :-
    intent(C1,I1),
    intent(C2,I2),
    conforms(I1,I2),
    visib(C1,V1),
    visib(C2,V2),
    implies(V1,V2).
```

Here, **conforms** may be defined according to the conformance rules of Emerald [BHJL86], and **implies** defined according to logical implication, meaning that the visibility of class $C_1$ is within the visibility of $C_2$ (cf. Section 5.6.3).

Guidance may also be given to the generation of ACRs. The **poss_acg** rule below shows that if there is an ACG between two domains, two attributes of those domains may have the same ACG between them:

```
poss_acg(A1,A2,ACG) :-
    domain(A1,D1),
    domain(A1,D2),
    acg(D1,D2,ACG).
```

The general rules given for class integration in the steps of the methodology are ad hoc. The intention with the customised guidance is, as an integrator gets more experience with class integration, the experience may be recorded by making new guidance rules. Thus, the customised rules may be in a knowledge base recording
class integration knowledge. An example of this is to give feedback to the term space in Figure 6.7 from the experience with integration based on the term space.

Other possible application areas for the customised guidance are to recognise typical abstractions (e.g., list and tree structures will usually have links to themselves) and for domain-specific integration. When a database application area utilises the object model in special ways, which again requires special ways of integrating classes, this knowledge could be recorded in a knowledge base. Cf. the experiences from the Carnot project [CHS91].

6.7 Comparison

6.7.1 The framework as a canonical data model

[SCG91] discusses the factors that a data model should possess to be a canonical data model (CDM) for federated databases. Although [SCG91] does not claim to complete or definitive, we will use it to discuss the suitability of the framework as a CDM. [SCG91] identifies two important factors for canonical data models: the expressiveness of the model and the degree that it supports semantic relativism. Expressiveness is defined to be the degree the model can directly represent a conceptualisation, no matter how complex this conceptualisation is. A basic requirement here is that the CDM is rich enough to at least express the semantics of the component schemas (cf. Figure 2.1). At a more detailed level [SCG91] finds the following characteristics of expressiveness that should be supported by a CDM:

Classification/instantiation: This is supported by the main constructs of our basic object model: classes and objects.

Generalisation/specialisation: As we have seen in Section 6.2.1, this is supported by appropriate use of EPLs, $\exists$ functions, ACRs and OCRs.

Aggregation/decomposition: This is supported by different constructs: The basic object model provides record as a construct for aggregating attributes, and containment links to make composite objects. As seen in Section 6.2.2, we may apply the evolution constructs of the framework (EPLs, $\exists$ functions, ACRs and OCRs) for aggregation and group integration.

Operations and integrity constraints: The basic object model supports the definition of new operations as properties of objects. We think of operations as the basic means for expressing integrity constraints in object-oriented data models. This is done by letting the representation be encapsulated and by having public operations implementing (and respecting) the integrity constraints.

Semantic relativism is defined to be the degree to which the data model can support the different conceptualisations (of the real world) different users make. On a detailed level, this means that the CDM must support:
**Integration operators:** The framework supports “integration operators” by the use of the evolution constructs: EPLs, 3 functions, ACRs and OCRs. We think this is one of the strongest points of our framework, since we allow for integration using any combination of the framework components. We are not limited to a fixed set of predefined operators.

**View mechanism:** As seen in Section 4.6.3, and as will be seen in Section 7.3, the semantics of our classes is very similar to the semantics of views. Thus, our classes can be partly used as views.

**One basic structure:** Objects are used as the basic structure of our framework. The basic object model supports some attribute domain constructors (e.g., record and setOf) as well. But these are not intended to be used as the main structure in integration.

**Multiple semantics:** By letting an object be member of multiple classes, it may have multiple semantics. However, the dependencies between these semantics must be described using ACRs/OCRs.

We think this shows that our framework meets the requirements of [SCG91] very well. [SCG91] draws the conclusion that an object-oriented data model equipped with an adequate language for expressing views, is well suited as a CDM.

### 6.7.2 Generalisation by upward inheritance

Generalisation is the most obvious technique used for class integration. Generalisation by upward inheritance [SN88, SN90] is concerned with integrating existing classes by defining new abstract, generalised classes, which “smooth” over the existing classes. When an object is requested through a generalisation class, the request will be forwarded to the class the object is represented in (this is the upward inheritance). Our class integration is rather different, since it defines the generalisation classes as normal classes with their own normal attributes and operations. The integration information is provided by defining EPLs with 3 functions, ACRs and OCRs attached. However, the object correspondence rules of [SN88] is quite similar to our object correspondences provided by 3 functions.

The differences can be seen in the power plant example in Section 3.5 and 6.2.1. While the upward inheritance approach defined two special integration classes, PowerPlant as a category generalisation class and MJoule as a data type generalisation class, our framework provided for the integration by two external EPLs using ACRs and OCRs.

The generalisation classes of [SN88] are mere behaviour to operate on other objects, while our classes are normal classes which may be used as any other class. This means that in our approach a generalised class may be further evolved. We do not know if this is possible in the upward inheritance approach. That approach incorporates rather complex rules for forwarding operation request, and it is not clear to us whether these will hold upon further class evolutions.
The upward inheritance approach may be more efficiently implemented than our OCRs. Upward inheritance is tailored towards the task of forwarding operation requests, while our OCRs are more or less general database triggers. If we facilitate shared representation of ACGs (cf. Section 4.4.2), our approach becomes very similar to the upward inheritance approach. Thus, in this case it may possibly achieve the same efficiency.

6.7.3 ViewSystem

ViewSystem integrates classes by creating derived classes using a predefined set of class constructors, which are based on a set of known semantic relationships between classes: Specialisation, category generalisation, role generalisation, grouping, and aggregation. The derived classes are views to the existing classes, but possibly extended with their own operations. The operations of the derived classes will forward messages to the classes being integrated. The integration capability of ViewSystem is limited to the predefined set of operators, while our approach allows for any kind of integration expressible with the components of the framework. Another difference is that the derived classes of ViewSystem are views with no other purpose than integrating classes, while our classes are normal classes which can be used for creation of objects and further evolution. ViewSystem may also derive derived classes, but they cannot be used for creation of objects.

6.7.4 Attribute equivalences vs. ACGs

Our work on ACGs is influenced by the attribute equivalences of [LNE89]. Attribute equivalences are partitioned into strong and weak equivalences between two attribute sets A and B. For both categories there exist a function from a subset of the domain of A to a subset of the domain of B. For strong equivalences there exist an inverse function (from a subset of B to a subset of A). Functions defined on the whole domain of an attribute set correspond to totally derivable ACRs, while functions defined on a subset of the domain of an attribute set correspond to partially derivable ACRs. ACGs only involving totally and partially derivable ACRs correspond to strong equivalences, while ACGs involving non-derivable ACRs correspond to weak equivalences.

[LNE89] further categories equivalences into either α or β equivalences, which are equivalences holding at some and all points in time, respectively. We mainly use our ACGs as specification of dependencies which hold at all points in time. In this sense, they correspond to β equivalences. However, an ACR is only a specification of a dependency, but being accompanied by a function implementing the dependency. As will be seen in Section 7.1.3, we allow for the implementation of a specification to change, as long as it preserves the specification. An ACR specification only tells that it exist a function being partial or total in between the domains of the attributes, or that it does not exist a function.

[LNE89] defines several characteristics of attributes which should hold in attribute equivalences. E.g., every allowable operation on the attributes in A must have a
corresponding operation on $B$, all integrity constraints are preserved by the dependencies, etc. These extra characteristics of attributes are results of the semantic data model underlying the work of [LNE89], and are less relevant for a more object-oriented data model.
Chapter 7

Implementation Issues

This chapter treats some issues connected to implementation of the framework presented in Chapter 4. We will focus on how existing high-level database constructs can be used to implement the different components. We start by discussing implementation techniques for the components which are specifically related to class evolution. Then, we present the prototype we have implemented on top of the database system developed in the EPOS project. At the end of this chapter we will show how object-oriented views may be used to implement class evolution. This chapter does not address architectural issues, neither does it address file-level implementation details. Section 7.3 is partly published in [Bra92].

7.1 Implementation issues for the framework

Chapter 4 presented the framework for class evolution, including some constructs which were oriented towards implementation. This section treats additional issues concerned with implementation. We divide this presentation into three: First we discuss how the semantics of extent propagation links will be implemented. Then we present issues connected to consistency maintenance due to ACRs/OCRls. At the end we present some issues concerning management of ACRs.

7.1.1 Extent propagation

Recapturing from Section 4.4.1, extent propagation is the process of ensuring that all objects being member of the class $C$, have a corresponding object in all classes in $EPS(C)$. This means propagating objects in between extents due to EPLs, or checking whether corresponding objects exist in the extents connected by EPLs. The framework defines the following existential operations in connection with extent propagation: create, delete, add, and remove. In Section 4.2.2 we described the semantics of these operations with respect to extent graphs.

The framework assumes objects are allowed to be added to classes with given OIDs. Iris [F+89] allows for this by having a construct for adding a class (type in Iris) to
an object. However, most object-oriented database systems do not. This restriction is tightly connected to the restriction to not allow objects to have multiple "most specific" classes (discussed in the context of dynamic binding in Section 4.2.4). As argued for in [Ste91], the name binding problem with multiple most specific classes is essentially equivalent to the name binding problem with multiple inheritance. If an object model allows for multiple inheritance using a name conflict resolution technique, there is no reason for not supporting multiple most specific classes as well.

For object-oriented databases not supporting the capability of adding an object to a class, a simulation of the addition ability must be taken, e.g., by having links between corresponding objects.

