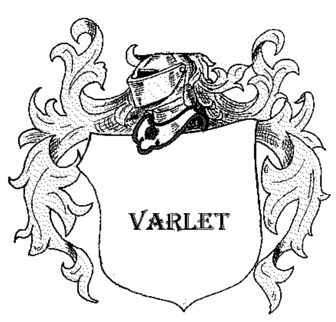
MANAGEMENT OF UNCERTAINTY AND INCONSISTENCY IN DATABASE REENGINEERING PROCESSES



Dissertation submitted in partial fulfillment
of the requirements for the degree of
"Doctor of Science"
in the Department of Mathematics and Computer Science
at the University of Paderborn

Schriftliche Arbeit
zur Erlangung des Grades
"Doktor der Naturwissenschaften"
im Fachbereich Mathematik-Informatik
der Universität Paderborn

Dipl. Inform. Jens H. Jahnke
Universität Paderborn
Fachbereich Mathematik-Informatik
D-33095 Paderborn
Germany

August 1999

© Jens H. Jahnke, University of Paderborn, Germany. All rights reserved.

ABSTRACT

This dissertation tackles one of the most urgent problems in today's information technology, namely the renovation and migration of legacy information systems to modern platforms and net-centric architectures. In this context, several methods, tools, and processes have been proposed to support reengineering and modernizations of legacy database applications. This can be a complex task because many legacy databases have grown over several generations of programmers and lack a sufficient documentation. Computer-aided reengineering methods and processes have a great potential to reduce the complexity and risks involved in database design recovery and migration projects. Still, current reengineering tools are hardly adopted for practical problems in industry because they often make idealistic assumptions about the structure of legacy systems and the characteristics of reengineering processes. The goal of this thesis is to provide concepts and techniques to overcome these severe limitations. In particular, our focus is on developing mechanisms to manage uncertainty and inconsistency in computer-aided databases reengineering processes. In practice, uncertain knowledge plays an important role in activities aiming to recover conceptual design documents for large idiosyncratic implementation structures. This fact is neglected in current database reengineering methods and tools.

In this dissertation, we identify and extend a theory that provides a suitable basis to deal with uncertain reengineering knowledge and allows to implement practical tools and environments to support reengineering processes. The requirement for consistency management considers the fact that it is unrealistic to presume that database reengineering processes can be executed in a number of sequential phases or steps without iterations. In practice, larger reengineering projects comprise many process iterations due to various reasons like incomplete knowledge about legacy implementation structures or necessary "on-the-fly" modifications of the legacy system. Detecting and removing inconsistencies caused by such iterations significantly increase costs and durations of current reengineering projects. In this thesis, we employ graph transformation theory to develop mechanisms which allow to detect and eliminate inconsistencies between legacy schema implementations and their abstract representation, automatically. Our results have been implemented in the database reengineering environment *Varlet* and evaluated with an industrial project. They are suitable to complement many existing approaches in the domain of information system reengineering and migration. As an example, we describe the integration of *Varlet* with an existing middleware product for data integration.

ACKNOWLEDGMENTS

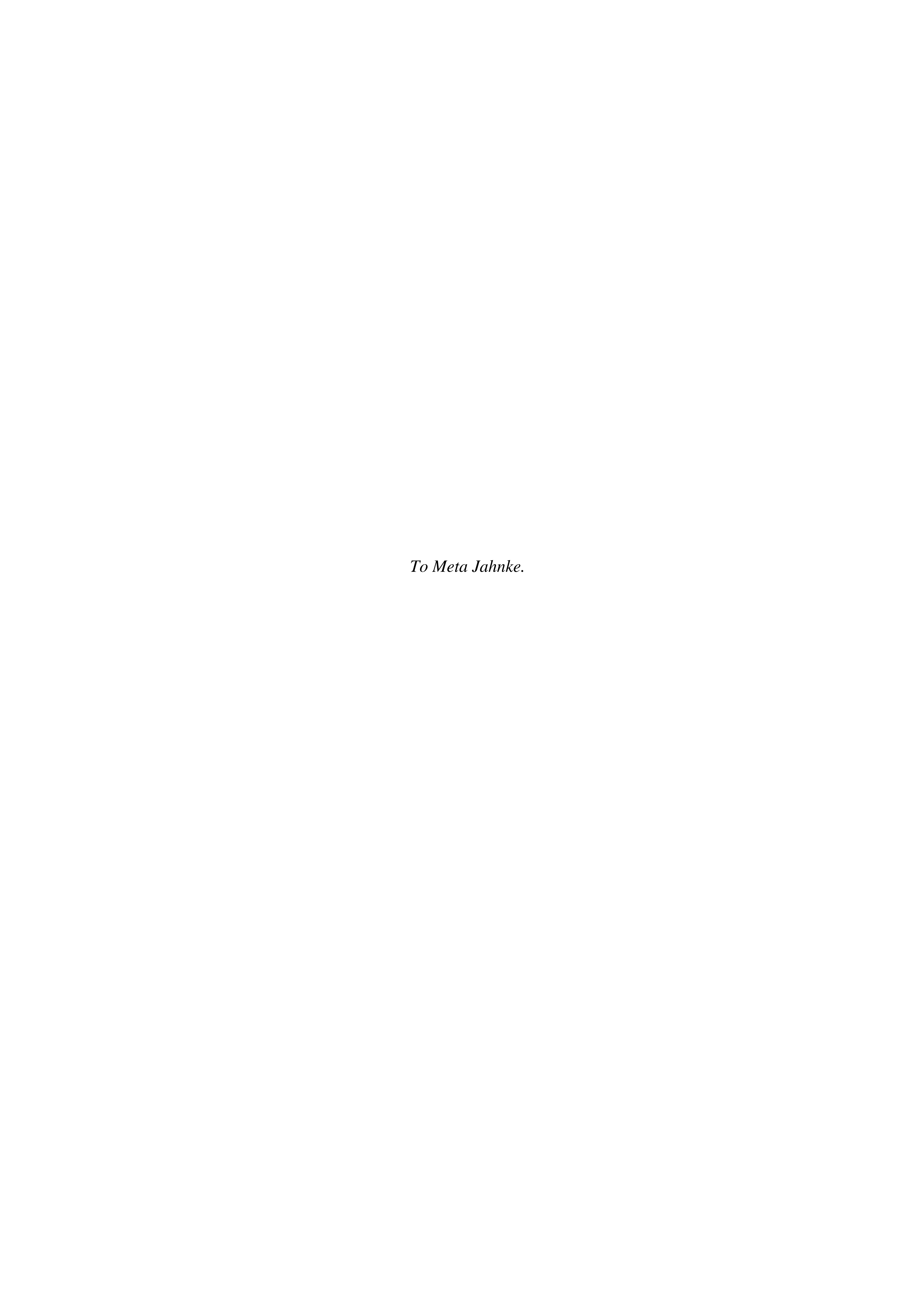
During the last five years in Paderborn, many people have influenced my research and significantly contributed to the results presented in this dissertation. I am especially obliged to Wilhelm Schäfer who supported me throughout this period and provided an interesting and constructive working environment. I learned a lot from our fruitful discussions. Many thanks to Hausi Müller who welcomed me as an external member of his research group at the University of Victoria, B.C., for nearly four months. His advice in the early phase of structuring my written thesis as well as his detailed comments on a preliminary version were very valuable and had a significant impact on the final result. I am grateful to Gregor Engels for many interesting comments and discussions in our weekly graduate seminars. My special thanks go to Albert Zündorf with whom I shared one office, several thoughts, and many beers. Over the years, our collaboration has been productive as well as pleasant. Albert did a great job in proofreading this dissertation.

The achievements made in this dissertation would not have been possible without the practical and theoretical contributions of a number of graduate students at the University of Paderborn. I was particularly inspired by supervising several master theses in the context of this project. In his thesis, Jens Holle implemented the very first prototype of the *Varlet* schema migration environment. His work gave us important insight into the benefits and limitations of using triple graph grammars for conceptual abstraction. Christian Rummel formalized a conceptual data model and elaborated a catalog of schema redesign transformations. Jörg Wadsack realized the mechanism for incremental change propagation presented in this dissertation. Heike Schalldach developed a generator for object-relational middleware components based on graph-oriented schema dependencies. In collaboration with another local research institute (C-Lab), Ulrich Nickel applied our techniques to a practical case study in the domain of engineering information systems. Markus Westerfeld extended *Varlet* by generic mechanisms (e.g., XML views) to integrate commercial middleware components like *ObjectDRIVER*. Melanie Heitbreder implemented the core of the *Varlet* schema analysis tool, namely the possibilistic inference engine. Christoph Strebin extended this inference engine by concepts and techniques that enable self-adaptation of reverse engineering heuristics. Barbara Bewermeyer developed a flexible detection mechanism for stereotypical source code patterns. Additionally, I would like to thank all students who implemented the current *Varlet* user interface, namely Martin Bierschenk, Frank Eckhardt, Sven Meyer zu Eißen, Hajo Köhler, Ralf Langer, Carsten Matysczok, Jens Rehpöhler (who also designed the *Varlet* Web page at *www.upb.de/varlet*), and Swen Thümmler. Special thanks to Michael Kisker and Felix Wolf who worked on this project as research assistants.

I would like to thank *eps Bertelsmann*, Gütersloh, Germany, for providing us with an industrial-strength case study for our research prototype. Thanks to the Database Team at *CERMICS*, Sophia Antipolis Cedex, France, and the *Progres* Team at RWTH Aachen, Germany, for their technical support.

Thanks to Reiko Heckel, Jörg Niere, Heike Schalldach, Jörg Wadsack, and Anke Weber for proofreading parts of this dissertation. Thanks also to Olaf Neumann and Sabine Sachweh who have always been eager to discuss new ideas. Jutta Haupt has been a great help in navigating through the "jungle of paper works" at the University of Paderborn. Many thanks to Jürgen Maniera who did a great job in keeping the machines running and spreading good mood. I will definitely miss this team.

Still, most important during my studies in Dortmund and Paderborn has been the support and love of my family and my wife Anke Weber.



To Meta Jahnke.

CONTENTS

List of Figures xiii

List of Definitions xvii

1 Introduction 1
- 1.1 Background: the dilemma of software legacies . . . 1
- 1.2 Database reengineering . . . 3
- 1.3 Problem definition . . . 4
- 1.4 The approach . . . 7
- 1.5 Dissertation outline . . . 10

2 Database Reengineering - A Case Study 11
- 2.1 A legacy product and document information system . . . 11
- 2.2 Migration target: a distributed marketing information system . . . 11
  - 2.2.1 Functional requirements . . . 12
  - 2.2.2 Technical requirements . . . 13
- 2.3 Migration strategy . . . 13
- 2.4 The reengineering process . . . 14
  - 2.4.1 Legacy schema analysis . . . 15
  - 2.4.2 Conceptual schema migration and redesign . . . 24
  - 2.4.3 Implementation of changes and a middleware for data integration . . . 26
- 2.5 Summary and concluding remarks . . . 30

3 A Theory to Manage Imperfect Knowledge 33
- 3.1 Requirements on formalisms to manage DBRE knowledge . . . 33
  - 3.1.1 Quantitative representation of uncertainty . . . 35
  - 3.1.2 Representation and indication of contradicting knowledge . . . 36
  - 3.1.3 Reasoning about incomplete knowledge . . . 36
  - 3.1.4 Representation of ignorance . . . 36
  - 3.1.5 Computational tractability . . . 37
- 3.2 Evaluation of theories . . . 37
  - 3.2.1 Production systems with confidence factors . . . 38
  - 3.2.2 Probabilistic reasoning . . . 40
  - 3.2.3 Credibilistic reasoning . . . 42
  - 3.2.4 Fuzzy reasoning . . . 44
  - 3.2.5 Possibilistic reasoning . . . 49
- 3.3 Summary and conclusion . . . 52

4 GFRN as a Basis for Legacy Schema Analysis 55
- 4.1 Supporting human-centered schema analysis processes . . . 55
- 4.2 Specification of database reengineering knowledge . . . 57

4.2.1 Informal introduction to GFRNs 58
4.2.2 Integration of automatic analysis operations 63
4.2.3 Formal definition 66
4.3 Knowledge inference with GFRN specifications 76
4.3.1 A fuzzy Petri net model for non-monotonic reasoning 77
4.3.2 The inference process 81
4.4 Implementing the Varlet Analyst 99
4.4.1 Architecture 99
4.4.2 User interface 101
4.5 Evaluation 107
4.6 Related work 109
4.7 Summary 111

5 Conceptual Schema Migration and Data Integration 113
5.1 The migration graph model 115
5.1.1 Graph-based representation of logical and conceptual schema 116
5.1.2 The schema mapping graph model 119
5.2 A graphical formalism to implement schema translators 121
5.2.1 Triple graph grammars 122
5.2.2 Mapping variants to class hierarchies 127
5.2.3 Mapping columns to class attributes 132
5.2.4 Mapping inclusion dependencies to relationships 133
5.2.5 Discussion 136
5.3 Conceptual schema redesign 137
5.3.1 Schema redesign transformations 137
5.3.2 An extensible catalog of schema redesign transformations 138
5.3.3 Complex schema redesign transformations 144
5.4 Incremental change propagation 145
5.4.1 The history graph 146
5.4.2 The propagation mechanism 150
5.5 Implementing the Varlet Migrator 157
5.5.1 Architecture 157
5.5.2 User interface 159
5.6 Data integration 164
5.6.1 Generating descriptions for relational and object-oriented schemas . 166
5.6.2 Generating object-relational mapping descriptions 167
5.7 Evaluation 176
5.8 Related work 178
5.8.1 Conceptual schema migration and consistency management 178
5.8.2 Data integration 179
5.9 Summary 180

6 Conclusions and Future Perspectives 181
6.1 Major contributions 181
6.2 Transferability of results 182
6.3 Open problems 183
6.4 Future perspectives 184

A Additional Definitions and Specifications **187**
A.1 Interpretation of a logical schema 187
A.2 Specification of the migration graph model 188

B A Catalog of Redesign Transformations **195**

References **217**

Index **233**

Abbreveations **237**

LIST OF FIGURES

Figure 1.1. Reengineering process . . . 3
Figure 1.2. Conceptual schema as a starting point for subsequent DBRE activities . . . 4
Figure 1.3. CARE tool classification according to the role of human knowledge . . . 6
Figure 1.4. Proposed DBRE approach . . . 8
Figure 2.1. Existing Product and Document Information System (PDIS) . . . 12
Figure 2.2. Planned Marketing Information System (MIS) . . . 13
Figure 2.3. Gradual migration strategy . . . 14
Figure 2.4. The planned reengineering process . . . 15
Figure 2.5. Constraints resulting from the schema catalog . . . 16
Figure 2.6. Potential constraints indicated by naming heuristics . . . 16
Figure 2.7. Detail of PDIS . . . 17
Figure 2.8. Contradicting indicators for key constraint . . . 18
Figure 2.9. Potential foreign keys indicated by *join* patterns . . . 18
Figure 2.10. Result of the structural completion . . . 19
Figure 2.11. Assumed hidden common domain relation . . . 20
Figure 2.12. Labeled variants and additional foreign keys of table *PRODREF* . . . 20
Figure 2.13. Variants of table *PRODREF* . . . 20
Figure 2.14. Detected optimization and aggregation structures . . . 21
Figure 2.15. Implication of relational constraints on the cardinality of relationships . . . 22
Figure 2.16. Summary of analysis results . . . 23
Figure 2.17. Conceptual schema for PDIS (detail) . . . 25
Figure 2.18. Extended conceptual schema for MIS (detail) . . . 26
Figure 2.19. Extended conceptual schema for MIS after iteration (detail) . . . 27
Figure 2.20. Implemented extensions of the logical schema . . . 28
Figure 2.21. MIS architecture . . . 28
Figure 2.22. Design of the middleware layer (detail) . . . 29
Figure 3.1. Reference architecture of KBS . . . 34
Figure 3.2. Sample fuzzy sets with continous and discrete membership functions . . . 45
Figure 3.3. Sample fuzzy sets for fuzzy predicates *AName*, *LargeExt*, and MediumExt . . . 47
Figure 3.4. Evaluation summary . . . 53
Figure 4.1. The proposed schema analysis process . . . 56
Figure 4.2. Simple GFRN . . . 59
Figure 4.3. Implication with constraint and negation . . . 59
Figure 4.4. Implication with conjunction . . . 59
Figure 4.5. Similarity measures for the seven sample attribute names with the string *userid*. . . 60
Figure 4.6. Implication with threshold . . . 60
Figure 4.7. Premise with universal quantifier . . . 61
Figure 4.8. Variable aggregation and composition . . . 62
Figure 4.9. Combination of heuristics in a single GFRN . . . 62
Figure 4.10. Characteristics for classifying automatic analysis operations . . . 64
Figure 4.11. GFRN with data- and goal-driven predicates . . . 64
Figure 4.12. Goal-driven analysis operation *validate_IND* . . . 65

Figure 4.13. *N(validIND^2(i,v))* for the case of no counterexamples 65
Figure 4.14. GFRN to illustrate the formalization 68
Figure 4.15. Translation algorithm *GFRN2NPL^1* 69
Figure 4.16. Translation algorithm *Impl2NPL^1* 70
Figure 4.17. Algorithm *OperateGFRN* 73
Figure 4.18. Excerpt of case study 75
Figure 4.19. Necessity degrees for the facts produced by Ω*(ANameIsRSName+ID^1)(B)*. 75
Figure 4.20. Necessity degrees for the facts produced by ω*(validKey^1)(B,validKey^1(x))*. 76
Figure 4.21. Belief revision phase 1: computation of fuzzy truth tokens 79
Figure 4.22. Belief revision phase 2: Computation of FBMs 79
Figure 4.23. The proposed iterative and interactive inference process 82
Figure 4.24. Representation of an expanded GFRN implication (sample) 83
Figure 4.25. Forward and backward expansion (sample) 84
Figure 4.26. Information sources for inference example 85
Figure 4.27. GFRN to exemplify the inference process 86
Figure 4.28. FPN after the first expansion/evaluation cycle 86
Figure 4.29. FPN after second expansion/evaluation cycle 87
Figure 4.30. FPN after third expansion/evaluation cycle 88
Figure 4.31. Additional implications to specify necessary conditions for R-INDs 88
Figure 4.32. FPN after considering human input 89
Figure 4.33. Final analsysis result 89
Figure 4.34. Representation of human assumptions 90
Figure 4.35. Algorithm *GFRNInference* 91
Figure 4.36. Algorithm *CreatePlace* 93
Figure 4.37. Algorithm *ExpandFPN* 94
Figure 4.38. Algorithm *ComputeBindingsForImpl* 96
Figure 4.39. Algorithm *ComplementBindings* 97
Figure 4.40. Example GFRN for termination problem 98
Figure 4.41. Architecture of the *Varlet Analyst* 100
Figure 4.42. *Customization Front-End* 102
Figure 4.43. *Customization Front-End* (2) 103
Figure 4.44. *Analysis Front-End* (overview) 104
Figure 4.45. *Analysis Front-End* (detail view) 105
Figure 4.46. Graphical and textual documentation of an analyzed logical schema 107
Figure 5.1. Incremental schema migration and generative data integration 115
Figure 5.2 Migration graph model 117
Figure 5.3. Graph test *DuplicateClassName* 119
Figure 5.4. Sample situation: correspondence among variant and inheritance structures 120
Figure 5.5. Graph production *AddRSToLSchema* 122
Figure 5.6. Mapping rule *MapRSToClass* 123
Figure 5.7. Reverse production $MapRSToClass_{rv}$ 124
Figure 5.8. Forward production $MapRSToClass_{fw}$ 125
Figure 5.9. Startgraph for schema migration 126
Figure 5.10. Algorithm *MapSchema* 126
Figure 5.11. Mapping rule *MapVariantToConcreteClass* 127
Figure 5.12. Example RS *Tenant* with two variants an their conceptual representation 128
Figure 5.13. Example application of rules *MapRSToClass* and *MapVariantToConcreteClass* . . 128
Figure 5.14. Production $MapVariantsToAbstractClass_{rv}$ 129
Figure 5.15. Example application of production $MapVariantsToAbstractClass_{rv}$ 130
Figure 5.16. Production $MapVariantsToInheritance_{rv}$ 131

Figure 5.17. Example application of production *MapVariantsToInheritance*$_{rv}$ 132
Figure 5.18. Mapping rule *MapColToAttr* 133
Figure 5.19. Mapping rule *MapRINDToAssoc[1:1]* 134
Figure 5.20. Mapping rule *MapRINDToAssoc[N:0,1]* 135
Figure 5.21. Mapping rule *MapRINDToAssoc[0,N:0,1]* 135
Figure 5.22. Mapping rule *MapIINDToInheritance* 136
Figure 5.23. Catalog of conceptual redesign transformations 139
Figure 5.24. Schema transformation *SplitClass* 140
Figure 5.25. Schema transformation *MoveAttribute* 141
Figure 5.26. Schema transformation *Generalize* 142
Figure 5.27. Schema transformation *PushUpAttribute* 143
Figure 5.28. Complex transformation *MoveOverAggregation* 145
Figure 5.29. Basic structure of a history graph 146
Figure 5.30. Template of transformation *Generalize* 147
Figure 5.31. History graph model 149
Figure 5.32. Phase I: forward propagation 151
Figure 5.33. Phase II: backward propagation 152
Figure 5.34. Phase III: reevaluation 152
Figure 5.35. Phase IV: translation 153
Figure 5.36. Transaction *PropagateChange* 154
Figure 5.37. Graph test *Generalize_getParams* 155
Figure 5.38. Production *Generalize_withParams* 156
Figure 5.39. Architecture of the *Varlet Migrator* 157
Figure 5.40. Using the Progres environment to extend module *Redesign Transformation* 158
Figure 5.41. Logical schema after first analysis step and its initial conceptual translation 160
Figure 5.42. Redesigned conceptual schema (*Migration Front-End*) 161
Figure 5.43. Completed logical schema (top) and updated logical schema (bottom) 162
Figure 5.44. Implementation of conceptual extensions (*Analysis Front-End*) 164
Figure 5.45. *ObjectDRIVER* overview 165
Figure 5.46. Integration of the *ObjectDRIVER* middleware generator as a back-end for *Varlet* . 166
Figure 5.47. Relational schema description for *ObjectDRIVER* 167
Figure 5.48. Object schema description for *ObjectDRIVER* 168
Figure 5.49. Mapping description for classes and subclasses 169
Figure 5.50. Test *getClassInstantiationConstraint* 169
Figure 5.51. Mapping description for base table attributes 170
Figure 5.52. Test *getAttrMappedToColInBaseTable* 170
Figure 5.53. Mapping description for remote attributes 171
Figure 5.54. Test *getAttrMappedToColInRemoteTable* 171
Figure 5.55. Mapping description for base table relationships 172
Figure 5.56. Test *getRelMappedToBaseTable* 172
Figure 5.57. Mapping description for remote relationships 173
Figure 5.58. Test *getRelMappedToRemoteTable* 173
Figure 5.59. Mapping description for IND-based inheritance relationships 174
Figure 5.60. Test *getInheritMappedToI_IND* 174
Figure 5.61. Mapping Description for *ObjectDRIVER* 175
Figure 5.62. MIS application code (example) 176
Figure 6.1. Self-adapting analysis process 185
Figure B.1. Transformation *Aggregate* 196
Figure B.2. Transformation *AssociationToClass* 197
Figure B.3. Transformation *ChangeAssocCardinality* 198

Figure B.4. Transformation *ChangeAttributeType* 198
Figure B.5. Transformation *ClassToAssociation* 199
Figure B.6. Transformation *CreateAssociation* 200
Figure B.7. Transformation *CreateAttribute* 200
Figure B.8. Transformation *CreateClass* 201
Figure B.9. Transformation *CreateInheritance* 201
Figure B.10. Transformation *CreateKey* 202
Figure B.11. Transformation *ConvertAbstract* 202
Figure B.12. Transformation *ConvertConcrete* 203
Figure B.13. Transformation *DisAggregate* 204
Figure B.14. Transformation *Generalize* 205
Figure B.15. Transformation *MergeClass* 206
Figure B.16. Transformation *MoveAttribute* 207
Figure B.17. Transformation *PushDownAttribute* 208
Figure B.18. Transformation *PushDownAssociation* 209
Figure B.19. Transformation *PushUpAttribute* 210
Figure B.20. Transformation *PushUpAssociation* 211
Figure B.21. Transformation *Remove* 212
Figure B.22. Transformation *RenameClass* 212
Figure B.23. Transformation *RenameAttribute* 212
Figure B.24. Transformation *RenameRelationship* 213
Figure B.25. Transformation *SplitClass* 213
Figure B.26. Transformation *Specialize* 214
Figure B.27. Transformation *SwapAssocDirection* 215

LIST OF DEFINITIONS

Definition 1.1 Legacy software system ... 1
Definition 1.2 Software reengineering ... 2
Definition 1.3 Reverse engineering ... 2
Definition 1.4 Database reengineering ... 4
Definition 3.1 Data model ... 37
Definition 3.2 Database ... 38
Definition 3.3 Relational database ... 38
Definition 3.4 (Notation) ... 38
Definition 3.5 Flattening ... 38
Definition 3.6 Basic probability assignment, focal proposition ... 42
Definition 3.7 Combination of evidences ... 43
Definition 3.8 Belief function ... 43
Definition 3.9 Plausibility function ... 44
Definition 3.10 Fuzzy set ... 45
Definition 3.11 t-norm and t-conorm ... 46
Definition 3.12 Fuzzy relation ... 47
Definition 3.13 Fuzzy logical operators ... 48
Definition 3.14 Fuzzy inference ... 48
Definition 3.15 Necessity-valued formula ... 49
Definition 3.16 Classical projection ... 50
Definition 3.17 α-cut ... 50
Definition 3.18 Partial contradicting set of formulae ... 51
Definition 3.19 Best model ... 51
Definition 3.20 Formal system for NPL[1] ... 51
Definition 4.1 Signature of an analyzed logical schema ... 58
Definition 4.2 Signature of a GFRN ... 66
Definition 4.3 Context sensitive syntax ... 67
Definition 4.4 Declarative semantics of GFRNs ... 68
Definition 4.5 Extent of a predicate ... 71
Definition 4.6 Data-driven analysis operation ... 71
Definition 4.7 Goal-driven analysis operation ... 71
Definition 4.8 Application context ... 72
Definition 4.9 Expansion of formulae over a finite universe ... 72
Definition 4.10 Occurrence of literals ... 72
Definition 4.11 Semantics of automatic analysis operations ... 72
Definition 4.12 Fuzzy Petri net ... 78
Definition 4.13 Stability ... 80
Definition 4.14 Predecessor ... 80
Definition 4.15 Axiom ... 80
Definition 4.16 Axiom-based marking ... 81
Definition 4.17 Grounded place ... 92
Definition 4.18 Derivability ... 95

Definition 4.19 Derivation sink 95
Definition 5.1 Graph 115
Definition 5.2 Graph production 121
Definition 5.3 Application of a production 121
Definition 5.4 1-context of a set of nodes 148
Definition 5.5 Context of a transformation application 148
Definition 5.6 History graph 149
Definition 5.7 Application of transformations to a history graph 150
Definition A.1 Interpretation of a logical schema 187

CHAPTER 1 INTRODUCTION

To better meld into the software development practice, CASE tools should adopt a programmer's mental model of software projects. In particular, CASE tools should support ***soft*** *aspects of software development as well as rigorous modeling, provide a* ***natural process-oriented*** *development framework rather than a method-oriented one, and play a more* ***active*** *role in software development than current CASE tools.*

Jarzabek and Huang, CACM 8/98. [JH98b]

1.1 Background: the dilemma of software legacies

role of information management

Effective and efficient information management is a crucial factor for the competitiveness of today's companies. It enables them to respond to changing conditions on a global market, quickly. Emerging key technologies like the *World Wide Web* (Web), *Object-Orientation* (OO), *Client/Server* (CS) *applications*, and *open system standards* (e.g., CORBA [Vin97], DCE [Fou92]) greatly influence modern business processes. Besides new applications in the area of *Electronic Commerce* [Ten98], there has been increasing interest in using enterprise-wide data integration to build management information systems and decision support systems [JP92]. While new company start-ups are able to purchase software that takes advantage of the latest technology, longer established enterprises have to deal with various pre-existing software systems. In many cases, these pre-existing systems have to be extended or modified to fit new requirements and exploit emerging technologies. Such modifications are often difficult to achieve in older software. These systems are usually called *legacy software systems* (LSS).

legacy systems characteristics

Many LSS have evolved over several generations of programmers. They do not satisfy the flexibility and growth requirements of modern enterprises. They were built with focus on efficiency rather than on interoperability and maintainability. LSS are often badly documented, which adds to the complexity of achieving modifications. In many cases, technical design documents are obsolete, have been lost, or have never existed. But even without the need for extensive modifications, LSS are increasingly expensive to maintain, because they are usually operated centrally and based on old hard- and software platforms. On the other hand, LSS are of great value if they incorporate important business knowledge and manage a vast amount of *mission-critical* business data. The described characteristics reflected in the following definition which is a combination of the definitions given by Schick [Sch95a] and Umar [Uma97], respectively.

Definition 1.1 Legacy software system

Any software system of value that significantly resists modification and evolution to meet new and constantly changing business requirements is called legacy software system (LSS).

◻

dealing with legacy systems

cold turkey

Enterprises have to handle the dilemma of software system legacies in order to remain competitive and respond to emerging business requirements. One solution is to replace LSS completely by new systems that have been rebuilt from scratch and meet the current requirements. This strategy is called *cold turkey* [Uma97, BS95]. For complex systems, such a project might require up to several years and implies a significant risk. During this time, the LSS is likely to evolve according to urgent business requirements and additional new features. It is often a problem to ensure that the development of the new software system evolves in step with the evolving LSS. In general, *cold turkey* is only cost-effective for software with long expected lives and high demands for flexibility. However, in case of *mission-critical* applications, e.g., systems that have to be operational for 24 hours a day (e.g., billing systems), this solution is hardly viable. This is due to the fact that new complex systems are typically far from being faultless.

reengineering

Because of the aforementioned problems to replace existing software systems there has been increasing interest in concepts, methods, and tools to migrate LSS gradually to fit new requirements. The corresponding process is usually called *reengineering* (RE). This strategy tries to analyze and decompose LSS in order to migrate some of their subcomponents to new technologies while other legacy components are replaced or remain unchanged [BS95]. Typical candidates for such components are user interfaces, data management components, and data processing units. Compared to *cold turkey*, RE aims to achieve *step-wise* improvements in shorter time and with minimized risk. The following definition of the RE process has been adopted from Chikofsky and Cross [CI90]:

Definition 1.2 Software reengineering

The analysis of an LSS to reconstitute it in a new form and the subsequent implementation of the new form is called software ***reengineering*** *(RE).*

reengineering process

According to Definition 1.2, software reengineering processes mainly consist of two subsequent phases, namely *reverse engineering* (RvE) and *forward engineering* (FE). RvE activities investigate an LSS to gain abstract information about its internal structure. The purpose of these activities are to improve the human understanding of an LSS. Chikofsky and Cross give the following definition of the RvE process [CI90]:

Definition 1.3 Reverse engineering

Reverse engineering *is the process of analyzing a subject system with two goals in mind:*

- *to identify the system's components and their interrelationships; and,*
- *to create representations of the system in another form or at a higher level of abstraction.*

The formulation of the above definition (...*two goals in mind*...) reflects on the fact that the RvE task is generally considered to be human-intensive, i.e., it requires a well-trained staff and a high amount of expert knowledge [ALV93, Big90]. Subsequently, the produced abstract information is used as a basis to plan the necessary modifications of the LSS and estimate the required effort. Such *planning* activities are crucial to manage the risk of RE projects [Sne95]. The forward engineering phase aims to implement the planned changes. Often several iterations between the different phases are needed to yield the desired target system. Figure 1.1 illustrates the described evolutionary reengineering process.

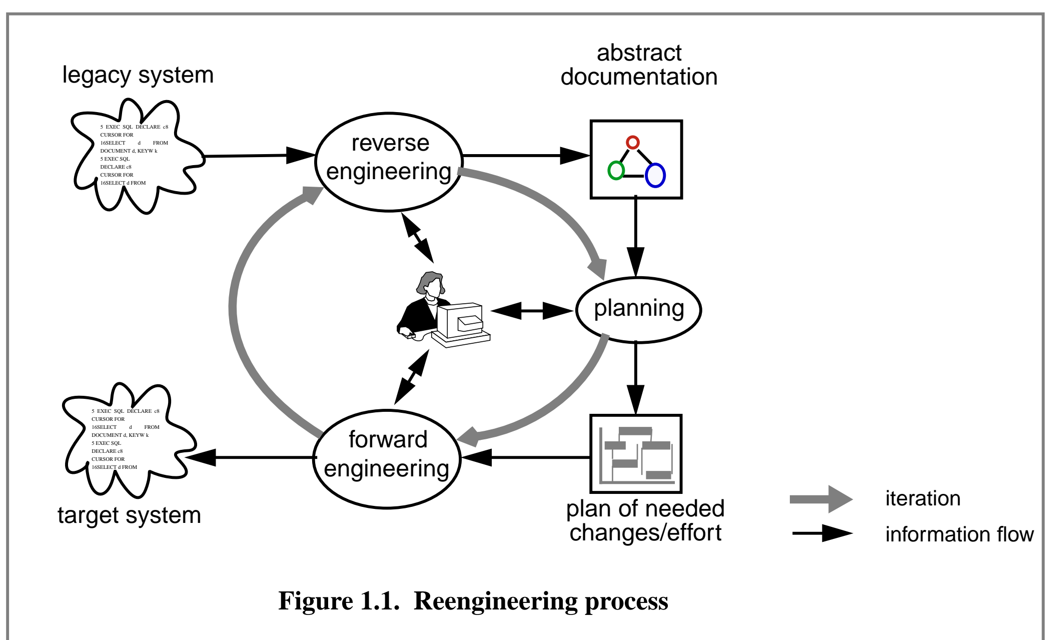


Figure 1.1. Reengineering process

1.2 Database reengineering

mass changes w.r.t. data representation

Software evolution and maintenance problems might be caused by all kinds of new or changed requirements. However, in his keynote for the 1998 Working Conference on Reverse Engineering, McCabe has identified a number of requirements, which are currently of special importance because they are responsible for significant *mass changes* in today's business software [McC98]. Among these central requirements are the *Year-2000* problem [Mar97], the *Euro-conversion* problem [Gro98], and the ability to compete on a global, electronic market. The primary concern of all these requirements is the issue of how business data should adequately be represented in software systems. The addressed problems range from simple questions, e.g., for the number of digits that are necessary to represent a date (Year-2000 problem), up to complex architectural decisions, e.g., how to federate data maintained by diverse (formerly autonomous) information systems and integrate these systems with the Web to facilitate electronic commerce.

importance of data structures

If an LSS has to be adapted to one of these requirements, a conceptual documentation of its data structure is thus often a necessary prerequisite to achieve the maintenance goal. Moreover, a conceptual data structure is an excellent starting point for the migration to modern programming languages, as they are usually data-oriented [GK93]. This is because it reflects major business rules but is fairly independent from procedural application code.

database reengineering

The importance of a sound understanding of legacy data structures in RE projects has been pointed out by several researchers and practitioners [Aik95, HEH^{+}98, GK93]. Unfortunately, a corresponding documentation is missing, obsolete, or inconsistent for many existing LSS. The process of recovering such a documentation from an LSS is called *data reverse engineering* (DRvE) [Aik95]. Today, many existing LSS in business applications maintain data in some kind of database management system (DBMS). In these cases the described design recovery process is also more specifically called *database reverse engineering* (DBRvE), respectively, *database reengineering* (DBRE) if a subsequent modification of the LSS is considered.

Definition 1.4 Database reengineering

Database reengineering *(DBRE) aims on recovering a consistent conceptual model for the persistent data structure of an legacy database (LDB). Subsequently, this conceptual model is used to reconstitute the LDB in a new form. In addition to the LDB schema catalog and the stored data, DBRE processes might consider the same information sources (and employ similar techniques) as general RE processes.*

In general, DBRvE processes are in general more structured than arbitrary DRvE processes [Aik95]. Consequently, the potential for tool support and automation is much higher in DBRE. The main reason for this is that the used DBMS already provides the reengineer with some basic information about the implemented physical data structure in form of a *schema catalog*. Still, important structural and semantic information about the data structure might not be explicit but indicators for this information might be found in different parts of the LDB, including its procedural code, stored data, and obsolete documentation. Moreover, domain experts and developers might be able to contribute further valuable information about the LDB. The DBRE problem is to find, assess, and merge these indicators to create a consistent conceptual DB schema (cf. Figure 1.2). In many cases, heuristics and uncertain expert knowledge have to be employed.

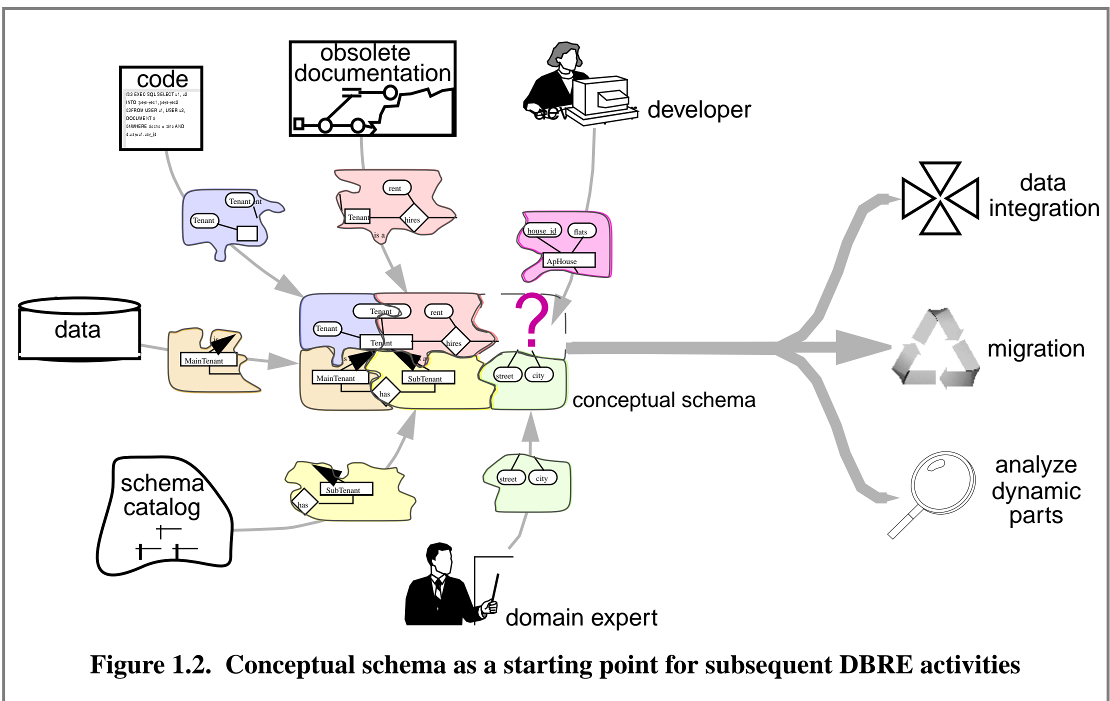


Figure 1.2. Conceptual schema as a starting point for subsequent DBRE activities

1.3 Problem definition

tool support

In the last two decades, many researchers have developed concepts and techniques for automating certain DBRE activities, in order to reduce the complexity of the DBRE task [vdBKV97]. Many of these approaches have been implemented in computer-aided RE (CARE) tools and some of them have proven to be useful for practical applications. CARE tools seem to have great potential to assist the reengineer, e.g., by performing laborious analysis steps, browsing information about legacy software artifacts, and guiding the DBRE process. However,

such tools are rarely used in industry [Sto98]. Researchers and practitioners have identified the most significant reasons for this as their lack of *customizability* [MNB+94] and *human-awareness* [JH98b, Sto98]. Furthermore, they do not allow for *incremental* and *iterative* RE processes [WSK97].

customizability

Customizability is a crucial requirement on CARE tools, because LDBs differ with respect to many technical and non-technical parameters. They are based on diverse (old) hard- and software platforms, use miscellaneous programming languages, contain various optimization structures and arcane coding concepts (*idiosyncrasies* [BP95, HHEH96]), and comprise different naming conventions. Furthermore, DBRE projects may be driven by a great variety of different goals. Such goals range from fixing defects (e.g., Year-2000-Problem [Mar97]), over extending or integrating data structures, up to completely changing the architecture of an LDB, e.g., migrating from a procedural, monolithic, and autonomous legacy system to an open, distributed, and object-oriented application.