As said in Section 4.4.1, extent propagation may either be done eagerly upon requests to the existential operations, or lazily upon retrieval of objects. For eager extent propagation we see the following possibilities of implementation:

**Hard-coded in the existential operations:** If there is a creation operation associated with every class, e.g., the constructors of C++ [Str86] or the new methods of Smalltalk-80 [GR83], a set of add requests can be hard-coded in the creation operations. Upon a class evolution, this solution requires possibly all creation operations to be modified to capture the new classes. This solution allows the propagation to be efficiently implemented, and it does not create problems when there are cycles in the extent graph.

**Interpreted by the existential operations:** When having the extent graph explicitly represented in the database system, extent propagation may be implemented by letting the existential operations interpret the extent graph. This solution means that new EPLs can easily be added to the system. However, the interpretation of the extent graph may be inefficient, as will be seen in the prototype (Section 7.2).

**Triggers connected to the existential operations:** In Section 4.4.1 we said that extent propagation can be implemented by OCRs. This can be done by connecting them to the the existential operations, e.g., by having one OCR per EPL. When there are cycles in the extent graph, this approach must explicitly ensure the propagation to terminate.

The following approach implements lazy extent propagation:

**Views:** Extent propagation can be implemented by letting classes be implemented as views querying the extent of other classes upon retrieval of objects. This is only possible in its pure form where objects can be completely represented in their creation classes.

An EPL $C_1 \Rightarrow C_2$ may be interpreted as "an object in $C_1$ has a corresponding object in $C_2$". According to Section 6.3.2 these correspondences are registered automatically, manually, or in a combined manner. Automatic registering requires the objects
in $C_1$ and $C_2$ to have attributes which were unique within the class. This correspondence information can be embedded in the extent propagation implementation. This may be implemented in the following ways: Hard-coded in the extensional operations, in the explicitly represented extent graph, or in the triggers connected to the extensional operations.

Manual registering may be done be either embedding corresponding object references in each object, or by having external registering of object correspondence tuples. The choice between these two depends on the nature of the evolution. If an object often acquires new corresponding objects, it may be better to use the latter approach, while in a more stable situation the former solution may be best. When corresponding objects are existing in different databases, this registering must either be done in both component databases, or in a separate “integration database”.

### 7.1.2 Implementing ACRs and OCRs

Chapter 4 treated implementation of ACRs both for shared and separate representation of attributes. The main techniques we used to implement consistency maintenance were:

1. OCRs were used to propagate mutation of attributes between corresponding objects in classes being connected by EPLs. OCRs could either be used directly between operations, or as generated implementation of ACRs. The consistency maintenance could be done either eagerly or lazily using *after* and *before* timing modes for OCRs.

2. For lazy consistency maintenance “dirty” flags were used to mark objects having “dirty” attribute values.

3. Shared representation of attributes was hidden inside attribute request operations.

The technique of embedding the consistency maintenance into the request operations of attributes may actually be used for all kinds of consistency maintenance, not only for the shared attribute representation. Similar to the implementation of the extent propagation, this can either be hard-coded or interpreted by the attribute request operations. The hard-coded way requires the operations to be reflected by the changes done upon class evolution, i.e. large portions of the schema must be recompiled and get new attribute request operations. OCRs are more modular than this technique, because they can be added to a schema without modifying the existing classes.

In this section we will focus on consistency maintenance by OCRs, and how an Event-Condition-Action (ECA) model of database triggers [MD89] may be used. We will also discuss some approaches to interdatabase dependencies which may be used to extend our ACRs/OCRs to multidatabases.

An *ECA rule* consists of three parts: an *event*, a *condition*, and an *action*. When the database system detects an event to have happened, it evaluates the condition,
and if the condition is satisfied, it executes the action. This model can be used to implement our OCRs. The events of our OCRs are operation requests. A condition will either always be true, or it could be used to check for “dirty” or “visited” flags. The action corresponds to the body of our OCRs. Given the following OCR:

```c++
exitprop C2 => C1 {
  ocrs:
    timing op (i1, ..., ik) body
};
```

If we assume op to be an operation of C2, the OCR would be expressed as an ECA rule as follows:

```
ON C2::op (i1, ..., ik)
IF true
DO body
```

The event is here the request of op of class C2. The body of this OCR needs to know which object op was requested for. This information could either be a part of the signature, or it could be provided through an implicit reference to the object, e.g., this in C++ or self in Smalltalk-80. The timing of our OCRs is expressed by the coupling modes in the ECA model. It supports three coupling modes: immediate, deferred, and decoupled, which may be specified between the event and the condition, and between the condition and the action. Our before and after correspond to ECA’s immediate and deferred, respectively.

When corresponding objects exist in different databases, we must extend the consistency maintenance scheme to capture interdatabase dependencies. [Geo92] introduces dependency descriptors (DD) on the distributed object management (DOM) (multidatabase) system level. A dependency descriptor consists of a dependency specification (DS) and a dependency implementation (DI). The dependency specification allows for the specification of equality between attributes, and may thus be seen as special cases of ACRs. Dependency implementations are ECA rules specified on the DOM system level. [Geo92] claims the typical coupling modes of ECA rules cannot be enforced without compromising the autonomy of the component databases.

[RSK91] presents a slightly richer approach to interdatabase dependencies. It defines a data dependency descriptor $D^3$ as a specification $(S, U, P, C, A)$, where $S$ and $U$ are the source and target objects, $P$ is an interdatabase dependency predicate, $C$ is a mutual consistency predicate, and $A$ is a consistency restoration procedure. $C$ corresponds to the event, $P$ to the condition, and $A$ to the action of an ECA rule. $D^3$ extends the ECA model by allowing $C$ to be connected to specific events in the database or to temporal terms, e.g., on a specific time or at intervals. Unlike ECA, where the objects involved are implicitly given, $S$ and $U$ explicitly denote which objects are involved. This suits the implementation of OCRs where corresponding objects do not share OIDs.
7.1.3 Management of ACRs

In Chapter 6 we argued for making the attribute domain definitions global. The framework depends much on ACRs, being specification of dependencies between attributes. Our intention is that ACRs should not have to be implemented from scratch for every class evolution. Here, we will suggest to also represent ACRs globally together with the implementation of their functions. This has two main advantages: First, ACRs may be defined between the relevant domains, making them reusable for all attributes of these domains. Second, it makes class integration easier, because it may automatically lead to integration of attributes. EPLs must be organised such that they contain references to the ACRs they include.

The functions implementing ACRs would be specified in the same language as operations. Such a function could be implemented intensionally, e.g., an algorithm that holds for all values of the connected attribute domains. The example below converts from feet to inches:

```c
domain feet = float;
domain inch = float;

inch foot_to_inch (feet f) {
    return 12*f;
}
```

They may also be specified extensionally, i.e. they list the correspondences between the domain explicitly. The example below converts from departments to faculties:

```c
facDmn dep_to_fac (depDmn dep) {
    if (dep == math) return math_phys;
    else if (dep == phys) return math_phys;
    else if (dep == cs) return ee_cs;
    else if (dep == cyb) return ee_cs;
    else if (dep == ee) return ee_cs;
    else return null;
}
```

Maps [ALPR91] may also be available as a primitive in the underlying system.

In Section 4.4.6 we stressed that ACRs are specifications, which gave them some advantages compared to OCRs. One of the main advantages of separating specification from implementation in programming languages, is that the implementation of a specification may be changed without affecting the clients of the specification [LG86]. This can be used for ACRs as well: As long as the implementation of the function preserves the specification, the implementation may be changed. E.g., given the ACR department → faculty, and the implementation above. If an existing department is split into two, the dep_to_fac function should be changed to reflect this. However, this does not change the specification of the ACR.
7.2 CEFP: Class evolution framework prototype

We have developed a simple prototype named CEFP (Class Evolution Framework Prototype) atop the EPOS database, which is a prototype database developed to support software engineering environments. CEFP implements all components of the framework by creating a thin layer on top of the EPOS database. The intention with CEFP is to verify the completeness of the design itself, i.e. does it contain holes? In addition to this, it serves as an example of an implementation of the framework. We give a small introduction to the EPOS database before we proceed with the presentation of the prototype CEFP.

7.2.1 The EPOS database

The EPOS database is a prototype of a database supporting software engineering environments [Lie90]. It has an entity-relationship data model extended with inheritance for both entity and relation types, and it supports the change-oriented versioning (COV) model. The data model is structural and has a set of predefined generic operations. It allows for attributes of various domains both for entity and relation types. Entity types allow for creation of objects with unique OIDs, and having the set of attributes described by the entity type. Entity types have system-maintained extents, which are available for queries. Entity types are organised in subtype hierarchies, which facilitate both inheritance of attributes and inclusion of extents. Relation types are binary, i.e. they are defined between two entity types. The two roles of a relation type may be restricted by upper cardinalities. Relation types do also exist in type hierarchies, facilitating what we call relation refinement [BO92]. A special entity type for long-fields is predefined to allow the representation of bulks of data not interpreted by the database. A very useful facility of the EPOS database is its support for type descriptors, being represented as ordinary database objects. Thus, any application can query the schemas as normal objects.

The EPOS database has been developed since 1989, and is now existing in its third major revision. It is developed in C on top of C-ISAM, an index-sequential record storage system. The database is implemented according to a client-server model over a local network using Sun RPC. The main bulk of work is done by the server, the clients are mainly programming language interfaces with some minor caching of schema information. Currently, a high-level Prolog interface and a low-level C interface are supported.

7.2.2 The object model of CEFP

The object model supported by CEFP is a subset of the framework presented in Chapter 4. The limitations are mainly concerning the richness of attribute domains. We adopt the attribute domains supported by the EPOS database, which mainly correspond to the basic attribute domains of the framework. Link domains are implemented as relation types, but with no support for containment links. The EPOS database does not support user-defined operations. The prototype extends
this by allowing for user-specified operations connected to classes. The operations are implemented in Prolog.

7.2.3 The interface of CEFP

The interface to CEFP, named OBI, has a set of generic operations working on the CEFP classes defined in a schema. Ideally a schema should be defined in a syntax similar to the one used in the examples of this thesis, and from this be compiled into the definitions necessary for the database system. Presently we do not support any schema compiler – the schema must be manually translated into the EPOS type definitions. The EPOS database does neither have any schema compiler – schemas are created in the EPOS database by creating a set of objects of the predefined type descriptors.

CEFP applications use the prototype through OBI, which is a thin layer built on top of the Prolog interface of the EPOS database. We adopt the special notation of SWI-Prolog [Wie91], where + in front of a variable means that it must be instantiated, - means that the variable will be instantiated inside the predicate, while ? means that both these may be used. OBI has the following generic operations:

- **cre_O (+Class, -O, +Attr.List)**
  
  Create an object O in a class Class with a given set of initial attribute values Attr.List. This operation ensures the object to be propagated to all classes in EPS(Class).

- **del_O (+O)**

  Delete an object O from all classes it is member of. This holds for corresponding objects as well.

- **add_O (+Class, +O, +Attr.List)**

  Add an object O to a class Class. The object achieves the initial attribute values in Attr.List.

- **rmv_O (+Class, +O)**

  Remove an object O from a specific class Class.

- **[+Class, ?O] => [+OP1, [?PL1], +OP2, [?PL2], ...]**

  Request a list of operations (OP1, OP2, ...) with parameter lists (PL1, PL2, ...) for an object O in a given class Class.

There are no generic operations to mutate and observe attributes directly. cre_O and add_O allow for the attributes to achieve user-specified initial values. In addition to this, the schema compiler should for every attribute generate attribute request operations. E.g., the attribute rho has rho (+NewRho) to mutate and rho (-Rho) to observe. The generic operations are presented in more detail in Appendix B.
7.2.4 The schema of the prototype

The EPOS schema implementing the prototype is shown in 7.1. Entity types are illustrated as rectangles and relation types as ovals. The types which are filled gray in the figure, are predefined in the EPOS database. We have not included attribute definitions in the figure.

![EPOS Schema Diagram](image)

Figure 7.1: The EPOS schema implementing the class level of CEFP.

A class in CEFP will be represented by an object of `ent_td` (entity type descriptor). An attribute definition will be implemented as an object of `vattr` (attribute with visibility mode), and an operation definition as an object of `voper` (operation with visibility mode). `vattr` is rather special since it is created as a subtype of the predefined type `attr`. The EPOS database has the semantics of the predefined relation type `hasa` hard-coded: it ensures that an object will be allocated with space for the attributes, and that it can interpret queries and updates to the attributes. An object of `vattr` will be an object of `attr` as well, and will be understood by the EPOS database as a normal attribute definition. However, the attribute definitions added in `vattr`, will be plain attributes of the objects of `vattr`, i.e. with no special meaning to the EPOS database.

An EPL is described by an object of `ext_prop`, which is an entity type related to the two `ent_tds` it propagates from and to. An `ext_prop` has a set of `ocrs` (OCR)s.
A two-way EPL (\( \Leftrightarrow \)) will be represented as two distinct `ext.props`, but they are allowed to share `ocrs` (due the N cardinalities of `has.ocr`). The `derived_from` and `derived_to` relation types are used when OCRs implement ACRs. They are used to relate dependent attributes, such that the right attributes may be marked “dirty” and “visited” during consistency maintenance. `triggered_by` is used to connect OCRs to their operations.

Until now we have described the EPOS schema implementing CEFP on the class level. In Figure 7.2 we have illustrated the EPOS schema representing objects in CEFP. Objects in the EPOS database are only allowed to have one most specific type. This is done to allow for dynamic binding between a request of an attribute and the attribute of an object. Since we allow for objects to be members of multiple most specific classes, we have to implement one CEFP object by multiple EPOS database objects. Every CEFP object will be represented as an object of `identity`. The OID of this EPOS object will be the OID of the CEFP object as well. Every class in an CEFP schema will be implemented by a corresponding type and relation type in the EPOS database. In Figure 7.2 we have illustrated how two classes `PolarLoc` and `CartLoc` are implemented by EPOS types. `PolarLoc` will be implemented by the entity type `polarLoc` and the relation type `polarLocr`. The EPOS type `polarLoc` will have the same attributes as the CEFP class `PolarLoc`, but will in addition have the attributes of the predefined supertype `frag_type`. `frag_type` is used to contain administrative information for consistency maintenance (i.e. “dirty flags”). The relation type `has_corr` relates corresponding objects.

The EPOS type `polarLoc` is represented through a type-descriptor object of type `ent_td`. For each attribute of `PolarLoc`, the type-descriptor object will have a relationship to an “attribute descriptor” object of `vattr`. Similarly, for each operation it will have a relationship to an “operation descriptor” object of `voper`.

---

**Figure 7.2: The EPOS schema implementing CEFP at the object level.**
7.2.5 Consistency maintenance

According to the definition of object consistency in Section 4.3.6, object consistency maintenance has to ensure the following:

1. Objects are consistent in all classes.
2. Objects are propagated to their classes’ EPSe.
3. All ACGs are satisfied.

The definition of class evolution transparency extends this by telling that objects should be consistent upon observer requests.

The first problem is solved by relying on the attribute initialisation capability of the EPOS database. Every attribute may be defined to have a default value. If a default is not specified for an attribute, the attribute will achieve the default value of the attribute domain, e.g., 0.0 for float and false for bool. The EPOS database ensures that no attribute holds a value outside its domain.

The second problem is solved by letting the generic operations create, delete, remove, and add ensure this to be true. The classes to propagate to are available by traversing the prop_to and prop_from relations between the type descriptors of the corresponding types, and the corresponding objects are available through has_cor and frag_rel. Cycles in the extent graph do not cause problems for the extent propagation, because objects are not propagated further if a corresponding object is found.

The third problem is solved both according to shared representation (presented in Section 4.4.2) and separate representation (presented in Section 4.4.5). Shared representation is implemented by embedding the derivation of attributes into the attribute request operations. Given the ACG \((x, y) \leftarrow \rightarrow (\text{rho}, \text{theta})\), where we choose to have shared representation in \((x, y)\), we embed the functions of the ACG into the attribute request operations of \text{rho} and \text{theta} according to the scheme outlined in Section 4.4.3.

For separate representation we have introduced the entity type \text{frag_type}, which has defined an attribute to hold “dirty flags”. For each class an object is a member of, it will have an EPOS object of a subtype of \text{frag_type}, e.g., \text{polarLoc} and \text{cartLoc} in Figure 7.2. Separate representation is implemented according to the scheme outlined in Section 4.4.5. We use the “dirty flags” in \text{frag_type} for two different purposes: For lazy consistency maintenance we use them as flags marking the attributes being “dirty”. For eager consistency maintenance we use them as “visited flags” to ensure termination of OCR triggering.

7.2.6 Operation requests and OCR evaluation

We will now describe what happens when operations are requested and evaluated, and how they trigger the OCRs used to maintain consistency. We will explain eager consistency maintenance in the coordinate example (also illustrated in Figure 4.11).
We illustrate an object as a tuple \((\text{Class}, \text{OID}, \text{AttrValues})\), e.g., \((\text{CartLoc}, \text{OID}, x, y)\) and \((\text{PolarLoc}, \text{OID}, \text{rho}, \text{theta})\). An object \(o200\) is consistent in these two classes by the following two “tuples”: \((\text{CartLoc}, o200, 3.0, 4.0)\) and \((\text{PolarLoc}, o200, 5.0, 0.9279)\). When \(o200\) is requested with the mutation operation \(\text{rho}(7.0)\),

\[
[polar\text{Loc}, 200] \Rightarrow [\text{rho}-[7.0]].
\]

the following takes place:

1. \(\text{rho}(+\text{NameRho})\) is queried from the database by traversing the \textit{haso} relation from the type descriptor object of \text{polar\text{Loc}}.

2. This operation is asserted as a clause in the Prolog workspace.

3. It is checked whether there are any \textit{before} OCRs connected to \(\text{rho}(+\text{NameRho})\) by traversing the \textit{triggered\_by} relation. In this case there are none.

4. The operation \(\text{rho}(7.0)\) is executed, resulting in \(\text{rho}\) to be mutated to 7.0.

5. It is checked whether there are any \textit{after} OCRs. It is one OCR, which is queried from the database and asserted in the Prolog workspace.

6. The OCR marks \text{rho} and \text{theta} as visited, and requests \(x(4.2)\) and \(y(5.6)\) for the same object.

7. For each of these operations, a new OCR will be requested, which tries to propagate new values to \text{rho} and \text{theta} again. Since these are marked, the two OCRs terminate without mutating \text{rho} and \text{theta}.

8. The first OCR can finish by unmarking \text{rho} and \text{theta}.

9. The first OCR and \text{rho()} are retracted from the Prolog workspace. Since we have implemented operations by asserting them in the context of the application as normal Prolog predicates, we retract the operations after they have been executed to prevent name crashes with predicates of the application.

### 7.2.7 Evaluation of prototype

The prototype consists of about 600 lines of Prolog code including the schemas of the examples. It was created and tested in about two weeks of time. The prototype showed that it was possible to implement the framework.

The prototype also revealed the consistency management overhead to be substantial. We think the main sources of this overhead lie in the simplicity of the prototype. All information needed to do consistency maintenance is stored in the database server, and is queried from the server every single time it is needed. The eager consistency maintenance example above, where we mutated \text{rho} to a new value for an object of \text{Polar\text{Loc}}, resulted in 68 calls to the database server. The prototype does absolutely no caching itself. The only caching is done in the client side of the EPOS database,
where type descriptors are cached. Thus, every request to an attribute being exposed to consistency maintenance, will cause substantial work for the database.