While current compiler technology allows to generate parsers for different programming languages based on abstract specifications [Slo95], most existing CARE tools still employ general-purpose programming languages to implement RE heuristics, analysis operations, and processes. As a consequence, these heuristics and processes can hardly be customized for changing application contexts. Some more advanced approaches aim to tackle this problem by providing application programming interfaces (APIs) [BM98] or interpreters for scripting languages [Rat98, TWSM94]. Such interfaces offer the flexibility to adapt CARE tools with less effort or even without the need for recompilation. Still, a limitation of these approaches is their low level of abstraction: RE heuristics and processes have to be programmed in form of procedural scripts, even though they would be more adequately described in a declarative formalism, e.g., in form of textual or graphical rules [SLGC94, HK94, PS92]. Moreover, CARE tools should provide an *open architecture*, i.e., they should allow the integration of other tools (e.g., parsers, analyzers, extractors, and transformers).

human-awareness

One of the most valuable information sources in RE are humans. Developers, operators,[a] and domain experts might be able to contribute important knowledge about a subject LDB. Hence, CARE tools should be *human-aware*, i.e., they should consider human knowledge and interaction in the supported (DB)RE process. The human-awareness of existing CARE tools can be characterized according to two main aspects. The first aspect regards the *role* of human knowledge in the RE process, while the second aspect regards its *representation*.

role of human knowledge

A comparison of CARE tools according to the role of human knowledge concerns the question: *at which point in the supported RE process is human knowledge considered?* We classify existing approaches as either *human-excluded*, *human-involved*, or *human-centered* (cf. Figure 1.3).

a In order to clearly distinguish between the user of a CARE tool and a user of an LSS, we use the term *operator* whenever we refer to an LSS user.

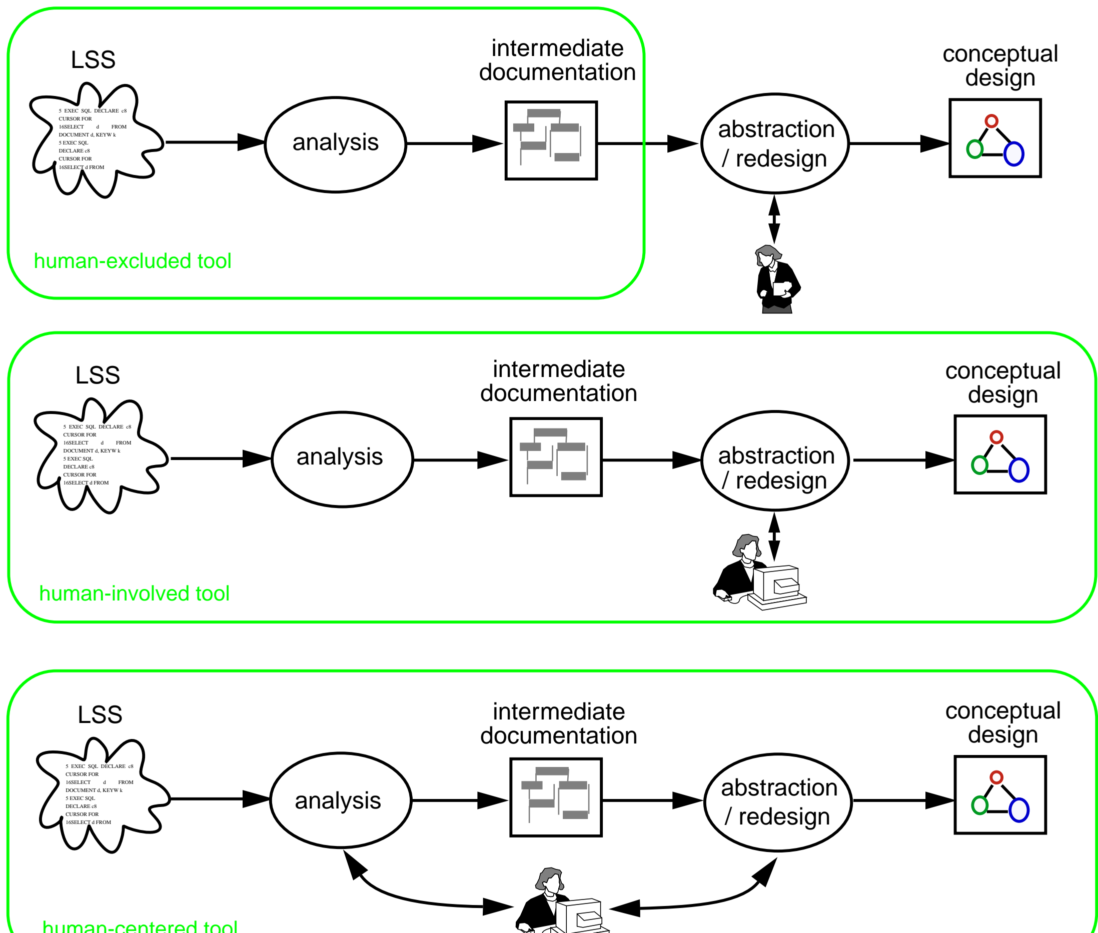


Figure 1.3. CARE tool classification according to the role of human knowledge

Human-excluded CARE tools perform fully-automatic analysis or conceptual abstraction operations on a subject LSS (e.g., [Fon97, RH97, Hüs98, FV95, MCAH95, MN95, SLGC94, Wil94, RHSR94, LS97, vDM98]). As an output, they produce (a number of) analysis reports that can be used as a starting point for manual semantic abstraction and redesign activities. Human knowledge and intervention is not considered in these batch-oriented tool processes.

Many CARE tools involve humans in partly interactive RE processes. Such approaches usually start with an automatic analysis of the LSS in order to extract important structural information. Based on the analysis results, the user can subsequently explore the LSS, and interactively add further semantic abstractions. Examples for such more sophisticated approaches are [Hol98, KWDE98, LO98, Nov97, FHK+97, BGD97, YHC97, ONT96, SM95, MAJ94, AL94, MWT94]. We call these tools *human-involved* as opposed to *human-centered*, because human knowledge is only considered in the second stage of the supported RE process (*abstraction/ redesign*, cf. Figure 1.3). Finally, we denote CARE tools as *human-centered*, if they enable interactive RE processes including both kinds of activities, *software analysis* and *abstraction/ redesign*, e.g., [HEH+98, AG96, MNS95].

representation of human knowledge

The second aspect of *human-awareness* considers the way how human knowledge is represented in CARE environments. RvE activities deal with various heuristics that deliver uncertain analysis results and reengineers have uncertain assumptions about the internal realization of LDBs. Existing CARE tools do not consider this human mental model and represent assumptions and analysis results without a measure for their confidence. Furthermore, RvE activities generally have an evolutionary and explorative nature. It frequently occurs that heuristics deliver contradicting analysis result, i.e., the reengineer discovers indicators in favor of a given hypothesis as well as against it. Current CARE tools do not tolerate such contradictions and most of them do not even indicate them. This is a severe limitation because in a later stage of the RvE process it might become clear that the hypothesis that has been chosen in such a situation has to be refuted. In this case, the knowledge about the indication of its alternative has been lost due to the inability to represent "both sides of the coin".

iterations

Another problem of currently existing CARE tools is their limited support for process iterations. They usually assume that the process of knowledge accumulation is *monotonic* and prescribe a strictly phase-oriented methodology. In practice, this is an important limitation as iterations between analysis and abstraction steps occur frequently: When a reengineer learns more about the abstract design of an LSS, (s)he often refutes some initial assumptions or does some further investigations. For example, as soon as an intermediate abstraction of an LSS has been created it can be discussed with domain experts which might elicit additional information. In many cases, this new information contradicts to some initial assumptions. Strictly phase-oriented tools do not aid the reengineer in detecting and resolving such inconsistencies. In case of iterations with early analysis activities the reengineer loses the work (s)he has performed interactively in later abstraction and redesign activities.

1.4 The approach

In this dissertation, we developed concepts and techniques that allow to build CARE environments which overcome the aforementioned limitations of current approaches in the DBRE domain. We propose a process that consists of two main phases, namely *schema analysis* and conceptual *schema migration* (cf. Figure 1.4). In the first phase, the different parts of the LDB are analyzed to obtain a consistent and complete logical schema for the implemented physical representation. In the second phase (migration), this logical schema is transformed into a conceptual schema that is a suitable basis for subsequent maintenance activities like schema extension and federation, data integration, distribution, and code migration.

GFRN to achieve customizability

We developed a dedicated graphical language named *Generic Fuzzy Reasoning Nets* (GFRN) to customize the analysis process of CARE tools according to their specific application context. GFRN specifications separate declarative knowledge from operational aspects. They provide a high level of abstraction and extensibility. Analysis operations that have been developed in other (DB)RE approaches can easily be integrated with GFRN specifications. We implemented a prototype CARE environment that is parameterized by GFRN specifications and includes a *customization front-end* for this purpose.

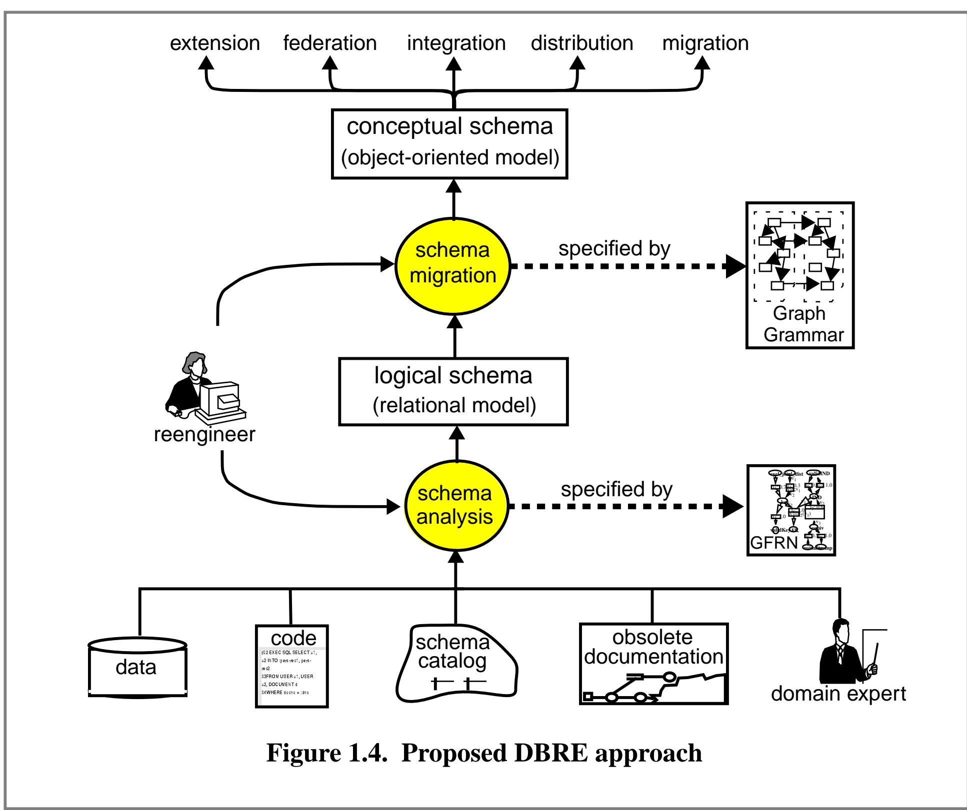


Figure 1.4. Proposed DBRE approach

analysis guided by possibilistic inference engine

We reflect the mental model of the reengineer by representing RvE knowledge in the framework of possibility theory [DLP94]. This approach allows to deal with uncertain and contradicting analysis results. We developed a non-monotonic *inference engine* (IE) that supports the reengineer in his/her DBRvE activities by propagating and indicating measures of credibility and contradiction. For this purpose, the IE interprets the declarative knowledge that has been specified in the GFRN specification. In addition, the IE is also capable of executing the analysis operations that are specified in the GFRN. This is done automatically during the DBRE process to search for indicators or validate intermediate hypotheses. With this approach, we obtain a CARE tool that plays a more *active* role in the DBRE process than existing tools.

user interaction

A graphical *dialog component* visualizes the current knowledge about the persistent structure of an LDB to the user. This component provides powerful abstraction and query mechanisms to focus the reengineers attention on the most controversial parts of the legacy schema. It enables the reengineer to enter the results of manual investigations or add new hypotheses that might be falsified or supported by the IE. Hence, our approach intertwines automatic and manual analysis activities in an explorative and evolutionary process that is guided by the IE until a consistent (and definite) logical schema is obtained.

iterations between analysis and migration

We applied *graph grammars* [Roz97] to map the analyzed logical schema into a conceptual (OO) data model. The resulting conceptual schema can interactively be enhanced and redesigned to exploit additional abstraction mechanisms and migrate to new requirements. The available redesign operations are formally defined by graph transformation rules. Based on this formalization, we developed a consistency management component that incrementally propagates modifications of the logical schema to its (redesigned) conceptual representation in

case of process iterations. This unburdens the reengineer from the error-prone and time-consuming task to determine such inconsistencies manually. The developed consistency management component can be viewed as an adaption of general techniques described in [Nag96] to the (DB)RE domain.

The particular research contributions of this dissertation have partly been published in [JSZ96, JSZ97, JZ97, JNW98, JH98a, JZ98, JW99b, JZ99, JS99, JW99a, JW99c, JSWZ99] and can be summarized as follows:

- We elaborated a catalog of requirements on a theory to manage imperfect knowledge in human-centered DBRE environments.
- We used these requirements to evaluate the appropriateness of major theories for managing imperfect knowledge with respect to the application domain of DBRE. We showed that fuzzy set theory and possibilistic logic [DP88] provide a suitable basis for our application.
- We defined *Generic Fuzzy Reasoning Nets* (GFRNs), as an abstract, graphical formalism to specify domain-specific DBRE knowledge and schema analysis processes.
- In order to interpret GFRN specifications, we developed a non-monotonic inference engine (IE) that allows for user interaction and automatically executes specified analysis operations to refute or support hypotheses. As a basis for this IE, we extended the *fuzzy Petri net* (FPN) model described by Konar and Mandal [KM96] to represent and propagate contradicting situation-specific knowledge.
- We employed *graph grammars* [Roz97] to map the relational data model to a formally defined conceptual data model.
- We formalized conceptual redesign operations by graph transformation rules.
- We defined a data structure (called *migration graph model*) that represents the mapping between the logical (relational) schema and its conceptual (object-oriented) representation. The migration schema is updated incrementally with every redesign operation and allows to develop
  - a consistency management mechanism that incrementally propagates changes in the logical schema to the conceptual schema to support iterations in the DBRE process;
  - an update mechanism that automatically implements semantic modifications of the conceptual schema to the logical schema;
  - an automatic generator for textual schema mapping descriptions as the input for commercial off-the-shelf (COTS) middleware components. In particular, we describe the integration of the object-relational middleware product *ObjectDRIVER* [CER99] which provides seamless integration of distributed object-oriented applications and legacy data according to the *ODMG* standard [CBB$^+$97].
- We implemented our approach in a prototype DBRE environment (called *Varlet*) and we use an industrial case study for evaluation purposes.

1.5 Dissertation outline

Chapter 2

This dissertation is organized as follows. In the next chapter, we characterize the application domain of DBRE with a motivating sample scenario that summarizes our experiences with an industrial case study. By means of this scenario, we point out a number of observations in order to provide the motivation for CARE tools that are able to deal with uncertain reengineering knowledge and process iterations. We revisit details of this scenario throughout the dissertation to exemplify and evaluate different aspects of our approach.

Chapter 3

Chapter 3 elaborates central requirements on a formalism to represent and reason about DBRE knowledge in human-centered CARE environments. We use these domain-specific requirements to evaluate different theories on managing imperfect knowledge with respect to their suitability for the application domain of DBRE.

Chapter 4

Subsequently, we introduce *Generic Fuzzy Reasoning Nets* (GFRN) as a dedicated formalism to specify, execute, and customize DBRE heuristics and processes. The execution of GFRN specifications is based on an inference mechanism that employs an FPN model which enables non-monotonic reasoning under uncertainty and contradiction. The last part of Chapter 4 describes an implementation of these concepts and mechanisms in a customizable DBRE tool that guides the user in an evolutionary schema analysis process.

Chapter 5

In Chapter 5, we describe a flexible approach to database schema migration based on graph transformation systems. We employ graphical mapping rules to yield an initial translation that is subsequently enhanced by applying an extensible set of conceptual redesign transformations. Redesign transformations are defined by graph productions which fosters human comprehension and provides a sound basis for their semantics. Furthermore, we describe a mechanism for incremental consistency management to support iterations between intertwined analysis and redesign steps. After the LDB schema has been analyzed and migrated to a suitable conceptual target schema, we describe the integration of middleware components that encapsulate LDBs with object-oriented interfaces. This flexible approach facilitates gradual migration to new technologies like object-orientation, *Java*, and the Web, because autonomous legacy applications can be preserved.

Chapters 6

Both technical chapters (4 and 5) are closed with a discussion and evaluation of our results with practical case studies and a comparison with related work. Finally, Chapter 6 provides a summary of our major contributions and identifies a number of open problems and future directions of our approach.

CHAPTER 2 DATABASE REENGINEERING- A CASE STUDY

In this chapter, we introduce a database reengineering (DBRE) sample scenario that summarizes some experiences we made in an industrial project with two German companies. The reason for this chapter is to elicit characteristic observations about DBRE activities to motivate our approach. It presents one coherent application example that integrates the different aspects covered in this thesis. We will revisit this example throughout the dissertation and it will be used to evaluate the developed concepts and techniques. Even though the background of the presented scenario is a concrete industrial case study, the presented implementation details have been changed to protect copy rights, simplify the presentation, and consider experiences with other projects. We presume that the reader is familiar with the basic terminology of relational DBs. An excellent introduction to this subject is given by Elmasri and Navathe [EN94].

2.1 A legacy product and document information system

PDIS overview

The case study deals with a legacy *product and document information system* (PDIS) of an international enterprise that produces a great variety of drugs and other chemical products. The original version of PDIS was developed in Cobol [McC75] as an information system to maintain the company's product catalog using a relational DB (DB2) [Dat84]. Subsequently, the functionality was extended to support the maintenance of information about documents related to stored products. PDIS has evolved over more than ten years and has been subject to many modifications (e.g., according to new national and international safety regulations and changing organizational structures). PDIS has become the central source for information about more than 100,000 products currently available and it contains more than 50,000 references to documents including product specifications, safety classifications, research reports, and marketing statistics. PDIS is accessed by members of the central hotline at the company headquarters. They use it to answer questions of customers, product managers, and researchers about products and available documents (cf. Figure 2.1). The functionality of the system is considered to be *mission-critical*, i.e., the company depends on the service provided by PDIS.

PDIS problems

Today, PDIS has become increasingly expensive to operate. It requires a huge staff of hotline members at the company headquarters to answer all ingoing inquiries. Different international time zones demand for extended business hours of this hotline service. Furthermore, old hard- and software platforms and the lack of a consistent technical documentation result in serious problems in maintaining the system.

2.2 Migration target: a distributed marketing information system

migration objectives

Because of the problems described above, the Information Technology (IT) department plans to employ Internet-technology to establish a distributed Web-based marketing information

system (MIS) that covers and extends the functionality of PDIS. The aim of this project is to reduce expenses and increase the availability and currency of the stored data.

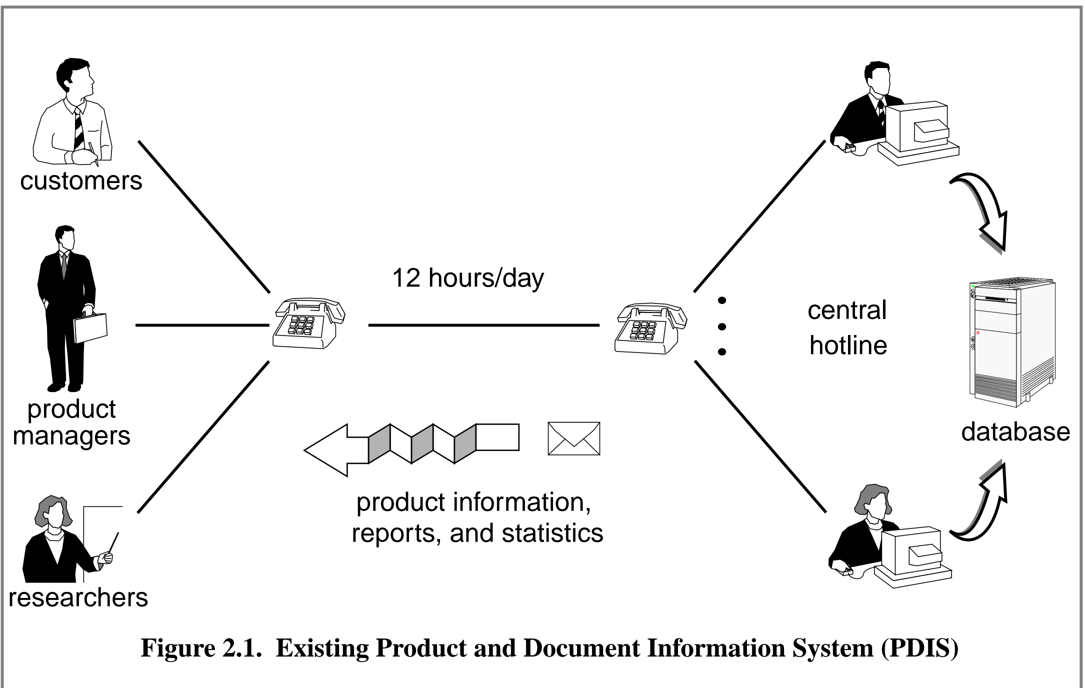


Figure 2.1. Existing Product and Document Information System (PDIS)

2.2.1 Functional requirements

Analogously to the old PDIS, the primary purpose of the planned MIS is to store and retrieve up-to-date product information. However, the new system aims to provide customers and employees with *direct* access to product data (24 hours a day). This will drastically reduce the expenses to operate the central hotline service at the company headquarters. Another goal is to improve the availability of information and unburden the company's marketing department by providing on-line versions of frequently accessed documents. Moreover, the IT department aims to integrate their sales, distribution and financial controlling system (SAP R/3, [KKM98]) with the new MIS, to increase the currency of marketing information. User statistics created by MIS will be transferred to the company's data warehouse that is used for strategic planning. A schematic overview of the planned MIS is given in Figure 2.2.

Other functional requirements on MIS are implied by the heterogeneity of its prospected users. In contrast to the old PDIS, the new system will not be accessed by well-trained staff but by geographically distributed users in various roles and with different knowledge about the system. Thus, MIS has to provide a simple user-interface with on-line help for inexperienced customers, as well as more sophisticated access mechanisms for experts (e.g., product managers). Furthermore, there have to be authorization and security mechanisms, as certain information may not be publicly available.

2.2.2 Technical requirements

The most important technical requirement on MIS is for high availability. This is due to the fact that its functionality is considered to be mission-critical. Consequently, the MIS client has to run on a wide range of different hard- and software platforms and the MIS server has to be reliable and fault tolerant. Of course, the new MIS should be extendable and overcome the maintenance problems of PDIS.

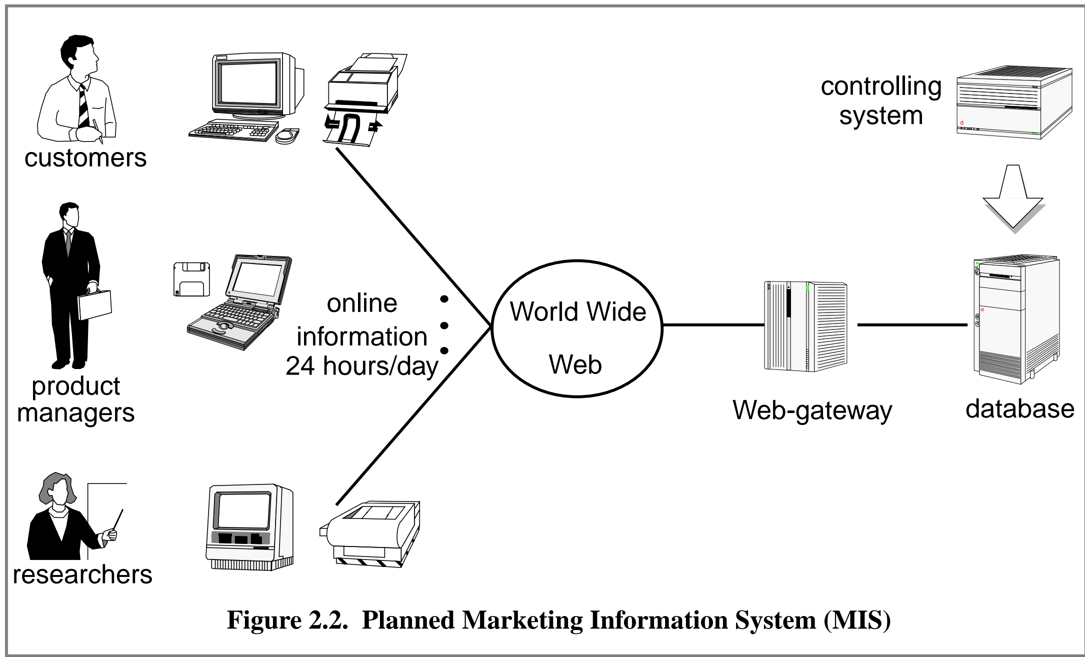


Figure 2.2. Planned Marketing Information System (MIS)

2.3 Migration strategy

The object-oriented programming language *Java* [GJS97] was chosen to implement the MIS. It facilitates to meet most technical requirements, because of its support for distributed, heterogeneous architectures and its built-in security concept. In addition, the IT department has selected the *Unified Modeling Language* (UML) [RJB99, BRJ99] to specify and document MIS.

In order to be able to test and improve the reliability of the new system, the company plans to migrate gradually from the old PDIS to the new MIS. This means that both systems have to be operated in parallel until the MIS runs stable. During this period, customers and employees will still have the possibility to access information via the hotline service. This strategy entails that both systems must access the same up-to-date information. One possible solution was to create a completely new DB for the MIS and periodically replicate and propagate information changes between the MIS and the PDIS. However, this would result in temporary inconsistencies and a low currency of stored information. Thus, the IT department decides to integrate both systems by using a common DB. The plan is to decompose the PDIS data management component to extend and reuse it in the MIS. The current realization of this component in DB2 facilitates this decomposition. However, a conceptual design of the implemented data structure is not available. Thus, the data management component has to be reverse engineered before it can be extended.

As the information provided by the MIS will be an extension of the data stored in the PDIS, the necessary changes of the DB schema can be made in a way that preserves compatibility with the procedural rest of the PDIS. This allows to run the legacy application with no or only little modification on top of the extended DB schema. The integration of the common DB with the object-oriented rest of the new MIS will be realized by an object-relational *middleware* layer. The purpose of this layer is to perform the transformation between *Java*-objects and relations. The implementation of the middleware will be based on the standardized *Java database connectivity* (JDBC [PM96, YLQ98]). The resulting gradual migration strategy is illustrated in Figure 2.3.

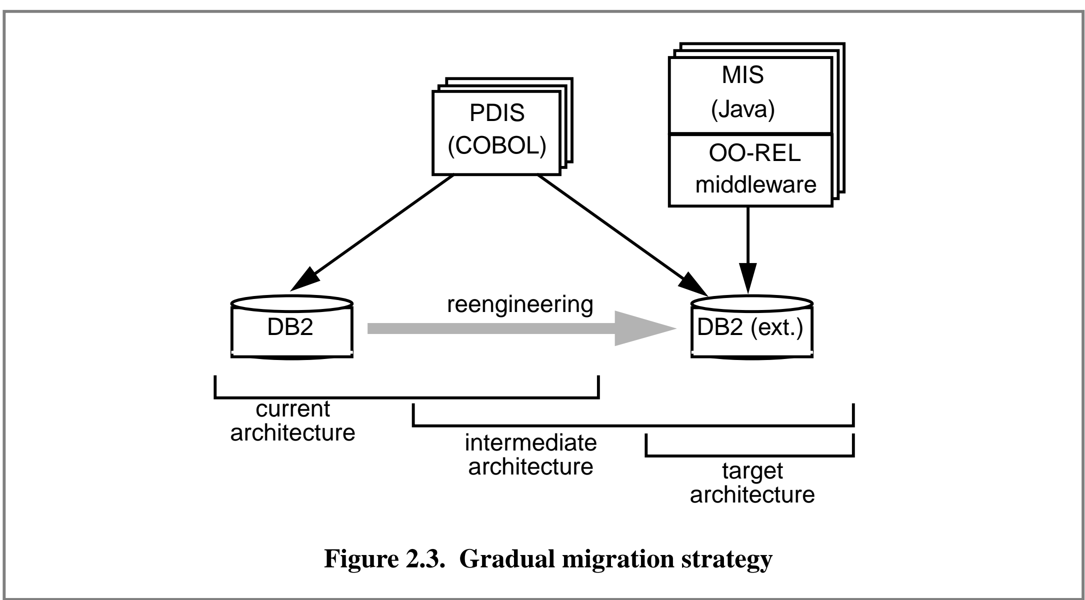


Figure 2.3. Gradual migration strategy

2.4 The reengineering process

The planned RE process consists of several subsequent activities, which are shown in Figure 2.4. The first two activities aim to recover an abstract model for the persistent data structure of PDIS. The first step is to extract the available schema catalog from DB2. Then, the resulting physical schema is structurally completed and semantically enriched by adding further information gained in *analyzing* PDIS and querying human experts. This analysis activity produces the logical schema (cf. Figure 1.4). Subsequently, the next activity (*migration*) maps the logical schema to a (semantically equivalent) conceptual schema in UML-notation.

The following activities are forward engineering steps. The first step (*redesign*) extends the reverse engineered conceptual schema according to the additional requirements on MIS. The resulting extended conceptual schema is the basis to employ UML to specify dynamic properties of MIS (e.g., object interaction and classes for transient objects). Moreover, the extended conceptual schema also serves as input for an activity to modify the implemented DB schema accordingly and develop the object-relational middleware for MIS. Finally, the last two activities aim to implement the *Java* classes for MIS and migrate the legacy data to the extended relational schema.

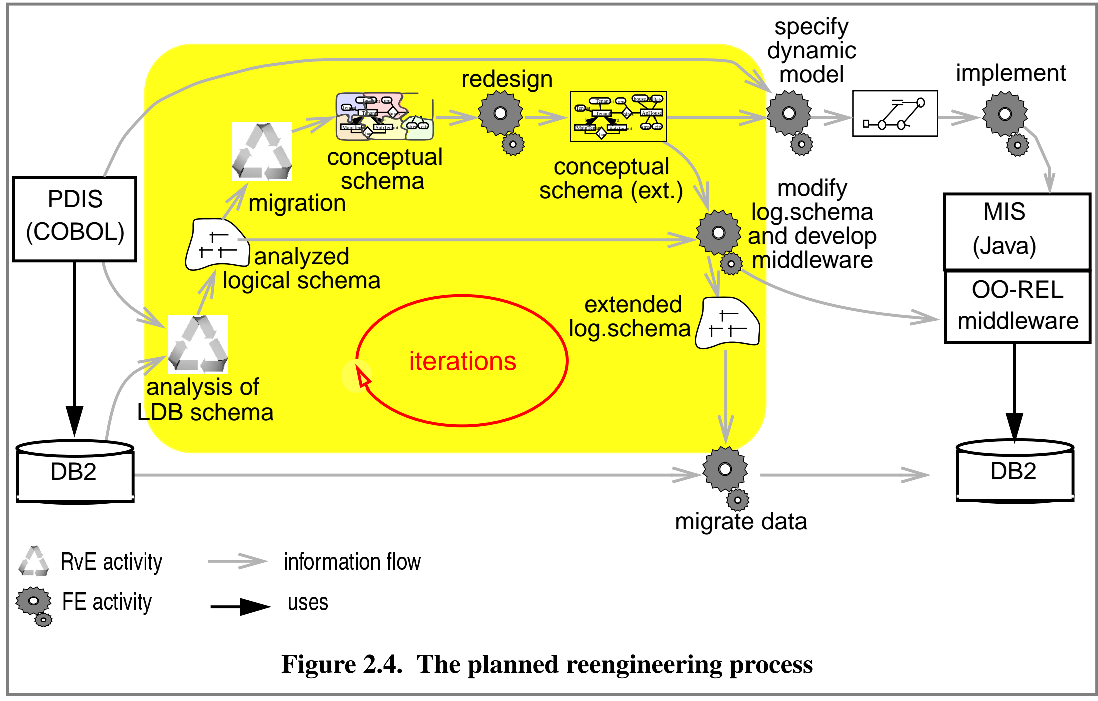


Figure 2.4. The planned reengineering process

problem of inconsistency

Subsequently, we will use our sample scenario to exemplify each activity in the shaded area of the displayed RE process (Figure 2.4). We will point out typical observations and experiences made with each activity. In particular, we will also show that problems might be caused by inconsistencies among documents[a] used in different stages of this process. Such inconsistencies might arise for several reasons, e.g., due to process iterations or human failures during manual activities. In our sample scenario, we will exemplify one instance of such an iteration. However, according to the evolutionary and explorative nature of DBRE processes, often several such iterations occur in practice.

2.4.1 Legacy schema analysis

activity overview

This activity starts with the physical schema catalog extracted from the LDB. In most cases, this schema catalog lacks important structural and semantical information that is needed in order to recover a correct logical schema. Furthermore, legacy schemas often comprise optimization structures and de-normalizations that obscure their original meaning. The goal of this first activity is thus to detect these de-normalizations and hidden constraints in order to produce a *structural complete* and *semantically enriched* legacy schema (cf. [FV95]).

PDIS example

Let us revisit our sample scenario to exemplify this activity. Figure 2.7 shows a small detail of different parts of PDIS, including its schema catalog, a small snapshot of its data, and four selected segments of its procedural code. Moreover, Figure 2.7 illustrates that human experts are another valuable source of information about an LDB. For the sake of simplicity, we will refer only to this detail of PDIS throughout this sample scenario. That means we will only consider the eight relational tables shown, instead of all 85 tables of the real case study.

a We use the common term *documents* to denote the various representations of information used as in- or output of RE activities.

Structural completion

In case of a relational database, the activity of *structural completion* mainly consists of detecting all key *completion* mainly consists of detecting all key candidates and foreign keys, which are not declared explicitly in the schema catalog. According to Figure 2.7, the schema catalog of PDIS includes definitions of tables and some index structures. A definition of a unique index trivially implies a key constraint on the corresponding table. Hence, the schema catalog gives information about the five key constraints displayed in Figure 2.5.

Key: COMGRP(cgid)
Key: PRODGRP(cg,pg)
Key: DOCUMENT(docno)
Key: PRODUCT(no,pg,cg)
Key: KEYW(keyw,seqn)

Figure 2.5. Constraints resulting from the schema catalog

heuristics as indicators

In order to detect additional key candidates, the reengineer has to make further investigations. Typically, (s)he has to make use of heuristics to find indicators for the desired information. Such heuristics for relational information systems are described for example in [HHEH96, FV95, BP95, PKBT94, PB94, SLGC94, And94, ALV93].

naming conventions

Simple, commonly used heuristics check column names for typical naming conventions. In our example, the reengineer assumes that the *id* columns of tables *DOCREF* and *PRODREF* represent key values, as many programmers use similar names to label key columns. Likewise, (s)he assumes *usrid* to represent a key column of table *USER*, because keys are often named after their tables with a supplemented string "*id*". Furthermore, the reengineer might check column names in different tables for similarities in order to detect foreign key constraints. Obviously, such potential foreign keys have to be type compatible with their referenced columns. Moreover, there should be a key constraint over the columns referenced. One possible application of this heuristic to our example is to infer a potential foreign key from columns *cg* and *pg* of table *PRODUCT* to the equally named columns in table *PRODGRP*. Analogously, the reengineer might also compare column names with names of tables. For example, (s)he might notice that the name of column *usr* of table *DOCUMENT* is very similar to the name of table *USER*. This suggests a foreign key constraint between these two tables, even despite the fact that column *usr* (*DOCUMENT*) is not *exactly* type compatible with the referenced key column *usrid* (*USER*). Such slight type incompatibilities among columns with identical meaning occur frequently in LDB applications. Naming heuristics similar to those described above can be used to indicate the rest of the potential constraints listed in Figure 2.6.