We could do some improvements to this. Our method for class evolution is additive, i.e. it is only added new classes and EPLs. We believe evolution at the schema level to be rather infrequent compared to operation requests, such that it is reasonable to assume that the information about classes and EPLs could be cached. Therefore, the objects of the schema illustrated in Figure 7.1 would be cached or hard-coded together with the application. For eager consistency maintenance, an additional improvement would be to hold the “visited marks” for objects and the relationships between corresponding objects cached (since they are only used temporarily during OCRs execution). Together these two improvements would in the eager consistency maintenance example above have reduced the number of calls to the server from 68 to 4, i.e. the resulting database calls are: updating polarLoc, querying polarLocr, querying cartLocr, and updating cartLoc. We have not extended the prototype to do this caching, because it would have made the prototype considerably more complex, and would not provide any deeper insight into the issue of consistency maintenance.

### 7.2.8 Alternative consistency maintenance schemes

In Section 4.4.1 we showed two different timing models for consistency maintenance. These timing models offered to maintain the consistency of objects either immediately after every mutator request, or just before every observer request. The experience with the prototype indicates the straightforward trigger implementation of consistency maintenance to be rather expensive. This section shows how more flexible notions of timing can be supported by the transaction model of the EPOS database.

In Section 4.4.2 we saw that ACRs could be used to infer that shared representation of attributes was possible. Thus, we may view the sharing imposed by ACRs equivalently to the sharing of objects between different clients. Therefore, for more flexible notions of consistency we think looking at “design” or “long” transactions (e.g. [BKK85, K+85]) might be feasible. While traditional “short” transactions supporting the ACID properties [HR83] let users believe they are alone when using a shared database, “long” transactions often relax the concurrency control and make the concurrency visible. We propose two extensions to make the consistency maintenance more flexible using the mechanisms of the transactions of the EPOS database [LCM92]:

**Delayed consistency maintenance by merging change sets:** The EPOS database offers a nested transaction model with long, non-serialisable transactions. During their execution subtransactions will work on distinct copies of the overlapping objects (the same or corresponding objects). Objects are copied out from the parent transaction upon updates (“copy-on-write”). The updates will not be made visible to the parent transaction before the subtransaction has committed. The set of updated objects is explicitly recorded in a change set, which is copied back into the parent transaction upon commit of the subtransaction. When all subtransactions have committed and there are conflicting
updates, the change sets must be merged. Related to our framework, this means that the same objects or corresponding objects appearing in different change sets must be merged. This involves merging of structured objects (e.g., see [Wes91]), which is a generalisation of the traditional merging of variants of text files. The gain of this approach is to reduce the overhead of consistency maintenance while the transactions are running, by delaying this until commit time. However, for objects or corresponding objects being observed within the same transaction, the consistency must be maintained according to the original approach. Thus, this approach is best when a client only requests the same or corresponding objects through one class.

Cooperating transactions: The EPOS database provides mechanisms to send asynchronous messages and to propagate objects between sibling transactions. This technique is known from software configuration management, where a developer may “listen” to another developer to be notified about new intermediate versions of objects, e.g., see [LRW91]. In this way, one reconciled version should be committed from all subtransactions. The gain of this approach is to let clients have more control of the consistency maintenance by giving them primitives to exchange information about changes.

7.3 Implementation by object-oriented views

As we have seen in Section 4.6.3 the classes of our framework are very similar to views. We will now show how object-oriented views may implement our framework. To accomplish this we will develop an “objectified” version of a relational query algebra, which will be used in the definitions of the views. There are several rationales for giving this design:

- It shows how our framework may be implemented on top of a database system supporting views, and some of the peculiarities the database must support to do this.
- It explains our framework using existing database concepts. Hopefully, the semantics of the framework becomes clearer to the reader.
- It gives us the possibility to take advantage of techniques known from views and query processing, e.g., materialised views [BLT86] and rewrite rules.

The design presented here is not complete. But still it serves to illustrate the main lines and present some of the peculiarities of the views needed to implement our classes.

We borrow the view definition from [HZ90], and define a view to consist of an intent and a query\(^1\). The intent of a view corresponds to an intent of class, and consists of both attribute and operation definitions being public or private. The query of a view corresponds to the extent of a class, and it is expressed in a query algebra

\(^1\) Type and query in [HZ90].
presented later. The query is issued against a set of other views, and returns a set of objects. There are some special views called base views, which are not represented by queries, but materialised by objects.

The idea is to let classes be implemented as follows. For each class $C$ there are two views: a local view $C^L$, which is a base view used to represent the object created in the class $C$, and a class view $C$ having the same intent as the base view. Let there be $N$ classes $C_i \ (i = 1..N)$ that directly propagate objects to the class $C$, i.e. there are $N$ EPLs $C_i \Rightarrow C$. The class view $C$ will then be defined as a query to the local view and the views $C_i$. To make this implementation reasonable, it must be possible to derive the properties in the intent of $C$ from the properties of the intent of $C_i$. We have two situations that allow for this:

1. For each EPL $C_i \Rightarrow C$, it is an ACG $B_i \oplus A$. When querying the attributes $A$ of the objects in $C_i$, $A$ will be derived from $B_i$.

2. For each class $C_i$, such that $C_i \Rightarrow C$, there are operations in $C_i$ that are “the same” as the operations of $C$. When querying the operations in $C$ for objects of $C_i$, the operations of $C_i$ will be used instead. By “the same as” we mean that they at least have the same signatures.

In many class evolution situations, not all properties of $C$ may be derived from $C_i$. Thus, some of the properties in $C$ must be represented in $C^L$ also for objects created in $C_i$.

### 7.3.1 Query algebra

We will now describe a query algebra used to express the extents in our design of classes by object-oriented views. We will present those operators which we will use in the ensuing sections.

The query algebra is set-oriented, i.e. the input and output of queries are sets of objects. The different operators will be characterised by two aspects. Every query will have an intent, which is the set of public properties of the objects in the result of a query. Given a query $Q$, the intent of $Q$ is denoted by $\text{intent}(Q)$. Unlike intents of classes, an intent of a query has no name. Every query will also have a cardinality, which is the number of objects in the result of the query. The cardinality of the query $Q$ is denoted by $\#Q$.

- **Projection:**

  \[ S_r = \pi_{p_1,p_2,...,p_N}(S_s) \]

  \[ \#S_r = \#S_s \]

  \[ \text{intent}(S_r) \subseteq \text{intent}(S_s) \]

  This operator projects the properties $p_1$ through $p_N$ from the set of objects $S_s$. The result is the same objects as the input, but the intent of the result is a subset of the intent of the input. $p_i \ (i = 1..N)$ may be the name of an attribute
or an operation in the intent of $S_r$. We will allow the properties of $\textit{intent}(S_r)$ to be to be “renamed” from $\textit{intent}(S_r)$ by letting $p_i$ be an ACG.

- **Inner join:**
  \[
  S_r = S_1 \land^i S_2 \\
  0 \leq \#S_r \leq \min(\#S_1, \#S_2) \\
  \textit{intent}(S_r) = \textit{intent}(S_1) \cup \textit{intent}(S_2)
  \]

  The objects in $S_1$ are joined with the objects in $S_2$, where the join criterion is equal OIDs. The result set of objects is the intersection of the sets of objects $S_1$ and $S_2$, and the intent of the result is a superset of both the intent of $S_1$ and $S_2$. If there are two overlapping properties between $\textit{intent}(S_1)$ and $\textit{intent}(S_2)$, it is not specified which of them will be chosen. If one of them is preferred, a projection can be used before the join to remove the undesired property. If an object is only member of one of the input sets, it will not appear in the result.

- **Outer join:**
  \[
  S_r = S_1 \land^o S_2 \\
  \max(\#S_1, \#S_2) \leq \#S_r \leq \#S_1 + \#S_2 \\
  \textit{intent}(S_r) = \textit{intent}(S_1) \cup \textit{intent}(S_2)
  \]

  This is the same as the previous join, but with the following differences: The result set of objects is the union of the sets of objects $S_1$ and $S_2$. If an object is only member of one of the input sets, null values will be inserted for the properties coming from the other intent. This corresponds to an outer join in relational algebra [Dat83].

- **Union:**
  \[
  S_r = S_1 \cup S_2 \\
  \max(\#S_1, \#S_2) \leq \#S_r \leq \#S_1 + \#S_2 \\
  \textit{intent}(S_r) = \textit{intent}(S_1) = \textit{intent}(S_2)
  \]

  The resultant set is the union of the two input sets, where each object is identified by their OIDs. The cardinalities of the sets satisfy the given equation. The intents of the input sets must be equal, and the intent of the result is equal to the intents of the input sets.

- **Difference:**
  \[
  S_r = S_1 - S_2 \\
  (\#S_1 - \#S_2) \leq \#S_r \leq \#S_1 \\
  \leq \#S_r \leq \#S_1 \\
  \textit{intent}(S_r) = \textit{intent}(S_1)
  \]

  The resultant set is the difference between the first and second input set, where objects are identified by OIDs. The cardinalities of the sets satisfy the two given equations. The intent of the result is equal to the intent of the first input set.
If \(\text{intent}(S_1) = \text{intent}(S_2)\) in the two join situations, outer join is the same as union, and inner join the same as intersection. Several other operators are natural parts of this algebra, e.g., selection and intersection, but we have only introduced those operators used later in this design.

In all binary operators until now, objects in the two input sets are identical if they have the same (proxy) OID. To manage object integration we need to extend the query algebra to manage user-defined object correspondences.

- **Entity join:**
  \[ S_r = S_1 \bowtie_3 S_2 \]

  This is the same join as previously, but it is extended with a user-defined join criterion \(\exists\), which corresponds to the automatic registering\(^2\) of object correspondences. \(\exists\) is a condition defined on properties of \(\text{intent}(S_1)\) and \(\text{intent}(S_2)\), and decides which pairs of objects are going to be joined. If the properties used in \(\exists\) are not unique within their class, all pairs of “matching objects” will be included in the result.

  In the entity join it is also decided which OID to be used in the resultant object. Generally, it may be the OID of the object in \(S_1\) or \(S_2\), or a newly generated OID. In the examples presented below we will use the following notation for \(\exists\): \((\text{ResultOID}, \text{Condition})\). ResultOID is one of the following symbols: \(\$1\) (OID from the first set), \(\$2\) (OID from the second set), or \(\text{new}\) (new OID). 

  Condition is the predicate to evaluate which objects are corresponding. Like the normal join, the entity join exists in two versions: *inner* and *outer entity join*.

Similarly to join, the other binary operators, e.g., *union* and *difference*, will have their entity counterparts.

### 7.3.2 View implementation and class evolution

The basic idea in this design is that a class \(C\) should be represented as a view \(C\), which queries a local view \(C^L\) and other views representing classes propagating objects to \(C\). Hence, a concrete class \(C\) having no connected EPLs will be implemented by the following view: \(C = C^L\).

Upon class evolution we must possibly extend the view definitions implementing the classes. The extent evolution part of some of the typical class evolution situations will be implemented as follows:

- **Category generalisation:** \(C_G\) is created as category generalisation of \(C_1\) and \(C_2\).
  
  The views \(C_1\) and \(C_2\) will be retained as they are, while \(C_G\) is defined as follows:
  
  \(C_G = C_G^L \cup C_1 \cup C_2\). If \(C_G\) is abstract, \(C_G^L\) may be superfluous in \(C_G\).

- **Role generalisation:** \(C_R\) is created as a role generalisation of \(C_1\) and \(C_2\). \(C_1\) and \(C_2\) are retained as they are, while \(C_R\) is defined as follows:
  
  \(C_R = C_R^L \cup (C_1 \bowtie^o C_2)\).

\(^2\)Here, we only consider automatic object integration.
Specialisation: $C$ is specialised in the class $C_S$. The view $C$ is extended as follows:

$$C = C^L \cup C_S.$$  

Class versioning: $C_2$ is created as a new version of $C_1$. $C_1$ will be extended to the following definition: $C_1 = C_1^L \cup C_2$ and $C_2$ to: $C_2 = C_2^L \cup C_1$.

The objects in the result of the query of a view must have the same properties as the intent of the view. To respect this requirement we adopt two techniques. The first is to “rename properties” by the use of ACGs in the projection operator, e.g., $C_2 = \pi_p(C_1)$ where $p$ is defined as $B \oplus A$. This essentially means that $A$ in $C_1$ is renamed to $B$ in the query of $C_2$. In these cases $B$ has to have the same (name, domain) pairs as the attributes of $C_2$.

The second technique is to represent parts of objects created in other classes in the local view. E.g., consider the EPL:

$$\begin{array}{c}
C \leftarrow \quad C_1 \\
A \quad \quad \quad \quad A
\end{array}$$

$C$ also has the attribute set $I$ being independent of $C_1$. The class view $C$ may be defined as follows:

$$C = (C^L - C_1) \cup (\pi_I(C^L) \land \pi_A(C_1))$$

The left side the union retrieves all objects created in $C$, and the right side all objects of $C_1$. The inner join ensures that only the objects existing in both $C^L$ and $C_1$ will be returned from the right side.

We have two different aspects of updatability to treat. The first problem is creation of objects. If a class is concrete, it is possible to create objects in it. This is simply accomplished by creating and representing the objects in the local view. However, if an object have to be represented in several local views, the object must be manually propagated to these other local views. E.g., the previous example requires the objects created in $C_1$ to be manually added to $C^L$, and to get values for the attributes $I$.

The second updatability problem is mutation of objects. For objects represented in the local view, this is the same as for normal classes. Objects queried from other views will be mutated in their respective local views. This shows a second purpose of the use of ACGs in projections. E.g., given the projection $C_2 = \pi_p(C_1)$, where $p = B \oplus A$. When $B$ is mutated for objects represented in $C_1$, $A$ will be derived from $B$. The ACG may include a non-derivability mutation symbol. This means that mutation may be denied or that null values will be inserted. We assume the null value approach if no symbol is specified.

We will understand null values as missing information, which may be interpreted as sets of possible values. Given the EPL US $\equiv$ FS with the attached ACG (faculty) $\leftarrow \quad \rightarrow$ (department). Every department has a faculty, and every faculty may have several departments. The correspondence between the elements of the domains of these two attributes is illustrated in Figure 7.3. Let $FS = FS^L \cup \pi_p(US)$, where $p = (department) \leftarrow \quad \rightarrow$ (faculty). Assume faculty to be mutated to ee.cs for an object o100. This means that department will be set to null for o100, which we
Figure 7.3: The correspondence between the faculty and department domains.

will understand as a marked null [Gra91]. This marked null value is a denotation for the set of possible values: cs, cyb, and ee. This information may be exploited in the following way: When querying o100 from US, we will get the set of possible values returned. This result could be presented either as one object with a set of possible values \( \{o100, \{cs, cyb, ee\}\} \), or as three possible objects \( \{o100, cs\} \), \( o100, cyb \)\), and \( \{o100, ee\} \).

Another peculiarity with this design is that we may have recursive queries, e.g., due to the class versioning situation. One way of avoiding this is to let each class be implemented by a view querying from all classes propagating to this class (both directly and transitively). Upon every class evolution, the view implementation for all existing classes must possibly be extended to consider the newly added classes. If we allow for recursive queries, we could take advantage of the domain-specific information in the extent graphs. \( C_1 \Leftrightarrow C_2 \) does not mean that objects should be propagated indefinitely, but that these two classes mutually propagate objects to each other. Thus, we could let the query evaluator use the extent graph, and stop to evaluate a query when a cycle is found. General solutions known from recursive query evaluation could also be adopted, but this could possibly give different semantics than the one we intend with our EPLs and ACRs/OCRs.

As said in Section 7.1.1 this approach assumes lazy propagation of objects in between extents. Actually, objects are retrieved from the other extents every time they are requested. By introducing materialised views [BLT86], the local view may be used as a “cache” for the objects retrieved from the other views (extents). When the objects are mutated in their creation classes, the materialised views must be signaled to be incorrect, and have to be recomputed on the first occasion they are requested.

This design of classes preserves class evolution transparency. For clients it should not be possible to tell whether an object is represented in the local view or it is queried from other views. This is done by retaining all objects upon class evolution, and by modifying the query definitions. As the queries are intensional in nature, this should be less work than propagating all objects manually. The introduction of “renaming” in the projection operator is necessary to support class evolution transparency, because the classes to be integrated may be independently developed, and thus cannot be expected to have “matching” intents.
### 7.3.3 Examples

Consider the two classes given in Figure 4.18 as an example of a class derivation, where *UniversityStudent* is derived from *FacultyStudent*. The view *UniversityStudent* is initially defined by querying the local view, indicated by the name of the class with a $L$ superscript:

$$ \textit{UniversityStudent} = \textit{UniversityStudent}^L $$

A class derivation lets *FacultyStudent* be created as a new “version of” *University-Student*, given by the following EPL and ACG definitions:

<table>
<thead>
<tr>
<th>FacultyStudent</th>
<th>⇔</th>
<th>UniversityStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td>−→</td>
<td>(name)</td>
</tr>
<tr>
<td>(startYear)</td>
<td>−→</td>
<td>(enrolmYear)</td>
</tr>
<tr>
<td>(department)</td>
<td>−→</td>
<td>(faculty)</td>
</tr>
</tbody>
</table>

The *name* attribute is the same in both classes – thus they constitute an ACG. *enrolmYear* is “renamed” to *startYear*, and *department* is introduced in an ACG with *faculty*. We can use the three ACGs to have shared representation of attributes as follows. Objects created in the two classes will be represented fully in their creation classes, and not represented at all in the other class. The views implementing the two classes will be defined as the local view union a projection of the view representing the other class.

The view of *UniversityStudent* is changed to have the following definition:

$$ \textit{UniversityStudent} = \textit{UniversityStudent}^L \cup \pi_{\text{name},p_1,p_2}(\textit{FacultyStudent}) $$

where

$$ p_1 = (\text{enrolmYear}) \leftarrow \rightarrow (\text{startYear}) $$

$$ p_2 = (\text{faculty}) \leftarrow \rightarrow (\text{department}) $$

The view of *FacultyStudent* is defined as follows:

$$ \textit{FacultyStudent} = \textit{FacultyStudent}^L \cup \pi_{\text{name},p_1,p_2}(\textit{UniversityStudent}) $$

where

$$ p_1 = (\text{startYear}) \leftarrow \rightarrow (\text{enrolmYear}) $$

$$ p_2 = (\text{department}) \leftrightarrow \rightarrow (\text{faculty}) $$

The choice to let objects being created in *UniversityStudent* to not be represented in *FacultyStudent*, results in the *department* attribute of those objects to always be *null*. According to Section 4.4.2, it would have been advantageous to represent *faculty* and *department* shared in *department* for all student objects, irrespectively of the creation
class. Consequently, all student objects will have representation in \texttt{FacultyStudent}, and those created in \texttt{UniversityStudent} will be only partly represented in their local views. The new view definition of \texttt{UniversityStudent} is shown below:

\[
\text{UniversityStudent} = \\
(\pi_{\text{name}, \text{enrolmYear}}(\text{UniversityStudent}^f)) \bowtie^p \pi_{p_2}(\text{FacultyStudent}) \\
\cup \\
\pi_{\text{name}, p_1, p_2}(\text{FacultyStudent} - \text{UniversityStudent})
\]

where

\[
p_1 = (\text{enrolmYear}) \leftrightarrow \rightarrow (\text{startYear}) \\
p_2 = (\text{faculty}) \leftrightarrow \rightarrow (\text{department})
\]

The view definition of \texttt{FacultyStudent} must be modified similarly. One important difference from the first solution is that objects created in \texttt{UniversityStudent} must be manually added to \texttt{FacultyStudent} and with a value for the \texttt{department} attribute.