Key?: DOCREF(id)
Key?: PRODREF(id)
Key?: USER(usrid)
Foreign key?: PRODUCT(cg,pg) -> PRODGRP(cg,pg)
Foreign key?: DOCUMENT(usr) -> USER(usrid)
Foreign key?: PRODGRP(cg) -> COMGRP(cgid)
Foreign key?: DOCREF(sdoc), DOCREF(tdoc) -> DOCUMENT(docno)
Foreign key?: KEYW(doc1)...KEYW(doc5) -> DOCUMENT(docno)

Figure 2.6. Potential constraints indicated by naming heuristics

schema catalog

```
create table COMGRP (
        cgid: INTEGER;
        name: CHAR(18));
create unique index COMGRPIDX
        on COMGRP(cgid);

create table PRODGRP (
        cg: INTEGER;
        manager: CHAR(40);
        pg: INTEGER;
        grpname: CHAR(18));
create unique index PRODGRPIDX
        on PRODGRP(cg,pg);

create table DOCUMENT (
        dname: CHAR(255);
        docno: INTEGER;
        valid: CHAR(8);
        author: CHAR(255);
        usr: CHAR(30);
        rd: INTEGER);
create unique index DOCIDX
        on DOCUMENT(docno);

create table DOCREF (
        id: INTEGER;
        sdoc: INTEGER;
        tdoc: INTEGER)
create index DREFIDX
        on DOCREF(sdoc);

create table PRODUCT (
        name: CHAR(50);
        no: INTEGER;
        pg: INTEGER;
        cg: INTEGER));
create unique index PRODIDX
        on PRODUCT(no,pg,cg);

create table PRODREF (
        id: INTEGER;
        pg: INTEGER;
        prod: INTEGER;
        cg: INTEGER;
        doc: INTEGER)
create index PREFIDX
        on PRODREF(doc);

create table USER (
        usrid: CHAR(10);
        name: CHAR(50);
        dpt: CHAR(18);
        sname: CHAR(18);
        addr: CHAR(40);
        telo: CHAR(18);
        telp: CHAR(18))

create table KEYW (
        keyw: CHAR(20);
        seqn: INTEGER;
        doc1: INTEGER;
        doc2: INTEGER;
        doc3: INTEGER;
        doc4: INTEGER;
        doc5: INTEGER)
create unique index KEYWIDX
        on KEYW(keyw,seqn)
```


data

COMGRP

cgid	name
3	lacquer
10	laboratory chemicals
8	medical supply

PRODGRP

cg	manager	pg	grpname
3	sch03	80	metal adhesive lacquers
10	mul08	20	liquid laboratory chemicals
8	bes01	60	pharma

DOCUMENT

dname	docno	valid	author	usr	rd
rHT sales rep.8	67487	1.2.98	Kruger	10	1
specific. H_2O(d)	47639	31.06.99	Niere	8	0
flyer metal lac 4	12004	31.12.97	Steel	19	0
cost statmt 1/9	98586	31.12.98	Thun	10	1

DOCREF

id	sdoc	tdoc
2	98586	67487
4	98586	94763
1	12004	76380
2	82656	03654

PRODUCT

name	no	pg	cg
H_2O (dest.) 10 L.	100	20	10
lacFlex-R1-red8	218	80	3
relief-HT-50	163	60	8
X-500 reflection	289	50	3

PRODREF

id	pg	prod	cg	doc
3	NULL	NULL	5	98586
3	60	163	8	67487
8	80	NULL	3	12004
2	NULL	NULL	NULL	98586

USER

usrid	name	dpt	sname	addr	telo	telp
3	John Best	MRD	bes01	MLab 340	6020	NULL
10	Manfred Schmitz	PCD	sch02	OfficeW 450	3530	58787
8	Heinrich Muller	CRD	mul08	ChemB A350	8331	52718

KEYW

keyw	seqn	doc1	doc2	doc3	doc4	doc5
automobile	40	12004	12005	12006	1210	8902
automobile	41	8903	8904	NULL	NULL	NULL
statistic	304	67487	98586	48563	NULL	NULL

procedural code

```
01 PROCEDURE DIVISION.
...
code segment 1:
02 EXEC SQL SELECT u1, u2 INTO :pers-rec1, pers-rec2
03  FROM USER u1, USER u2, DOCUMENT d
04  WHERE docno = :dno AND d.usr=u1.usrid
05        AND u2.dpt=u1.dpt AND u2.addr=u1.addr
06        AND NOT u1.sname=u2.sname END-EXEC.
...
code segment 2:
07 EXEC SQL SELECT DISTINCT * INTO :pers-rec
08  FROM USER WHERE sname = :SN and dpt= :DEP
09 END-EXEC.
...
code segment 3:
10 EXEC SQL DECLARE c1 CURSOR FOR
11  SELECT p FROM PRODUCT p, DOCUMENT d, PRODREF r
12        WHERE title =:T AND r.doc=d.docno
13              AND r.prod=p.no AND r.pg=p.pg AND r.cg=p.cg
14 END-EXEC.
...
code segment 4:
15 EXEC SQL DECLARE c8 CURSOR FOR
16  SELECT d FROM DOCUMENT d, KEYW k
17        WHERE k.keyw = :searchword AND
18              (doc1=docno OR doc2=docno
19              OR doc3=docno OR doc4=docno
....
```

Figure 2.7. Detail of PDIS

code patterns

Indicators for potential constraints might also be found in the procedural code of the legacy system, e.g., in form of stereotypical *code patterns*. Andersson (informally) describes the idea of searching legacy code for typical SQL[a] queries that serve as indicators for constraints about the database schema [And94]. The four code segments of COBOL-embedded SQL presented in Figure 2.7 are instances of such code patterns.

code segment 1: cyclic join pattern

Code segment 1 is an instance of the so-called *cyclic join pattern* [And94]. The purpose of the query is to deliver contact information about the user (*u1*) who is responsible for a given document and another person (*u2*) who works close to this user. The corresponding query is called *cyclic join*, because it selects two rows in the same table. At this, an inequality condition assures that these two rows are not identical. Hence, a cyclic join serves as an indicator for a key candidate. In code segment 1, the inequality condition is applied to column *sname* of table *USER*, which indicates an alternative key for this table.

code segment 2: select distinct pattern

The second code segment is an example for the fact that indicators might even be contradicting. It is an instance of the so-called *select distinct pattern*. It selects a person record according to a given value for columns *sname* and *dpt*. At this, the keyword *DISTINCT* is used in SQL to eliminate multiple equal rows in the result of a query. However, such equal rows can only occur, if columns *sname* and *dpt* do *not* represent keys in table *USER*. This code segment is thus a negative indicator that *sname* represents a key and contradicts to the previously found cyclic join indicator (cf. Figure 2.8). However, the select distinct indicator generally has a lower credibility than the cyclic join pattern, because many programmers tend to use the keyword *DISTINCT* even if it is not needed. The reengineer has to make further investigations (e.g., examine the available data) in order to resolve such contradicting analysis results.

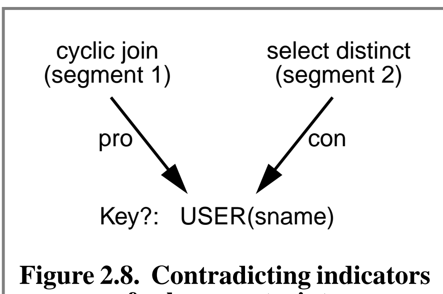


Figure 2.8. Contradicting indicators for key constraint

code segment 3: join pattern

Code segment 3 is an instance of a so-called *join pattern*. It joins table *PRODREF* with table *DOCUMENT* and table *PRODUCT*, respectively. In our scenario, the reengineer uses this join statement as an indicator for two potential foreign keys in table *PRODREF* (cf. Figure 2.9). Likewise, joins in other queries might be found as indicators for the additional potential foreign keys listed in Figure 2.9.

Foreign key?: PRODREF(doc) -> DOCUMENT(docno)
Foreign key?: PRODREF(prod,pg,cg) -> PRODUCT(no,pg,cg)
Foreign key?: PRODGRP(cg) -> COMGRP(cgid)
Foreign key?: PRODGRP(manager) -> USER(sname)

Figure 2.9. Potential foreign keys indicated by join patterns

data analysis

The reengineer can use a snap shot of the available data (DB extension) to validate assumed (potential) constraints about the legacy schema. Of course, hypotheses can only be falsified but not proved by means of data. Still, the fact that an assumed constraint holds for a huge amount of data can provide further support for this hypothesis. In our sample scenario, we will only use the small amount of sample data represented in Figure 2.7. It shows that most of the potential

a *Structured Query Language* [Dat89]

constraints cannot be falsified via the available extension. However, the entries in tables *PRODREF* and *DOCREF* contain counterexamples for the initial belief of the reengineer that the name *id* might label key columns in these tables (cf. Figure 2.6 and Figure 2.7). By talking to PDIS users in the company hotline, the reengineer learns that references from documents to products or other documents are uniquely numbered per each referencing document. This additional knowledge leads to the assumption that columns *id* and *doc*, respectively, *id* and *sdoc*, represent keys for the corresponding tables *PRODREF* and *DOCREF.* A new investigation of the available data supports this assumption, because it holds for the huge extension of these two tables (which is not shown in Figure 2.7). Consequently, Figure 2.10 summarizes the result of the structural completion of our schema detail.

Keys	Foreign keys
Key: COMGRP(cgid)	Foreign key: PRODUCT(cg,pg) -> PRODGRP(cg,pg)
Key: PRODGRP(cg,pg)	Foreign key: DOCUMENT(usr) -> USER(usrid)
Key: PRODGRP(manager)	Foreign key: PRODGRP(cg) -> COMGRP(cgid)
Key: DOCUMENT(docno)	Foreign key: DOCREF(sdoc), DOCREF(tdoc) -> DOCUMENT(docno)
Key: PRODUCT(no,pg,cg)	Foreign key: KEYW(doc1)...KEYW(doc5) -> DOCUMENT(docno)
Key: USER(usrid)	Foreign key: PRODREF(prod,pg,cg) -> PRODUCT(no,pg,cg)
Key: USER(sname)	Foreign key: PRODGRP(manager) -> USER(sname)
Key: PRODREF(id,doc)	
Key: DOCREF(id,sdoc)	
Key: KEYW(keyw,seqn)	

Figure 2.10. Result of the structural completion

Semantical enrichment

Semantical enrichment aims to classify and annotate LDB schema components according to detected optimization structures and higher level concepts like inheritance and aggregation [EN94]. This activity usually starts with an LDB schema that has already been structurally completed [FV95]. Nevertheless, both activities, structural completion and semantic enrichment, are highly intertwined in practice. This means that often important structural information is discovered during semantic enrichment, that has not been detected before. Our sample scenario will reflect on this experience.

inheritance structures

Relational data models often contain indicators for hidden inheritance structures. Similar or synonymous names of tables or groups of columns might represent hints for such structures. In our sample scenario, the reengineer knows that PDIS maintains references among different documents and products. Due to this domain knowledge and the similarity of the names of tables *PRODREF* and *DOCREF*, (s)he assumes a hidden inheritance structure. (S)he beliefs that the purpose of tables *PRODREF* and *DOCREF* is to store references from documents to other documents and from documents to products, respectively. Thus, her/his first assumption is that there might be a (hidden) common *domain relation* [FV95] *REFERENCE* that covers all references in general (cf. Figure 2.11).

variant records

However, when the reengineer considers other available information sources, (s)he has to refute the initial assumption that tables *PRODREF* and *DOCREF* serve for separate specialized concerns: the DB extension shows an overlapping among key values (*id*,*doc*) of both tables. In fact, each key value in table *DOCREF* seems to imply an equal key value in table *PRODREF*. Furthermore, all of these implied rows seem to comprise NULL-values in all other columns (cf. Figure 2.7). After a more detailed investigation of the available data of table *PRODREF* the reengineer discovers that there are actually four different variants of entries in this table, which are displayed in Figure 2.13. By talking to PDIS users, the reengineer learns that the system not only allows for references from documents to products but also to product groups and commodity groups.

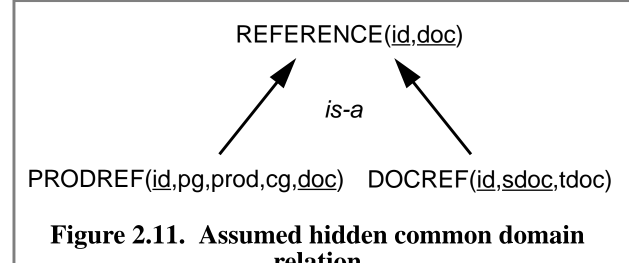


Figure 2.11. Assumed hidden common domain relation

Moreover, (s)he learns that all references, either among different documents or among documents and products, have unique numbers with respect to the referencing document. This requirement for unique reference numbers is a plausible explanation for the hypothetical inclusion dependency between tables *DOCREF* and *PRODREF*: each entry in table *DOCREF* implies a numbering place holder in table *PRODREF*. Together, the additional domain knowledge allows to label the detected variants in table *PRODREF* and entails the three new foreign key constraints shown in Figure 2.12. The fact that Variant 4 of table *PRODREF* represents place holders for document references allows to classify the corresponding foreign key as an inheritance (is-a) relationship [HHEH96].

Variant 1 (product reference):
PRODREF(id,pg,prod,cg,doc)
Variant 2 (product group reference):
PRODREF(id,pg,cg,doc)
Variant 3 (commodity group reference):
PRODREF(id,cg,doc)
Variant 4 (placeholder for document reference):
PRODREF(id,doc)

Foreign key: PRODREF(pg,cg) (var. 2) -> PRODGRP(pg,cg)
Foreign key: PRODREF(cg) (var. 3) -> COMGRP(cg)
Foreign key: DOCREF(id,sdoc) -> PRODREF(id,doc) (var. 4) *(classified as is-a relationship)*

Figure 2.12. Labeled variants and additional foreign keys of table *PRODREF*

variant#	id	pg	prod	cg	doc
1	...	...	...	...	...
2	...	...	NULL	...	...
3	...	NULL	NULL	...	...
4	...	NULL	NULL	NULL	...

Figure 2.13. Variants of table *PRODREF*

optimization structures

Another objective of semantic schema enrichment is to detect optimization structures. The five foreign keys between table *KEYW* and table *DOCUMENT* are a typical example for such an optimization structure. A similar structure is described by Premerlani and Blaha [PB94]. Conceptually, it represents a *many-to-many* relationship between keywords and documents. Such a relationship is normally implemented as a simple join-table. However, in this case, the developer implemented it by a number of (five) foreign keys borrowed by the keyword table. The role of column *seqn* is to enable a carry over in case that one keyword is associated to more than five documents. This means that more than five references are represented by additional rows with the same keyword in column *keyw* but increasing values in column *seqn*. Again, there are various possibilities to detect optimization structures, e.g., naming conventions in the schema, characteristic procedural access code (code segment 4), and special value combinations in the available data.

code segment 4

artificial keys

It is common practice to introduce additional key columns when a relational data model is implemented. Often, sequence numbers are used for these columns, because they provide a simpler notion of identity than composite keys which carry real application data. Joins among tables are generally more efficient using such *artificial keys*. Hence, they can be considered as another kind of optimization structure. When an LDB is reengineered such artificial implementation structures have to be identified, as they should be suppressed in the recovered conceptual schema. In case of PDIS, the reengineer recognizes column *usrid* of table *USER* as an artificial key because users are conceptually identified by their short name (*sname*).

aggregations

Foreign key:	KEYW(doc1)...KEYW(doc5) -> DOCUMENT(docno)
Key:	KEYW(keyw,seqn) *(optimized many-many relationship)*
Key:	USER(usrid) *(artificial key)*
Complex:	telephone(USER(telo,telp))

Figure 2.14. Detected optimization and aggregation structures

According to the first normal form [BCN92], the relational data model does allow for atomic values in columns of tables, only. This implies that if a complex (aggregated) object structure has to be stored in a relational DB, it has to be decomposed into relations over atomic values. When a legacy relational DB is reengineered, knowledge about these aggregate relationships is important to recover its conceptual design. In our sample scenario, the reengineer annotates the information that column *telo* and *telp* of table *USER* conceptually represent a complex attribute that maintains different telephone numbers of users. More complex examples of detected aggregate relationships are described by Soutou [Sou98] and Vossen [FV95]. The annotations of our sample schema detail according to the detected optimization and aggregation structures is shown in Figure 2.14.

cardinality constraints

In general, a foreign key implements a *many-to-one* relationship between two tables. However, this cardinality information might be defined more precisely by investigating further relational constraints. Figure 2.15 gives an overview on the implication of such relational constraints on the cardinality range of a relationship implemented as a foreign key $A(a_1...a_n)\text{->}B(b_1...b_n)$. Some of these relational constraints are already included in the results of previous analysis activities (e.g., key or not-NULL constraints), while others have not been investigated, previously. For example, by analyzing the data a reengineer can try to find out if a given foreign

key also entails an *inclusion dependency* (IND) [EN94] in the reverse direction.[a] In this case, the minimum lower bound of the left side of the corresponding relationship is one, i.e., the relationship is *left-total* (cf. last row of Figure 2.15).

Cardinality range of represented relationship: $[x_l,x_u] : [y_l,y_u]$

relational constraint	$Min(x_l)$	$Max(x_u)$	$Min(y_l)$	$Max(y_u)$
Foreign Key: $A(a_1...a_n)$->$B(b_1...b_n)$	0	N	0	1
Not NULL: $A(a_1...a_n)$	0	N	1	1
Key: $A(a_1...a_n)$	0	1	1	1
IND: $B[b_1...b_n] \subseteq A[a_1...a_n]$	1	N	0	1

Figure 2.15. Implication of relational constraints on the cardinality of relationships

problems of scale: completeness and consistency

The primary goal of the presented sample scenario is to characterize the involved activities. Hence, it is not intended to be complete but it describes the most important steps of schema analysis for relational LDBs. There are many other analysis activities and methods dealing with various data models (cf. Section 2.5). For obvious reasons, our sample scenario covers only a small detail of the real case study. Figure 2.16 summarizes the results of the legacy schema completion and enrichment activity for this detail. The real system consists of 85 relational tables, 347 attributes, 111 foreign keys, several hundred thousand lines of procedural code and a huge database extension. As a consequence of this scale, performing the described process manually becomes a time-consuming, tedious, and error-prone task. The reengineer is likely to overlook some indicators for important semantic information and it is difficult to keep the resulting semantic information consistent.

iterative process

Because of the reasons described above, it is idealistic to presume a strictly phase-oriented, waterfall-type DBRE process. In practice, reengineers often start abstraction and forward engineering activities based on analysis information about the logical schema which still might be incomplete or inconsistent. During these activities reengineers accumulate additional knowledge about abstract concepts of the LDB. With this understanding, they often go back in the RE process and make some further investigation to refute or add some analysis results. In order to reflect on this experience in our sample scenario, we assume that in an initial analysis of PDIS the reengineer has not noticed the different variants of table *PRODREF* and the alternative key of table *USER*. The resulting incomplete analysis information is shown in black color in Figure 2.16.

a This special kind of inclusion dependency is called *C-IND* by Vossen and Fahrner [FV95].

Variant 1 (product reference):
PRODREF(id,pg,prod,cg,doc)

Variant 2 (product group reference):
PRODREF(id,pg,cg,doc)

Variant 3 (commodity group reference):
PRODREF(id,cg,doc)

Variant 4 (placeholder for document reference):
PRODREF(id,doc)

Complex: telephone(USER(telo,telp))

Key: COMGRP(cgid)

Key: PRODGRP(cg,pg)

Key: PRODGRP(manager)
(alternative key)

Key: DOCUMENT(docno)

Key: PRODUCT(no,pg,cg)

Key: USER(usrid)
(artificial key)

Key: USER(sname)
(alternative key)

Key: PRODREF(id,doc)

Key: DOCREF(id,sdoc)

Key: KEYW(keyw,seqn)

Foreign key: PRODUCT(cg,pg) -> PRODGRP(cg,pg)
(cardinality range [1,N]:[1,1])

Foreign key: DOCUMENT(usr) -> USER(usrid)
(cardinality range [0,N]:[1,1])

Foreign key: PRODGRP(cg) -> COMGRP(cgid)
(cardinality range [1,N]:[1,1])

Foreign key: DOCREF(sdoc), DOCREF(tdoc) -> DOCUMENT(docno)
(cardinality range [0,N]:[1,1])

Foreign key: KEYW(doc1)...KEYW(doc5) -> DOCUMENT(docno)
(optimized many-many relationship with cardinality range [1,N]:[1,N])

Foreign key: PRODREF(prod,pg,cg) (var. 1) -> PRODUCT(no,pg,cg)
(cardinality range [0,N]:[1,1])

Foreign key: PRODREF(pg,cg) (var. 2) -> PRODGRP(pg,cg)
(cardinality range [0,N]:[1,1])

Foreign key: PRODREF(cg) (var. 3) -> COMGRP(cg)
(cardinality range [0,N]:[1,1])

Foreign key: DOCREF(id,sdoc) -> PRODREF(id,doc) (var. 4)
(classified as IS-A relationship with cardinality range [1,1]:[1,1])

Foreign key: PRODGRP(manager) -> USER(sname)
cardinality range [0,1]:[1,1])

Figure 2.16. Summary of analysis results

Observations

With the presented sample scenario we can observe a number of typical characteristics about the activity of analyzing legacy schemas. They are summarized in the following statements:

Legacy schema analysis

O1. *involves heuristics and imprecise facts, i.e., it deals with uncertain knowledge;*

O2. *deals with idiosyncratic coding concepts and optimization structures;*

O3. *involves heuristics with credibilities that depend on technical and non-technical parameters of the LDB (e.g., used hard/software platforms and personal programming style or naming conventions, respectively);*

O4. *combines contradicting indicators and assumptions from various information sources, which may result in contradicting analysis results;*

O5. *deals with incomplete information and non-monotonic reasoning processes, i.e., new analysis results might refute initial hypotheses;*

O6. *is a human-intensive process that can be supported by semi-automatic analysis operations; and*

O7. *produces abstract information about the LDB by aggregating and classifying legacy schema components.*

2.4.2 Conceptual schema migration and redesign

conceptual migration

In this activity, the reengineer uses his/her domain knowledge and the analysis results about the logical schema to produce a corresponding conceptual schema. This is a creative design task because, in general, there are many possible conceptual models for one single logical schema. For the selected detail of our case study, the reengineer designs the conceptual schema presented in Figure 2.17 as a UML [RJB99] object model. It contains classes for the central entities of our schema detail. Associations have been created mostly according to the detected foreign keys with their annotated cardinality information. However, the conceptual schema abstracts from optimization structures and artificial keys. The two tables *PRODREF* and *DOCREF* are conceptually represented as ordered *many-to-many* associations, where the order is determined by their columns named *id*. Furthermore, the reengineer has decided to represent a user's department as a separate class.[a]

conceptual redesign

After creating an up-to-date conceptual schema for PDIS, the reengineer makes the necessary modifications and extensions to meet the new requirements for MIS. Figure 2.18 shows such an extended conceptual schema for our sample scenario. According to the requirement to maintain documents on-line, the reengineer has introduced two specializations of class *Document*, namely *OfflineDocument* and *OnlineDoccument*. The qualified association (*master*) among these new classes specifies that each on-line document must have exactly one archived master document. On the other hand, an off-line document might also be available on-line in different formats. Moreover, Figure 2.18 contains two new classes (*Employee* and *Customer*) and some new attributes to represent the different roles of users in MIS. Finally, the reengineer refined some cardinality constraints in the extended conceptual schema.

iteration

By discussing the designed conceptual schema with developers of MIS and users of PDIS, the reengineer learns that PDIS maintains not only cross references from documents to products and other documents, but also from documents to product groups and commodity groups. Therefore, (s)he returns to the analysis phase to investigate this indication. During this investigation, (s)he detects the different variants of table *PRODREF*, the alternative key of table *USER*, and the missing foreign key between *PRODGRP* and *USER* (cf. Figure 2.16).

a Class *Department* represents a so-called *weak entity* [BCN92].

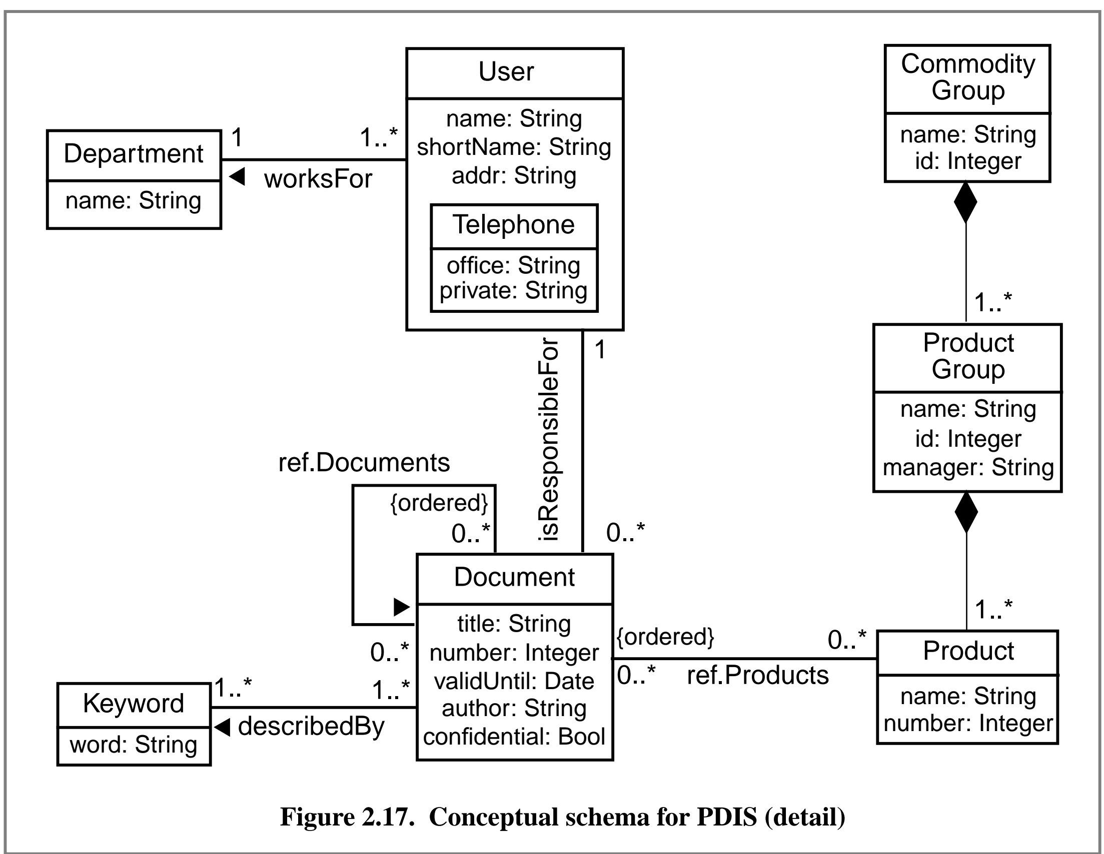


Figure 2.17. Conceptual schema for PDIS (detail)

update of conceptual schema

After modifying the analysis results about the logical PDIS schema, consistency with the redesigned conceptual schema has been lost. In order to re-establish consistency, the reengineer has to trace the impact of the new analysis results on the conceptual schema and to perform necessary changes and extensions. Figure 2.19 shows a detail of the conceptual schema for MIS that has been updated according to the current analysis results. Due to the common unique numbering of cross references (cf. Section 2.4.1), it is no longer correct to represent document and product references as distinct *many-to-many* associations. Hence, the reengineer introduces a new abstract class *XRef* as a generalization of all types of references. Moreover, the newly detected foreign key implies a partial *one-to-one* association between classes *ProductGroup* and *Employee*. However, the former attribute *manager* of class *ProductGroup* had to be deleted because its relational counterpart has been borrowed from table *USER* to implement this association.

problems of scale: correctness and consistency

For larger examples, the design of a correct conceptual schema for a given legacy schema is a complex task that is prone to error. Modifications of the LDB and iterations between schema migration and reverse engineering activities often result in inconsistencies between the logical LDB schema and the corresponding conceptual schema. Resolving these inconsistencies early is crucial for the success of any DBRE project, as the conceptual schema is the basis for many subsequent migration and forward engineering activities. However, the complexity of real-world systems makes it difficult to keep track of the impact of changing information about the LDB on the designed conceptual schema, manually.

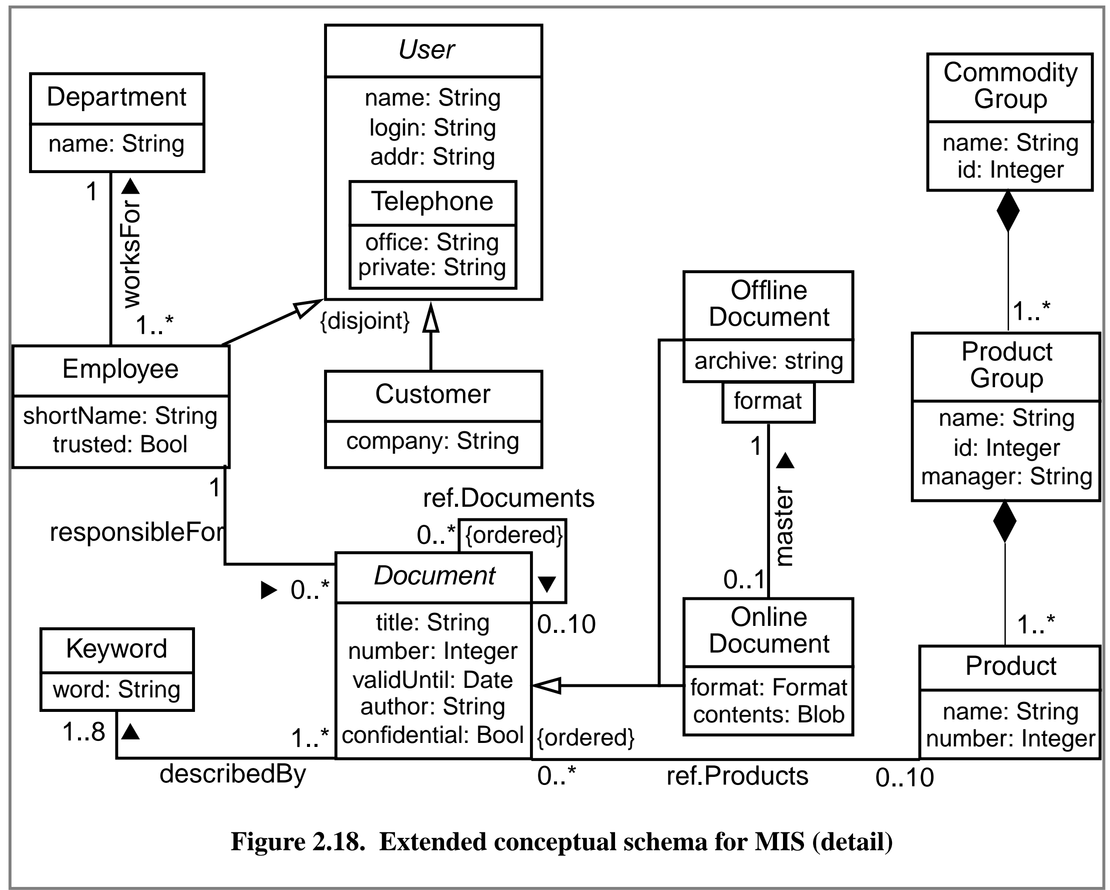


Figure 2.18. Extended conceptual schema for MIS (detail)

2.4.3 Implementation of changes and a middleware for data integration

The next step in our RE process is to implement the extended conceptual schema for MIS in the relational DB. According to the requirements defined in Section 2.2.1, the necessary schema modifications should be performed in a way such that the legacy PDIS will still be able to access the stored data (with no or only minor changes to its application code). Figure 2.20 shows that the conceptual changes can be implemented in a canonical way for our sample scenario. The new classes *OnlineDocument* and *OfflineDocument* are implemented as extensions of the existing table *DOCUMENT*. This solution has been chosen because the current legacy data about documents actually represents *off-line* documents which are available at the company head quarter (*"HQ"*). Hence, the data in table *DOCUMENT* does not have to be reorganized and the legacy PDIS application code can still be used to access the document data. Similar compatibility reasons have driven the decision to implement the new subclasses *Employee* and *Customer* as two variants in the existing table *USER*. (All users which are stored by PDIS so far are in fact employees, namely the members of the telephone hotline service.) The additional columns in tables *DOCUMENT* and *USER* are added using the SQL modification command *alter table* and providing a default value. The new association *master* is implemented as a foreign key which implies a referential integrity constraint.

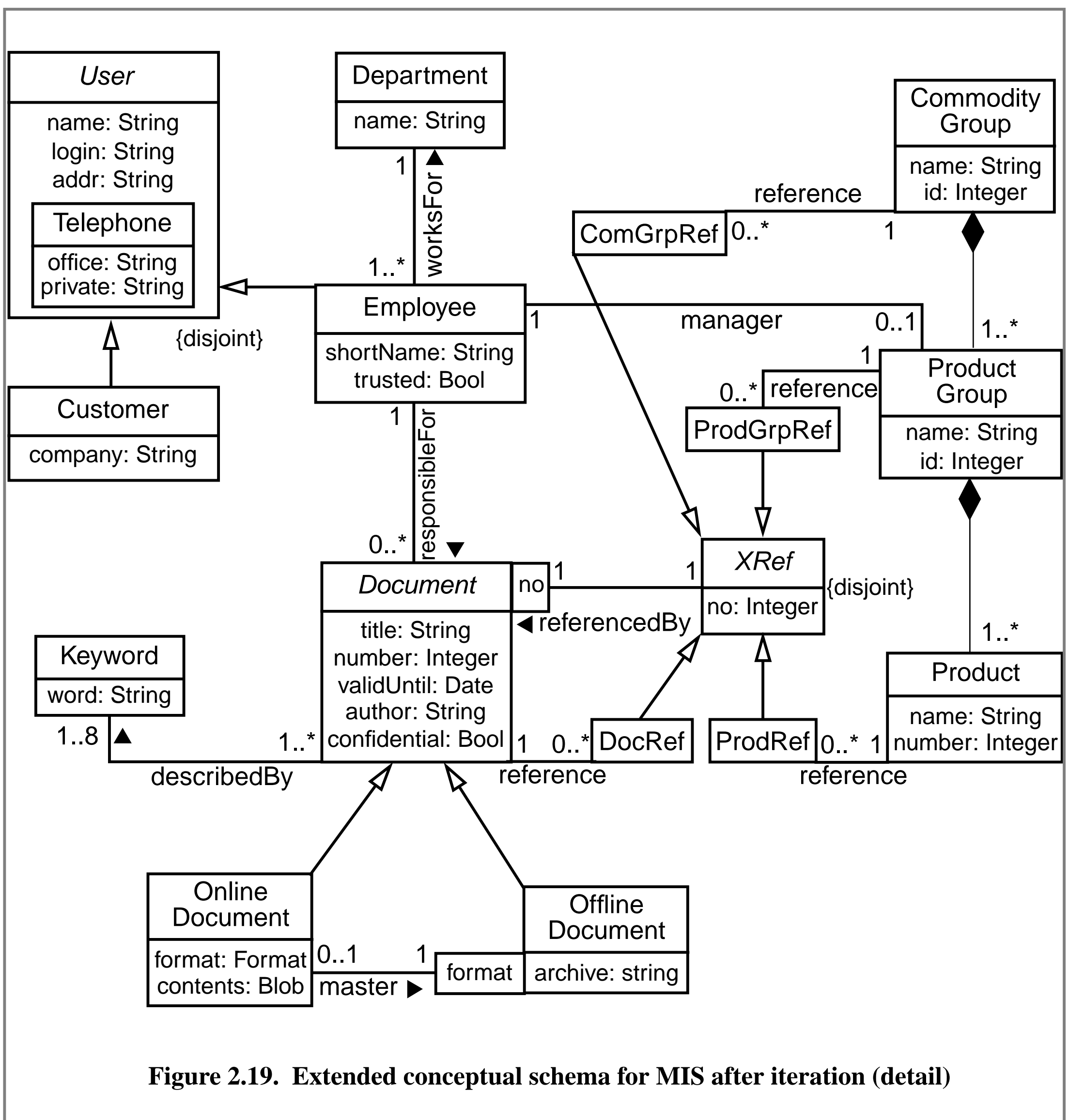


Figure 2.19. Extended conceptual schema for MIS after iteration (detail)

implementation alternatives

There are other possible implementations of the extended conceptual schema, e.g., new tables for each subclass, column replication, etc. Selecting one alternative is mostly a trade-off between minimized redundancy (well-defined schema structure) and efficiency. A comprehensive overview on relational implementation alternatives of conceptual structures is given by Fussell [Fus97]. In particular, in the DBRE domain, reengineers often have to compromise between well-designed schema modifications and the need for compatibility with legacy applications in order to enable gradual migration of legacy systems.

implementation of class* OfflineDocument*:
alter table DOCUMENT add column
archive: VARCHAR(80) with default "HQ";
(legacy documents are OfflineDocuments)

implementation of class* OnlineDocument*:
alter table DOCUMENT add column
format: INTEGER with default NULL;
alter table DOCUMENT add column
contents: BLOB with default NULL;
alter table DOCUMENT add column
master: INTEGER with default NULL;

implementation of association* master*:
foreign key DOCUMENT(master) references
DOCUMENT(docno) on delete cascade;

additional attribute login of class* User*:
alter table USER add column
login: CHAR(12) with default NULL;

implementation of class* Employee*:
alter table USER add column
trusted: BOOLEAN with default FALSE;
(legacy user data represents Employees)

implementation of class* Customer*:
alter table USER add column
company: VARCHAR(80) with default NULL;

Figure 2.20. Implemented extensions of the logical schema

MIS architecture and rationales

In order to exploit the benefits of the conceptual schema migration the IT department decided to employ the object-oriented paradigm [JEJ95] to develop MIS. However, this implied that the developers may not use one of the various available DB Web-gateways [EKR97] or DB access libraries (e.g., JDBC) in their application code. These solutions deal with direct textual queries to the legacy schema. Using them in the application code would violate the most important principle of object-oriented systems, namely *encapsulation*. As a consequence, every change in the legacy schema would entail changes to the MIS application code. Therefore, the developers decided to implement an object-relational middleware layer that hides the concrete representation of objects in the DB from the application code. The objective is to increase the maintainability of the new MIS by designing a middleware API that is compliant to the *Java* language binding specified in the *Object Database Management Group* (ODMG) *s*tandard [CBB+97]. Figure 2.21 shows that the planned object-oriented API is *internally* based in the JDBC-gateway for DB2.

middleware design

An outline for the design of the desired middleware is given in Figure 2.22. Of course, there are other possible designs that realize an ODMG compliant API, e.g., the *Java* mechanisms of class and interface inheritance could be used differently. Still, all possible designs will have certain classes which are specific to the application (MIS) and other classes, which are application independent (generic). The shaded outer part of Figure 2.22 contains examples for generic ODMG classes, including a common root class for all persistent objects (*ODMGObject*), a transaction manager (*Transaction*), and a class to create and look up named database objects (*Database*).

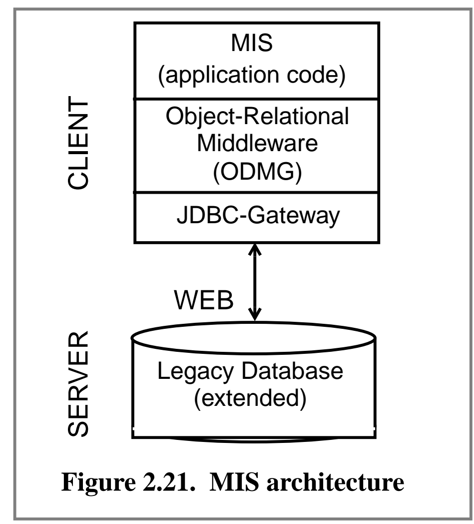


Figure 2.21. MIS architecture

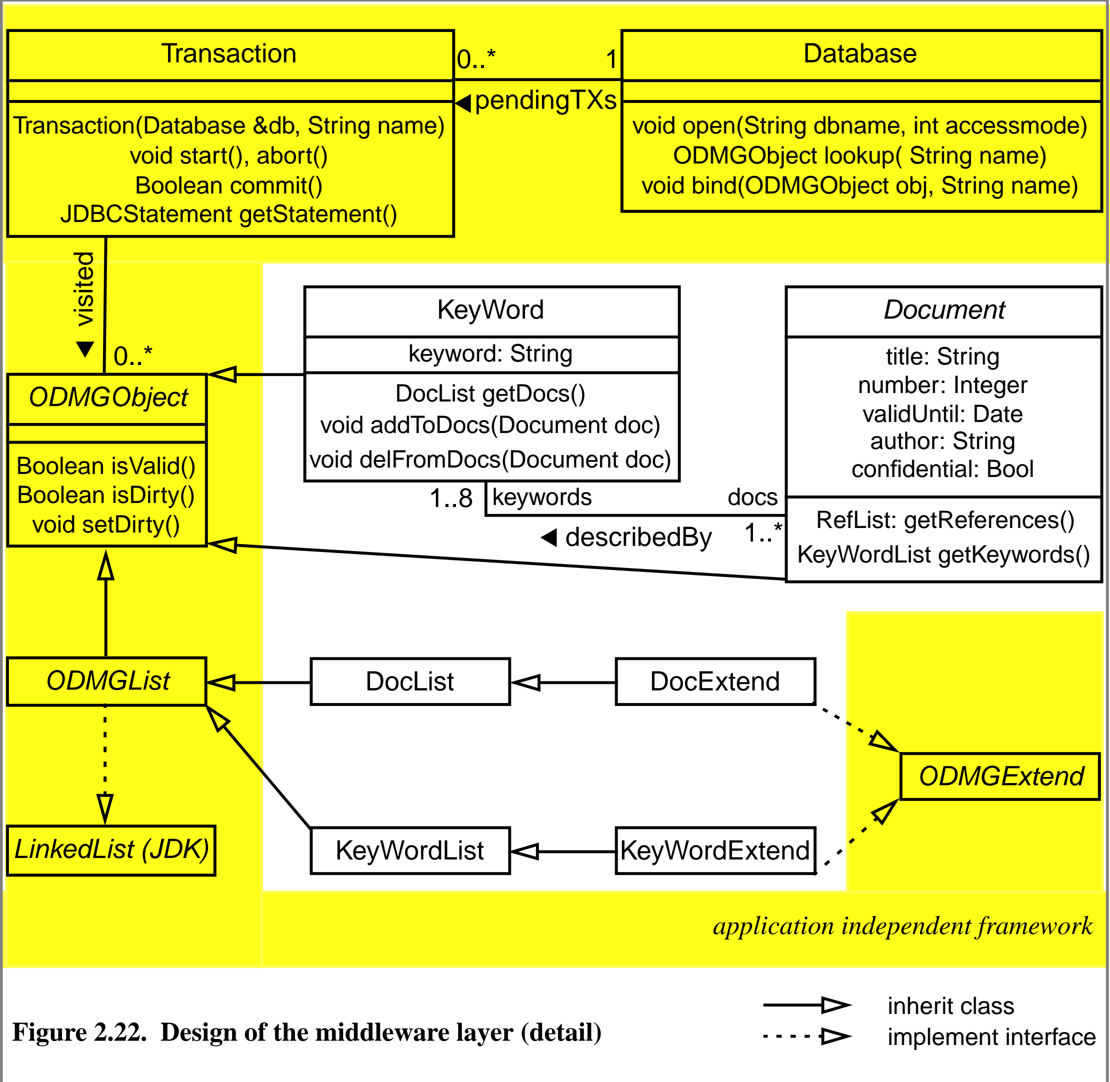


Figure 2.22. Design of the middleware layer (detail)

The inner (unshaded) part of Figure 2.22 exemplifies that application specific classes are not only implemented for each class in the conceptual schema, but also for collections of their instances and their entire extends, respectively. The class members of application specific API classes are mainly designed in a canonical way by adding *read* and *write* accessor methods for each class attribute and association.[a] Using these methods to traverse an association automatically triggers the translation of relational data to *Java*-objects in the run-time cache of the current transaction. Hence, the implementation details of the LDB are completely hidden from the user of the middleware. Of course, application specific middleware classes may also contain additional methods that encapsulate more sophisticated queries to the legacy database (LDB).

a For the sake of simplicity, we did not list all methods and attributes of the classes depicted in Figure 2.22.

Observations

This case study led to the following observations:

O8. *Conceptual abstraction (and redesign) of a logical schema is a creative (re-)design task; there are many alternative conceptual schemas.*

O9. *Conceptual translation of complex schemas is error-prone; due to the semantic gap between conceptual and logical data models it is often non-trivial to decide whether a created conceptual schema is correct, i.e., semantically equivalent to the implemented data structure.*

O10. *Increasing conceptual knowledge about LDBs often cause iterations with further analysis activities. Other causes for iterations are on-the-fly modifications of the LDB schema during an ongoing RE project.*

O11. *Iterations cause inconsistencies between the logical and conceptual schema which are often difficult to detect and resolve.*

O12. *Modifications to the original schema can be performed canonically according to the redesigned conceptual schema.*

2.5 Summary and concluding remarks

relevance of the scenario

The case study presented in this chapter describes a typical example for current industrial DBRE projects. The emerging requirement to compete on a global electronic market has become one of the major driving forces to integrate existing LDBs with Web-based technologies [LS98b]. An increasing number of companies seek a strategic business advantage in establishing Web-based information systems. Lederer et al. give an overview on the benefits expected from this technology [LMS98]. There are various reports about similar projects of reengineering LDBs to the Web. Many of them deal with data stored in relational DBMS, e.g., Umar [Uma97, pp. 461-464], Fryer [Fry95], and Simpson [Sim94] report on the integration of DB2-based mainframe applications with the Web. Other case studies deal with different data models, e.g., the hierarchical data model (IMS[a]) [Uma97, pp. 464-468] and the network data model [FH97, p. 227]. For some scenarios it is not necessary to make the transition to a fully object-oriented API. In these cases, HTML[b] language extensions or Web-gateways can be used to integrate LDBs with Web services. However, these solutions are insufficient for companies which aim on encapsulating and integrating various heterogeneous DBs to achieve enterprise-wide IS infrastructures.

migration strategy

Like the migration strategy described in our scenario, many approaches make use of the fact that LDBs are *data-decomposable* [Uma97], i.e., they maintain their data in some kind of DBMS which can be integrated with the new technology. Similar strategies can also be applied to migrate other components of a legacy system, e.g., its user interface [BS95]. Recently, this idea of decomposing legacy systems in order to reuse certain components and substitute or enhance others has been described in general in terms of the so-called *divide-and-modernize* reengineering pattern [SP98].

---

a Information Management System - hierarchical DBMS on mainframes (IBM)

b HyperText Markup Language [Bar94]

In our scenario, we only describe the integration of one *single* LDB with new technologies. Additional problems arise if we also consider the integration of *several* autonomous LDB, e.g., the problem of mediating among different component schemas. This issue is tackled by Lincke and Schmid [LS98a], again, by using the example of electronic product catalogs.

DBRE process

The process described in our case study covers the major aspects of typical DBRE activities in the area of relational systems. For each of these activities we pointed out a number of important observations. A more general discussion about DBRE processes has been presented by Hainaut et al. [HCTJ93]. However, even if we consider other data models and architectures our observations are still appropriate, as they reflect on inherent characteristics of the DBRE domain [ALV93, Big90].

role of the scenario

Analogously to our observations, most techniques presented in this thesis are not restricted to the described scenario only, i.e., the integration of relational LDBs with object-oriented, Web-based technologies. The primary purpose of the described scenario is rather to motivate these techniques and to provide a coherent application example for their evaluation. We will refer to the elicited observations in the following technical chapters to define the major requirements for DBRE tool support.

CHAPTER 3 A THEORY TO MANAGE IMPERFECT KNOWLEDGE

knowledge-based system

A major goal of this dissertation is to develop a formalism to specify and customize database reengineering (DBRE) knowledge and to implement mechanisms that allow to apply this knowledge in human-centered CARE environments. In principle, the desired system is similar to a *knowledge-based* or *expert system* [Kas96]. However, the term *expert system* has originally been introduced in the community of artificial intelligence (AI) to describe a computer program which imitates human experts [BC90]. In this sense, the desired DBRE environment can hardly be considered as an expert system. This is because its primary task is to support the reengineer by unburdening her/him from stereotypical and error-prone activities and focussing her/his attention on the parts of the LDB where human common-sense and intuition is required. Consequently, the reengineer will not be replaced but the goal of the desired DBRE environment is to employ her/his capabilities in a more efficient and effective way. Recently, the term *knowledge-based system* (KBS) has been used in a broader sense: the main characteristic of a KBS is that it consists of a formal description of domain knowledge, a fact base, and a separate component including a number of problem solving strategies to execute the knowledge [BL97].

In order to develop a knowledge-based tool to assist legacy schema analysis, it is crucial to characterize the problem domain of DBRE, carefully. With our case study, we have already made some important observations about the nature of DBRE processes and activities (cf. Section 2.4.1, on page 15). Other researchers and practitioners report on similar experiences with DBRE projects, e.g., [BP95, BRH95, PKBT94, And94, PB94, AMR94, Sne91]. It is the purpose of Section 3.1 to use these observations to derive central requirements on a formalism that is suitable to manage DBRE knowledge in human-centered CARE environments. Subsequently, we will review different theories of imperfect knowledge representation and reasoning, and evaluate their suitability for our specific application domain (Section 3.2). In Section 3.3, we compare the reviewed approaches based on the evaluation results. This comparison enables us to conclude which general theories and techniques are most suitable to be applied in the DBRE context.

3.1 Requirements on formalisms to manage DBRE knowledge

The requirements that have to be fulfilled by a formalism that is suitable to manage imperfect knowledge in human-centered DBRE environments cover different aspects. Some aspects like *clarity* and *maintainability* should generally be considered whenever a language for knowledge representation is developed. Still, other aspects depend on our specific class of problems. A good way to introduce these aspects is to look at a typical reference architecture for KBS in Figure 3.1 [Kas96].

A core component of each such systems is a *knowledge base* which contains situation-independent domain knowledge, that has explicitly been specified by experts and/or implicitly

been learned from sample situations. Some of the central questions are: *What kind of knowledge has to be represented? How can the knowledge be specified or adapted? How is the knowledge represented internally?*

Other core components of a KBS are the *inference engine* and the *fact base*. The inference engine is an implementation of a set of problem solving strategies. It interprets the knowledge represented in the knowledge base and applies it to the available data residing in the fact base in order to infer additional facts about the current situation. We have to cover questions like: *What kind of data has to be stored in the fact base? Where does this data come from? What is the right problem solving strategy?*

Finally, user interaction plays an important role in KBS. On one hand, the inferred data has to be communicated to the user (*user dialog*), and on the other hand, the system should be able to explain the way *how* this result has been obtained (*explanation*). Questions of interest include *How can facts adequately be presented to the user? What explanation or query functionality is required?*

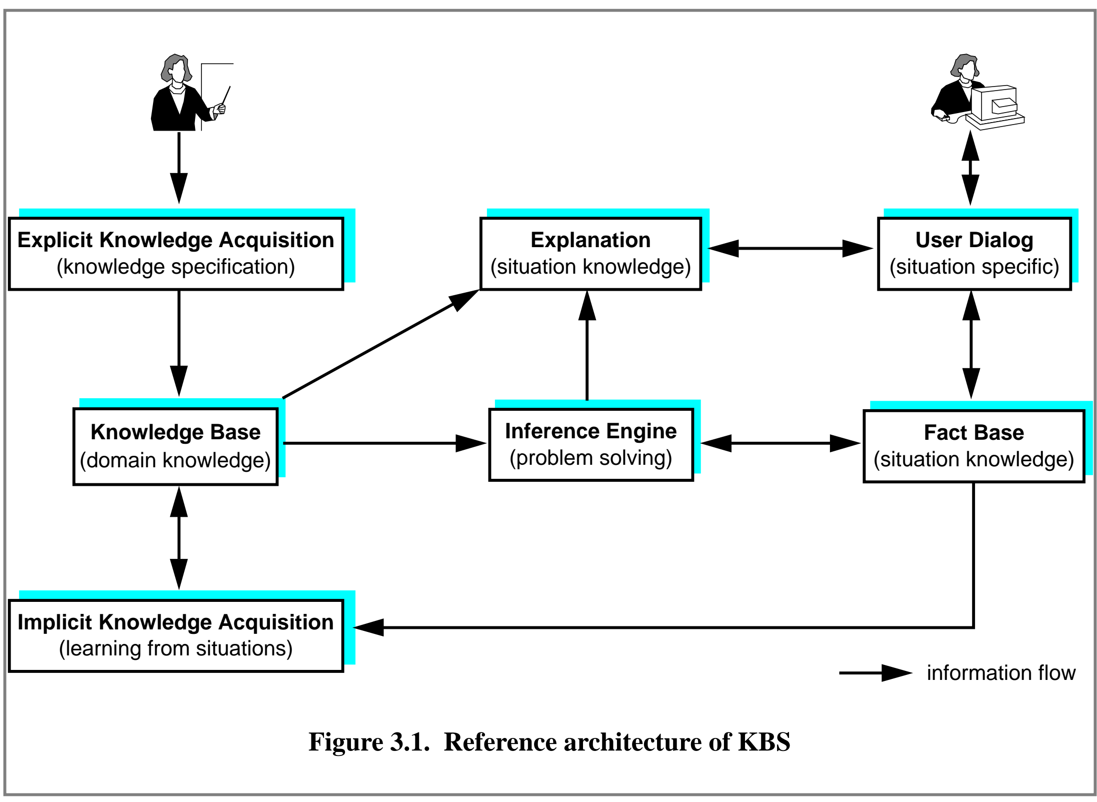


Figure 3.1. Reference architecture of KBS

In the following subsections, we will investigate the mentioned questions more thoroughly for the application domain of DBRE and elaborate central requirements that have to be fulfilled by a theory to manage DBRE knowledge.[a]

a We do not claim that the following requirements are *sufficient* for an approach to manage DBRE knowledge but they are *necessary* and allow to identify the most promising theory.

3.1.1 Quantitative representation of uncertainty

Our case study has demonstrated that DBRE expert knowledge includes various heuristics (cf. O1 in Section 2.4.1). In general, heuristics represent a form of *imperfect knowledge* about the real world under consideration. They are employed when it is not tractable to use definite knowledge, e.g., in the case when necessary information is unknown, or when it takes too much effort to use definite knowledge. The drawback of using heuristics is that they might not be valid in some situations, i.e., they might lead to unsatisfactory results.

quantitative vs. qualitative approaches

The approaches which have been introduced to formalize uncertain knowledge can be distinguished in two major categories, namely *quantitative* and *qualitative* approaches [Hül96]. In quantitative approaches, each piece of knowledge has an associated *valuation* which is represented by a real number in a closed interval. The concrete semantics of this valuation depends on the underlying theoretical framework of each approach, e.g., probability theory. However, the different valuations in all quantitative approaches have in common that they define a measure for the degree of validity of the corresponding pieces of knowledge. When a new piece of knowledge p is derived from a combination of existing pieces of knowledge $\{p_1,..,p_n\}$, the valuation for p is computed by combining the valuations of $\{p_1,..,p_n\}$.

Many critics have argued that real numbers are not adequate to represent the uncertainty of human knowledge [Her94, Sch92]. They point out that common sense reasoning has a qualitative, rather than numerical nature. Hence, qualitative approaches to uncertain knowledge representation allow to make propositions like "*p is likely*" or "*p is possible*". In principle, qualitative approaches are specializations of quantitative approaches with a small, finite domain of possible values [BC90]. Some qualitative approaches use an internal knowledge representation based on real numbers to represent uncertainty. In these cases, it is often possible to specify or obtain a quantitative measure for the certainty of a piece of knowledge, which is in contrast to "purely" qualitative approaches.

In order to decide whether a qualitative or a quantitative approach is more suitable for the DBRE domain, we have to consider the primary purpose of our application. In the introduction to this chapter, we stated that a central functionality of the projected DBRE environment is to direct the reengineer‘s attention to the most controversial parts of the legacy system. Hence, we can classify our application as a *selection problem*. Purely qualitative approaches are less suitable for these kinds of problems, as their small domain of possible truth-values is less selective than real numbers. For example, it is less informative for a reengineer to know that a given foreign-key might *possibly* represent *either* an association *or* an inheritance relationship, than to know that the confidence of the association is measured to 0.2, whereas the inheritance relationship has a confidence measure of 0.7 (in a valuation interval from 0 to 1). As a consequence of this discussion, we impose the first requirement on the desired formalism.

R1. *A formalism to specify DBRE expert knowledge has to allow for a quantitative representation of uncertain domain and situation-specific knowledge.*

3.1.2 Representation and indication of contradicting knowledge

In Section 2.4.1, we have exemplified that in order to recover an up-to-date documentation about an LDB the reengineer has to combine indicators and assumptions that stem from various sources. Furthermore, we have observed that this information is likely to be (partly) contradicting (cf. Observation O4). In general, dealing with contradicting domain and situation specific knowledge seems to be inevitable in our application domain. On one hand, contradicting domain specific knowledge is often introduced by acquiring heuristics from different DBRE experts. On the other hand, contradictions in situation specific knowledge might be injected, e.g., by combining the output of different automatic software analysis procedures [MNL96, AT98] or by considering diverse human (situation-specific) assumptions about the LDB. This leads to the following requirement:

R2. *A formalism to manage DBRE knowledge has to allow for the representation of contradicting knowledge with domain-specific and situation-specific character.*

Still, *tolerating* contradicting knowledge is not enough, because a major goal of the DBRE process is to produce a documentation of the LDB that has to be *consistent*, eventually. Hence, a central task of the reengineer is to find and resolve all contradictions in the situation-specific knowledge. A knowledge-based DBRE environment should support this process by indicating contradicting knowledge to the user.

R3. *A formalism to manage DBRE knowledge has to be able to indicate contradicting situation-specific knowledge.*

3.1.3 Reasoning about incomplete knowledge

Traditionally, many approaches to knowledge representation and reasoning make the so-called *closed world assumption*, which entails that *all relevant* information is known before the reasoning process starts. This kind of reasoning process is called *monotonic* because no additional information that might become available later can lead to the falsification of a conclusion. From the DBRE case study, we learn that we have to give up the *closed world assumption* for our application domain (cf. Observation O5, on page 24). On the contrary, we have shown that reengineers generally begin their reasoning process with *incomplete* information in terms of initial assumptions and analysis results. This information might lead to intermediate hypotheses which might be refuted or supported as soon as new information becomes available from further investigations. Consequently, we have to make an *open world assumption*, which involves a *non-monotonic* reasoning process.

R4. *A formalism to manage DBRE knowledge has to be able to deal with incomplete knowledge in a non-monotonic reasoning process.*

3.1.4 Representation of ignorance

DBRE uses heuristic knowledge in combination with positive and negative indicators and human assumptions to infer new (uncertain) situation specific knowledge. For example, in our case study, we show that an instance of a *cyclic-join* pattern over a given attribute x serves as a positive indicator for the fact that x is a key, whereas an instance of a *select-distinct* pattern over x represents an indicator against this assumption (cf. page 18). However, in the absence of

any such indication nothing is known about whether or not attribute *x* might be a key. This state of (partial) ignorance cannot be described with statements like "*x is not a key*" or "*x is a key with 50% chance*". Thus, we require the following criteria.

R5. *A formalism to manage DBRE knowledge has to be able to represent partial ignorance about situation-specific knowledge.*

3.1.5 Computational tractability

The criteria discussed above reflect on qualitative properties of the desired formalism for knowledge management. However, we are interested in selecting this formalism in order to solve a particular class of problems in DBRE. Even if we employ a knowledge representation that satisfies all of the above requirements but cannot be executed on a computer with the efficiency that is necessary for practical applications we have not solved the problem. In the DBRE domain, we have to deal with a large amount of information in terms of several hundred tables, millions of lines of code, and a vast amount of business data. It is crucial to find a solution that scales up to practical applications. Therefore, we need to take into account another criterion:

R6. *A formalism to manage DBRE knowledge should scale up to practical applications.*

3.2 Evaluation of theories

This section contains a survey of major approaches to manage imperfect knowledge, namely *production systems with confidence factors* (Section 3.2.1), *probabilistic reasoning* (Section 3.2.2), *credibilistic reasoning* (Section 3.2.3), *fuzzy reasoning* (Section 3.2.4), and *possibilistic reasoning* (Section 3.2.5). We use the requirements that have been elaborated in the previous Section 3.1 to evaluate each approach according to its suitability for the application domain of DBRE. We concentrate only on quantitative (or hybrid) approaches because of our first requirement. Even though purely qualitative formalisms (e.g., *modal logic* [Lem77, Gär75, HR87], *default logic* [MT93, Poo88], and *multi-valued logic* [BB92]) are not suitable for our particular purpose they have proven useful in many other application contexts. An interesting comparison of modal and many-valued logic with most quantitative approaches evaluated in this dissertation has been presented by Hajek [Haj94].

Notation and basic definitions

Before we start our discussion of the different approaches in Section 3.2.1, we need a more formal notion of a (relational) DB. Furthermore, we define some notational conventions that are used throughout this dissertation.

Definition 3.1 Data model

A ***data model*** *is a tuple M:= (C, O), where C is a set of* ***concepts*** *that are used to describe the structure of data and O is a set of* ***operations*** *to handle the data represented by elements of C.*

Definition 3.2 Database

*A **database** is a tuple $DB:=(M,S,\delta(S),C,D)$, where M is a data model, S is a data structure that is represented by concepts of M (S is also called **schema**), $\delta(S)$ is the extension of S (also called **data**), C is an application program that uses operations of M (C is often represented by its source **code**), and D is the **documentation** of DB.*

Definition 3.3 Relational database

*A **relational database** is a database $RDB:=(M,S,\delta(S),C,D)$, where M is the **relational algebra**; $S=(R, \Delta)$ is a **relational database schema** where $R=\{r_1,...,r_n\}$, $n\in \mathbb{N}$, is a finite set of **relation schemas** (RS); and Δ is a set of **inclusion dependencies** (INDs). Each $r\in R$ is a tuple $r=(X, \Sigma, \Theta)$, where X is a finite set of **attributes**; Σ is a finite set of **key dependencies**; and Θ is a finite set of **not-null constraints**.*

Definition 3.4 (Notation)

- *$\mathcal{U}$ denotes the **universe** of discourse.*
- *$\overline{SET}$ denotes the infinite set of all **sets**.*
- *$\overline{REL}\subset\overline{SET}$ denotes the infinite set of all **relations**.*
- *$\overline{LIST}$ denotes the infinite set of all **lists**.*
- *We extend the applicability of the set **operators** $\{\in,\subseteq,\subset,\supseteq,\supset\}$ on lists, e.g., for a list $<e_1,..,e_n>\in\overline{LIST}$ we define $e\in<e_1,..,e_n>\Leftrightarrow e\in\{e_1,..,e_n\}$, etc.*
- *$\mathcal{P}(S)\in\overline{SET}$ denotes the **power set** of a given $S\in\overline{SET}$.*
- *For a given set S, let $|S|$ denote the **cardinality** of S.*
- *$\overline{FUN}$ denotes the infinite set of all **functions**.*
- *For a given function $f\in\overline{FUN}$ and an argument $a\in\mathcal{U}$ we denote $def(f(a))$ iff f is defined for a.*
- *L^0, L^1 denote the language of **propositional logic** and **first-order logic** [Rog71], respectively.*
- *$\mathcal{L}\{L\}$ denotes the infinite set of all valid expressions in a given language L.*
- *Let $\models$ denote the **tautology** and let $\not\models$ denote the **contradiction**, i.e., the logical formula that is always and never fulfilled, respectively.*
- *Let $\overline{RDB}$ denote the infinite set of all possible relational databases.*

Definition 3.5 Flattening

*We define a function that transitively **flattens** nested sets and lists, i.e., $flatten:\overline{SET}\cup\overline{LIST}\to\overline{SET}$, with*

$$flatten(S) = \left\{ x | (s \in S) \wedge \begin{array}{ll} x \in flatten(s) & \text{if } def(flatten(s)) \\ x = s & \text{else} \end{array} \right.$$

3.2.1 Production systems with confidence factors

An early approach to represent and reason about uncertain expert knowledge has been proposed by Shortliffe et al. [Sho74, BS84]. It has been implemented in the well-known expert system called *MYCIN* and applied to problems of medical diagnosis. In Shortliffe's approach,

propositions and implication rules are associated with a *measure of belief* (MB) and a *measure of disbelief* (MD), both being numbers between *0* and *1*. These measures are then combined into a single number called *certainty factor* (CF). The CF of a proposition $u \in \mathcal{U}$ is computed as $CF(u) := MB(u) - MD(u)$, $CF(u) \in [-1,1]$. The general form of an implication rule with CF is

$$\textbf{IF } u_1 \textbf{ THEN } u_2 \text{ , with } CF = c$$

where $u_1 \in \mathcal{U}$ can be an arbitrary complex proposition and $u_2 \in \mathcal{U}$ has to be atomic. Confidences of complex propositions are calculated using the minimum operation for conjunctions and the maximum operation for disjunctions. A negation results in a change of the sign of the corresponding CF. This means for $u_3, u_4 \in \mathcal{U}$

$$CF(u_3 \wedge u_4) = min(CF(u_3), CF(u_4)) \quad (EQ\ 1)$$

$$CF(u_3 \vee u_4) = max(CF(u_3), CF(u_4)) \quad (EQ\ 2)$$

$$CF(\neg u_3) = -1 * CF(u_3) \quad (EQ\ 3)$$

The above equations can be used to determine the confidence for the entire antecedent of a rule. This confidence is then multiplied by the confidence factor of the corresponding rule itself to obtain the confidence for the conclusion. Generally, several rules can have the same conclusion. In this case, the confidences that result of each rule application have to be combined to obtain the new CF for the common conclusion. For the rules

$$R_1 \text{: } \textbf{IF } u_1 \textbf{ THEN } u \text{ , with } CF = c_1$$

$$R_2 \text{: } \textbf{IF } u_2 \textbf{ THEN } u \text{ , with } CF = c_2$$

let $CF(u|u_i)$ denote confidence for proposition u resulting from the application of rule R_i, with $CF(u|u_i) = c_i * CF(u)$. The combined confidence for the common conclusion u is then computed as

$$CF(u|u_1 \wedge u_2) = \begin{cases} CF(u|u_1) + CF(u|u_2) - CF(u|u_1)CF(u|u_2) & \text{for } CF(u|u_1), CF(u|u_2) > 0 \\ CF(u|u_1) + CF(u|u_2) + CF(u|u_1)CF(u|u_2) & \text{for } CF(u|u_1), CF(u|u_2) < 0 \\ \dfrac{CF(u|u_1) + CF(u|u_2)}{1 - min(|CF(u|u_1)|, |CF(u|u_2)|)} & else. \end{cases} \quad (EQ\ 4)$$

Evaluation

only monotonic reasoning (R4)

Many researchers have criticized the unclear semantics of the measures defined in MYCIN [Ada76, Joh86]. However, the most significant problem of Shortliffe's approach with respect to our application domain is the inability to represent incomplete knowledge and execute non-monotonic reasoning processes (Requirement R4, on page 36). All relevant knowledge has to be known before the inference starts. In the general case of cyclic rule bases, recursive rule applications cause problems with constantly growing confidences [BL97]. In contrast to the original application domain of MYCIN (medical diagnosis) where acyclic rule bases were sufficient to solve many practical problems, we need the general case of cyclic rule bases to support the desired incremental and evolutionary DBRE process. This means that in our particular application domain there is no strict separation of "symptoms" and "diagnoses" but the DBRE environment should enable the reengineer to add information on arbitrary levels of abstraction.

3.2.2 Probabilistic reasoning

Probabilistic logic extends classical logic [Rog71] with probability theory to reason about uncertain information [NH95, Nil93, Paa88b]. Analogously to Shortliffe's approach, knowledge representation in probabilistic logic usually involves the specification of weighted implication rules of the form

"***IF*** *A* ***THEN*** *B, with probability* π".

semantics

A and *B* represent propositions in L^1.[a] Given the semantics of an implication in classical logic "$\rightarrow$", the probability might be interpreted as the probability that the condition $(A \wedge B) \vee \neg A$ holds. However, this semantics has not been adopted by most approaches to probabilistic logic. The main argument against this semantics is that the probability of the above condition is of little meaning in the mental model of an expert who tries to model his/her domain knowledge [Paa88b, pp. 216]. Therefore, the probability associated with a rule is defined as a *conditional probability* of the consequent given the fulfillment of the antecedent, i.e.,

$$\pi = \frac{p(A \wedge B)}{p(A)} \qquad \text{(EQ 5)}$$

subjective probability

The traditional estimation of probabilities involves a large number of repetitions of a given situation. The estimated probability for an event is then based on the frequency of occurrences of this event divided by the total number of experiments performed. However, this frequency concept is not applicable in most applications of probabilistic expert systems, because it is rarely possible to observe a large number of identical experiments. For these cases, the theory of *subjective probabilities* [Ber80, pp. 61ff] has been created. A subjective probability reflects a human expert's personal belief in the chance that the corresponding proposition is true. Consequently, there might be different subjective probabilities for a single proposition which have been defined by different human experts depending on their personal experience and background.

The primary aim of probabilistic logic is to combine the probabilities provided by human experts in order to define and evaluate a joint *probability measure p* over the universe of all relevant propositions $\mathcal{U}=\{u_1,...,u_n\}$. This universe is defined by all propositions that occur in a probability (or conditional probability) specified for the resulting inference net.

The joint probability measure can then be defined as $p:(\mathcal{U}) \rightarrow [0,1]$

$$p(u_i) = \sum_{j \in J_i} p(\omega_j), \quad \text{where } u_i = \bigvee_{j \in J_i} \omega_j \qquad \text{(EQ 6)}$$

$\Omega=\{\omega_1,..,\omega_k\}$ represents the set of all interpretations, i.e., all possible worlds with respect to the problem of interest. These worlds have to be exclusive and exhaustive. Consequently, the fundamental axioms of *Kolmogorow* have to be fulfilled for the probability measure [Loe78].

---

a In order to keep this survey simple, we restrict this introduction to the propositional case, which is mostly used in probabilistic KBS. Approaches to define the semantics of probability-valued formulae in L^1 are described, e.g., by Halpern [Hal90] and Fenstad [Fen67].

Inference methods for probabilistic KBS usually employ *Bayesian inference networks* [HMW95, Pea98] to represent causal information. They are based on the *Bayesian formula* [Loe78] which is used to calculate the so-called *posterior probability* of a each interpretation $\omega_i \in \Omega$ for a given event $u_j \in \mathcal{U}$:

$$p(\omega_i | u_j) = \frac{p(\omega_i) p(u_j | \omega_i)}{\sum_{\omega \in \Omega} p(\omega) p(u_j | \omega)} \qquad (EQ\ 7)$$

Evaluation

limited support for contradiction (R3): error models

An often cited problem of probabilistic reasoning is that it is unrealistic to expect that a human expert is able to specify exact probabilities for axioms and implication rules. One approach to tackle the problem of contradicting probabilities is the specification of *error models* [Paa88b]. This solution entails that each judgement of an expert has to be assessed w.r.t. its certainty. An error model is represented by a conditional distribution $p(\pi_i/\bar{\pi}_i)$, where π_i denotes the subjective probability for rule i provided by an expert and $\bar{\pi}_i$ represents the "correct" probability estimate that would be given by a rational expert with complete information about all aspects of the problem. In the general case, error models are specified for the subjective beliefs of many rules. In this case, the *maximum-likelihood* approach can be employed to yield the *most probable* solution [Paa88b]. This optimization procedure resolves contradictions in such a way that for less reliable subjective probabilities the extent of the modification is largest.

Still, a major limitation of this approach is that the errors for different probabilities are assumed to be statistically independent. This is only reasonable if the experts use distinct sources of information and do not collaborate, which cannot be generally assumed in our application domain. Clemen and Winkler [CW85] show that dependent sources of information considerably reduce the precision of estimates. An inherent feature of using error models and the maximum-likelihood approach is that inconsistencies are automatically resolved during the inference process, i.e., probabilities of contradicting rules are adjusted to obtain consistency with the available data (cf. [Paa88b]). If the available data (situation-specific knowledge) is uncertain itself, error models can be used in the same way to specify this uncertainty. However, definite probability values are calculated for deduced situation-specific knowledge. This means that this approach does not allow to represent contradicting inference results explicitly, but it *adapts* the uncertain input knowledge such that the inference results are consistent.

no representation of ignorance (R5)

Furthermore, it is not possible to represent ignorance in probabilistic logic. This is because the state of knowledge where there is an equally lack of certainty about all events (including non-elementary ones) that are liable to occur cannot be expressed by a single probability measure [DP88 p. 287].

computational intractable for DBRE (R6)

Bayesian inference is typically employed with a number of severe structural restrictions, e.g., events (axioms) are required to be conditionally independent, conclusions have to be exclusive, the inference net has to be acyclic, prior probabilities are required for final and intermediate results, or the desired probability distribution is expected to belong to a restricted class of distributions (cf. [Paa88b]). The general problem of finding the posterior probability of a proposition in a Bayesian network is in NP [Coo90]. Some authors have proposed inference procedures that are less accurate and have a lower complexity for the average case, e.g.,

[Poo93]. Haber and Brown present an iterative algorithm for the general case of cyclic inference nets [HB86], while Pearl discusses simplified procedures with special interaction patterns [Pea86]. Some approaches employ optimization heuristics that yield approximate solutions with less computational effort [AdBHL86]. Given the fact that in our particular application domain (DBRE) we have to deal with cyclic inference networks and a vast amount of propositions, probabilistic inference seems to be computationally intractable for our purpose. Moreover, the effort that is spent in probabilistic reasoning in order to comply to the basic axioms of probability theory does not seem to be justifiable for our application. This is because the credibility of DBRE heuristics vary according to diverse technical and nontechnical parameters of the current application context. Hence, experiments are not *repeatable* which, according to von Mises [vM19], makes the computation of numerical probabilities meaningless. Even, if we employ the subjectivistic approach, e.g., a gambling situation [Nea92], instead of the relative frequency of events to define the semantic of (subjective) probabilities, the significance of the inference results might be questionable for our application domain. This is because the multiplicative combination of probabilities could lead to an unreasonable amplification of estimation errors for longer inference paths.

3.2.3 Credibilistic reasoning

The mathematical theory of *credibilistic reasoning* based on the quantification of pieces of *subjective evidences* has been introduced 1976 by Shafer [Sha76, Sha90]. This theory, which is often referred to as *Dempster-Shafer model*, has been further generalized by Smets [Sme88] to deal also with incomplete information, i.e., non-monotonic reasoning. It has been applied to several practical problems of uncertain reasoning [Sme88, Nea92, Bau94, Bau95].

The central assumption of Shafer‘s theory is that there is a finite amount of belief (or credibility) that is spread among the universe of all relevant propositions $\mathcal{U}$. This belief is distributed according to the available pieces of evidence. Without any loss of generality, the total amount of belief induced by a single piece of evidence is usually scaled to 1. For each available piece of evidence, the expert distributes this total amount of belief to a number of so-called *focal* propositions. The functions that define these basic distributions are referred to as *basic probability assignment*:

Definition 3.6 Basic probability assignment, focal proposition

*Let $\mathcal{U}$ denote the set of all relevant propositions and let $\{E_1,..,E_n\}$ be a set of pieces of evidence. The amount (mass) $m_i(u_1)$ of belief which has been allocated by evidence E_i to a proposition $u_1 \in \mathcal{U}$ (and which cannot be allocated to any other proposition $u_2 \in \mathcal{U}$ that implies u_1) is called a **basic probability number**. Any proposition $u \in \mathcal{U}$ with $m_i(u)>0$ is called **focal proposition** of evidence E_i. Basic probability numbers are assigned to propositions by a function $m: \mathcal{U} \rightarrow [0,1]$ called **basic probability assignment**, with*

$$\sum_{u \in U} m_i(u) = 1, \quad (1 \le i \le n) \tag{EQ 8}$$

◘

difference to probabilistic logic

The main difference of Shafer‘s model compared to probability theory is the way how credibilistic reasoning handles evidence which supports complex focal propositions, e.g., $u_1 \vee u_2$. In probabilistic logic, the total assigned mass of belief $m(u_1 \vee u_2)$ has to be split between

the two component propositions u_1 and u_2. If it is unknown how to distribute $m(u_1 \vee u_2)$, probabilists usually invoke the *principle of insufficient reason* [Sme88] or an argument of symmetry to decide that $m(u_1 \vee u_2)$ has to be split in two equal parts $m(u_1)$ and $m(u_2)$. Credibilistic reasoning does not rely on this principles, i.e., it allows to allocate basic probability numbers for complex propositions.

The combination of different pieces of evidence is performed by applying *Dempster's rule of combination* on the basic probability assignments [Sme88]. At this, the mass assigned to a conjunction of (focal) propositions is defined as the product of the basic probability assignments derived for both propositions from all available pieces of evidence.

Definition 3.7 Combination of evidences

For a proposition $u \in \mathcal{U}$ *and a set of pieces of evidence* $\{E_1,..,E_n\}$, *let* $m_i(u)$ *denote the basic probability numbers assigned to u which have been derived from evidence* E_i. *The combined mass of two evidences* E_1 *and* E_2 *supporting proposition* $u_1 = u_2 \wedge u_3$, *denoted as* $m_{12}(u_1)$, *is then defined as:*

$$m_{12}(u_1) = \sum_{\substack{a \wedge b = u_1 \\ a, b \in U}} m_1(a) m_2(b) \qquad \text{(EQ 9)}$$

The combination of n+1 pieces of evidences is recursively defined by applying Dempster's rule to combine the combination of n pieces of evidence with the next piece of evidence, e.g., m_{123} *is computed by combining* m_{12} *with* m_3 *in the same way.*

□

Using Dempster's rule of combination, we yield a combined measure for the belief $m(u)$ that has specifically been committed to each proposition $u \in \mathcal{U}$. However, if we w ant to obtain the total degree of belief that we have about the fact that u is true, we have to add all masses of belief that have been allocated to propositions $u' \in \mathcal{U}$ that imply u. This total belief is quantified by the so-called *belief function*:

Definition 3.8 Belief function

Let $m: \mathcal{U} \rightarrow [0,1]$ *be the mass function that is obtained by applying Dempster's rule of combination for all available pieces of evidence. The function* $bel: \mathcal{U} \rightarrow [0,1]$ *is called* ***belief function****, with*

$$bel(u_1) = \sum_{u_2 \rightarrow u_1} m(u_2) \qquad \text{(EQ 10)}$$

□

semantics of belief and plausibility

In [Sme88], Smets describes the semantics of the degree of belief as *the degree of minimal or necessary entailment*. Besides the degree of belief, Shafer introduces another measure, which is called *plausibility*. The plausibility of a proposition $u_1 \in \mathcal{U}$ is defined as the sum of the belief allocated to all other propositions $u_2 \in \mathcal{U}$ that do not contradict to u_1. Its meaning can be described as *the degree of minimal or potential entailment*.

Definition 3.9 Plausibility function

Let $m: \mathcal{U} \rightarrow [0,1]$ be the mass function that is obtained by applying Dempster's rule of combination for all available pieces of evidence. The function $pl: \mathcal{U} \rightarrow [0,1]$ is called ***plausibility function****, with*

$$pl(u_1) = \sum_{u_2 \wedge u_1 \neq \not\models} m(u_2) \qquad \text{(EQ 11)}$$

The plausibility function is related to the belief function by the following equation:

$$pl(u)=bel(\models)-bel(\neg u)=1-m(\not\models)-bel(\neg u) \qquad \text{(EQ 12)}$$

Evaluation

limited support for non-monotonic reasoning (R4)

Shafer's original theory did not consider incomplete knowledge. This closed-world assumption has been too restrictive for many practical applications. In [Sme88], Smets describes a theory of credibility that deals with incompleteness. Whenever new evidence becomes available all basic probability assignments are changed to take this new evidence into account. This revision is performed by *Dempster's rule of conditioning* [Sme88].

computational intractable (R6)

A more severe problem that arises with the application of credibilistic reasoning in the DBRE domain is, however, that Shafer's approach has proven unfeasible even for moderately-sized problems (cf. [Pro89, Voo89]). The general problem of inferring belief functions is NP hard, because they are defined on the power set of possible answers to a question which is the complete *Boolean* algebra of events with 2^k elements [Paa88a].