The next example is the role generalisation example of Section 6.2.1, where \texttt{Student} is created as a role generalisation of \texttt{UniversityStudent} and \texttt{FacultyStudent} using the following EPLs and ACGs:

\[
\begin{array}{c|c}
\text{Student} & \Leftrightarrow & \text{FacultyStudent} \\
\hline
\text{(name)} & \leftrightarrow \rightarrow \exists & \text{(name)} \\
\text{(startYear)} & \leftrightarrow \rightarrow & \text{(startYear)} \\
\text{(department)} & \leftrightarrow \rightarrow & \text{(department)} \\
\text{(faculty)} & \leftrightarrow \rightarrow & \text{(department)} \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{Student} & \Leftrightarrow & \text{UniversityStudent} \\
\hline
\text{(name)} & \leftrightarrow \rightarrow \exists & \text{(name)} \\
\text{(startYear)} & \leftrightarrow \rightarrow & \text{(enrolmYear)} \\
\text{(faculty)} & \leftrightarrow \rightarrow & \text{(faculty)} \\
\text{(department)} & \leftrightarrow \rightarrow & \text{(faculty)} \\
\end{array}
\]

In this example we assume the two classes being generalised to be independently developed (and not connected as in the previous example). If we assume \texttt{Student} to be concrete, the view \texttt{Student} will be defined as follows:

\[
\text{Student} = \\
\text{Student}^f \cup \\
\pi_{\text{name}, \text{startYear}, p_1, \text{department}}(\text{FacultyStudent}) \\
\bowtie^p \pi_{\text{name}, p_2, p_3, \text{faculty}}(\text{UniversityStudent})
\]
where

\[
\begin{align*}
  p_1 &= \text{(faculty)} \leftarrow \rightarrow \text{(department)} \\
  p_2 &= \text{(startYear)} \leftarrow \rightarrow \text{(enrolmYear)} \\
  p_3 &= \text{(department)} \leftarrow \rightarrow \text{(faculty)}
\end{align*}
\]

The objects from the two classes being generalised, are outer-joined in \textit{Student} based on \textit{name} equals \textit{name}. The objects in \textit{Student} will acquire new OIDs when being retrieved. The view definitions of the two classes being generalised are not changed.

This example has some delicate mutation problems. The intents of both input sets in the join have all properties of \textit{intent(Student)}. According to the definition of \textit{\&}, it is undefined which of these properties will appear in the result. However, for mutation of these attributes, the most reasonable semantics is to mutate all properties. E.g., if \textit{startYear} of \textit{Student} is mutated for objects created in either of the other classes, both \textit{startYear} in \textit{FacultyStudent} and \textit{enrolmYear} in \textit{UniversityStudent} should be mutated. This problem is a result of \textit{UniversityStudent} and \textit{FacultyStudent} being semantically overlapping classes, and that they are \textit{not} properly connected by EPLs and ACRs/OCRs.

The next example is a category generalisation of the classes \textit{Teacher} and \textit{Student} into the \textit{Person} class. \textit{Teacher} has encapsulated the name and address representation by two observer operations \textit{getName} and \textit{getAddress}, and has no public mutation operations. This is possible by letting the teachers acquire their name and address through the creation operation, and disallow any changes to the attributes later. The private properties of \textit{Teacher} are not shown.

```cpp
class Teacher {
public:
  string getName ()
  string getAddress ()
  set-of <link <Course>> teaches;
};
```

\textit{Person} has three public attributes:

```cpp
class Person {
public:
  string name;
  string address;
  string phone;
};
```

The integration of \textit{Student} and \textit{Person} involves the following EPL with an ACG attached:

\[
\begin{array}{c}
\text{Student} \quad \Rightarrow \quad \text{Person} \\
\text{(name)} \quad \leftarrow \rightarrow \quad \text{(name)}
\end{array}
\]
Person adds address and phone with respect to Student. Thus, these two attributes are independent of Student. These two attributes will not be given for objects created in Student. All objects in Student have to be added to Person manually with values for these two attributes.

The EPL between Teacher and Person will have following “ACGs”:

<table>
<thead>
<tr>
<th>Teacher</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name)</td>
<td>getName()</td>
</tr>
<tr>
<td>(address)</td>
<td>getAddress()</td>
</tr>
</tbody>
</table>

name and address of Person have corresponding information in Teacher available by two public operations. We have “renamed” these operations to the attributes using two “ACGs”. Since these are only observer operations, we have disallowed mutation of name and address for objects created in Teacher. The view Person is defined below:

Person =
\[\pi_{\text{name}}(\text{Student}) \Join^i \pi_{\text{address}, \text{phone}}(\text{Person}_L)\]
\[\cup\]
\[\pi_{p_1, p_2}(\text{Teacher}) \Join^i \pi_{\text{phone}}(\text{Person}_L)\]

where

\[p_1 = (\text{name}) \xrightarrow{\text{MD}} \xleftarrow{} \text{getName}()\]
\[p_2 = (\text{address}) \xrightarrow{\text{MD}} \xleftarrow{} \text{getAddress}()\]

phone of Person is independent of Teacher, which means that objects in Teacher must also be added to Person manually with a value for phone. We assume that there are no entity identical objects in Student and Teacher. Therefore, we use normal union between the two parts of the query.
Chapter 8

Evaluation, Conclusions, and Further Work

8.1 Evaluation

8.1.1 Integrating class evolution techniques

In Chapter 4, 5, and 6 we have seen that the framework can cover for a wide range of class evolution and integration situations. Does this mean that the framework provides remedies for all needs for class evolution?

To this question we would say no. A part of the experience from implementing the framework in a small prototype (Section 7.2), was that the consistency maintenance of the framework was expensive to implement. E.g., eager consistency maintenance requires to calculate new values and mutate the object at once in all classes the object is a member of. A solution could be to work more on optimisation of consistency maintenance, e.g., by exploiting ACRs along the lines of Section 4.4 and 5.6.1. Another solution could be to have more flexible notions of timing for consistency maintenance (see Section 7.2.8).

However, we see transparent class evolution only as a part of an integrated set of class evolution techniques:

Transparent class evolution: This should be provided along the lines of the framework presented in this thesis.

Class modification: Transparent class evolution is always additive – it never modifies the contents of classes, or removes any EPL. The main quality of this is that it retains existing clients. Sometimes it may wiser to “modify” classes. Imprecisely stated, class modification should be used when the cost of modifying clients is less than the cost of maintaining object consistency. Transparent class evolution and class modification may be combined by phasing out classes and EPLs as they become less frequently used. This means that we do not modify classes directly, but remove classes when they are not being used any-
more. It is also possible to phase out new classes which were only used as test versions. The effect of phasing out classes is almost achieved by lazy consistency maintenance. By “almost” we take reservation on the cost of marking objects “dirty”.

**Class hierarchy modification:** This is primarily the capability of removing EPLs from the extent graph. This allows for restructuring extent graphs.

**View integration:** When classes are not populated, integrating different schemas into one may be reasonable. This may include merging and splitting of classes.

**Class reorganisation:** Class reorganisation is different from class hierarchy modification, because it allows for splitting classes and changing properties of objects. We find it very hard to support class reorganisation (in the sense of [Cas91, Opd92]) in object-oriented databases, because they do not consider the extensional dimension of classes. However, these class reorganisations may very well be used during design of schemas.

### 8.1.2 Object model or mechanisms?

Is the framework another object model, or is it a set of mechanisms applicable to existing object models? We think the framework may be used for both purposes.

The framework may be seen as an object model, because it replaces object model constructs like subclassing and generalisation. With this understanding of the framework, the different components of the framework will appear in a schema definition language. E.g., ACRs will appear in a specification language. We have not specified a complete schema definition language for our framework, but the examples presented throughout this thesis indicate a possible language. The framework may also be used as a canonical data model of a federated database system. In this situation, ACRs can be used by a class integration tool to register dependencies between attribute domains for later use (Sections 6.4.3 and 7.1.3).

The framework may also be used by incorporating parts of it into a database system with a traditional object model. E.g., it may be used to extend an object-oriented database system with versioning of classes. In this situation, ACRs can be used internally in the database system and in accompanying tools, e.g., the class editor (Section 5.5).

### 8.2 Conclusions

The main reason for class evolution and integration is that we cannot foresee all needs on the time of design of a schema. Despite the qualities of transparent class evolution, it is not meant as a substitute for good schema design! Putting much effort into the design of a schema will always pay in the long run.

This thesis expresses the view that in the general case it is impossible to avoid
evolution at the schema level. Therefore, this thesis has contributed to the state of the art in database research, by facilitating the following points:

- It examines and unifies several existing class evolution and integration techniques. This unification is motivated from several problems which are similar, but usually are relevant in different contexts. E.g., integration of two classes is very similar to merging two versions of a class in a version graph.