3.2.4 Fuzzy reasoning

Fuzzy logic is a relatively young theory that can be viewed as a form of *multi-valued logic* [Got88]. During the last two decades there has been a tremendous amount of research in this area. Many practitioners have used this technology to implement and reason about vague knowledge in a variety of application domains. We refer the interested reader to [Kas96] and [Nov92] for a comprehensive introduction to this theory. Furthermore, [NAF99] and [FUZ98] give a general overview on the latest results and research directions in this area. The following introduction to the basic principles of fuzzy reasoning (and the next section on possibilistic reasoning) is a little more detailed than the description of the previous theories as they will provide the theoretical framework for the approach developed in Chapter 4.

fuzzy sets

Fuzzy reasoning is based on the central notion of *fuzzy sets* introduced in 1965 by Zadeh [Zad65]. A fuzzy set is a generalization of the concept of a set in classical mathematics. Traditionally, each object in the universe may either be included in a given set S or excluded from S. In this sense, a set S can be represented by its characteristic function

$$\mu_S: \mathcal{U} \rightarrow \{0,1\}, \text{ with } \mu_S(u) = \begin{cases} 1 & u \in S \\ 0 & u \notin S \end{cases}. \qquad \text{(EQ 13)}$$

Zadeh's theory generalizes this concept by allowing objects to belong to a (fuzzy) set only partially. Hence, the values of the characteristic function of a fuzzy set, which is called *membership function* in fuzzy set theory, are real numbers in the interval *[0,1]*.

Definition 3.10 Fuzzy set

A set of pairs $F:=\{(u,\mu(u)) \mid u\in \mathcal{U}\}$ is called ***fuzzy set*** *in a universe $\mathcal{U}$. The function $\mu_F: \mathcal{U}\rightarrow[0,1]$ is called* ***membership function*** *of F.*

For a given object $u\in \mathcal{U}$ and a fuzzy set F the value $\mu_F(u)$ is called *membership degree* of u in F. A membership degree of $\mu_F(u)=0$ means that u is not a member of F and $\mu_F(u)=1$ means that u entirely belongs to F. Membership functions might be continuous or discrete. Figure 3.2 shows two examples from our application domain. The continuous membership function on the left-hand side defines the fuzzy set of large software systems according to their total number of lines of code (LOC). The second example is a fuzzy set of pairs of type compatible string attributes. It is described by a discrete membership function that is defined over the absolute difference of length of both attributes. The diagram on the right-hand side of Figure 3.2 illustrates this fuzzy set for the case that the first attribute has a length of 80 characters.

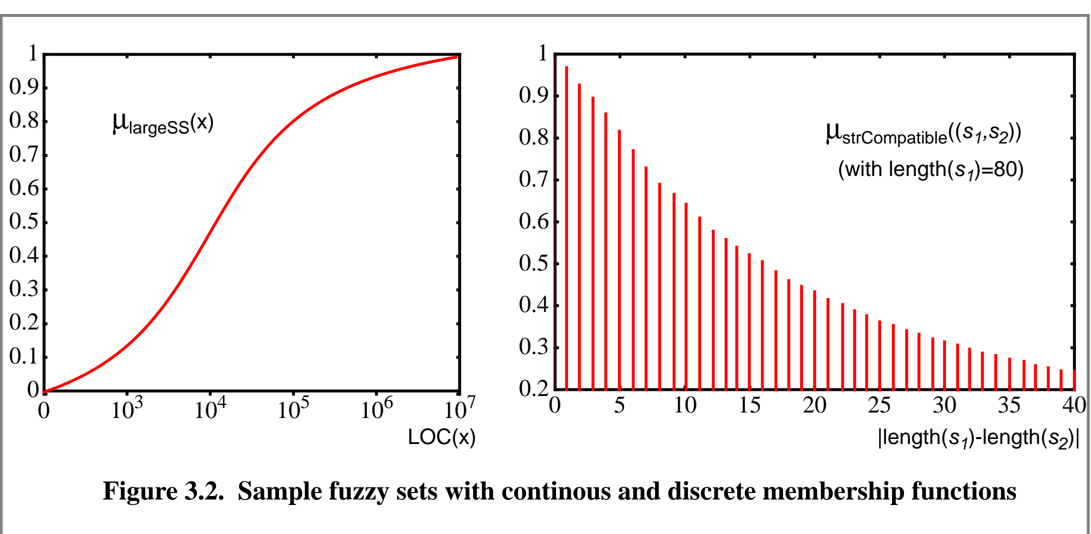


Figure 3.2. Sample fuzzy sets with continous and discrete membership functions

These simple examples already demonstrate that a major benefit of using fuzzy sets is a more adequate formalization of aspects of human reasoning. Using traditional set theory to describe the set of type compatible string attributes, we would have to use a strict threshold value to define the corresponding membership function. Each pair of string attributes with a difference in length that is lower than the chosen threshold would then be considered to be (completely) compatible, while all other pairs of attributes would be considered to be (completely) incompatible. Obviously, this solution does not adequately represent the notion of type compatibility in the mental model of human DBRE experts.

α-cut

Several operations have been defined to convert fuzzy sets to traditional (*crisp*) sets. The most important example is the so called *α-cut*, which is defined to be the (classical) subset of a given

fuzzy set *F* that consists of all elements in *F* with a membership degree greater or equal a given value $\alpha \in [0,1]$.

operations on fuzzy sets

Analogously to crisp sets, it is possible to define operators like *intersection* ($\cap$) and *union* ($\cup$) on fuzzy sets. The *intersection* operator is generally defined by an operation called *t-norm*, while the *union* operator is defined by an operation called *t-conorm*.

Definition 3.11 t-norm and t-conorm

t-norm/t-conorm

Two binary functions $\top,\bot:[0,1]\times[0,1]\rightarrow[0,1]$ *are called* ***t-norm*** *and* ***t-conorm****, respectively, if they fulfill the following properties:*

Symmetry	$\top(x,y)=\top(y,x)$	$\bot(x,y)=\bot(y,x)$
Associativity	$\top(x,\top(y,z))=\top(\top(x,y),z)$	$\bot(x,\bot(y,z))=\bot(\bot(x,y),z)$
Neutral Element	$\top(x,1)=x$	$\bot(x,0)=x$
Null/one element	$\top(x,0)=0$	$\bot(x,1)=1$
Monotony	$x \le z \Rightarrow \top(x,y)\le\top(z,y)$	$x \le z \Rightarrow \bot(x,y)\le\bot(z,y)$

◘

There is a functional dependency between *t-norm* and *t-conorm* operations. With the help of a negation operation *n*, a *t-conorm* can uniquely be derived from a given *t-norm* and vice-versa, i.e., $\bot(x,y)=n(\top(n(x),n(y)))$ and $\top(x,y)=n(\bot(n(x),n(y)))$. Hence, *t-norms* and *t-conorms* are called *dual operations*. In practice, the most commonly used *t-norm* is the minimum function, $\top(x,y)=min(x,y)$. The corresponding *t-conorm* is the maximum function, $\bot(x,y)=max(x,y)$. With these functions, we are able to define the following operations on two fuzzy sets *A* and *B* which are defined over the same universe $\mathcal{U}$. (Other possible choices for *t-norms* (*t-conorms*) can be found in [Gra95]).

$$\text{Union, } A\cup B := \{(x, max(\mu_A(x), \mu_B(x))) \mid x\in \mathcal{U}\} \quad \text{(EQ 14)}$$

$$\text{Intersection, } A\cap B := \{(x, min(\mu_A(x), \mu_B(x))) \mid x\in \mathcal{U}\} \quad \text{(EQ 15)}$$

$$\text{Equality, } A=B := (\forall x\in \mathcal{U})\ (\mu_A(x)=\mu_B(x)) \quad \text{(EQ 16)}$$

$$\text{Complement, } \neg A := \{(x, 1-\mu_A(x)) \mid x\in \mathcal{U}\} \quad \text{(EQ 17)}$$

fuzzy rules and inference

Fuzzy rules are vague implication rules that use fuzzy sets as predicates to express common-sense reasoning. The most common form of such rules is "***IF*** *A(x)* ***THEN*** *B(y)*" (*Zadeh-Mamdani-rules* [Kas96]) or, more general, "***IF*** $A_1(x_1)$ ***and*** $A_2(x_2)$ ***and ...*** $A_n(x_n)$ ***THEN*** *B(y)*".

Example 3.1 Fuzzy rule

In this example, we use a fuzzy rule to describe the following sample DBRE heuristic:

*"**If** the name of an attribute x is similar to its RS R, supplemented with the string 'id' **and** if all tuples in the extension of R have unique values in attribute x **and** if the extension of R is large **then** x might be a key."*

In the following, we will use fuzzy predicates to reason about a given attribute *x* that belongs to a relation schema *RS(x)*:

IF *ANameIsRSName+ID(x)* ***AND*** *Unique(x)* ***AND*** *LargeExt(δ(RS(x)))*
THEN *Key(x)*

Let us abbreviate the first predicate used in the above implication rule as *AName*. Each of those four predicates are described by a (fuzzy) set that contains all objects in the universe which (gradually) comply to this predicate. Figure 3.3 illustrates this for predicates *AName* and *LargeExt*. The left-hand side of Figure 3.3 shows similarity degrees for seven sample attributes of an RS named *user*. In this definition of μ_{AName}, we use the *Levenshstein-distance* [Lev66] (*Levensh()*) to calculate a measure of similarity of two strings. The right-hand side of Figure 3.3 shows a sample definition of fuzzy sets that define the predicates *LargeExt* and *MediumExt* for possible extensions δ*(RS(x))*.

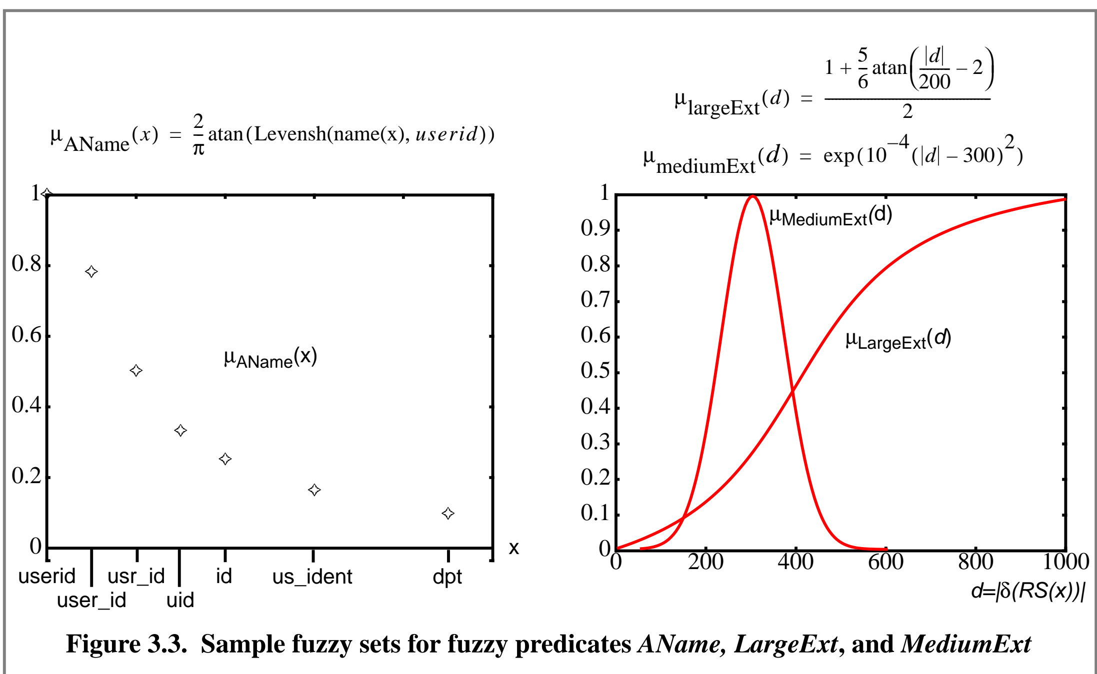


Figure 3.3. Sample fuzzy sets for fuzzy predicates *AName, LargeExt*, and *MediumExt*

If the fuzzy values in the antecedent of a fuzzy rule are known, it is possible to compute a fuzzy value for its consequent by using methods of *fuzzy inference*. Fuzzy inference is based on the notion of *fuzzy implications* and *fuzzy compositions*. In order to define these terms we have to introduce the formal concept of a *fuzzy relation*.

Definition 3.12 Fuzzy relation

fuzzy relations

Let $F_1,..,F_n$ be n fuzzy sets over objects of the universe $\mathcal{U}_1,..,\mathcal{U}_n$, respectively. A ***fuzzy relation*** *$R(F_1,..,F_n)$ is then defined as a fuzzy set over the cross product of the universes $\mathcal{U}_1 \times .. \times \mathcal{U}_n$, i.e., $R(F_1,..,F_n)=\{((x_1,..,x_n), \mu_R(x_1,..,x_n)) \mid x_1 \in \mathcal{U}_1,..,x_n \in \mathcal{U}_n\}$*

A *fuzzy implication*, denoted as $A \rightarrow B$, is a fuzzy relation over two fuzzy sets A and B over the universes $\mathcal{U}_A$ and $\mathcal{U}_B$, respectively. In fuzzy logic there are different ways to define an implication. This is in contrast to propositional logic where the implication is defined by a

single truth table. Mizumoto and Zimmermann compare 15 different fuzzy implications [MZ82]. One commonly used implication is defined by the minimum function and has been introduced by Mamdani [EM77] (for a discussion of other implications, we refer to [Ker92]):

$$A \rightarrow B := \{((a,b), min(\mu_A(a), \mu_B(b))) | a \in \mathcal{U}_A,\ b \in \mathcal{U}_B\} \qquad (EQ\ 18)$$

Analogously to propositional logic, fuzzy logic uses *and-* ($\wedge$), *or-* ($\vee$), and *not-* ($\neg$) operators to compose logical expressions. These operators are defined by the following compositions:

Definition 3.13 Fuzzy logical operators

Conjunction, $A \wedge B := \{(x, min(\mu_A(x), \mu_B(x))) \mid x \in \mathcal{U}\}$

Disjunction, $A \vee B := \{(x, max(\mu_A(x), \mu_B(x))) \mid x \in \mathcal{U}\}$

Negation, $\neg A := \{(x, 1 - \mu_A(x)) \mid x \in \mathcal{U}\}$

MAX-MIN composition

A *fuzzy composition* of two fuzzy relations $R_1(A,B)$ and $R_2(B,C)$, denoted as $R_1 \bullet R_2$, is a relation $(R_1 \bullet R_2)(A,C)$ obtained by applying R_1 and R_2 after one another. A typical composition is the MAX-MIN composition [Zad65]:

$$(R_1 \bullet R_2)(A,C) = \{((a,c), max\{min(\mu_{R1}(a,b), \mu_{R2}(b,c)) \mid b \in \mathcal{U}_B\}) \mid a \in \mathcal{U}_A,\ c \in \mathcal{U}_C\} \qquad (EQ\ 19)$$

Analogously to the fuzzy implication, there are other composition operators that have been successfully applied to fuzzy reasoning [Kas96]. A fuzzy implication and composition allow for fuzzy inference according to the following *compositional inference law*. (Other possible laws of inference are given in [Kas96].)

Definition 3.14 Fuzzy inference

Given an implication $A \rightarrow B$ *and a composition* $\bullet$*, a fuzzy set* B' *can be inferred when a fuzzy set* A' *is known, with* $B' = A' \bullet (A \rightarrow B)$

3.2.4.1 Evaluation

limited support for uncertainty (R1)

Fuzzy logic has been introduced as an approach to *linguistic approximation* of human knowledge. It allows to describe and reason about *vague* concepts but it is less suitable to deal with *uncertain* knowledge. Some extensions of this theory have been proposed to overcome this deficiency. Analogously to the approach described in Section 3.2.1, a popular approach is to assign *confidence factors* (CF) to fuzzy rules and facts [Kas96, pp. 194ff]. However, this solution has similar limitations like production systems with CF in the general case of cyclic rule bases. Another approach to handle uncertainty is *type-2 fuzzy logic* [KM98]. It is based on the concept of *type-2 fuzzy sets*, introduced in [Zad75]. While in normal (*type-1*) fuzzy sets membership degrees are represented as real numbers, in type-2 fuzzy sets, membership degrees are fuzzy themselves, i.e., they are defined by (type-1) fuzzy values in [0,1]. Hence, type-2 fuzzy sets can be used in situations where there is uncertainty about the membership degrees, e.g., when the exact shape of the membership function is unknown. This approach can be viewed as a *second-order approximation*, compared to type-1 fuzzy logic which represents a first-order approximation. A qualitative disadvantage of using type-2 fuzzy logic to describe

uncertain DBRE knowledge is that second-order approximations are more difficult to handle and compute than other approaches which include a direct notion of uncertainty.

limited support for contradiction and ignorance (R3,R5)

Fuzzy reasoning does not meet our requirements for representation of contradicting knowledge (*R3*) and partial ignorance (*R5*). Recently, Zhang proposed to use *bipolar fuzzy sets* to overcome this limitation [Zha98]. A bipolar fuzzy set consists of two traditional fuzzy sets which represent degrees of *compatibility* or *incompatibility* with the associated predicate, respectively. Hence, they allow to reason about the coexistence and interaction of contradicting relationships. In addition, they are suitable to express partial ignorance.

3.2.5 Possibilistic reasoning

Possibility theory has been introduced by Zadeh [Zad78] in 1978 as a means for approximate reasoning with uncertain and incomplete information. Since then, possibility theory has been systematically developed as a calculus of uncertain logics, mainly by Dubois et al. [DP83, DP88, DLP92, PD93, DLP94, DP97]. Like fuzzy logic, possibilistic logic has its roots in the theory of fuzzy sets. However, both calculi serve distinct purposes. While fuzzy logic is used to reason about vague knowledge, possibilistic logic has been developed primarily to reason about uncertain and incomplete knowledge. In this section, we will introduce the main idea behind the concept of possibility and we will introduce a calculus for *necessity-valued possibilistic logic*. For a comprehensive introduction to possibility theory, we refer to [DLP94].

possibility and necessity

Possibilistic logic deals with weighted formulae of the form (f,β), where f is a closed formula in L^1 and the *valuation* $\beta \in [0,1]$ is a positive real value. The valuation represents a lower bound on so-called degrees of *necessity* $N(f)$ or degrees of *possibility* $P(f)$ of the corresponding formula f. The value of $N(f)$ expresses to what extent the available evidence entails the truth of f, whereas $P(f)$ expresses to what extent the truth of f does not contradict to the available evidence.[a] The degree of necessity and the degree of possibility are dual measures, i.e., $N(f)=1-P(\neg f)$. It is important to note that $N(f)=0$ or $P(f)=1$ represent the state of complete ignorance, i.e., nothing is known about the truth of f. The following properties hold:

$$P(\not\models)=0;\ P(\models)=1;\ N(\not\models)=0;\ N(\models)=1; \qquad (EQ\ 20)$$

$$N(f_1 \wedge f_2)=min(N(f_1),N(f_2));\ P(f_1 \wedge f_2)=max(P(f_1),P(f_2)) \qquad (EQ\ 21)$$

$$N(f_1 \vee f_2) \geq max(N(f_1),N(f_2));\ P(f_1 \wedge f_2) \leq min(P(f_1),P(f_2)) \qquad (EQ\ 22)$$

$$min(N(f),N(\neg f))=0;\ max(P(f),P(\neg f))=1 \qquad (EQ\ 23)$$

necessity-valued formulae

In the following, we will only consider *necessity-valued* formulae, because this fragment of possibilistic logic is powerful enough for our application.

Definition 3.15 Necessity-valued formula

A ***necessity-valued formula*** *is a pair* $\phi := (f, \beta)$*, where* f *is a well-formed formula in* L^1 *and* $\beta \in [0,1]$ *is a lower bound for the necessity degree of* f*, i.e.,* $N(f) \geq \beta$*. Let* NPL^1 *denote the* ***language*** *of necessity-valued formula.*

◻

a Other possible (physical) interpretations of this mathematical model are summarized in [DP88].

Sometimes it is desired to convert a set of necessity-valued formulae Φ to a set of classical formulae or to extract those formulae from Φ that are at least certain to a given degree. The following two operations serve these purposes.

Definition 3.16 Classical projection

For a given set of necessity-valued formulae $\Phi \subseteq \mathcal{L}\{NPL^1\}$, the **classical projection** *Φ^* is defined as $\Phi^* := \{f \,/\, (f,\beta) \in \Phi\}$.*

Definition 3.17 α-cut

For a given set of necessity-valued formulae $\Phi \subseteq \mathcal{L}\{NPL^1\}$, the **α-cut** *$\Phi_\alpha$ is defined as $\Phi_\alpha := \{(f,\beta) \,/\, (f,\beta) \in \Phi \wedge \beta \geq \alpha\}$.*

semantics

The semantics of a set of closed formulae $\mathcal{F}$ in L^1 is defined by the subset ω of all interpretations Ω that satisfy all formulae in $\mathcal{F}$. Each such interpretation $\omega \in \Omega$ is called a model of $\mathcal{F}$. In case of a set of formulae Φ in NPL^1 the interpretation is given by a so-called *possibility distribution* over Ω that is represented by fuzzy set π of all models for Φ. π can be viewed as a preference relation over Ω. Based on the possibility distribution π, we can define the possibility measure P as a function

$$P: \mathcal{L}\{L^1\} \rightarrow [0,1], \text{ with } P(f) = sup\{\pi(\omega), \omega \models f\}, \ \omega \in \Omega \qquad (EQ\ 24)$$

Consequently, the dual necessity measure $N(f)=1-P(\neg f)$, induced by π is defined by

$$N: \mathcal{L}\{L^1\} \rightarrow [0,1], \text{ with } N(f) = inf\{1-\pi(\omega), \omega \models \neg f\}, \ \omega \in \Omega \qquad (EQ\ 25)$$

A possibility distribution π is said to *satisfy* a formula $(f,\beta) \in \mathcal{L}\{NPL^1\}$, iff $N(f) \geq \beta$. Consequently, a set of formulae $\Phi = \{\phi_1,..,\phi_n\} \subseteq \mathcal{L}\{NPL^1\}$ is satisfied by a possibility distribution π, denoted as $\pi \models \Phi$, iff $\forall i \in [1,n]$, π *satisfies* ϕ_i. Then, a logical formula $\phi_{n+1} \in \mathcal{L}\{NPL^1\}$ is a *logical consequence* of a set of formulae $\Phi \subseteq \mathcal{L}\{NPL^1\}$, iff all possibility distributions that satisfy Φ also satisfy ϕ_{n+1}, i.e., the following condition holds.

$$\forall \pi \ (\pi \models \Phi) \Rightarrow (\pi \models \phi_{n+1})) \qquad (EQ\ 26)$$

partial contradiction

For a *consistent* set of possibilistic formulae Φ, we require the fuzzy set that represents the possibility distribution π induced by Φ to be normalized, i.e., $sup\{\pi(\omega) / \omega \in \Omega\} = 1$. Possibilistic logic is also able to deal with partial *contradiction* if we give up the above normalization condition, i.e., if we allow for $sup\{\pi(\omega) / \omega \in \Omega\} = 1-i$, $i \in (0,1]$. Consequently, the axiom $N(\bot)=0$ (*EQ 20*) (given for the consistent case) is no longer valid, because $N(\bot) = N(f \wedge \neg f) = min(N(f), N(\neg f)) = i > 0$. However, the following properties still hold:

$$N(\models) = 1 \qquad (EQ\ 27)$$

$$N(f_1 \wedge f_2) = min(N(f_1), N(f_2)) \qquad (EQ\ 28)$$

$$N(f_1 \vee f_2) \geq max(N(f_1), N(f_2)) \qquad (EQ\ 29)$$

$$(f_1 \models f_2) \Rightarrow N(f_2) \geq N(f_1) \qquad (EQ\ 30)$$

Definition 3.18 Partial contradicting set of formulae

A set of formulae $\Phi=\{\phi_1,..,\phi_n\}\subseteq \mathcal{L}\{NPL^1\}$ is said to be partial contradicting (inconsistent), if there is no normalized possibility distribution that satisfies Φ, i.e.,

$$Cons(\Phi)=sup_{\pi \models \Phi} sup_{\omega \in \Omega} \pi(\omega)<1 \quad (EQ\ 31)$$

*$Cons(\Phi)$ and $Incons(\Phi)=1-Cons(\Phi)$ are called the **degree of consistency** or **inconsistency** (contradiction) of Φ, respectively.*

deduction problem

According to [DLP94], the deduction problem in possibilistic logic can be stated as follows: Given a set of formulae $\Phi\subseteq \mathcal{L}\{NPL^1\}$ and a classical formula f that we would like to deduce from Φ, we have to compute the *best valuation* β (i.e., the best lower bound of a necessity degree) such that (f,β) is a logical consequence of Φ. This means, we have to compute $Val(f,\Phi)=sup\{\beta\in (0,1] / \Phi \models (f,\beta)\}$.

least specific poss. distribution

This valuation is defined by the necessity measure $Val(f,\Phi)=N_\Phi(f)$ which is induced by the *least specific* possibility distribution π_Φ satisfying Φ. For a given set of formulae $\Phi=\{(f_1,\beta_1),..,(f_n,\beta_n)\}\subseteq \mathcal{L}\{NPL^1\}$ the least specific possibility distribution π_Φ is defined as $\pi_\Phi(\omega)=min\{1\text{-}\beta_i \,/\, \omega \models \neg f_i,\ i\in [1,..,n]\}$.

best model

π_Φ imposes a preference relation over all models of Φ. In order to solve the aforementioned deduction problem, we have to select a best model which means to choose an interpretation $\omega^*\in\Omega$ that is most compatible with Φ. The degree of compatibility of a given model ω is defined by $\pi_\Phi(\omega)$. Note, that such a best model always exists; a proof can be found in [DLP94].

Definition 3.19 Best model

*Let $\Phi\subseteq \mathcal{L}\{NPL^1\}$ be a set of possibilistic formulae. Any interpretation $\omega^*\in\Omega$ that maximizes π_Φ is called **best model** of Φ, i.e., $\pi_\Phi(\omega^*)=sup\{\pi_\Phi(\omega)/\omega\in\Omega\}$.*

inference

In [Lan91] and [DLP94], Lang et al. propose a formal system in terms of a set of axioms and inference rules, that implements the described semantics of NPL^1, i.e., that fulfills the condition that every possibilistic formula $\phi\in \mathcal{L}\{NPL^1\}$ is a consequence of a set of possibilistic formulae $\Phi\subseteq \mathcal{L}\{NPL^1\}$, iff ϕ can be derived from Φ using the proposed formal system. Similar versions of the following inference rule GMP (*graded modus ponens*) have been used in many theoretical frameworks for uncertain reasoning, e.g., [Res76, FG90].

Definition 3.20 Formal system for NPL^1

Axioms:

- (A1) $(f_1 \Rightarrow (f_2 \Rightarrow f_1)\ 1)$
- (A2) $((f_1 \Rightarrow (f_2 \Rightarrow f_3)) \Rightarrow ((f_1 \Rightarrow f_2) \Rightarrow (f_1 \Rightarrow f_3))\ 1)$
- (A3) $((\neg f_1 \Rightarrow \neg f_2) \Rightarrow ((\neg f_1 \Rightarrow f_2) \Rightarrow f_1)\ 1)$
- (A4) $((\forall x(f_1 \Rightarrow f_2) \Rightarrow (f_1 \Rightarrow (\forall x f_2))\ 1)$, *if x does not appear in f_1 and is not bound in f_2.*
- (A5) $((\forall x f) \Rightarrow f_{x/t}\ 1)$, *if x is free for t in f.*

Inference rules:

- (GMP) $(f_1,\beta), (f_1 \Rightarrow f_2,\gamma) \vdash (f_2, min(\beta,\gamma))$
- (G) $(f,\beta) \vdash ((\forall x f),\beta)$, *if x is not bound in f*
- (S) $(f,\beta) \vdash (f,\gamma)$, *if $\gamma \le \beta$*

3.2.5.1 Evaluation

The theory of possibilistic reasoning meets all requirements identified in Section 3.1. It is well-suited to describe and reason about uncertain knowledge (*R1*). The deduction mechanism described above deals with contradicting domain-specific and situation specific knowledge. Dubois et al. show that for a given set of formulae $\phi \in \mathcal{L}\{NPL^1\}$ the degree of contradiction $Incons(\phi)$ acts as a *threshold* that inhibits all formulae of ϕ with a valuation equal to or under this threshold [DLP94, pp. 458ff]. If ϕ contains all domain-specific and situation-specific knowledge the contradicting part of ϕ can be isolated by selecting all formulae $\phi_i \subseteq \phi$ that have a valuation lower or equal to $Incons(\phi)$. The part of ϕ_i that represents the (contradicting) situation-specific knowledge can be indicated to the user. Hence, both requirements, *R2* and *R3*, are satisfied. Moreover, requirement *R5* is fulfilled because ignorance about the truth of a proposition $u \in \mathcal{U}$ can be expressed by $N(u)=N(\neg u)=0$ or $P(u)=P(\neg u)=1$, respectively.

The deduction operator $\vdash$ introduced in Definition 3.20 is monotonic. However, in [DLP94, pp. 466ff], Dubois et al. define the *nontrivial deduction operator* $\mid\!\sim$ that allows only for the deduction for formulae with a valuation greater than the degree of contradiction, i.e.,

$$\phi \mid\!\sim (f,\beta) \text{ iff } \phi \vdash (f,\beta) \text{ and } \beta > Incons(\phi). \qquad (EQ\ 32)$$

Hence, the nontrivial deduction operator enables non-monotonic reasoning (requirement *R4*), i.e., it is possible that $\phi \mid\!\sim (f,\beta)$ and $\phi \cup \phi^* \not\mid\!\sim (f,\beta)$.

Finally, the problem of inference in possibilistic rule bases has polynomial complexity. Of course, if general first-order formulae in NPL^1 are considered the complexity of inference is exponential with respect to the number of elements in the universe of discourse.

3.3 Summary and conclusion

In this chapter, we elaborated a catalog of major requirements on a formalism that is suitable to manage imperfect DBRE knowledge in human-centered CARE environments. Based on these requirements, we systematically evaluated five important approaches to represent and reason about uncertain knowledge. We would like to emphasize that this evaluation is not general but dedicated to our particular application domain. Other applications might impose different criterions. In the following, we summarize the result of our evaluation in order to decide which approach is most suitable for our purpose. Figure 3.4 shows a decision matrix that relates each approach with each requirement imposed. In this matrix a requirement is either fulfilled (✓), partly fulfilled (■) or failed (✗) by a given approach. Obviously, this kind of condensed classification represents a simplified view on the results of our evaluation, i.e., it does not show preferences between two approaches which both fulfill or fail a given requirement. Still, it serves our purpose to identify the formalism which is most appropriate for the application to DBRE. Moreover, a quantitative classification would be rather hypothetical without further experimental results.

The main reasons for the unsuitability of production systems with confidence factors for our application is their computational difficulties in the case of cyclic inference networks. Moreover, they lack mechanisms to deal with contradicting and incomplete knowledge. Due to their computational complexity, probabilistic and credibilistic reasoning do not scale up for applications to practical DBRE problems: the concept of *objective* probability which is based

on the relative frequency of events does not apply to the DBRE context. Even if a *subjectivistic* view on probabilities is used it is problematic to estimate their reliability (in terms of error models). The multiplicative combination of uncertainties amplifies estimation errors which might lead to unreasonable results. In addition, the credibilistic approach lacks an explicit notion of contradiction. The primary focus of fuzzy reasoning is to deal with *vague* rather than *uncertain* knowledge. Existing approaches to incorporate a notion of uncertainty in fuzzy logic (e.g., confidence factors and *Type-2* fuzzy sets) comprise significant limitations w.r.t. to our application domain (cf. Section 3.2.4). This is in contrast to possibility theory which allows to reason about uncertain, contradicting, and incomplete knowledge. Consequently, possibility theory turns out to be most suitable to implement and reason about DBRE knowledge. In the next chapter, we will use this theory as a basis to develop a dedicated, high-level formalism to specify, customize, and execute DBRE knowledge.

Approach / *Requirement*	**Production sytems with CFs**	**Probabilistic reasoning**	**Credibilistic reasoning**	**Fuzzy reasoning**	**Possibilistic reasoning**
R1 **(uncertainty)**	✓	■ (error models)	✓	(■) (CF and Type-2-logic)	✓
R2 **(represent. of contradiction)**	✓	✓ (deviation of uncertain probabilities)	✓	■ (interpolation)	✓
R3 **(indication of contradiction)**	✕ (no explicit notion of contradiction)	✕ (adaptation of domain-spec. knowledge)	✕ (no explicit notion of contradiction)	■ (bipolar fuzzy sets)	✓
R4 **(incomplete-ness)**	✕	✓ (belief revision [AGM85])	✓ (Dempster's rule of conditioning)	✓ (nonmonotonic fuzzy logic, e.g., *FNM3* [DD92])	✓ (non-trivial deduction operator)
R5 **(ignorance)**	✓	✕	✓	✓ (bipolar fuzzy sets)	✓
R6 **(computational tractability)**	■ (problem with cycles)	✕	✕	✓	✓

Figure 3.4. Evaluation summary

CHAPTER 4 GFRN AS A BASIS FOR LEGACY SCHEMA ANALYSIS

In our experience, lack of customizability is the single most common limiting factor in using tools for software analysis and transformation.

Markosian et al. [MNB+94]

This chapter introduces *Generic Fuzzy Reasoning Nets* (GFRNs) as a dedicated formalism to specify, customize, and execute database reengineering (DBRE) knowledge applied to schema analysis. The development of this formalism has been driven by the requirements elaborated in Section 3.1. It is based on possibilistic logic (and fuzzy set theory) which, according to our evaluation in Section 3.2, is most adequate to manage imperfect knowledge in our specific application domain. The GFRN approach enables to realize a CARE environment that supports partial automation of the schema analysis process but provides a high amount of customizability and extensibility. GFRNs facilitate the integration of various existing analysis methods and the adaption of domain-specific DBRE knowledge. Our approach is human-centered because it allows for (and depends on) human interaction in an evolutionary rather than a phase-oriented schema analysis process. It reflects on the mental model of the reengineer and guides her/him from initially incomplete and contradicting knowledge about a legacy database (LDB) to a complete and consistent model of the corresponding logical schema. This logical schema is the basis for subsequent conceptual migration and redesign activities discussed in Chapter 5.

The structure of this chapter is as follows. In the next section, we give an overview of the proposed schema analysis process that is supported by our approach. Section 4.2 introduces GFRNs as a dedicated formalism to specify domain-specific DBRE knowledge and processes. Subsequently, we develop an inference mechanism for GFRN specifications that can be implemented in a human-centered CARE tool (Section 4.3). Section 4.4 presents the *Varlet Analyst* which is a prototype implementation of the concepts developed in this chapter. Section 4.5 reports on our experiences with applying this implementation to evaluate our approach with practical DBRE problems. A discussion of related work in the domain of legacy schema analysis is presented in Section 4.6. Finally, a summary of this section and its results is given in Section 4.7.

4.1 Supporting human-centered schema analysis processes

The main purpose of this chapter is to clarify the role of GFRN specifications in the proposed schema analysis process before we introduce the actual formalism. Moreover, the structure of the rest of this chapter is directly motivated by this schema analysis process which is shown as a data flow diagram in Figure 4.1.

customization process

It is important to distinguish between activities that aim to *customize* the prospected CARE tool to a specific application context from activities that are involved in the actual process of *applying* the tool. The activities that belong to the customization process are displayed with a grey background in Figure 4.1. In this process, a knowledge engineer investigates the LDB in order to determine the specific application context of the CARE tool. The result of this *domain analysis* step is a number of technical and non-technical characteristics of the current LDB, e.g., properties of the employed hard- or software platform and applied coding conventions, respectively. Subsequently, the knowledge engineer *specifies* or adapts the domain-specific DBRE knowledge that is applied in the schema analysis process according to these characteristics. The corresponding knowledge is formally represented by a GFRN specification.

analysis process

After the tool has been customized with respect to its current application context it can be employed to analyze the schema of the LDB which is under investigation. This analysis process is performed semi-automatically. At first, automatic analysis operations are applied to different legacy software artifacts including the LDB's schema catalog, procedural code, and the available data. The result of this *initial automatic analysis* is a set of (situation-specific) facts about the LDB. Subsequently, these facts are taken as indicators which are combined with the domain-specific knowledge specified in the GFRN to *infer* new knowledge about possible schema constraints. This newly inferred knowledge might comprise definite facts as well as uncertain and contradicting hypotheses. Some of these hypotheses might be refutable using automatic analysis operations. We call such analysis operations *goal-driven* because they are performed "on-demand" to support or refute intermediate hypotheses. According to the domain-specific characteristics of the LDB, the GFRN specification determines which operations are available and when they are performed.

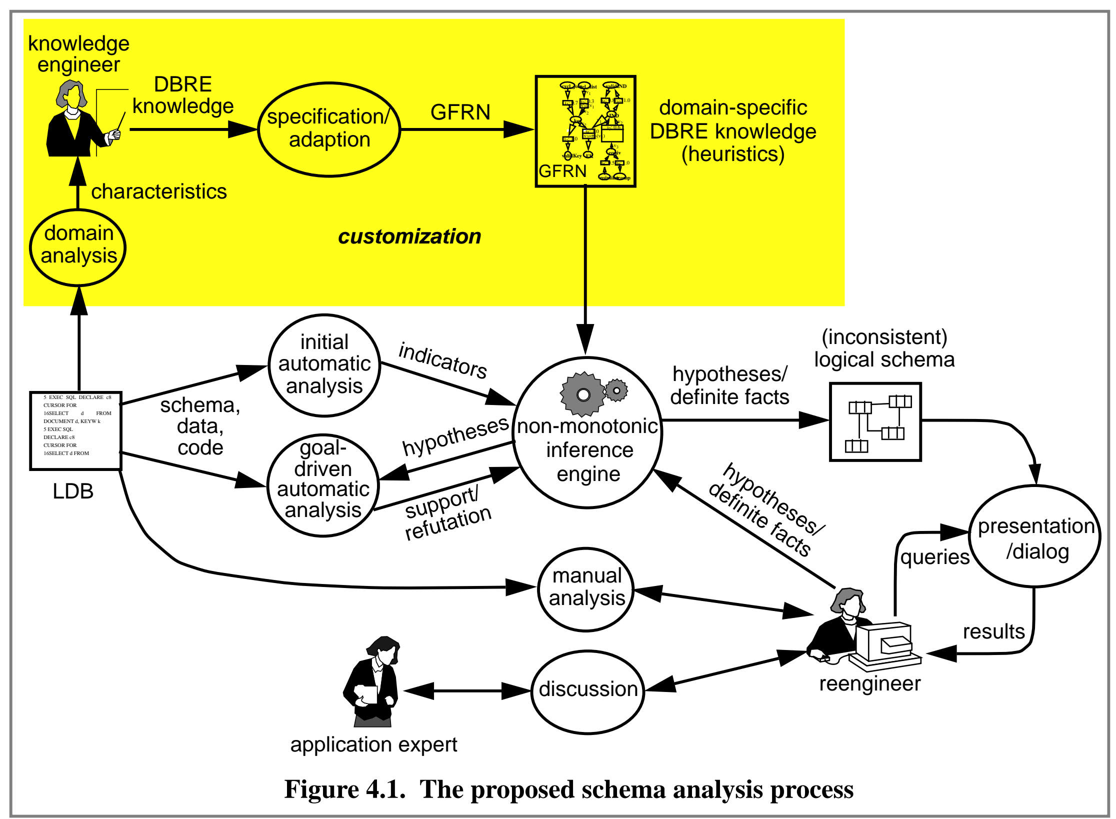


Figure 4.1. The proposed schema analysis process

user interaction

The output of this non-monotonic inference process is a logical schema which might still partially be inconsistent and uncertain. This schema is presented to the reengineer in a *dialog* process that provides interactive queries to indicate the sources of such imperfect knowledge. The reengineer might discuss this information with application experts (e.g., developers or operators) and do further manual investigations of the LDB. As a result of these manual activities the reengineer might enter additional hypotheses or definite facts about the LDB. Now, the inference process is resumed, i.e., new knowledge is inferred and automatic (goal-driven) analysis might be performed to validate hypotheses. The described semi-automatic schema analysis process is iterated until a complete and consistent logical schema is obtained.

role of the GFRN

From the above description it becomes clear that the domain-specific DBRE knowledge which is defined in a GFRN serves mainly three purposes: (1) it *unburdens* the reengineer from manually analyzing recurring situations and focuses her/his attention on non-standard situations, (2) it *controls the consistency* of the analysis result, i.e., the logical schema, and (3) it *facilitates the customization* of the CARE tool to changing application contexts.

4.2 Specification of database reengineering knowledge

In the previous section, we have described the proposed schema analysis process and clarified the role of predefined domain-specific DBRE knowledge. The current section is dedicated to the definition of GFRN as a formalism to specify this knowledge. According to the results of our evaluation in Chapter 3, we have chosen possibility theory as the formal framework for the definition of the GFRN semantics. As customizability is a crucial requirement in our application domain, we have developed GFRN as a graphical formalism that provides a high level of abstraction and, thus, facilitates human comprehension. In Section 4.2.1 and Section 4.2.2, we begin with an informal introduction of GFRNs followed by their formal definition in Section 4.2.3.

Basic definitions

Before we begin with the introduction of GFRNs as a formalism to support legacy schema analysis, we need a more precise notion of the actual analysis result, i.e., an *analyzed logical schema* of a relational database. In Section 2.4.1, we exemplified that such an analyzed schema basically consists of a relational schema with semantical annotations. Similar to an approach proposed by Fahrner and Vossen [FV95], we used annotations to classify INDs according to their semantics. In addition, we generalized the notion of attributes that can contain NULL-values to an explicit concept of different relational *variants* (cf. page 20). We formalize the signature of an analyzed logical schema in Definition 4.1. As the semantics of the relational data model is well-known we forego a formal definition of its interpretation in this chapter. However, such a formalization is included as Definition A.1 in Appendix A. From now on, we refer to an *analyzed logical schema* even if we use the expression *logical schema* for abbreviation.

Definition 4.1 Signature of an analyzed logical schema

An ***(analyzed) logical schema*** *for a relational DB is a tuple $(T, R, \Delta, \mathcal{A})$, where*

- $T=\{t_1,...,t_m\}$, $m\in \mathbb{N}$, *is a finite set of* ***attribute type names****;*
- $R=\{n_1,...,n_m\}$, $m\in \mathbb{N}$, *is a finite set of* ***relation schemas*** *(tables); each* $r\in R$ *is a tuple* $r:(n, X, \Sigma, V)$, *where*
  - *n is a unique* ***name*** *of a relation schema (RS);*
  - $X(r)=X=\{x_1,...,x_m\}$, $m\in \mathbb{N}$, *is a finite set of* ***column signatures****; each* $x\in X$ *is a tuple* $x:(n, c, t)$, *where c is a unique* ***attribute (column) name*** *and* $t\in T$ *is an* ***attribute type****;*
  - $\Sigma(r)=\Sigma=\{\sigma_1,...,\sigma_m\}$, $m\in \mathbb{N}$, *is a finite set of* ***keys****, with* $\sigma_j\subseteq X$, *for* $j\in [1,m]$*;*
  - $V(r)=V=\{v_1,...,v_m\}$, $m\in \mathbb{N}$, *is a non-empty, finite set of* ***variants****; each* $v_j\in V$, $j\in [1,m]$, *is a subset of X that includes all keys, i.e.,* $v_j\subseteq X$, $\forall\sigma\in \Sigma:\sigma\subseteq v_j$*;*
- $\Delta=\{d_1,...,d_m\}$, $m\in \mathbb{N}$, *is a finite set of* ***inclusion dependencies*** *(INDs); each* $d\in \Delta$ *is a tuple* $d:(l, r, I)$, *where*
  - $l\in V$ *is a* ***variant*** *of an RS* $(n, X, \Sigma, V)\in R$ *and represents the* ***left side*** *of the IND;*
  - $r:(\overline{n}, \overline{X}, \overline{\Sigma}, \overline{V})\in R$ *is the* ***RS*** *that represents the* ***right side*** *of the IND;*
  - $I=\{i_1,...,i_m\}$, $m\in \mathbb{N}$, *is a finite set of pairs of* ***equivalent attributes****, for each* $(x_l,x_r)\in I$, $x_l\in V$ *and* $x_r\in\overline{X}$.
- $\mathcal{A}:\Delta\rightarrow\{I\text{-}IND, R\text{-}IND, C\text{-}IND\}$ *is an* ***annotation function*** *that classifies each IND* $d\in \Delta$ *as*
  - *I-IND, if d semantically represents an inheritance relationship,*
  - *R-IND, if d semantically represents an association, and*
  - *C-IND, if d semantically represents a cardinality constraint (cf. [FV95]).*

For notational convenience, we define for any attribute x that RS(x) denotes the corresponding RS, i.e., $\forall r\in R, x\in X(r): RS(x):=r$.

◻

4.2.1 Informal introduction to GFRNs

The purpose of the GFRN language is to define domain-specific knowledge and analysis processes which are executed in a semi-automatic reverse engineering activity to recover a logical schema that is structurally complete and semantically enriched. In the following, we will informally discuss several example GFRN specifications that define parts of the knowledge employed in our DBRE case study in Section 2.4.1. For each of these examples, we denote the corresponding formal semantics in necessity-valued possibilistic logic (NPL^1) (cf. Section 3.2.5).

A GFRN specification is a graphical network of fuzzy *predicates* (represented as ovals) and uncertain *implications* (represented as rectangles). Predicates and implications are connected by directed arcs which are labeled by variable names. Figure 4.2 shows a simple example for a GFRN that represents the heuristic that an instance of a *cyclic-join* pattern over a set of relational attributes indicates a possible key constraint over these attributes (cf. page 18). The corresponding GFRN contains two predicates ($cyclicJoin^1$ and key^1) and one implication. Each predicate has a unique name which terminates with a number that denotes the arity of the corresponding predicate.

The premise of an implication is defined by all predicates that are the sources of ingoing arcs of this implication. Each implication has an associated *confidence value* (CV) between 0 and 1. Based on the theory of possibilistic logic, the semantics of a CV is a lower bound of the necessity that the corresponding implication is valid (cf. Section 3.2.5). The semantics of an implication in a GFRN is defined by a closed formula in NPL^1, i.e., all variables are implicitly quantified by a universal quantifier. Hence, the semantics of the GFRN in Figure 4.2 is defined by a formula $(f, 0.7) \in \mathcal{L}\{NPL^1\}$, with

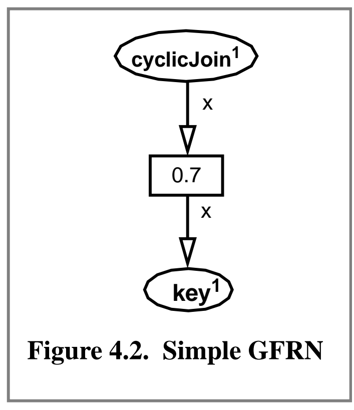


Figure 4.2. Simple GFRN

$$f := \forall x\ (cyclicJoin^1(x) \rightarrow key^1(x)).$$

constraints and negation

In order to express more complex heuristics, implications can be associated with constraints over variables that are attached to in- and outgoing arcs. As an example, the GFRN in Figure 4.3 represents the heuristic that an instance of a *select-distinct* pattern over a set of selected attributes *s* serves as an indicator against the assumption that one of the subsets $k \subseteq s$ might represent a candidate for a key (cf. page 18). Note, that in order to simplify the GFRN syntax we use prefix notation to denote operations defined in constraints of implications. The negation in the conclusion of the corresponding implication is represented by an arc with a solid arrow head. We would like to emphasize that the CVs presented in our examples are not absolute but depend on the specific characteristics of the LDB under investigation. They are adjusted according to the results of the domain analysis activity during the tool customization process (cf. Figure 4.1). The relatively low CV of the *select distinct* heuristic in Figure 4.3 might reflect on the fact that by investigating code samples the knowledge engineer has discovered that the programmers of the LDB had not been precise in using the *distinct* keyword only in queries where it is necessary to suppress duplicate tuples. The semantics of the GFRN in Figure 4.3 is defined by a formula $(f, 0.3) \in \mathcal{L}\{NPL^1\}$, with

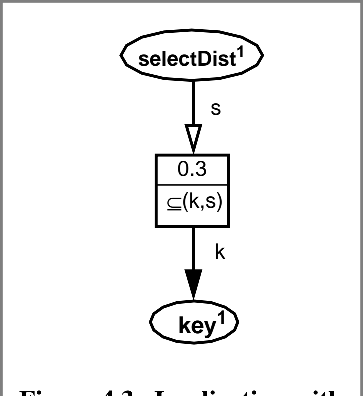


Figure 4.3. Implication with constraint and negation

$$f := \forall s \forall k\ ((k \subseteq s \rightarrow selectDist^1(s)) \rightarrow \neg key^1(k)).$$

conjunction

Logical conjunctions are represented in the GFRN formalism by connecting two or more predicates to the premise of an implication. An example for such a situation is given in Figure 4.4. The shown implication specifies the heuristic that an inclusion dependency (IND) can be classified as an R-IND if it is key-based, i.e., if there is a key constraint over the attribute set on its right-hand side. According to Definition 4.1, the signature of an IND $\Pi_{a_1, \ldots, a_n}\delta(R_l) \subseteq \Pi_{b_1, \ldots, b_n}\delta(R_r)$ is represented by a tuple *(l, r, i)* where *i* is a set of pairs of corresponding attributes, i.e., $i = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$. Operation $\Pi_2(i)$ applied in the constraint of the implication in Figure 4.4 represents the projection on the second component

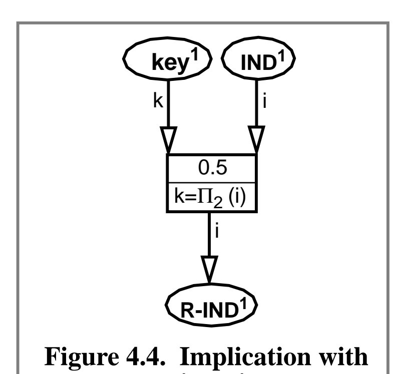


Figure 4.4. Implication with conjunction

of each tuple in a given relation i, i.e., $k=\Pi_2(i)=\{b_1,b_2,...,b_n\}$. Hence, k represents the set of attributes on the right-hand side of the IND which have to represent a key according to the second predicate in the implication's premise. The semantics of the GFRN in Figure 4.4 is defined by a formula $(f,0.5)\in \mathcal{L}\{NPL^1\}$, with

$$f:=\forall k\forall i\ ((k=\Pi_2(i)\rightarrow(key^1(k)\wedge IND^1(i))\rightarrow R\text{-}IND^1(i)).$$

thresholds

A problem that arises with the use of quantitative measures for uncertainty is that the inference might lead to a vast amount of hypotheses with a low certainty. For example, let us consider the heuristic that a key might be indicated by an attribute name that is similar to the name of its RS with the suffix "*id*" (cf. Example 3.1 on page 46). Using the *Levenshtein* distance [Lev66] to measure the similarity of strings we obtain the similarity measures displayed in Figure 4.5 for seven sample attribute names of table *USER* (cf. Example 3.1 on page 46). During manual analysis a reengineer would not consider an attribute like *dpt* with respect to the above heuristic, because the similarity of its name with the string "*userid*" is very low. Considering such indicators in the proposed automatic knowledge inference process would entail the generation of many false positives. Subsequently, the reengineer would have to validate each such hypotheses manually to obtain a definite analysis result. This contradicts to our goal to unburden the reengineer from stereotypical activities and focus her/his attention.

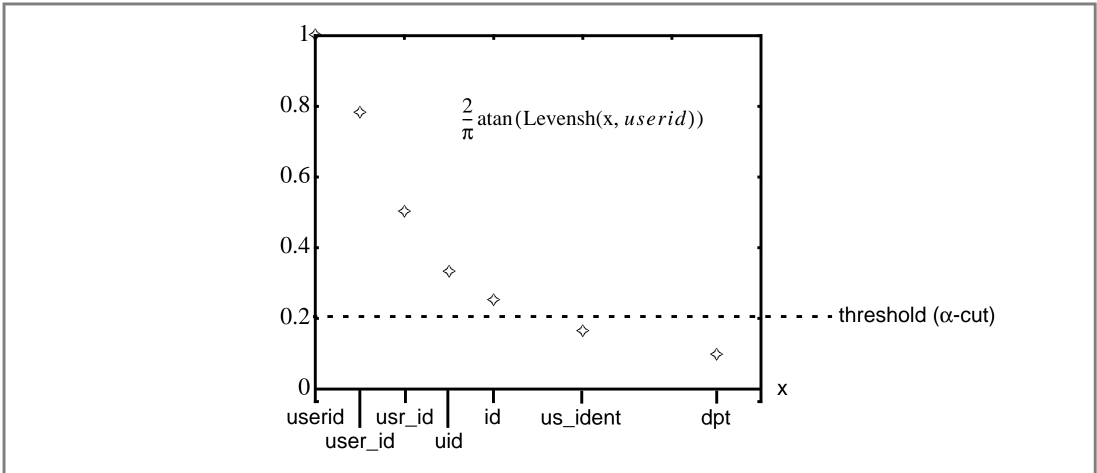


Figure 4.5. Similarity measures for the seven sample attribute names with the string *userid*.

In the GFRN approach, we allow to suppress incredible indicators by assigning a *threshold value* (TV) to each implication. A TV defines the minimum amount of certainty that is needed for a premise such that the corresponding implication is considered. The semantics of a TV is defined by an α-*cut* on the fuzzy set that represents the propositions in the premise of the corresponding implication. For example Figure 4.6 shows an implication that specifies the naming heuristic discussed above. It has an associated TV of 0.2 which is represented by another real number that is separated from the CV by a slash. The dashed line in Figure 4.5 illustrates how this threshold

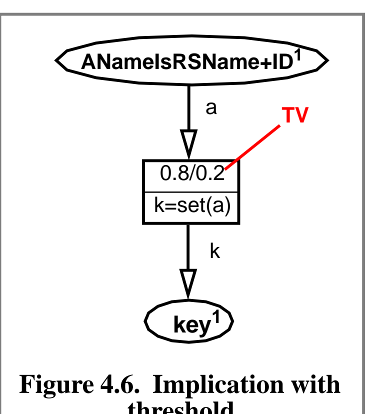


Figure 4.6. Implication with threshold

suppresses all propositions in the premise that have a certainty measure lower than 0.2. Furthermore, we have to consider that the naming heuristic uses names of *single* attributes as indicators for key constraints. However, key constraints are generally defined in *sets* of attributes. Hence, we have to restrict the hypothetical key *k* to be a set of only one attribute *a*. This restriction is represented by the constraint *k=set(a)*. The semantics of the GFRN in Figure 4.6 is defined by a formula $(f,0.8) \in \mathcal{L}\{NPL^1\}$, with

$$f := \forall a \forall k((k=set(a) \rightarrow ANameIsRSName{+}ID^1(a) \wedge N(ANameIsRSName{+}ID^1(a))>0.2) \rightarrow key^1(k)).$$

premise with inner universal quantifier

Some heuristics consider a set of indicators to infer new knowledge where the cardinality of this set depends on situation-specific knowledge. For example, in our case study, we applied a heuristic that uses an arbitrary set of pairs of similarly named attributes in two different RS as an indicator for a complex foreign key (IND) (cf. page 16). To be able to specify such heuristics we have to provide means to consider all situation-specific knowledge that fulfills certain specified constraints, i.e., we need an explicit notion of a universal quantifier within the premise of an implication. In the GFRN formalism, such an inner universal quantifier (IQ) is represented by an arc with a cancelled arrow head. Figure 4.7 shows an example that specifies the heuristic discussed in this paragraph. The predicate $NamSim^2$ is defined by the fuzzy set of pairs $p{:}(a,\overline{a})$ of attrib utes with similar names. The constraint $\in(p,i)$ restricts *p* to be a pair of corresponding attributes in the hypothetical IND $i{:}\{(a_1,\overline{a}_1),(a_2,\overline{a}_2),...,(a_n,\overline{a}_n)\}$. Furthermore, by using the constraint $disj(\Pi_1(s),\Pi_2(s))$ we restrict the left-hand side $\{a_1,a_2,...,a_n\}$ and the right-hand side $\{\overline{a}_1,\overline{a}_2,...,\overline{a}_n\}$ of the hypothetical IND to be disjoint. Finally, we require that at-a-time the left-hand side and the right-hand side of the concluded IND belong to one single RS (*sameRS(...)*). The semantics of the GFRN in Figure 4.7 is defined by a formula $(f,0.5) \in \mathcal{L}\{NPL^1\}$, with

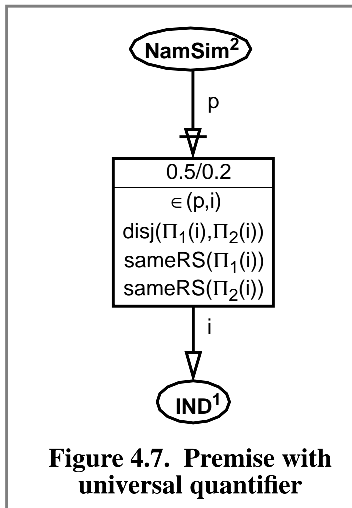


Figure 4.7. Premise with universal quantifier

$$f := \forall i(\forall p(p \in i \wedge disj(\Pi_1(i),\Pi_2(i)) \wedge sameRS(\Pi_1(i)) \wedge sameRS(\Pi_2(i)) \rightarrow NamSim^2(p) \wedge N(NamSim^2(p))>0.2) \rightarrow IND^1(i)).$$

The above examples show that compared with textual formulae the graphical GFRN formalism improves the understandability of specified knowledge, significantly. In the following examples, we will skip the translation of GFRN specifications to NPL^1 for the sake of readability. We will come back to this issue when we formally define the syntax and semantics of GFRN specifications in Section 4.2.3.

variable aggregation and composition

In the previous example, we already implicitly used the concept of *variable aggregation*: we used a single variable *p* to denote a tuple $(a,\overline{a})$ of attributes. In general, each arc in a GFRN is labeled either by a tuple of *n* variables which stands for the arguments of the connected *n*-ary predicate, or by a single variable that denotes the entire tuple. This notation can be used to aggregate variables as well as to compose new tuples. Figure 4.8 shows an application example that combines both techniques.

The left implication is a more sophisticated version of the implication in the previous example. While the implication in Figure 4.7 restricts the attributes on the left-hand side of the hypothetical IND to belong to the same RS, the left implication in Figure 4.8 strengthens this condition by restricting them to be in the same *variant*. This reflects on the experience with our DBRE case study which has shown that the extension of an RS might comprise different variants of tuples (cf. page 20). As a consequence, we have to extend the definition of predicate IND^2 by an additional parameter (v) which represents the variant for the IND's left-hand side. In the conclusion of the left implication in Figure 4.8 the argument of predicate IND^2 is composed by variables i and v. The implication on the right side of Figure 4.8 specifies the definite knowledge that a hypothetical IND can only by true if it is valid in the available data. Obviously, predicate $validIND^2$ has to be defined over the same formal parameters as predicate IND^2. However, for the right implication in Figure 4.8 we aggregate the pair of parameters in one variable (t).

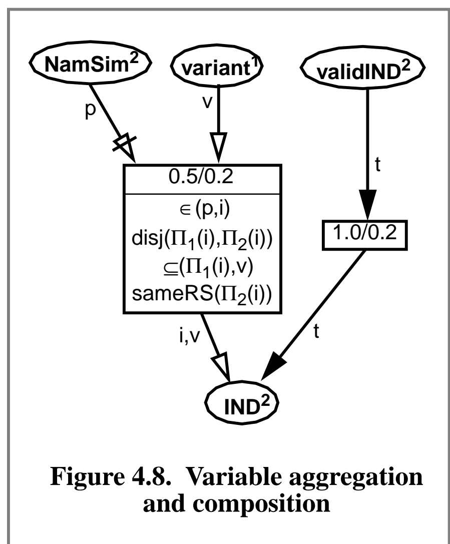


Figure 4.8. Variable aggregation and composition

Figure 4.9 combines the example heuristics discussed in this section in one single GFRN.

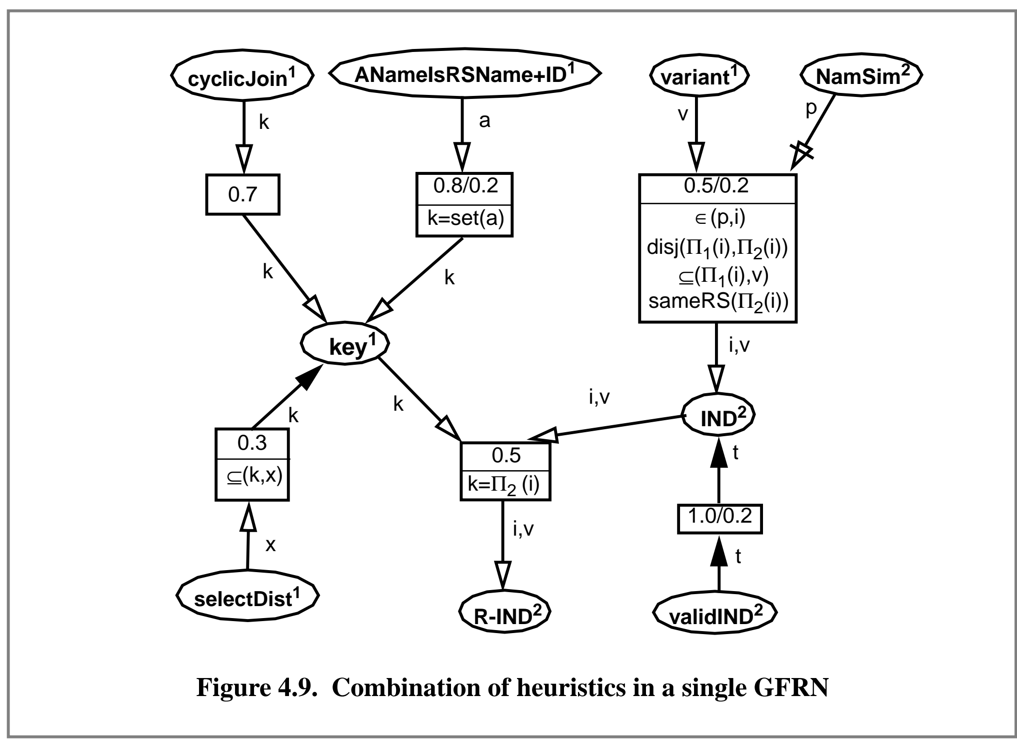


Figure 4.9. Combination of heuristics in a single GFRN

4.2.2 Integration of automatic analysis operations

The GFRN formalism described so far allows to define domain-specific heuristics to reason about situation-specific knowledge. If we want to employ this reasoning process in a semi-automatic schema analysis process (as described in Section 4.1) we have to provide means to integrate automatic analysis operations which retrieve situation-specific knowledge from the LDB.

existing operations

In the DBRE community, there is a great variety of programs and procedures that perform analysis of different parts of an LDB. For example, in [PB94], Premerlani and Blaha report on their experience in schema analysis using a simple but flexible tool set which mainly contains UNIX tools [RRF90] like *grep* and *awk* . Anderson [And94] defines a number of recurring patterns in the procedural LDB code that can be used as semantic indicators. In [Bew98], Bewermeyer extends this collection of patterns and employs graph grammars to recognize them in an abstract syntax graph representation. Petit et.al. [PKBT94] describe specific database queries that can be used to extract important information from the available legacy data.

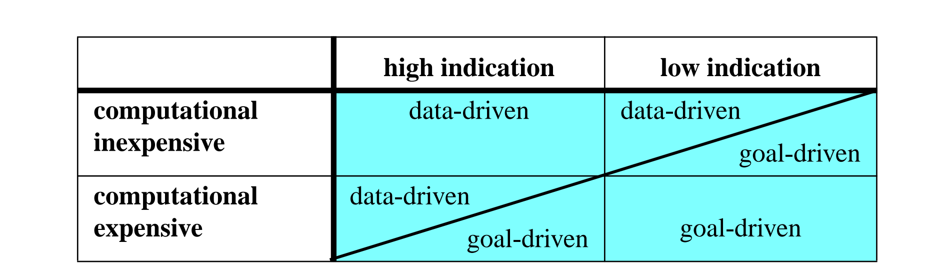
In this section, we describe how such existing operations can be integrated with the GFRN formalism to achieve the desired knowledge-driven analysis process. In Section 4.1, we have distinguished between two kinds of analysis operations: (1) operations that perform an initial analysis of the LDB, and (2) operations which are only executed on-demand to refute or support intermediate hypotheses. Let us revisit an example from our case study to motivate this distinction.

In Section 2.4.1, we have exemplified how indicators for foreign key constraints can be found by employing heuristics about naming conventions of LDB schema components. One of these heuristics searches the schema for pairs of RS that have (groups of) attributes with similar names (cf. page 16). If such a situation can be found in an LDB schema, our heuristic leads to an uncertain hypothesis that there might be a foreign key constraint between the two RS. However, such a foreign key might only exist if the corresponding IND is valid in the available data. Using the GFRN formalism we can specify this knowledge as shown in Figure 4.9. Both fuzzy sets that define the predicates $validIND^2$ and $NamSim^2$ can be determined by automatic analysis operations: the validity of INDs can be checked by predefined queries to the data and string similarity measures can be used to check the schema for naming conventions. Still, there is a qualitative difference between both predicates. While predicate $NamSim^2$ serves to *indicate* a semantic constraint, predicate $validIND^2$ is used to *validate* this indication. Hence, the validity of a hypothetical IND should only be checked when it has actually been indicated. Another rational for such a *goal-driven analysis* is that the computational effort which was involved in checking the validity of all possible combinations of INDs *beforehand* would grow exponentially with the size of the LDB's schema. Hence, this solution would contradict to our requirement for scalability (cf. *R6* on page 37).

data- and goal-driven operations

According to the above motivation, we classify automatic analysis operations as either *data-driven*, i.e., they are executed *before* the inference process starts to provide an initial set of indicators, or *goal-driven*, i.e., they are invoked on demand *during* the inference process. This classification can be performed according to the guidelines displayed in Figure 4.10. If an analysis operation delivers facts about an LDB that represent valuable indicators for semantic

constraints and this operation is computational inexpensive, then it should be classified as *data-driven*. On the other hand, if an operation delivers facts that are less suitable as indicators and this operation is computational expensive, then it should be classified as *goal-driven*. The classification of other analysis operations depends on the application context, i.e., on the concrete LDB under investigation. For example, an analysis operation that delivers valuable indicators but is computational expensive can be classified as *data-driven* for a small-scale LDB, but it should be classified as *goal-driven* if the LDB has a large scale.



	high indication	**low indication**
computational inexpensive	data-driven	data-driven / goal-driven
computational expensive	data-driven / goal-driven	goal-driven

Figure 4.10. Characteristics for classifying automatic analysis operations

different types of predicates

In the GFRN approach, predicates can be bound to data- and goal-driven operations. Consequently, such predicates are called *data-driven* or *goal-driven*, respectively. Predicates that are not bound to analysis operations are called *dependent*. Figure 4.11 shows that data- and goal-driven predicates are represented as bold ovals with different colors, where black means *data-driven* and grey stands for *goal-driven*. Furthermore, the left-most implication in Figure 4.11 exemplifies that the application of goal-driven predicates is not limited to the purpose of *refuting* hypotheses: the validity of a hypothetical IND in a large amount of data delivers a good *support* that this hypothesis is true. Still, hypotheses cannot be proved by means of data. Hence, we attached a CV lower than 1 to this implication.

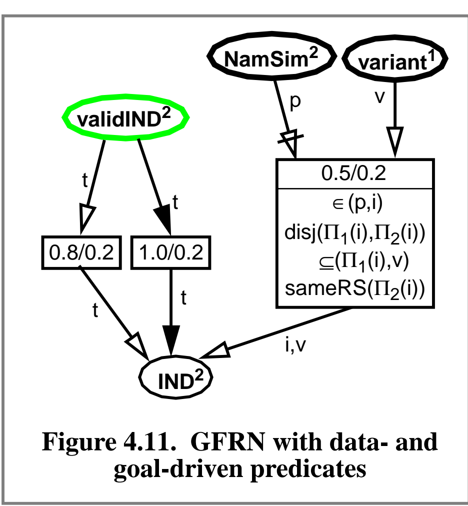


Figure 4.11. GFRN with data- and goal-driven predicates

Figure 4.12 displays an example for a goal-driven analysis operation named *validate_IND* which can be bound to predicate $validIND^2$ in Figure 4.11. The first argument of operation *validate_IND* (*B*) represents the LDB which is the current target of the analysis. Note, that in contrast to the other two arguments, parameter *B* is not represented explicitly in the GFRN. Operation *validate_IND* returns a degree of necessity for and against the proposition that the corresponding IND holds in *B*. The algorithm uses a local variable ψ to store all tuples that belong to variant *v* on the left side of the IND *i*. If these tuples contain no counterexample for the hypothetical IND the necessity of $validIND^2(i,v)$ is computed depending on the cardinality of ψ. A large amount of data entails a higher support for the hypothesis than just a few tuples. The generated membership function is illustrated in Figure 4.13. Otherwise, if a counterexample can be found, the hypothesis is refuted. Note, that we have presumed *correct*

legacy data for the definition of this analysis operation. If we expect the data to include some *incorrect* tuples, i.e., tuples that do not comply to INDs even if they exist, we should choose an analysis operation that *gradually* refutes IND hypotheses according to the relative number of contradicting tuples.

The purpose of the pseudo code in Figure 4.12 is simply to introduce our concept of integrating automatic analysis operations with knowledge represented in GFRN specifications. We employ the programming language *Java* for concrete implementations of goal- and data-driven analysis operations. This issue will be discussed in Section 4.4.

Operation validate_IND (B, i, v)
input: B *(* B is an RDB according to Definition 3.3 *)*
i:$(\{(a_1,\bar{a}_1),(a_2,\bar{a}_2),\ldots,(a_m,\bar{a}_m)\})$, v
($(v, RS(\bar{a}_1), i)$ is an IND signature w.r.t. Definition 4.1 *)*
output: $N(validIND^2(i,v)) \in [0,1]$, $N(\neg validIND^2(i,v)) \in [0,1]$
begin
let $r_1 = RS(a_1,\ldots,a_n)$;
let $r_2 = RS(\bar{a}_1,\ldots,\bar{a}_n)$;
let $\psi = \{x \in \delta(r_1) \mid \forall a \in X(r_1) : (a \in v \rightarrow \Pi_a(x) \neq NULL) \wedge (a \notin v \rightarrow \Pi_a(x) = NULL)\}$
(ψ represents all members of variant v *)*
if $\Pi_{a_1,\ldots,a_n}\psi \subseteq \Pi_{\bar{a}_1,\ldots,\bar{a}_n}\delta(r_1)$ *(* is the IND valid? *)*
then let $N(validIND^2(i,v)) = \frac{2}{\pi}\operatorname{atan}\left(\frac{|\psi|}{100}\right)$
let $N(\neg validIND^2(i,v)) = 0$
else let $N(validIND^2(i,v)) = 0$
let $N(\neg validIND^2(i,v)) = 1$
end.

Figure 4.12. Goal-driven analysis operation *validate_IND*

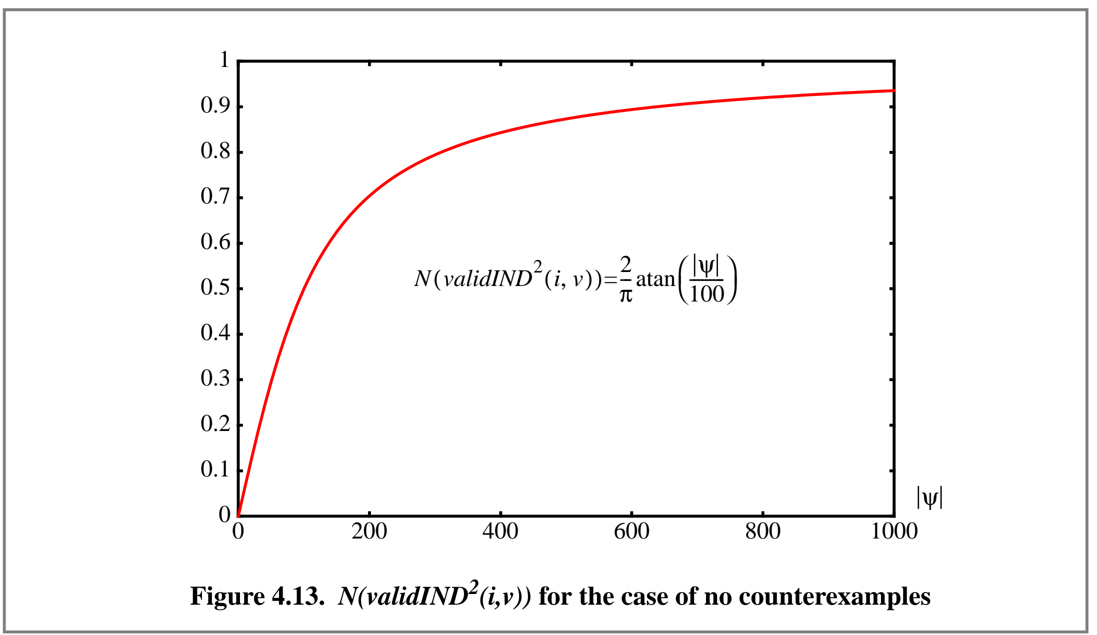


Figure 4.13. *N(validIND²(i,v))* for the case of no counterexamples

4.2.3 Formal definition

In the previous sections, we have informally introduced and exemplified GFRNs as a formalism to specify DBRE knowledge and semi-automatic analysis processes. In the following, we will give a formal definition of the syntax and semantics of this language.

4.2.3.1 Syntax of GFRN

In this section, we formalize the syntax of GFRN specifications by defining their signatures and a set of context sensitive constraints.

Definition 4.2 Signature of a GFRN

*A **generic fuzzy reasoning net** is defined by a 9-tuple* $GFRN:=(P, F^r, F^b, I, E, cf, th, \Omega, \omega)$,

- $P=(P^d,P^g,P^t)$, *with* $P^d \cup P^g \cup P^t = \{p_1^{u_1}, p_2^{u_2}, ..., p_x^{u_x}\}$, $x \in \mathbb{N}$, *a finite set of unique **predicate symbols** with arity* $u_q \in \mathbb{N}$, $q \in [1,x]$, *the disjoint sets* P^d,P^g,P^t *are called data-driven, goal-driven, and dependent predicates, respectively.*
- $F^r = \{f_1^{u_1}, f_2^{u_2}, ..., f_x^{u_x}\}$, $x \in \mathbb{N}$ *is a finite set of **relational function symbols** with arity* $u_q \in \mathbb{N}$, $q \in [1,x]$, *i.e., each* $f_q^{u_q} \in F^r$ *denotes a function* $f_q^{u_q} : \mathcal{U}^{u_q} \rightarrow \mathcal{U}^1$.
- $F^b = \{f_1^{u_1}, f_2^{u_1}, ..., f_x^{u_y}\}$, $x \in \mathbb{N}$ *is a finite set of **boolean function symbols** with arity* $u_q \in \mathbb{N}$, $q \in [1,x]$, *i.e., each* $f_q^{u_q} \in F^b$ *denotes a function* $f_q^{u_q} : \mathcal{U}^{u_q} \rightarrow \{True,False\}$.
  The boolean function symbol '$\in^2$' $\in F^b$ *is predefined.*
- $I=\{i_1, i_2, ..., i_x\}$, $x \in \mathbb{N}$, *is a finite set of **implications,** each implication* $i \in I$ *is a tupel* $i = (\iota, V, K)$, *with*
  - ι, *an unique **implication identifier**,*
  - $V=\{v_1, v_2, ..., v_x\}$, $x \in \mathbb{N}$, *a set of **parameter names**,*
  - $K=\{k_1,k_2,...,k_x\}$, $x \in \mathbb{N}$, *is a finite set of **constraints** over V, where each* $k \in K$ *has the form* $k=(w, f^u, <w_1,w_2,...,w_u>)$, *with* $w_1,..,w_u \in V$, $(w \in V \wedge f^u \in F^r) \vee (w=\varepsilon \wedge f^u \in F^b)$.
- $E=\{e_1,e_2,...,e_n\}$, $n \in \mathbb{N}^+$ *is a finite set of **arcs**, where each* $e \in E$ *is a tupel* $e=(\chi, l, s, d, \mathrm{A})$, *with*
  - χ *an unique **arc identifier**,*
  - $l:(p,(\iota,V,K)) \in (P \times I)$, *a **location**,*
  - $s \in \{\text{‘ ’}, \neg\}$, *a **sign**,*
  - $d \in$ *{premise, premise_quantified, conclusion}, a **type**; ‘premise’ and ‘premise_quantified’ means that the arc is in the premise of the connected implication, ‘premise_quantified’ denotes an arc with a variable that has been quantified with an IQ, ‘conclusion’ denotes an arc in the conclusion of the corresponding implication.*
  - $\mathrm{A}=\alpha$ *or* $\mathrm{A}=<\alpha_1, \alpha_2, ..., \alpha_{kq}>$ *an **actualization vector**, with* $\alpha,\alpha_u \in V$, *for* $1 \leq u \leq kq$.
- $cf: I \rightarrow (0, 1]$ *and* $th: I \rightarrow [0, 1)$ *are functions that associate integer values between 0 and 1 to implications. cf is called the **confidence function** while the th is called the **threshold function**.*
- $\Omega: P^d \rightarrow \overline{FUN}$ *and* $\omega: P^g \rightarrow \overline{FUN}$, *are two functions that associate **analysis operations** to data- and goal-driven predicates.*

◻

We define the following *context-sensitive* constraints on GFRN signatures in order to ensure their executability and simplify the formulation of the inference and translation algorithms in the following sections. We will denote a GFRN that complies to the following constraints as a *well-formed* GFRN.