- Specifically, it looks at class integration, class versioning and subclassing as essentially the same problems needing the same mechanisms! This discovery has resulted in transferring good technology between these fields. The ideas for our ACRs stem partly from theory of dependencies in schema integration [LNE89]. These have significant contributions when applied to subclassing and class versioning. The introduction of ACRs into subclassing has resulted in much richer relationships between subclasses and superclasses (see Section 5.7.3). For our work on class integration, class versioning has made us recognise the importance of integration without creating special integration classes, e.g., $C_1$ and $C_2$ may be integrated by $C_1 \Leftrightarrow C_2$, i.e. without creating a generalised class (cf. interdependencies vs. integration [SM92]).

- According to [Sje93], evolution results in more complex consequences on clients than managing the impacts on the extensional data. Our remedy to this problem is to provide class evolution transparency, which is a generalisation of class change transparency to include other kinds of transparent class evolution than class versioning (see Section 4.5.2).

- Unlike existing class evolution approaches, e.g., [KDN90, SN88, Ber91c], which embed the properties related to evolution into special evolution classes, we externalise the evolution information into EPLs and their associated Ξ functions, ACRs, and OCRs. This gives the schema designer more control of what the class should look like, independent on how it appeared: as a result of integration, class versioning etc. Therefore, the class may later be understood and evolved as a normal class. The class itself is not defined as a special class being a “join”, “union” or similar of a set of other classes. But through the external evolution information it becomes an integration class, a new version of a class, a subclass etc. See Section 4.6.4, 6.7.2, and 6.7.3 for more on this.

On a more detailed level, we see the following contributions:

- Extent graphs are introduced as a unification of the relationships between extents for subclassing, class versioning etc. This allows us to grow extent graphs in all directions without the hindrance of the partial order of inheritance hierarchies (see Section 4.5).

- We extend the work of providing class change transparency by introducing specifications of dependencies between attributes (Section 4.3). As seen in Section 4.4.6, these can be used to reason about representation of attributes, to choose a suitable timing model (without hard-coding it), to choose non-derivability mutation policy (also without hard-coding it), and finally to ensure
termination of consistency maintenance for some situations of cyclic dependencies.

- ACGs can be used to interpret marked null values (see Section 7.3.2).

- By the introduction of Ξ functions we unify multiple class membership with integration of entity identical proxy objects (Section 4.2.3 and 6.3). In class derivation Ξ functions exist implicitly by the OIDs resulting from object creation events, while in class integration they are provided to integrate objects being independently created. Ξ functions are used to relate objects for consistency maintenance.

- We introduce a cost model for deciding upon the timing policy for consistency maintenance when deriving classes. This is used in the class editor (see Section 5.3 and 5.5).

- We introduce a class integration methodology, which takes advantage of the framework by separately considering the intensional and extensional dimensions of the classes to be integrated (see Section 6.4).

- We propose ADTs as a semantic enrichment technique for the behavioural part of classes (see Section 6.4.2 and 6.4.3).

- By borrowing techniques from software reuse we suggest two different tools for help in class integration (see Section 6.6).

The main critique we see to our work is that it has not been evaluated using industrial size problems. This means that we really do not know how reasonable this approach is for these problems. The main excuse for not doing such an evaluation is the lack of resources and real cases. As object-oriented databases are yet not in wide-spread use in industry, there clearly is a shortage of real-world evolution cases for these. It is also not clear how such an evaluation should be done, because it represents a substantial work, and we can never know exactly how representative a rather small set of evaluation examples are.

Another critique we see to our work is that we address class evolution, and not schema evolution. By schema evolution we mean where many classes are involved simultaneously. E.g., a set of classes in one schema version corresponds to a set of classes in another schema version. While our sense of evolution is oriented towards situations where one object in one class corresponds to one or N objects in another class, schema evolution may have N objects in one version of the schema corresponding to N objects in another version. This limitation may be seen in the telescope evolution example in Section 5.2.3 and the aggregate integration in Section 6.2.2. It may also be seen by the limitation of the integration methodology to only consider two input classes (Section 6.4).
8.3 Further work

It still remains to evaluate the framework in "industrial size" evolution. To be able to do this, we are dependent on having an implementation of the framework with a reasonable performance.

We would also like to extend the work in this thesis in the following directions:

Framework as canonical data model: In Section 6.7.1 we discussed the suitability of the framework as a canonical data model. We would like to pursue this work by evaluating the framework as a canonical data model for different component data models.

Evaluation of integration methodology: In Section 6.4 we presented an integration methodology specially suited for our framework. We would like to evaluate this, and in particular we are interested in seeing the connection between the intensional and the extensional comparison for different integration situations. We would also like to work further on using ADTs as semantic enrichment of object-oriented classes.

Behavioural dependencies: We have introduced ACRs and OCRs as two mechanisms for describing dependencies between attributes and operations, respectively. The names of these mechanisms should indicate that these are on the same level of abstraction. However, as noted in Section 4.4.6, ACRs are favourable because they are specifications.

In Section 6.7.4 we saw that ACGs are similar to attribute equivalences, which state that attributes are equivalent if they have corresponding and derivable states. We would like to introduce a similar theory for operations, forming operation equivalences. Currently we do not know what such a notion should mean, but we would like to take a look at the notion of behaviour preservation in class reorganisations [Opd92].

Graphical environment: We would like to build a complete graphical development environment for class design, evolution, and integration. This environment should include a class browser, a class editor, an affinity editor, and a graphical interface to the database itself. Another PhD student at the Department of Computer Systems and Telematics, Bjørn Gulla, has created a prototype of a similar environment for the EPOS database. We think such a graphical environment would be a practicable work given Gulla's prototype.

Consistency maintenance and transactions: We would also like implement and evaluate the work presented in Section 7.2.8, where we looked into the application of the transactions of the EPOS database to have a more flexible notion of consistency maintenance.
Appendix A

The Components of the Framework

This appendix contains a summary of the components of the framework presented in Chapter 4. We will explain some of the constructs by referring to the example below, which shows two classes FacultyStudent and UniversityStudent being integrated:

<table>
<thead>
<tr>
<th>FacultyStudent</th>
<th>⇔</th>
<th>UniversityStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name, startYear)</td>
<td>⟷</td>
<td>(name, enrolmYear)</td>
</tr>
<tr>
<td>(department)</td>
<td>⟷</td>
<td>(faculty)</td>
</tr>
<tr>
<td>(credits)</td>
<td>⟷</td>
<td>(level)</td>
</tr>
</tbody>
</table>

This example is also found in Section 6.2.1, except for the bottom ACG, which is taken from Figure 4.9 with credits of domain B and level of domain A. The class evolution framework has the following components:

**Basic object model:** As a basis for the framework lies an object model capable of describing objects having attributes and operations. A class consists of an intent and an extent.

**Intent hierarchy:** This is an inheritance hierarchy between intents of classes. An intent may inherit a set of properties (attributes and operations) from other intents. The intent hierarchy mainly play the role as a reuse mechanism.

**Extent graph:** This is an organisation of extents of classes into graphs, where the vertices are classes and the edges are extent propagation links (EPLs). The graph expresses the set relationship between extents of classes. E.g., in the example above the two classes FacultyStudent and UniversityStudent represent vertices in the extent graph, while the symbol ⇔ represents two directed edges between these two vertices. Thus, the two classes share extents.

**锶 functions:** These are used to describe “corresponding pairs” of objects when classes are integrated. In the example above the first ACG serves as a definition of the 铷 function, i.e. students having equal name and enrollment year will be integrated by the 铷 function.
**Attribute consistency relations (ACRs):** These are used to specify dependencies between attributes of different classes which are connected by EPLs. They come in three categories: totally derivable \((\text{department}) \rightarrow \text{faculty}\), partially derivable \((\text{credits}) \rightarrow \text{level}\), and non-derivable \((\text{department}) \leftrightarrow \text{faculty}\). ACRs appear in pairs, which are named attribute consistency groups (ACGs).

**Operation consistency relations:** These are triggers used for consistency maintenance, either by implementing the specifications of ACRs, or directly between operations of the classes connected by an EPL.
Appendix B

The Interface of the Prototype

This is a detailed presentation of the interface (named OBI) of the prototype documented in Section 7.2. The interface has the following generic operations:

Create object: \texttt{cre}_0 (+\text{Class}, -0, +\text{Attr}\_list)
\texttt{cre}_0 creates an object of \texttt{Class} returning OID 0, and with the initial values of the attributes given in \texttt{Attr}\_list. The attributes not specified in the attribute list will achieve the default values specified in the schema, or ultimately the default values of the corresponding attribute domain (in the EPOS database). \texttt{cre}_0 ensures that 0 is propagated to all classes it should be propagated to (\texttt{EPS(Class)}), or it ensures that its corresponding objects already exist in these classes. \texttt{cre}_0 is non-backtracking.

Delete object: \texttt{del}_0 (+0)
\texttt{del}_0 deletes an object 0 and its corresponding objects entirely, i.e. in every class the object or its corresponding objects are members. The predicate is non-backtracking.

Add object to a class: \texttt{add}_0 (+\text{Class}, +0, +\text{Attr}\_list)
\texttt{add}_0 adds the object 0 to the class \texttt{Class} with the initial values for attributes specified in \texttt{Attr}\_list. This predicate ensures that the object exists in or has corresponding objects in all classes in \texttt{EPS(Class)}. The predicate is non-backtracking.

Remove object from a class: \texttt{rmv}_0 (+\text{Class}, +0)
\texttt{rmv}_0 removes 0 from \texttt{Class}. The predicate ensures that the object or its corresponding objects will be removed from all classes \texttt{C} such that \texttt{Class} \in \texttt{EPS(C)}.

Operation request: [+\text{Class}, ?0] => [+\text{OP1}-[?\text{PL1}],..., +\text{OPN}-[?\text{PLN}]]
The operation request operator evaluates a list of operation requests either for a given object 0 or backtracking for all objects in the given \texttt{Class}. \texttt{OP1} through \texttt{OPN} are the names of the requested operations, and \texttt{PL1} through \texttt{PLN} are the lists of parameters to these operations. The operations will be evaluated left to right. Note that all attributes are requested through attribute request operations, which implementations are using the EPOS database predicates (\texttt{qry}_E
and `upd.E`) directly. The operation request operator will for each operation in the list evaluate all associated before OCRs before actually evaluating the operation, and similarly for after OCRs after the evaluation of each operation.

There are several points where the design of the interface could be questioned. Attributes are given initial values through the `cre.O` and `add.O` operations. This means that all attributes are visible through these two operations, which may be seen as breaking encapsulation by letting all clients creating or adding objects use the attributes in their code. This means that the code of those clients are directly dependent on these attributes. Note that most initialisation constructs of object-oriented languages in some way do the same, e.g., the parameters of `new` in Smalltalk [GR83] and `constructors` in C++ [Str86].

Another lack of object-orientation in the model is that classes do not have operations themselves: The create, add and delete operations are not invoked as operations of specific classes, but are generic operations in the interface. We think this is mostly a question of the syntax of the operations. Secondary, it may also be a question of having a “pure” object-oriented system, where the system itself is implemented with the same constructs as the user-defined classes. Due to the implementation context (the EPOS database), such a uniform architecture would have been rather complex.

Since the applications themselves are written in Prolog, we decided to let the interface be suited for Prolog’s constructs. One consequence of this is that we provide an iterative operation request operator `⇒`, which is cleanly incorporated into Prolog’s backtracking. However, the interface is not designed to nicely navigate through object structures by following link attributes. Link attributes must be observed into Prolog variables, before they can be used to further request operations.
Bibliography


Index

abstract class, 102
abstract data type, 147, 153
abstraction,
shared, 147
ACG, 74
ACR, 70
active database, 85
add object operation, 65
aggregate integration, 138
aggregation, 32
assertion,
in schema integration, 155
attribute, 57
comparison of, 150
attribute consistency group, 74
definition of, 74
mathematical interpretation of,
75
satisfaction of, 77
attribute consistency relation, 70
global catalogue of, 122, 151
in integration, 156
interpretation of, 71
management of, 173
mathematical interpretation of,
73
non-derivable, 71
partially derivable, 71
satisfaction of, 77
totally derivable, 70
transitive, 78
vs operation consistency relation, 92
attribute,
domain, 57
domain constructor, 57
attribute equivalence, 167
attribute grammar, 124
attribute,
request operation, 58
attribute set, 74
attribute,
unit of mutation, 75, 84
autonomy, 144
basic object model, 55
behaviour preserving, 32
behavioural consistency, 29
canonical data model, 18, 165
suitability of, 165
category generalisation, 42, 100, 101,
135, 184, 189
CEFP, 174
change management, 4
change-oriented versioning, 126
class, 10, 56
class change transparency, 35, 98
class connection, 101
class derivation, 100, 109
class editor, 121
class evolution,
definition of, 96
class evolution framework, 95, 197
class evolution transparency, 98, 103,
191
class hierarchy, 10, 63
class integration, 39, 100, 133
domain-specific, 165
class,
learning from example, 51
class modification, 4, 27, 53, 191
class,
populated, 56
class refactoring, 31
class reorganisation, 5, 53, 192
incremental, 30
class transformation,
object-preserving, 32
class versioning, 4, 35, 53, 101, 110,
128, 185

221
client, 10, 56
coarse granularity, 12
complex data, 16
complex object, 10, 57
component database, 17, 104, 144
composite object, 57
computational completeness, 11
concrete class, 102
currency control, 10, 14, 180
configuration management, 14
conformance rules, 24
connecting
specialisation/generalisation, 138
connecting variants, 137
consistency maintenance, 79
alternative scheme, 180
execute, 86, 89
in CEPF, 178
lazy, 86, 89
optimisation of, 123
switching off, 123
termination of, 93
timing models, 86
constraint, 153
contravariance, 24
cost analysis, 114, 120
cost model, 114
create object operation, 64
database integration, 3
background for, 17
delete object operation, 65
dependency,
in the relational model, 129
derivation, 109
design environment,
database requirements of, 11
dirty flag, 89, 94
dynamic binding, 4, 11, 62, 67, 125, 130
example of, 23
explicit qualification, 67, 145
multiple operation request, 69
tagged link value, 69, 145
usage context, 68, 126
encapsulation, 10, 60, 131, 200
entity identical, 141
entity object, 44, 141
EPL, 63
EPOS, 5, 174, 180
EPS, 64
evolution, 17
additive, 96
anticipated, 17, 26, 52
background for, 3
telescope, 114
unanticipated, 17, 52
excuses, 26
existential operation, 64, 169
explicit relation, 106
extensibility, 4, 11
extensional comparison, 154
extent, 56
evolution of, 96
extent graph, 63, 96
extent inclusion, 23
extent propagation, 80
eager, 80
implementation of, 169
lazy, 80
extent propagation link, 63
interference of, 159
usage of, 63
extent propagation set, 64
faceted classification, 164
federated database system, 17, 104, 134
fine granularity, 12
FOOD, 5
framework, 95, 197
implementation by views, 181
generalisation, 5, 31, 53, 100, 110
by upward inheritance, 45, 166
geographical information systems,
database requirements of, 16
group integration, 140
heterogeneity, 17, 134
horizontal splitting, 119
IFO semantic model, 23
im function, 66, 141, 155
automatic, 142
im function (continued):
  combined, 143
  manual, 143
inheritance link, 61
inheritance,
  rationale of, 62
integration, 3, 19
  confirmation of, 155
  conflicts of, 20
  customised guidance, 164
  domain-specific, 165
  of classes, 39, 100, 133
  of objects, 141
intensional comparison, 148
  affinity browser, 163
  term-space tool, 161
intent, 56
  evolution of, 96
intent hierarchy, 61, 96
interdependency, 19
invariant, 132
IS-A relationship, 63
join,
  entity, 184
  inner, 183
  outer, 183
knowledge-based systems,
  database requirements of, 16
link, 57
  containment, 57
  inverse constraint, 57
link value,
  tagged, 125
longfield, 13
member, 56
meta-object/normal object
  integration, 140
methodology, 19, 145
multi-database system, 17, 134
multiple views, 36
mutation denial, 72
mutator, 58

name difference, 40, 147, 150
naming conventions, 150

nested relational database, 2
non-derivability mutation policy, 72,
  74, 76, 83, 93
non-derivability mutation symbol, 72
non-trivial cycle, 79, 91
null value, 58, 72, 76, 185
  marked, 76, 185
OBI, 175, 199
object, 56
  adding to a class, 56
  behaviour, 58
object consistency, 69
  definition of, 78
  interference of, 160
object,
  constrained conversion of, 49
  conversion of, 48, 125
object correspondence, 66, 141
object,
  entity, 141
    global, 142
object identifier, 56
object identity, 10, 56, 144
    global, 142
object integration, 19, 66, 141
    in CEFP, 177
object,
    long-lived, 14
object model, 9, 55
    rationale of, 59
object,
    multiply classified, 50
    proxy, 141
object table, 48
object,
    versioning of, 48
object-oriented,
    behaviourally, 1
object-oriented data model, 9
object-oriented database, 1, 9
object-oriented database system
    manifesto, 9
object-oriented design, 97, 147
object-oriented,
    fully, 1
    structurally, 1
observer, 58
OCR, 85
office information systems,
  database requirements of, 16
OID, 56, 141
operation, 58
  binding of, 67
  comparison of, 151
operation consistency relation, 85
  after, 86
  before, 87
  implemented by trigger, 172
  in integration, 156
  timing mode, 86
operation,
  implementation, 58
  request, 58
  string comparison of, 153
  tree comparison of, 153
parameterised class, 25, 53
persistence, 9
persistent programming language, 2
polymorphism, 11, 126
power plant example, 40, 42, 45, 135
private, 59, 131
product development database, 15
property, 57
  request, 58
prototype,
  evaluation of, 179
  implementation of, 174
  interface to, 199
proxy identical, 141
proxy object, 44, 141
public, 59
query algebra, 182
recovery, 9
remove object operation, 65
request,
  mutator, 58
  observer, 58
revision, 14
role generalisation, 42, 100, 101, 136,
  184, 188
scale difference, 40, 147, 162
schema, 98
schema evolution, 194
schema integration, 19, 38, 134
schema integration methodology, 19,
  145
schema versioning, 33
semantic data model, 1, 23
semantic enrichment, 153
semantic overlap, 19, 63, 66, 141
semantic set relationship, 145, 154
separate representation, 88
  cost of, 116
separated hierarchies, 6, 10, 104
shared representation, 80, 118, 123
  cost of, 115
  implementation of, 84
signature, 58, 152
Smalltalk, 62
software engineering database, 2, 12
specialisation, 32, 100, 110, 137, 185
static binding, 58, 145
static typing, 126
structural consistency, 29
subclassing, 4, 21, 52, 110, 130
  representation of, 118
  usage of, 131
subtyping, 24
timing model, 93
transaction,
  cooperating, 181
  design, 180
  long, 14, 180
trigger, 85
  ECA model, 171
trivial cycle, 76, 79
type, 10
  actual, 126
  required, 126
value, 57
variant, 14
versioning, 14, 48, 126
  purposes of, 33
vertical splitting, 119
view, 170
  base, 182
  class, 182
  definition of, 181

224
view integration, 3, 19, 192
view,
    integration by, 41, 43
    local, 182
    materialised, 186
    updatable, 37, 44, 105, 185
ViewSystem, 42, 167
visibility mode, 59
visited flag, 89, 94