Definition 4.3 Context sensitive syntax

A GFRN $((P^d,P^g,P^t),F^r,F^b,I,E,cf,th,\Omega,\omega)$ *is called* ***well-formed*** *if it admits to the following syntactic constraints:*

- *predicates are not isolated, i.e.,*

$$\forall p\in P^d\cup P^g\cup P^t\ \exists(\chi,(p,i),s,d,A)\in E)\in E$$

- *implications have at least one predicate in their premise and exactly one predicate in their conclusion, i.e.,*

$$\forall i\in I\ \exists(\chi,(p,i),s,d,A)\in E\ \exists!(\bar{\chi},(\bar{p},i),\bar{s},conclusion,\bar{A})\in E\ d\in\{'premise','premise_quantified'\}$$

- *data- or goal-driven predicates do not occur in the conclusion of any implication, i.e.,*

$$\neg\exists(\chi,(p,i),s,conclusion,A)\in E\ p\in P^d\cup P^g$$

- *all variables of all implications are actualized, i.e.,*

$$\forall i:(\iota,V,K)\in I\ \forall v\in V\ \exists(\chi,(p,i),s,d,A)\in E\ v\in A$$

- *IQs can only be used for single variable names, i.e.,*

$$\forall(\chi,l,s,premise_quantified,<\alpha_1,\ldots,\alpha_{kq}>)\in E\ \ kq=1$$

- *for each implication, there is at most one variable which is bound by an IQ, i.e.,*

$$\forall i\in I(\exists(\chi,(p,i),s,premise_quantified,a),(\bar{\chi},(\bar{p},i),\bar{s},premise_quantified,\bar{a})\in E\rightarrow\chi=\bar{\chi})$$

D

Example 4.1 Syntax of a GFRN

Figure 4.14 shows an example GFRN that consists of five implications and six predicates, including two data-driven and one goal-driven predicates.

According to Definition 4.2, the signature of the depicted GFRN is defined by a tuple $G:(P,F^r,F^b,I,E,cf,th,\Omega,\omega)$, with

- *predicate symbols* $P=(P^d,P^g,P^t)$*, with*
  - *data-driven predicates* $P^d=\{selectDist^1, ANameIsRSName{+}ID^1\}$,
  - *goal-driven predicates* $P^g=\{validKey^1\}$,
  - *dependent predicates* $P^t=\{IND^2, I\text{-}IND^2, key^1\}$,
- *relational function symbols* $F^r=\{\Pi_1{}^1,\Pi_2{}^1, set^1\}$,
- *boolean function symbols* $F^b=\{\in^2,\subseteq^2\}$,
- *implications* $I=\{ (\iota_1,\{s,k\}, \{(\varepsilon, \subseteq^2,<k,s>)\}), (\iota_2,\{k,a\}, \{(k,set^1,<a>)\}), (\iota_3,\{t\},\{\}), (\iota_4,\{t\},\{\}),$
  $(\iota_5,\{i,v,k_1,k_2\}, \{(k_1, \Pi_1{}^1,<i>), (k_2, \Pi_2{}^1,<i>)\}) \}$,
- *edges* $E=\{ (e_1, (selectDist^1,\iota_1),$ *'', premise,* $<s>), (e_2, (key^1,\iota_1), \neg,$ *conclusion,* $<k>),$
  $(e_3, (ANameIsRSName{+}ID^1,\iota_2),$ *'', premise,*$<a>), (e_4, (key^1,\iota_2),$ *'', conclusion,*$<k>),$
  $(e_5, (key^1,\iota_5),$ *'', premise,* $<k_1>), (e_6, (key^1,\iota_5),$ *'', premise,* $<k_2>),$

(e_7, (IND^2,ι_5), ", premise, <i,v>), (e_8, ($I\text{-}IND^2$,ι_5), ", conclusion, <i,v>), (e_9, (key^1,ι_3), ¬, conclusion,<t>), (e_{10}, ($valid_key^1$,ι_3), ¬, premise,<t>), (e_{11}, (key^1,ι_4), ", conclusion,<t>), (e_{12}, ($valid_key^1$,ι_4), ", premise,<t>),

- *confidence function* $cf(\iota_1)=0.3$, $cf(\iota_2)=0.8$, $cf(\iota_3)=1$, $cf(\iota_4)=0.8$, $cf(\iota_5)=0.6$,
- *threshold function* $th(\iota_1)=0$, $th(\iota_2)=0.2$, $th(\iota_3)=0.2$, $th(\iota_4)=0.2$, $th(\iota_5)=0$,
- *analysis operations* $\Omega(ANameIsRSName{+}ID^1)$, $\Omega(selectDist^1)$, *and* $\omega(validKey^1)$.

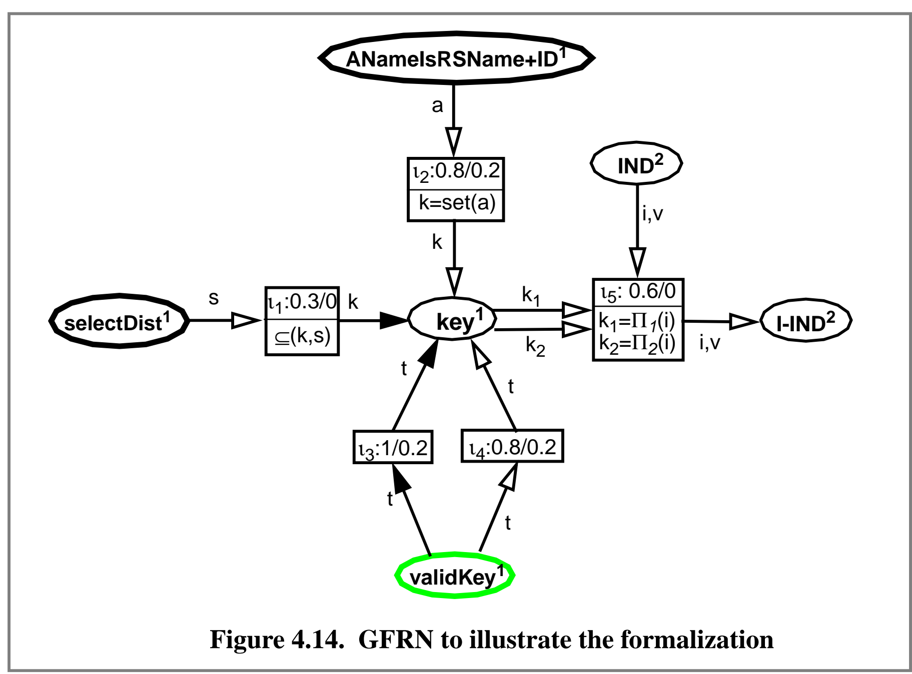


Figure 4.14. GFRN to illustrate the formalization

4.2.3.2 Declarative semantics

In Section 4.3, we will use algorithmic notation to define an inference and execution mechanism for GFRN specifications based on a *fuzzy Petri net* model. However, the level of abstraction of this *operational* definition of the GFRN semantics is too low to facilitate sound understanding of the meaning of GFRN specifications. Therefore, this section contains a declarative definition of the GFRN semantics based on a canonical translation of GFRN signatures to closed formulae in possibilistic logic. Subsequently, we formalize the semantics of integrating automatic analysis operations with data- and goal-driven predicates in the framework of this translation.

Definition 4.4 Declarative semantics of GFRNs

The declarative semantics of a well-formed GFRN $G:=(P,F^r,F^b,I,A,cf,th,\Omega,\omega)$ *is formally defined by a canonical translation of G to* NPL^1*. The translation algorithm is given in Figure 4.15 and Figure 4.16.*

algorithm *GFRN2NPL*^{1}

1) **input** $G:(P, F^r, F^b, I, E, cf, th, \Omega, \omega) \in \mathcal{L}\{GFRN\}$
2) **output** $\mathcal{F} \in \mathcal{L}\{NPL^1\}$
3) **local variables** $\mathcal{F} \in \mathcal{L}\{NPL^1\}, i \in I$
4) **begin**
5) **let** $\mathcal{F}$ = „ $\models$ “
6) **for each** $i \in I$ **do**
7) **let** $\mathcal{F}$= $\mathcal{F}$ „$\wedge$“ $Impl2NPL^1(G, i)$
8) **od**
9) **return** $\mathcal{F}$
10) **end**

Figure 4.15. Translation algorithm *GFRN2NPL*^{1}

algorithm explanation

Algorithm $GFRN2NPL^1$ in Figure 4.15 takes the signature of a well-formed GFRN G as input parameter and produces a closed formula $\mathcal{F}$ in NPL^1 as an output parameter. $\mathcal{F}$ is initialized by the tautology. For each implication i in G, $GFRN2NPL^1$ calls the algorithm $Impl2NPL^1$ in Figure 4.16, that creates a closed formula in NPL^1 representing the semantics of i. The semantics of the *entire* GFRN is defined as the logical conjunction of the translation of all its implications.

Algorithm $Impl2NPL^1$ uses five auxiliary variables ($\mathcal{F}_1$-$\mathcal{F}_5$) of type *String* to create the desired NPL^1 formula ($\mathcal{F}$). Strings (and formulae) are concatenated by using the assignment operation *let*, e.g., line 20 in Figure 4.16. Characters enclosed by quotes („“) are taken literally, while strings which are not enclosed by quotes have to be variables. Variables (like $\mathcal{F}_1$ and v in line 20) are evaluated and their current value is taken for the assignment operation.

If there exists an IQ in the premise of the current implication, the statement in line 20 creates a universal quantifier for the corresponding parameter tuple in variable $\mathcal{F}_2$. Likewise, the first loop uses $\mathcal{F}_1$ to store "outer" universal quantifications for all remaining variables of i. The second loop creates a string ($\mathcal{F}_3$) that represents a logical conjunction of all constraints of i. The last loop (lines 39-41) creates a string ($\mathcal{F}_4$) that represents a logical conjunction of all predicates in the antecedent of i, while the assignment in line 45 creates a string ($\mathcal{F}_5$) that represents the predicate in the consequent of i. Finally, the assignment operation in line 47 creates the resulting formula in NPL^1 that represents the semantics of i. We assign the identifier of the translated implication (ι) as an index to the implication operator ($\rightarrow_\iota$) to facilitate identification of the original GFRN implication. However, there is no additional semantics to this index.

```
algorithm Impl2NPL¹(G, i)
11)  input  G:=(P, Fʳ, Fᵇ, I, E, cf, th, Ω, ω) ∈ L{GFRN}, i:(ι,V, K) ∈ I
12)  output 𝓕 ∈ L{NPL¹}
13)  local variables 𝓕₁,𝓕₂,𝓕₃,𝓕₄,𝓕₅ ∈ String; e ∈ E; v,vᵢ ∈ V; V_c ⊆ V
14)  begin
15)      let 𝓕₁ = 𝓕₂ = 𝓕₅ = „“
16)      let 𝓕₃ = 𝓕₄ = „ ⊨“
17)
18)      // create „inner“ univ. quantifier (IQ)
19)      if ∃(χ,l, s, premise_quantified, vᵢ)∈ E
20)          then let 𝓕₂ = 𝓕₂ „∀“ vᵢ
21)          let V=V\vᵢ
22)      fi
23)
24)      // create „outer“ univ. quantifiers for all remaining variables
25)      for each v ∈ V do
26)          let 𝓕₁ = 𝓕₁ „∀“ v
27)      od
28)
29)      // create constraints
30)      for each (w,f^u, <w₁,..,w_u>) ∈ K do
31)          if w=ε then
32)              let 𝓕₃ = 𝓕₃ „∧“ f^u „(“w₁,..,w_u „)“
33)          else
34)              let 𝓕₃ = 𝓕₃ „∧“ w „=“ f^u „(“w₁,..,w_u „)“
35)          fi
36)      od
37)
38)      // create predicates in premise
39)      for each ((p^m,i),s,t,A)∈ E with t='premise' or t='premise_quantified' do
40)          let 𝓕₄ = 𝓕₄ „∧“ s p^m „(“ A „)“
41)      od
42)
43)      // create predicate in conclusion
44)      let ((p^m,i),s,conclusion, A)∈ E
45)      let 𝓕₅ = s p^m „(“ A „)“
46)
47)      let 𝓕 = „(“ 𝓕₁ „(“ 𝓕₂ „(“ 𝓕₃ „→“𝓕₄ „∧N(“𝓕₄ „)≥ “th(i) „)→ι“ 𝓕₅ „),“ cf(i) „)“
48)  return 𝓕
49)  end
```

Figure 4.16. Translation algorithm *Impl2NPL¹*

Example 4.2 Translation of GFRN to *NPL¹*

In this example, we revisit our sample GFRN in Figure 4.14 on page 68 to illustrate our translation algorithm in Figure 4.15 and Figure 4.16. The signature of this GFRN is presented in the previous example (4.1). It contains five implications, hence *Impl2NPL¹* is invoked five times. In the following, we select implication ι_5 to discuss one of these invocations in detail.

At first, the five auxiliary variables are initialized ($\mathcal{F}_1$-$\mathcal{F}_5$). Variable $\mathcal{F}_2$ remains empty because there is no IQ in the premise of implication ι_5. The first loop creates quantifiers for all parameters of ι_5, i.e., $\mathcal{F}_1$ = „$\forall i \forall v \forall k_1 \forall k_2$". The second loop considers the constraints of ι_5, i.e., $\mathcal{F}_3$ = „$\models \wedge\, k_1{=}\Pi_1(i) \wedge k_2{=}\Pi_2(i)$". In lines 39-45, variables $\mathcal{F}_4$ and $\mathcal{F}_5$ are defined to represent the predicates in the premise and the conclusion of ι_5, respectively. After this section, $\mathcal{F}_4$ has the value „$\models \wedge\, key^1(k_1) \wedge key^1(k_2) \wedge IND^2(i,v)$" and the value of $\mathcal{F}_5$ is „$I\text{-}IND^2(i,v)$". Finally, the translation of ι_5 to NPL^1 is created in line 47 as $Impl2NPL^1(\iota_5)=(f,0.6)$ with

$$f{=}\forall i \forall v \forall k_1 \forall k_2(\models \wedge k_1{=}\Pi_1(i) \wedge k_2{=}\Pi_2(i) \rightarrow \models \wedge key^1(k_1) \wedge key^1(k_2) \wedge IND^2(i,v) \wedge N(\models \wedge key^1(k_1) \wedge key^1(k_2) \wedge IND^2(i,v)) \geq 0) \rightarrow_5 I\text{-}IND^2(i,v).$$

The resulting formula can be simplified by pruning unnecessary brackets, conjunctions with tautologies, and preconditions due to thresholds which are equal zero (see below). The four other implications are translated likewise and the semantics of this sample GFRN is defined by the following formula in NPL^1.

$GFRN2NPL^1(G)=$

$$Impl2NPL^1(\iota_1)= (\forall s \forall k((k \subseteq s \rightarrow selectDist^1(s)) \rightarrow_1 \neg key^1(k)),0.3)$$

$$\wedge\ Impl2NPL^1(\iota_2)=(\forall t(ANameIsRSName{+}ID^1(t) \wedge N(ANameIsRSName{+}ID^1(t)) \geq 0.2 \rightarrow_2 key^1(t)),0.8)$$

$$\wedge\ Impl2NPL^1(\iota_3)=(\forall t(\neg validKey^1(t) \wedge N(\neg validKey^1(t)) \geq 0.2 \rightarrow_3 \neg key^1(t)),1)$$

$$\wedge\ Impl2NPL^1(\iota_4)=(\forall t(validKey^1(t) \wedge N(validKey^1(t)) \geq 0.2 \rightarrow_4 key^1(t)),0.8)$$

$$\wedge\ Impl2NPL^1(\iota_5)=(\forall i \forall v \forall k_1 \forall k_2((k_1{=}\Pi_1(i) \wedge k_2{=}\Pi_2(i)) \rightarrow (key^1(k_1) \wedge key^1(k_2) \wedge IND^2(i,v)) \rightarrow_5 I\text{-}IND^2(i,v)),0.6)$$

E

semantics of analysis operations

In the rest of this section, we formalize the semantics of automatic data- and goal analysis operations which have been attached to GFRN predicates. In Section 4.2.2, we have exemplified that automatic analysis operations deliver situation-specific facts about the LDB that are associated with degrees of necessity. The facts delivered by automatic analysis operations which have been bound to GFRN predicates represent applications of these predicates. Hence, we denote that these facts are in the *extent* of the corresponding predicates.

Definition 4.5 Extent of a predicate

For a given universe $\mathcal{U}$ the ***extent*** *of a possibilistic predicate p, denoted as $\langle p \rangle_{\mathcal{U}}$, is defined by the set of propositions $\langle p \rangle_{\mathcal{U}}=\{(p(u),x) | u \in \mathcal{U}, x \in [0,1]\} \subset \mathcal{L}\{NPL^0\}$.*

D

The concept of data- and goal-driven analysis functions is formalized as follows.

Definition 4.6 Data-driven analysis operation

For a given data-driven predicate $p \in P^d$ the associated ***data-driven analysis operation*** *$\Omega(p)$ is defined by a function $\Omega(p){:}\overline{RDB} \rightarrow \mathcal{P}(\langle p \rangle_{\mathcal{U}})$.*

D

Definition 4.7 Goal-driven analysis operation

For a given goal-driven predicate $p \in P^g$ the associated ***goal-driven analysis operation*** *$\omega(p)$ is defined by a function $\omega(p){:}\overline{RDB} \times \langle p \rangle_{\mathcal{U}} \rightarrow \langle p \rangle_{\mathcal{U}} \times \langle \neg p \rangle_{\mathcal{U}}$.*

D

The LDB which is under investigation defines a finite universe of discourse which we will call the *application context* of the LDB.

Definition 4.8 Application context

The ***application context*** $\mathcal{U}(B)$ *of a given LDB* $B{:}(M,S,\delta,C,D)\in\overline{RDB}$ *is defined by the finite power set of all software artifacts of B, i.e.,* $\mathcal{U}(B)=\mathcal{P}(flatten((M,S,\delta,C,D)))$.[a]

◻

In the following, we make use of the fact that a set of formulae in L^1 which is applied in a finite domain can be represented by an equivalent set of formulae in L^0 [BC90, pp. 35ff]. This is done by expressing each universal quantifier by a conjunction and each existential quantifier by a disjunction of propositions.

Definition 4.9 Expansion of formulae over a finite universe

Let $\Phi\subset\mathcal{L}\{NPL^1\}$ *be a set of closed formulae where all variables are bound with the universal quantifier. For a finite universe* $\mathcal{U}$*, let* $\Phi\Downarrow^{\mathcal{U}}\subset\mathcal{L}\{NPL^0\}$ *denote the* ***expansion*** *of* Φ *over* $\mathcal{U}$ *which represents an equivalent set of formulae where all quantifiers have been eliminated by using conjunctions, i.e.,*

$$\Phi\Downarrow^{\mathcal{U}}=\bigcup_{(f,\beta)\in\Phi}\{(g_i,\beta)/\,(\exists g_1,..,g_n\in\mathcal{L}\{NPL^0\})(f\equiv g_1\wedge,..,\wedge g_n \text{ in } \mathcal{U})\wedge i\in[1,n])\}.$$

◻

Definition 4.10 Occurrence of literals

Let $f\in\mathcal{L}\{L^0\}$ *be a propositional formula and let* $l\in\mathcal{L}\{L^0\}$ *be a literal. We denote occ(f,l) iff l* ***occurs*** *in f as a positive literal and we denote* $occ^{\neg}(f,l)$ *iff l occurs in f as a negative literal.*

◻

Now, we have the prerequisites to formalize the semantics of automatic analysis operations in GFRN specifications.

Definition 4.11 Semantics of automatic analysis operations

The semantics of a GFRN specification is defined by the algorithm OperateGFRN which is presented in Figure 4.17. OperateGFRN takes a GFRN and an RDB as its arguments and returns a consistent set of definite propositions about the RDB.

◻

algorithm explanation

Algorithm *OperateGFRN* uses a local variable (*exec*) that is a two dimensional array of boolean values which are initialized to *FALSE*. This array maintains information about which goal-driven analysis operations have already been applied. In line 5, algorithm *GFRN2NPL*^{1} is called to translate the passed GFRN to a set of formulae Φ in NPL^1. Then all data-driven analysis operations are executed on the RDB *B* and the resulting propositions are added to Φ (lines 7-9). The condition in lines 13-15 checks for the existence of an implication rule $(f_1\rightarrow_i f_2,\beta)$ in the expansion $\overline{\Phi}$ of Φ over the universe $\mathcal{U}(B)$ that represents the translation of an implication *i* in the GFRN. Furthermore, the condition requires that an instance of a goal-driven predicate *p(u)* occurs in the premise (f_1) of this rule and that its conclusion (f_2) can be deduced from $\overline{\Phi}$ with a necessity higher than the threshold of *i*. If this condition is fulfilled and

a For the definition of function *flatten* see Definition 3.5 on page 38.

the goal-driven analysis operation for *p(u)* has not yet been executed (*exec(p,u)=FALSE*) the corresponding operation is invoked and the results of the operation is added to Φ (line 19). Subsequently, the value of *exec(p,u)* is set to *TRUE* to avoid that the same goal-driven analysis operation is executed twice. Lines 23-26 consider hypotheses and definite facts entered by the reengineer.

algorithm *OperateGFRN(G, B)*
1) **input** $G:=((P^d,P^g,P^t),F^r,F^b,I,E,cf,th,\Omega,\omega) \in \mathcal{L}\{GFRN\}, B \in \overline{RDB}$
2) **output** $\overline{\mathcal{F}} \subset \mathcal{L}\{L^0\}$
3) **local variables** $\Phi \subset \mathcal{L}\{NPL^1\}$, $exec[p \in P^g, u \in \mathcal{U}(B)]:BOOLEAN=FALSE$
4) **begin**
5) **let** $\Phi = GFRN2NPL^1(G)$
6) *// execute data-driven analysis operations*
7) **for each** $p \in P^d$ **do**
8) **let** $\Phi = \Phi \cup \Omega(p)(B)$
9) **end**
10)
11) **loop**
12) **let** $\overline{\Phi} = \Phi \Downarrow^{\mathcal{U}(B)}$
13) **if** $(\exists (f_1 \rightarrow_i f_2, \beta) \in \overline{\Phi})\ (\exists p \in P^g)\ (\exists u \in \mathcal{U}(B))\ (\exists \gamma \in [th(i),1])$
14) $(occ(f_1,p(u))) \wedge \overline{\Phi} \vdash (f_2,\gamma))$ *// p(u) in the antecedent of an implication that implies a credible hypotheses */*
15) **then**
16) **if** $exec[p,u]=FALSE$
17) **then**
18) *// execute goal-driven analysis operations*
19) **let** $\Phi = \Phi \cup \omega(p)(B,p(u))$
20) **let** $exec[p,u]=TRUE$
21) **fi**
22) **fi**
23) **if** *exists user input* $\varphi \subset \mathcal{L}\{NPL^0\}$
24) **then**
25) **let** $\Phi = \Phi \cup \varphi$
26) **fi**
27) **until** *a definite analysis results is obtained, i.e.,*
28) $\neg(\exists p \in P^t)(\exists u \in \mathcal{U}(B))$
29) $((\overline{\Phi} \vdash (p(u),\gamma) \wedge \gamma \in (0,1) \wedge \overline{\Phi} \vdash (\neg p(u),\bar{\gamma}) \wedge \bar{\gamma} \neq 1) \vee (\overline{\Phi} \vdash (p(u),\gamma) \wedge \gamma = 1 \wedge \overline{\Phi} \vdash (\neg p(u),\bar{\gamma}) \wedge \bar{\gamma} = 1))$
30) **return** $\{f \mid (f,1) \in \Phi \wedge f \in \mathcal{L}\{L^0\}\}$
31) **end**

Figure 4.17. Algorithm *OperateGFRN*

The loop from line 11 to 29 is iterated until a definite analysis result is obtained. This condition is reached when each instance of a dependent predicate $\overline{\Phi} \vdash p(u)$ with a positi ve necessity degree is either necessarily true or false with a necessity degree of 1. Consequently, we take a necessity degree of 1 as a modal operator that overrules partial inconsistency, i.e., if $N(p(u))=1$ we ignore $N(\neg p(u))<1$ and vice-versa. In the following, we will denote this mechanism of overruling as *grounding*. Still, we have to exclude the case of complete inconsistency, i.e., $N(p(u))=N(\neg p(u))=1$. Grounding might occur due to the result of goal-driven analysis operations (e.g., the falsification of hypotheses with the available data) and by definite

knowledge entered by the reengineer. This non-monotonic inference process will be discussed in more detail in Section 4.3.

Example 4.3 Semantics of automatic analysis operations

In this example, we illustrate the formal semantics of automatic analysis operations in GFRN specifications by applying the GFRN in Figure 4.14 to a small excerpt of our case study which is given in Figure 4.18. Let this excerpt be formalized as an RDB $B{:}(M,(R,\Delta),\delta,C,D)\in\overline{RDB}$. We define the following automatic analysis operations for the data- and goal-driven predicates in our sample GFRN:

- $\Omega(ANameIsRSName{+}ID^1)(B)=$

  $$\bigcup_{\substack{r\in R\\ x\in X(r)}}\left(ANameIsRSName{+}ID^1(x),\ 1-\frac{2}{\pi}atan(Levensh(name(x),\ name(r){+}\text{id}))\right)$$

  where $Levensh(s_1,s_2)$ denotes the *Levenshstein-distance* [Lev66] of two strings s_1 and s_2.

- $$\Omega(selectDist^1)(B)\ =\ \bigcup\left\{\left(selectDist^1(\bigcup_{j\in[1,n]}\{a_j\}),\ 1\right)\middle|\begin{matrix}C\text{ contains select-distinct pattern}\\ \text{over attributes }a_1\ldots a_n\end{matrix}\right\}$$

- $\omega(validKey^1)(B,\ validKey^1(x))\ =$

  $$\begin{cases}(\neg validKey^1(x),\ 1) & if\ (\exists(t_1,t_2)\in\delta(RS(x)))\ (t_1\neq t_2\wedge\Pi_x(t_1)=\Pi_x(t_2))\\ \left(validKey^1(x),\ \frac{2}{\pi}\text{atan}\left(\frac{|\delta(RS(x))|}{100}\right)\right) & else.\end{cases}$$

In the first phase of algorithm *OperateGFRN*, *G* is translated to $\Phi\subset\mathcal{L}\{NPL^1\}$. The results of this translation is given in the previous Example 4.2. Subsequently, the data-driven analysis operations $\Omega(ANameIsRSName{+}ID^1)(B)$ and $\Omega(selectDist^1)(B)$ are executed. For each attribute in RS *USER*, $\Omega(ANameIsRSName{+}ID^1)(B)$ produces a fact with the necessity degrees shown in Figure 4.19. Furthermore, $\Omega(selectDist^1)(B)$ detects an instance of a *select-distinct* pattern over attributes *sname* and *dpt*, which results in the fact $(selectDist^1(\{sname,dpt\}),1)$.

In the first iteration of the inference loop from line 11-29, Φ is expanded to $\overline{\Phi}\subset\mathcal{L}\{NPL^0\}$ with respect to the application context $\mathcal{U}(B)$. In the following, we list the subset $\overline{\Phi}_s$ of formulae in $\overline{\Phi}$ which are relevant for this example. Note, that due to the threshold of implication ι_2 $(ANameIsRSName{+}ID^1(userid),0.8)$ is the only relevant fact in the analysis result of operation application $\Omega(ANameIsRSName{+}ID^1)(B)$.

$$\overline{\Phi}_s=\{\quad (selectDist^1(\{sname,dpt\}),1), \tag{EQ 33}$$

$$(ANameIsRSName{+}ID^1(usrid),0.8), \tag{EQ 34}$$

$$(\{sname\}\subseteq\{sname,dpt\}\rightarrow selectDist^1(\{sname,dpt\}))\rightarrow_1\neg key^1(\{sname\})),0.3), \tag{EQ 35}$$

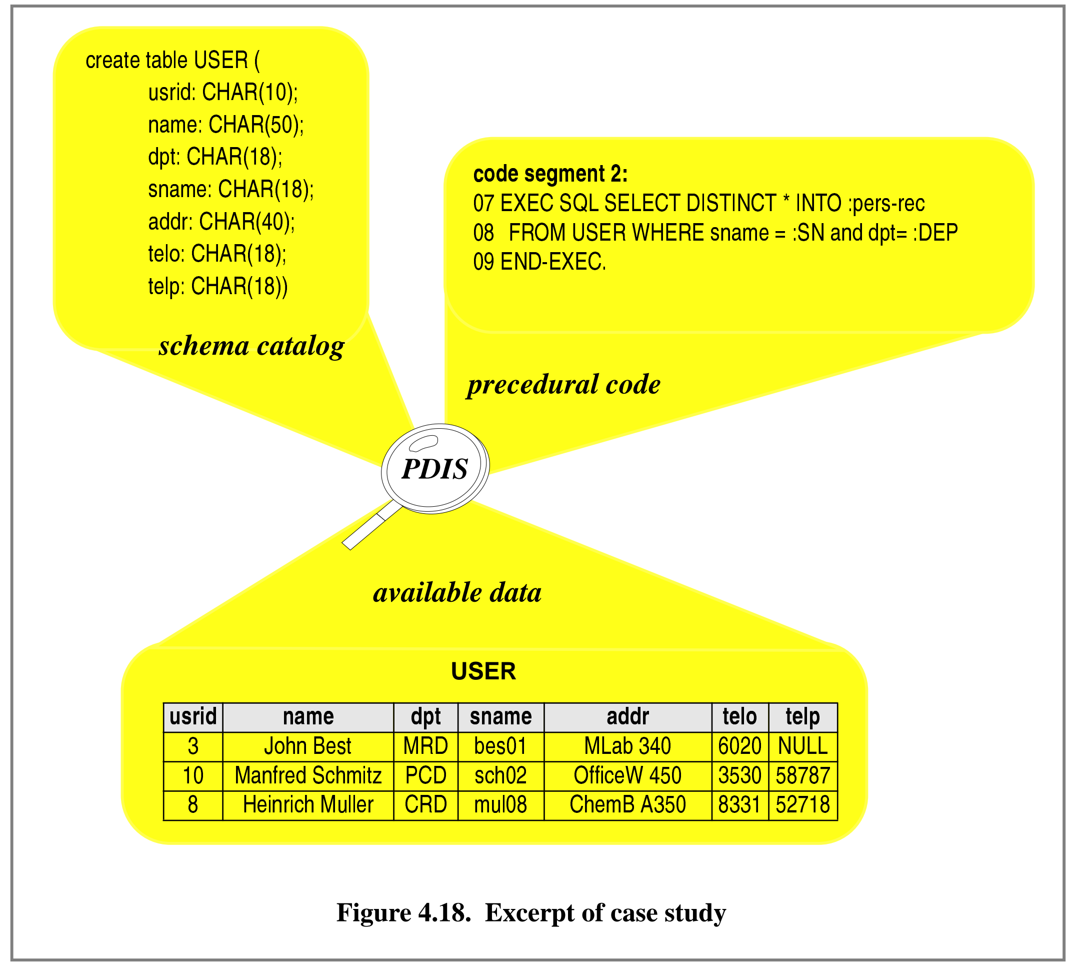
$$(\{sname,dpt\}\subseteq\{sname,dpt\}\rightarrow selectDist^1(\{sname,dpt\}))\rightarrow_1\neg key^1(\{sname,dpt\})),0.3), \tag{EQ 36}$$

$$(ANameIsRSName{+}ID^1(usrid)\wedge N(ANameIsRSName{+}ID^1(usrid)\geq 0.2)\rightarrow_2 key^1(usrid)),0.8), \tag{EQ 37}$$

$$(\neg validKey^1(usrid)\wedge N(\neg validKey^1(usrid))\geq 0.2)\rightarrow_3\neg key^1(usrid)),1), \tag{EQ 38}$$

$$(\neg validKey^1(sname)\wedge N(\neg validKey^1(sname))\geq 0.2)\rightarrow_3\neg key^1(sname)),1) \tag{EQ 39}$$

$\}$



usrid	name	dpt	sname	addr	telo	telp
3	John Best	MRD	bes01	MLab 340	6020	NULL
10	Manfred Schmitz	PCD	sch02	OfficeW 450	3530	58787
8	Heinrich Muller	CRD	mul08	ChemB A350	8331	52718

Figure 4.18. Excerpt of case study

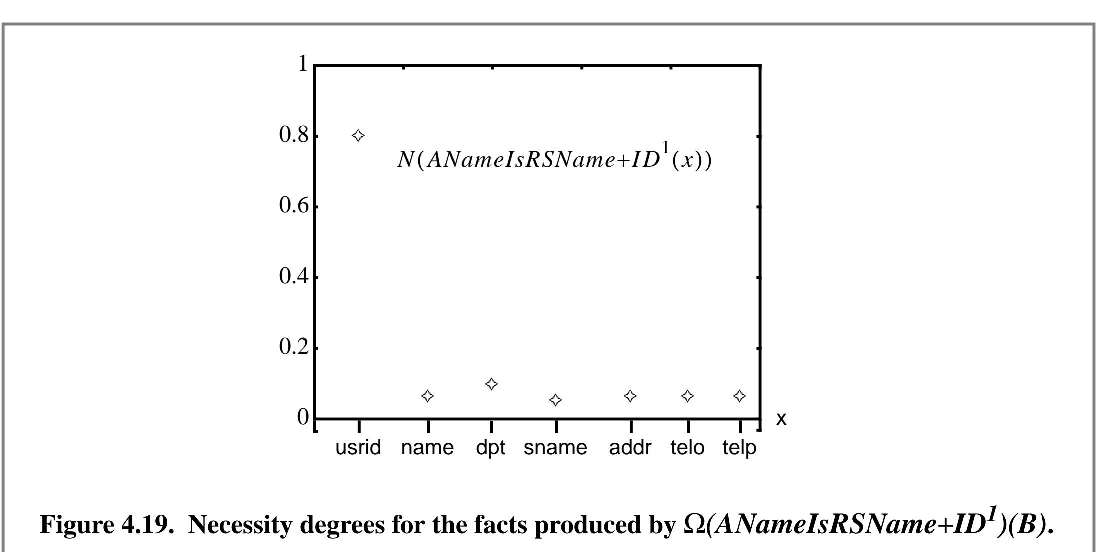


Figure 4.19. Necessity degrees for the facts produced by $\Omega(ANameIsRSName{+}ID^1)(B)$.

According to the formal system for NPL^1 defined in Definition 3.20 on page 51, we can deduce (GMP) *EQ34*, *EQ37* $\vdash$ $(key^1(usrid),0.8)$. Hence, the condition in line 13-14 is satisfied by $(f_1 \Rightarrow_i f_2, \beta) := EQ38$, because $p(u) := validKey^1(usrid)$ occurs in f_1 and f_2 is deducible from $\overline{\Phi}$ with a necessity of *0.8* which is greater than $th(i)=0.2$. Consequently, the query in the body of the conditional statement the goal-driven analysis operation $\omega(validKey^1)(B, validKey^1(usrid))$ is executed. The sample data in Figure 4.18 contains no counter-example for the hypothesis that *usrid* might be a key. Still, the function plot in Figure 4.20 shows that according to the small size of our sample data set we get also only little support for this hypothesis, i.e., $N(validKey^1(usrid)) \cong 0$.

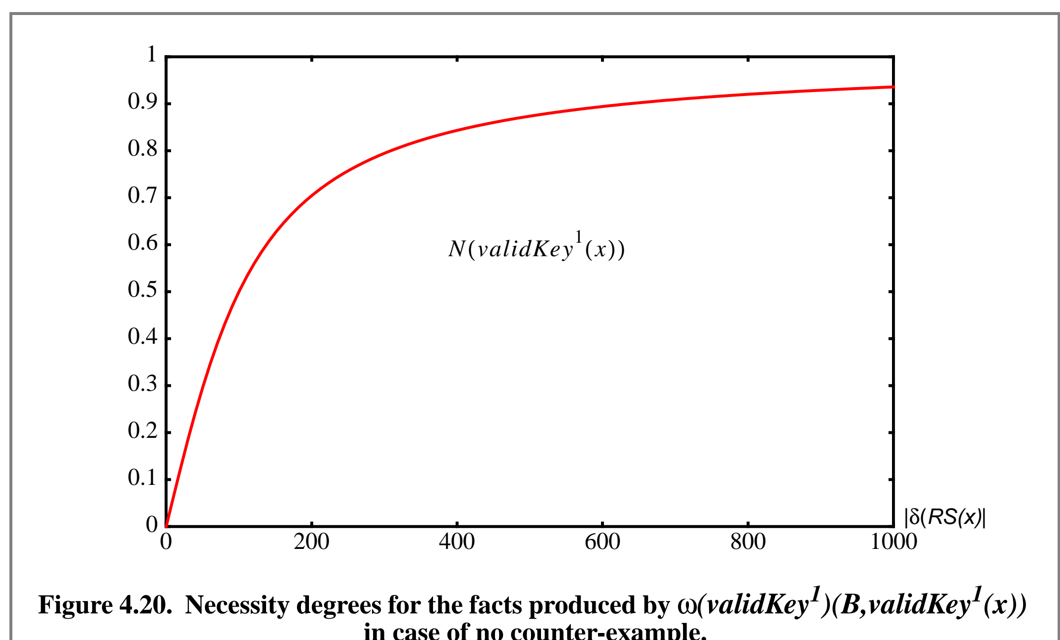


Figure 4.20. Necessity degrees for the facts produced by $\omega(validKey^1)(B,validKey^1(x))$ in case of no counter-example.

Let us now assume that the reengineer manually validates this automatically inferred hypothesis. As a result of this validation (s)he acknowledges the hypothesis by a definite proposition $(key^1(usrid),1)$. Consequently, at the end of the next iteration the inference loop terminates because we have obtained a definite result according to the criterion specified in lines 28 and 29.

The above example closes the formalization of the syntax and declarative semantics of GFRN specifications. In the next section, we will develop a non-monotonic inference engine that implements the described concepts and allows for efficient execution of GFRN specifications in CARE environments.

4.3 Knowledge inference with GFRN specifications

In the previous sections, we defined GFRNs as a dedicated language to specify and customize DBRE knowledge and processes. As described in Section 4.1, we aim to execute such specifications in semi-automatic schema analysis processes. A prerequisite for this execution is an *inference engine* that combines domain-specific GFRN specifications with situation-specific data about the LDB under investigation. Obviously, a suitable inference engine has to meet requirements *R2* and *R4* defined in Section 3.1, i.e., it has to allow for *non-monotonic*

reasoning over *inconsistent* knowledge. In addition, *efficiency* is a crucial requirement for the practical usability of the GFRN approach (cf. requirement *R6* on page 37).

forwards and backwards reasoning

In the AI literature, reasoning problems are often characterized by search problems, i.e., by the problem to select a method that efficiently finds a solution in the search space of all possible options [BB94, pp. 40ff]. Generally, search methods can be classified as either *forward-* or *backward-oriented*. Forward-oriented search methods start with initial data and successively apply reasoning operators until a certain goal is reached, while backward-oriented methods start with a predefined goal and try to find suitable data that allows to reach this goal. In our application domain, we aim to enable an incremental and explorative DBRE process that considers automatically retrieved indicators (initial data) as well as human assumptions on different levels of abstraction (goals). Hence, we have to aim for a *hybrid* approach that allows for forwards as well as backwards reasoning.

incremental reasoning

Another problem arises with the evolutionary character of the proposed schema analysis process (cf. Section 4.1). This process consists of iterative steps involving human interaction and automatic knowledge inference until a consistent and complete result is obtained. New knowledge is added in each of these iterations. However, this additional knowledge generally affects only a part of the results of the previous inference step. Consequently, we should avoid to recompute *every* inference result at each iteration. In contrast, we should aim for an *incremental reasoning mechanism* that uses inference results computed in previous iterations as far as they are not affected by the newly added knowledge.

In the following, we propose an inference engine that meets the above requirements. This inference engine is based on an operational knowledge representation in terms of a *fuzzy Petri net* (FPN) [FS97]. During the inference process, domain-specific knowledge in form of a GFRN and situation-specific knowledge about the LDB under investigation are compiled to an FPN that subsequently can be evaluated efficiently. This compilation process, which we will call *expansion* from now on, is performed incrementally, i.e., the FPN that has been expanded in a given iteration step is preserved and incrementally updated in subsequent iterations.

This section is devided in three parts. First, Section 4.3.1 introduces the used FPN model and reasons about the stability of the proposed non-monotonic belief revision process. Based on these results, we introduce and formally define the entire inference process in Section 4.3.2. Finally, Section 4.3.2.3 discusses the complexity and scalability of our approach.

4.3.1 A fuzzy Petri net model for non-monotonic reasoning

Traditionally, Petri nets (PNs) have been applied to formalize properties of dynamic systems [Pet81]. A rich theory of PNs has been developed since their invention in 1962 by Petri. Many different PN models have been proposed for a great variety of applications. Recently, PNs have been discovered for knowledge representation in rule-based expert systems [FS98]. They combine the advantage of a graphical representation of a rule base with a formal definition of its execution. Analogously to fuzzy rule-based systems, *fuzzy Petri nets* (FPN) have been proposed for applications that deal with imperfect knowledge. A good overview has been presented by Cardoso et al. [CVD96]. In this section, we define an FPN that is an extension of the model described by Konar and Mandal [KM96] which itself is based on Looney's approach [Loo88].

Like any PN an FPN is a directed, bipartite graph with active and passive elements. The active elements are usually called *transitions* while the passive elements are called *places*. In our FPN, places correspond to propositions and transitions represent implication rules. Each place carries a so-called *fuzzy belief marking* (FBM) which is represented by a real number between 0 and 1 in the original model of Konar and Mandal [KM96]. The actual extension of our model is that we use *two* real numbers to represent FBMs, one representing (a lower bound for) the necessity that the associated proposition is fulfilled, while the second represents the necessity against its fulfillment. Similar to GFRNs, we use *signed* arcs to determine whether the positive or the negative belief is propagated. This facilitates the representation of inconsistent knowledge. However, our model can easily be mapped to the original model of Konar and Mandal by using unsigned arcs and allocating two places per proposition (a positive and a negative one). Hence, we are able to transfer the theoretic results established for the original model to our extension. We will make use of this property when we analyze the belief revision model with respect to its stability in case of a cyclic FPN. The signature of the FPN model is defined in Definition 4.12.

Definition 4.12 Fuzzy Petri net

A ***fuzzy Petri net*** *(FPN) is a tuple FPN:=(S, T, F; D, b, v, c, t, m) where*

- *S is a finite set of elements called* ***places****,*
- *T is a finite set of elements called* ***transitions*** *disjoint from S,* $(S \cap T = \emptyset)$*,*
- $F \subseteq (S \times T) \cup (T \times S)$ *is a* ***flow relation****,*
- *D is a finite set of* ***propositions****,*
- *b:S→D is a bijective function that maps places to propositions,*
- *v:F→{‘’, ¬} is a* ***signing function****,*
- *cf: T→ (0, 1] and th: T→ [0, 1) are functions that associate integer values between 0 and 1 to transitions; cf is called the* ***confidence function*** *while the th is called the* ***threshold function****,*
- *m:S→ [0,1]×[0,1] is called the* ***marking function*** *that assigns a pair of real values to each place.*

D

For notational convenience, we use the auxiliary marking function *m:S×{‘’, ¬}→ [0,1]* defined as

$$m(s, x) = \begin{cases} a & \text{for } x = \text{‘‘} \\ b & \text{else} \end{cases} \quad \text{with } m(s) = (a, b)$$

belief revision

The process of propagating FBMs in a cyclic FPN is called *belief revision* [KM96]. It is performed in a number of subsequent *belief revision steps* (BRS). In the following, we will describe different markings of an FPN in different BRS’ by adding the number of the BRS as an index to the marking function *m*, i.e., m_{x+1} describes the marking of an FPN $(S, T, F; D, b, v, c, t, m_x)$ after performing one further BRS.

Each BRS consists of two subsequent phases illustrated in Figure 4.21 and Figure 4.22. In the first step, the output value of each transition in the FPN is computed. This output value is called *fuzzy truth token* (FTT) and is defined by the equations *EQ40* and *EQ41* below. At first, the minimum function is applied to the set of all incoming belief values depending on the signs of

the corresponding edges. Then, the resulting value $I_x(t_q)$ is compared to the threshold of the corresponding transition t_q. If the threshold is lower or equal to this intermediate result, the transition is said to be *enabled*. In this case, the transition fires with an FTT that is the minimum of the intermediate result $I_x(t_q)$ and its confidence value $cf(t_q)$. Otherwise, the new FTT is equal to zero.

$$I_x(t_q) = Min(\{m_x(s, v(s, t_q)) | \exists(s \in S)(s, t_q) \in F\}) \quad \text{(EQ 40)}$$

$$FTT_x(t_q) = \begin{cases} Min(cf(t_q), I_x(t_q)) & if\ I_x(t_q) \geq th(t_q)) \\ 0 & else \end{cases} \quad \text{(EQ 41)}$$

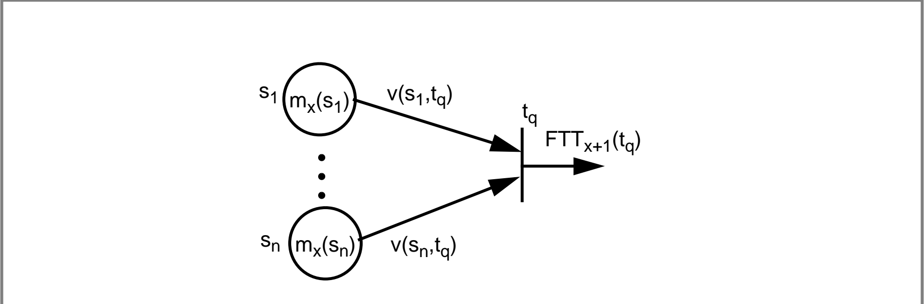


Figure 4.21. Belief revision phase 1: computation of fuzzy truth tokens

In the second phase of each BRS, the incoming FTTs are combined to compute the new FBMs at each place. This is done according to *EQ42* and Figure 4.22 by applying the maximum function over all incoming FTTs. Again, the signs of the arcs have to be respected.

$$m_{x+1}(s, w) = \begin{cases} Max(\{FTT_{x+1}(t) | (\exists t \in T)((t, s) \in F \wedge v(t, s)=w)\}) & if\ (\exists t \in T)((t, s) \in F \wedge v(t, s)=w) \\ m_x(s, w) & else \end{cases} \quad \text{(EQ 42)}$$

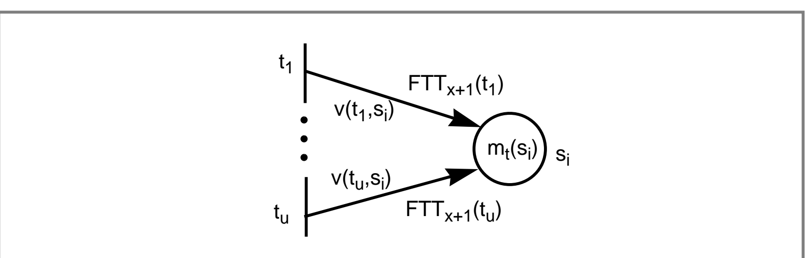


Figure 4.22. Belief revision phase 2: Computation of FBMs

It is important to note that a major difference of the introduced FPN model compared to classical PN models is that tokens are not removed from the input places of an enabled transitions that fires. On the contrary, input tokens are only copied and remain at their original

places. This procedure is necessary for logical inference since the truth of a proposition may imply the truth of several other conditions. Because of this speciality, well-known structural conflicts like deadlocks and traps [Pet81] cannot occur in our model. The characteristic of copying tokens entails another interesting property, namely the fact that belief revision can be performed for all places simultaneously.

termination and stability of belief revision

The belief revision process terminates when there is no change in the marking of the FPN in two subsequent BRS'. In this case, we say that the FPN has reached its *equilibrium state*. Still, an FPN might contain places on cycles that comprise *periodic oscillation* (PO) of FBMs. If such POs sustain for an infinite number of BRS' they prevent the FPN from reaching its equilibrium state. Such oscillating cycles are called *limitcycles* (LC) by Konar and Mandal [KM96]. An FPN with LCs is said to be unstable. This notion of stability is formalized in Definition 4.13. Furthermore, Theorem 4.1 is a result that has been established by Konar and Mandal.

Definition 4.13 Stability

An FPN N:(S, T, F; D, b, v, c, t, m_0) is said to be ***stable*** *iff its marking remains unchanged after a finite number of BRS* $(\exists \bar{x} \in I\!N)\ (\forall s \in S)(m_{\bar{x}}(s)=m_{\bar{x}+1}(s))$. *In this case, it is said that N has* ***equilibrium state*** *in BRS* $\bar{x}$ *and* $Min\{0,..,\bar{x}\}$ *is called the* ***equilibrium time****.*

D

Theorem 4.1 Equilibrium time

The number of transitions represents an upper bound for the equilibrium time of a stable FPN. (The proof of this theorem is given in [KM96].)

T

From Theorem 4.1 follows that after a maximum number of BRS that is equal to the number of transitions it can be decided whether an FPN is stable. Konar and Mandal present an algorithm that removes an LC from an unstable FPN by permanently inhibiting a selected transition on the LC from firing. This transition is selected in such a way that the inference result of the FPN is least affected by the modification. However, eliminating LCs might induce new LCs in neighborhood cycles. Hence, this procedure has to be performed iteratively, in general.

The following Theorem 4.2 shows that LCs cannot occur if we start the belief revision process with an initial marking that assigns non-zero FBMs only to places that do not have incoming arcs. Such places are called *axioms* and the described marking is called an *axiom-based marking*.

Definition 4.14 Predecessor

For a given place $s \in S$ *that is part of an FPN (S,T,F;D,b,v,c,t,m) the set of* ***predecessors****, denoted as pre(s), is given by* $pre(s)=\{z \in S / (\exists t \in T)((t,s),(z,t) \in F)\}$.

D

Definition 4.15 Axiom

A place $s \in S$ *that is part of an FPN (S,T,F;D,b,v,c,t,m) is called* ***axiom****, denoted as axiom(s), iff it has no incoming arc, i.e.,* $pre(s)=\emptyset$.

D

Definition 4.16 Axiom-based marking

An FPN (S,T,F;D,b,v,c,t,m) has an ***axiom-based marking****, iff the following condition holds:* $(\forall s \in S)(m(s) \neq (0,0) \Rightarrow axiom(s))$.

D

Theorem 4.2 Stability of FPN with axiom-based markings

An FPN $N:(S,T,F;D,b,v,c,t,m_0)$ *with an axiom-based marking is stable.*

Proof: If N is not stable there has to be at least one place $s \in S$ *with a fuzzy marking that exhibits infinite periodic oscillation starting from a given BRS* x_{lc}*, i.e.,*

$$(\forall x \in [x_{lc}, \infty))(m_x(s) = m_{x+p}(s) \wedge m_x(s) \neq m_{x+r}(s)) \quad \text{(EQ 43)}$$

with a period $p \in [2,\infty)$ *and* $r \in [1,p-1]$*. In the following, we denote* $(a,b) \geq (c,d)$ *for two tuples* $(a,b),(c,d) \in [0,1] \times [0,1]$ *iff* $a \geq c$ *and* $b \geq d$*. Obviously, EQ43 contradicts to the following condition*

$$(\forall x \in [0,\infty])(\forall s \in S)(m_{x+1}(s) \geq m_x(s)) \quad \text{(EQ 44)}$$

which can easily be proved: EQ44 is trivially fulfilled for axioms. The initial marking for all other (non-axiom) places s is set to $m_0(s) = (0,0)$*. Hence,*

$$(\forall s \in S)(m_1(s) \geq m_0(s)). \quad \text{(EQ 45)}$$

From EQ40-EQ42 follows that for any non-axiom places $s \in S$ *in any BRS x holds*

$$(\forall z \in pre(s))(m_{x+1}(z) \geq m_x(z)) \Rightarrow m_{x+2}(s) \geq m_{x+1}(s) \quad \text{(EQ 46)}$$

which together with EQ45 proves that EQ44 is also fulfilled for all non-axiom places $s \in S$ *in all subsequent BRS.*

T

Corollary 4.1

Each FPN $(S,T,F;D,b,v,c,t,m_x)$ *is stable that can be obtained by subsequently performing* $x \geq 0$ *BRS on an FPN* $(S,T,F;D,b,v,c,t,m_0)$ *with an axiom-based marking.*

C

The above corollary directly follows from the inductive proof of Theorem 4.2. It grants the stability of the FPN inference mechanism which will be employed in the next section.

4.3.2 The inference process

In this section, we develop an inference engine (IE) for GFRN specifications that allows for an iterative and human-centered DBRE process. The proposed IE is based on the FPN model which has previously been introduced. Again, this section is devided in two parts. In the first part (Section 4.3.2.1), we informally outline our strategy. Subsequently, we give a detailed formalization of the IE in Section 4.3.2.2.

4.3.2.1 Informal introduction

The control flow chart in Figure 4.23 shows the inference process that has been proposed in Figure 4.1 on page 56 in more detail. We will start with a general description of each step in

this process. Subsequently, we will discuss each step with an example that deals with an excerpt of our case study.

data-driven analysis

The entire inference process starts with the creation of a new FPN. Then, all data-driven analysis operations in the GFRN are executed and axioms are added to the FPN to represent the resulting initial knowledge about the LDB.

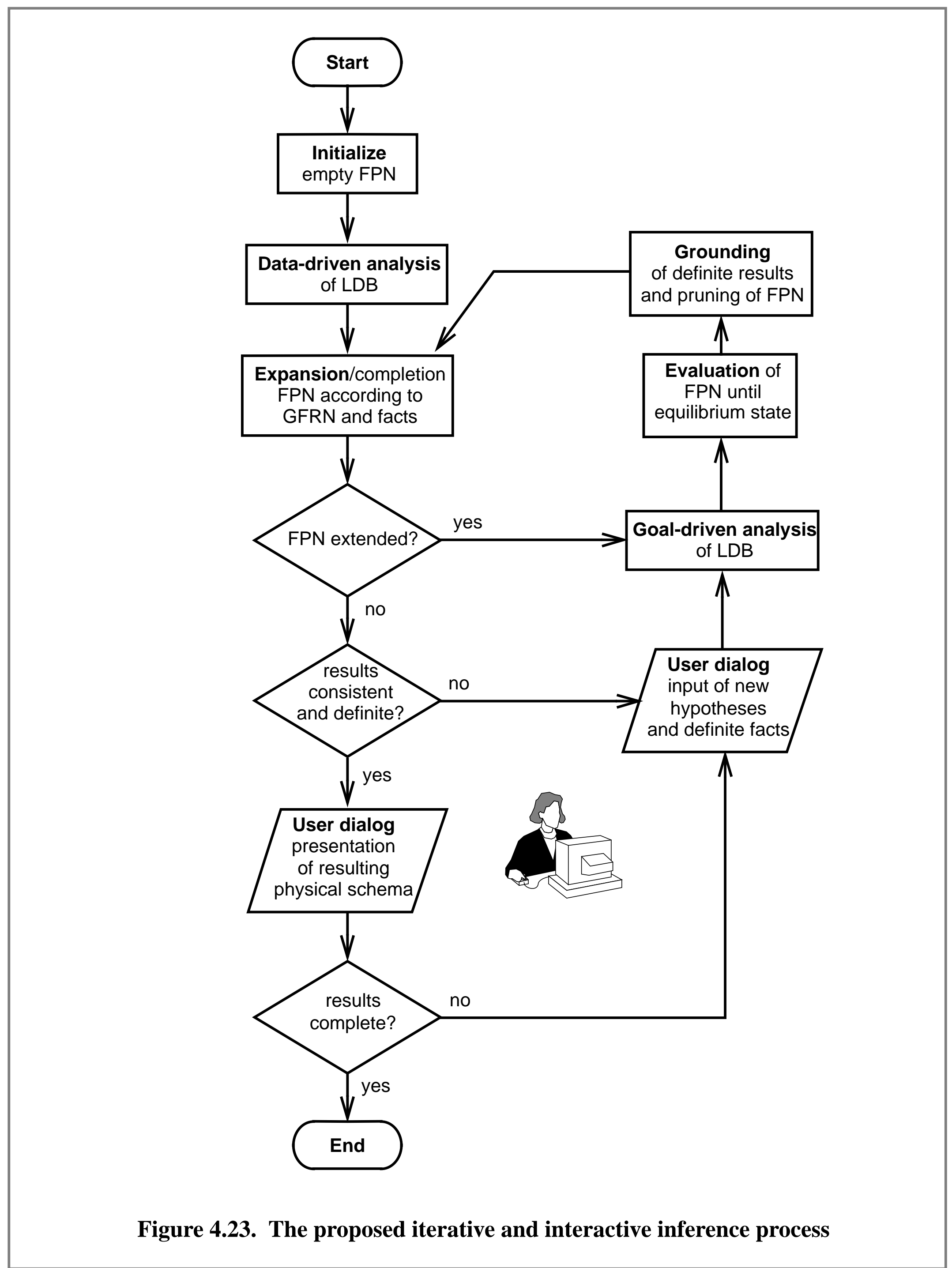


Figure 4.23. The proposed iterative and interactive inference process

expansion

In the next step, the FPN is expanded according to the places in the FPN. This step is illustrated in Figure 4.24 for a sample situation. It shows that an instance of a GFRN implication is represented by a number of transitions with the same CV and TV in the FPN: the solid transition represents the actual implication rule $p_2(u_2) \wedge p_3(u_3) \rightarrow p_1(u_1)$ and the other two transitions represent its contraposition $\neg p_1(u_1) \rightarrow \neg(p_2(u_2) \wedge p_3(u_3))$ that has been normalized with *deMorgan's* law to $\neg p_1(u_1) \wedge p_2(u_2) \rightarrow \neg p_3(u_3)$ and $\neg p_1(u_1) \wedge p_3(u_3) \rightarrow \neg p_2(u_2)$. Analogously to the GFRN formalism, we represent arcs with a negative sign by solid arrow heads. From now on, we refer to the transition that represents the actual implication rule as the *main transition* (MT) while we denote the other transitions as *contraposition transitions* (CTs). In general, the number of created CTs is equal to the number of places in the antecedent of the MT. In order to increase the readability of our FPN diagrams we use grey color for arcs that belong to CTs.

An implication can only be expanded if all its variables can be bound such that its constraints K are satisfied. If this precondition is fulfilled the implication can be expanded either in *forward* or *backward* mode (cf. Figure 4.25).

- The implication is expanded *forwards* if all necessary propositions in the antecedent of the MT to be created are present in the FPN, the MT would be enabled, and the MT would have at least one positive outgoing arc. (We do not expand MTs with negative consequents in *forward* mode because we are interested in inferring positive hypotheses. Such MTs are expanded in *backward* mode only to *refute* positive hypotheses.)
- The implication is expanded *backwards* if there exists a proposition in the consequent of the MT to be created that has a positive FBM that is greater or equal to the threshold of MT.

Note, that it is *not* required that *all* propositions in the antecedent and consequent of the transitions are already present in the FPN. It is sufficient for the expansion if variable bindings for missing propositions can be computed by applying the constraints K to the variables which can be bound to actual parameters of present propositions. For example, consider implication ι_2 from Figure 4.14 on page 68: if the FPN contains a proposition that is suitable to bind variable a, we can compute variable k by applying the constraint $k{=}set(a)$.

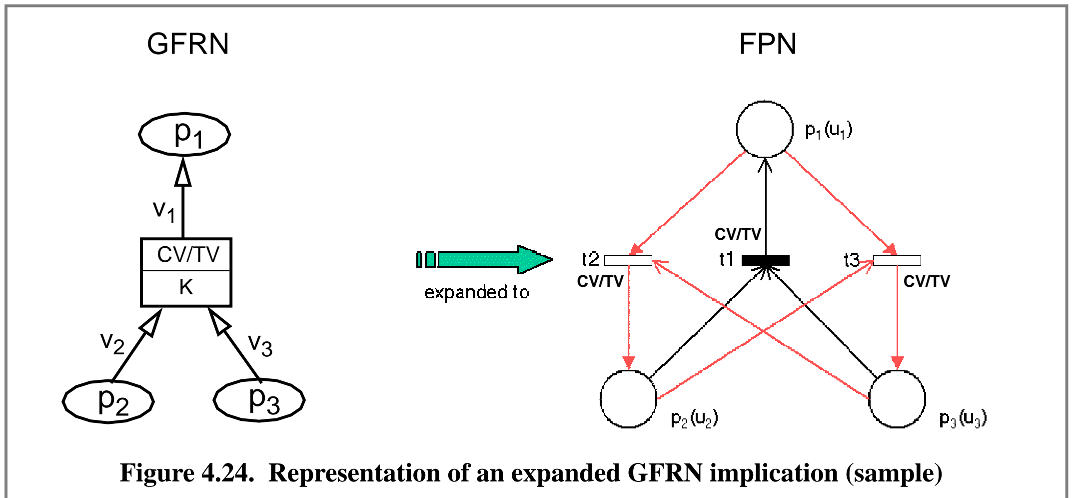


Figure 4.24. Representation of an expanded GFRN implication (sample)

goal-driven analysis

If the FPN structure has been modified in the expansion activity, goal-driven analysis operations are automatically executed for each newly created place that is an instance of a

goal-driven predicate. The result of each operation is stored in the FBM of the corresponding place. Furthermore, such places are converted to axioms, i.e., all incoming arcs are removed from the FPN. This is necessary because indicators delivered by goal-driven analysis operations are definite and may not be modified during the inference process.

evaluation

In the next step, the FPN is evaluated using the belief revision process defined in Section 4.3.1. The stability of the expanded FPN is guaranteed by Corollary 4.1, because we have created an axiom-based marking.

grounding

After performing the goal-driven analysis and evaluating the FPN there might be some definite analysis results, i.e., facts that have a positive or negative necessity degree of 1. These facts are converted to axioms in a subsequent activity that we call *grounding*.

automatic expansion and evaluation cycles

Now, the expansion of the existing FPN is resumed under consideration of the newly added facts and the results of the evaluation. Again, goal-driven operations are executed on demand if the FPN structure has been extended. Subsequently, the FBMs at all non-axiom places are reset to zero, in order to create an axiom-based marking before the evaluation process is resumed. These expansion/evaluation cycles are iterated automatically until the FPN structure remains unmodified after an expansion step.

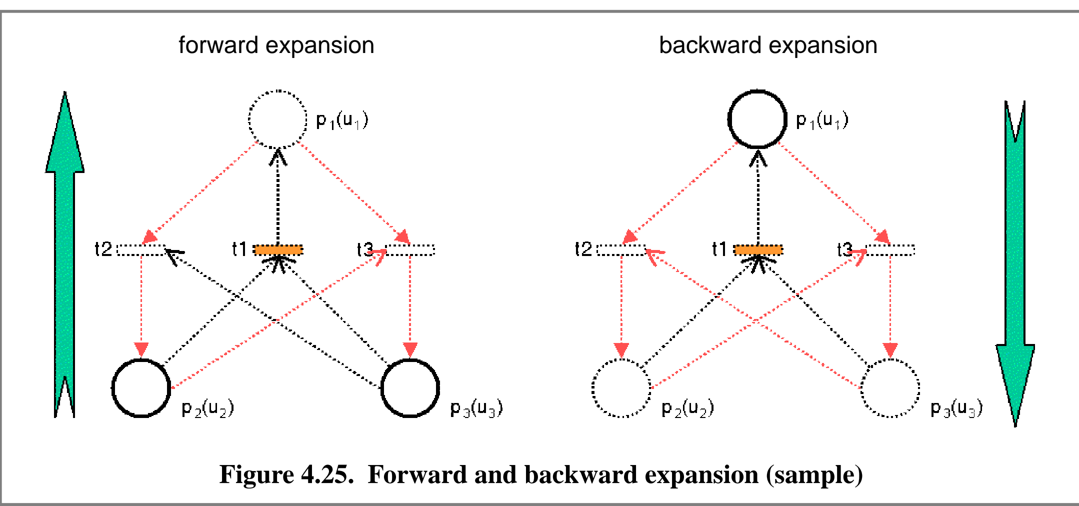


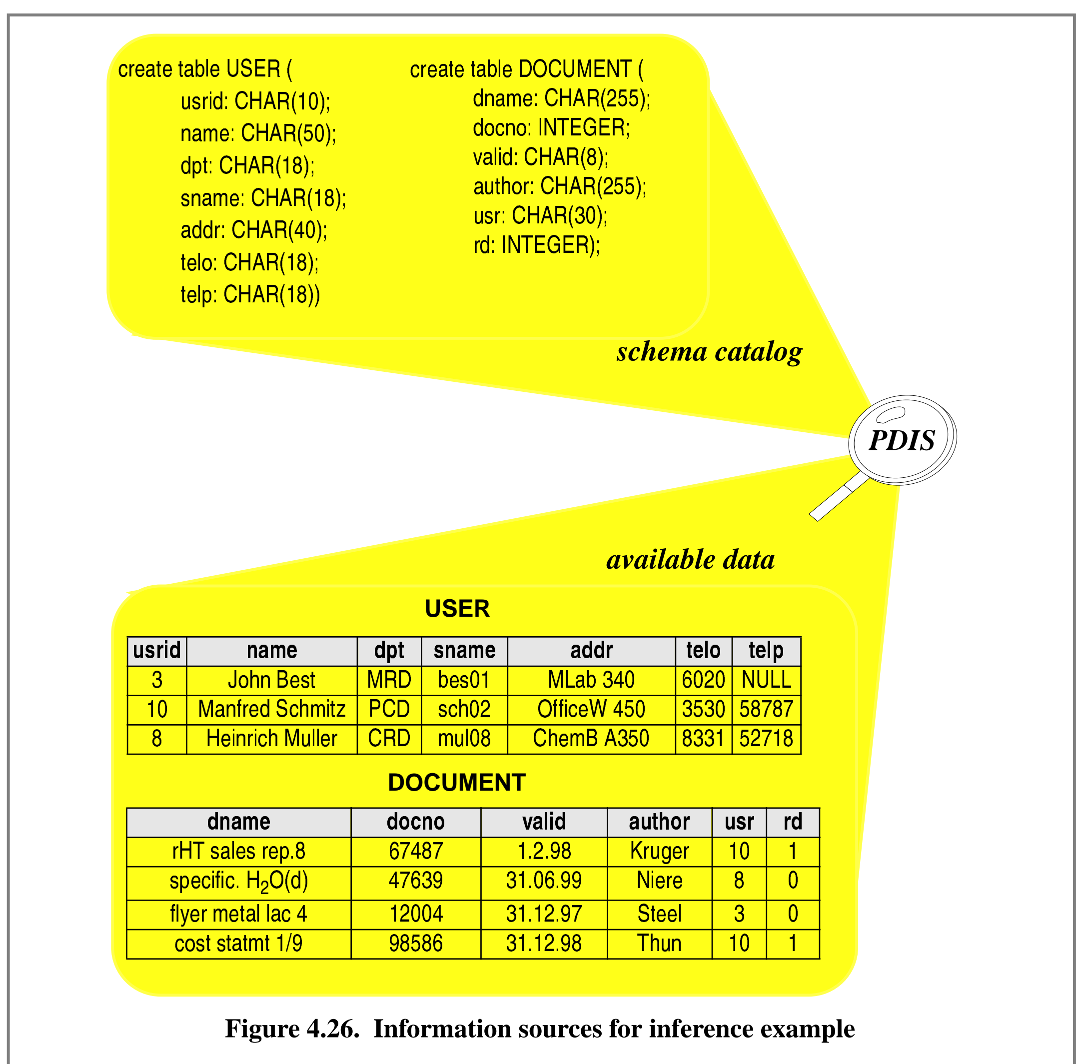
Figure 4.25. Forward and backward expansion (sample)

user dialog

When the automatic expansion/evaluation cycles terminate, the inference engine checks whether the produced analysis result is definite and consistent. If this is the case, the reengineer has to decide if the resulting information is complete. Otherwise, the reengineer has to do some further (manual) investigation of the LDB in order to support or refute intermediate analysis results or add new knowledge. After this interaction step, the automatic inference process is resumed. The entire semi-interactive process terminates when the analysis result is definite, complete, and consistent.

Example 4.4 Inference process

We will now illustrate the described semi-automatic inference process with an example that deals with an excerpt of our DBRE case study. Figure 4.26 shows that this excerpt consists of the two RS *USER* and *DOCUMENT*, including some sample data. In this example, we aim to detect foreign keys between these RS. We apply the GFRN presented in Figure 4.27 for this purpose.



usrid	name	dpt	sname	addr	telo	telp
3	John Best	MRD	bes01	MLab 340	6020	NULL
10	Manfred Schmitz	PCD	sch02	OfficeW 450	3530	58787
8	Heinrich Muller	CRD	mul08	ChemB A350	8331	52718

DOCUMENT

dname	docno	valid	author	usr	rd
rHT sales rep.8	67487	1.2.98	Kruger	10	1
specific. $H_2O(d)$	47639	31.06.99	Niere	8	0
flyer metal lac 4	12004	31.12.97	Steel	3	0
cost statmt 1/9	98586	31.12.98	Thun	10	1

Figure 4.26. Information sources for inference example

first expansion/ evaluation cycle

The initial analysis of the LDB is performed by executing the data-driven analysis operations that have been attached to predicates *ANameIsRelName+ID*[1], *NamSim*[2], and *variant*[1]. As a result of this automatic analysis, four axioms are created in the FPN, which are represented as doubled circles in Figure 4.28. At this, we use the following abbreviations for the actual parameters of the displayed propositions:

abbreviation	parameter
uu	*USER.usrid*
un	*USER.name*
du	*DOCUMENT.usr*
dn	*DOCUMENT.dname*
d	*{DOCUMENT.title,DOCUMENT.docno,DOCMU-NET.valid,DOCUMENT.author,DOCUMENT.usr}*

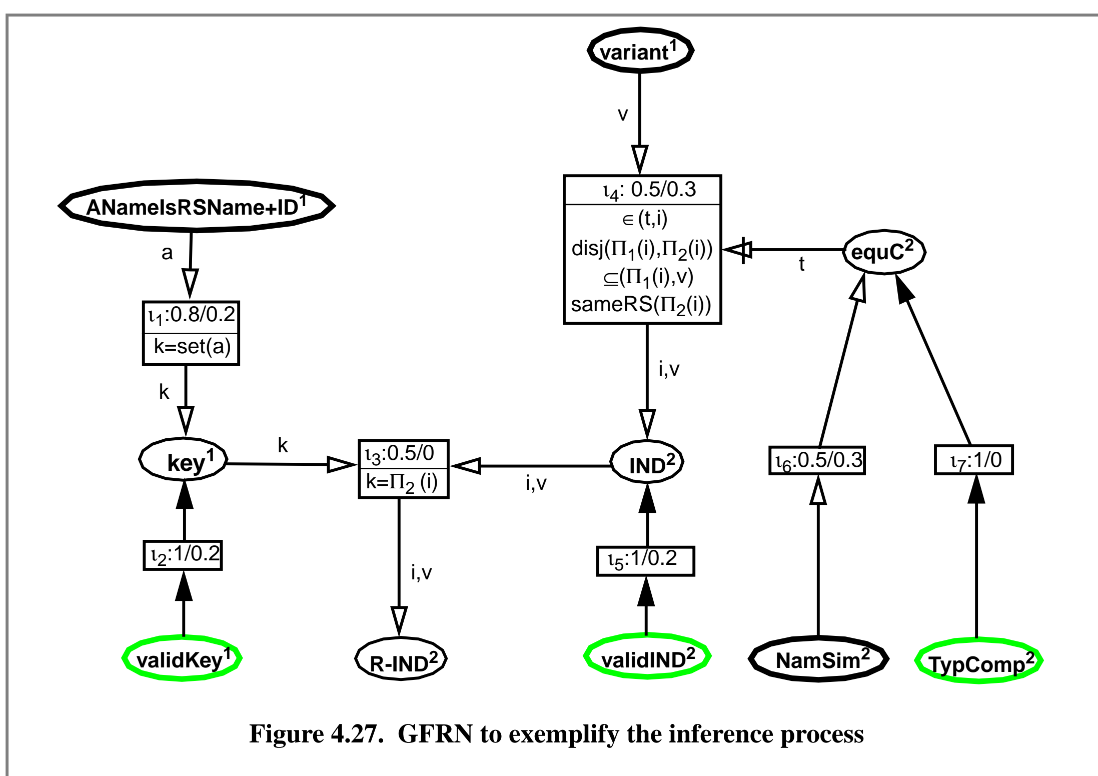


Figure 4.27. GFRN to exemplify the inference process

The axioms created in the FPN in Figure 4.28 show that the initial analysis has detected that predicate *ANameIsRelName+ID*[1] is valid to a degree of 0.8 for attribute *uu*. Moreover, it has detected that there are two pairs of similarly named attributes in these RS, namely (*du,uu*) and (*dn,nn*). An analysis of the available data shows only one variant of tuples that includes all attributes of RS *DOCUMENT*. (We skip the variant of RS *USER* as it is not relevant for our example.) In the first expansion step, the forward expansion rule can be applied once for implication ι_1 and twice for implication ι_6. During this expansion step variable *a* of implication ι_1 is bound to parameter *uu*. Using the function *set*, which is defined as a simple set constructor, the value of the second variable *k* is functionally determined by this binding. Note, that no CTs are created in this first expansion step, because incoming arcs are forbidden for axioms. The first automatic expansion/evaluation cycle finishes with the evaluation of the FBMs at the expanded places according to *EQ40-EQ42* on page 79.

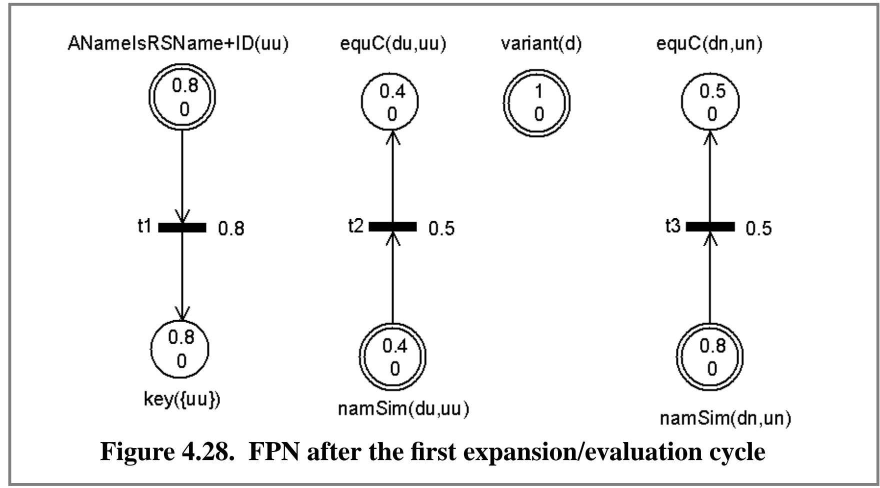


Figure 4.28. FPN after the first expansion/evaluation cycle

second expansion/ evaluation cycle

The FPN that results from the second expansion/evaluation cycle is presented in Figure 4.29. In this cycle, the backward expansion rule can be applied once for implication ι_2 and twice for implication ι_7. Furthermore, the forward expansion rule is applied three times to implication ι_4, because of the IQ in its premise. The corresponding MTs are labeled t_7, t_9, and t_{11}. After the expansion, the goal-driven analysis operations that are attached to predicates *validKey*[1] and *TypComp*[2] are executed for their newly added instantiations. However, the hypothetical key constraint over attribute *uu* cannot be falsified automatically by the available data and the compared pairs of attributes are (fairly) type compatible.

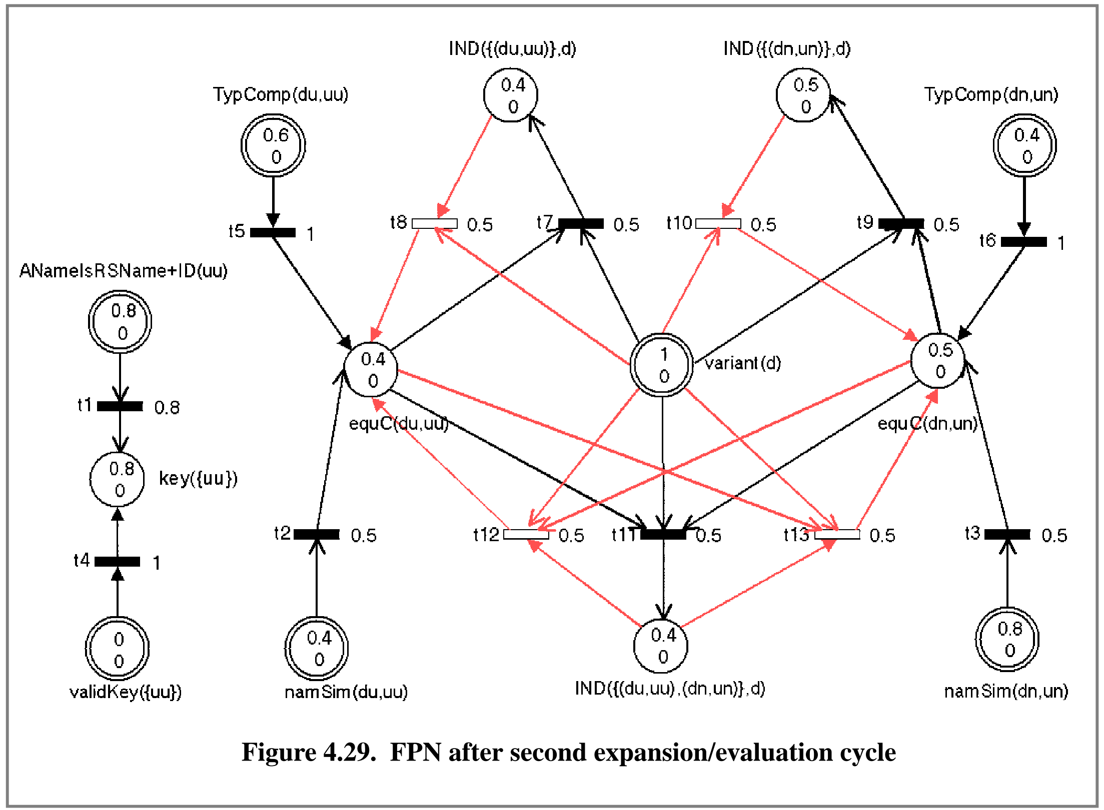


Figure 4.29. FPN after second expansion/evaluation cycle

third expansion/ evaluation cycle

In the third expansion/evaluation cycle, the forward expansion rule can be applied to implication ι_3 that combines the knowledge about the hypothetical key constraint and the IND over *du* and *uu* to infer an R-IND (cf. Figure 4.30). The two other INDs can be falsified by applying the backward expansion rule to implication ι_5 and executing the corresponding goal-driven analysis operation *validIND*[2]. We say that the two corresponding places have been *grounded*, because they represent definite facts (i.e., they have a negative necessity degree of 1). They are converted to axioms in the *grounding* activity at the end of this expansion/ evaluation cycle. In order to increase the readability of Figure 4.30 we display only enabled transitions and places that are connected to enabled transitions.

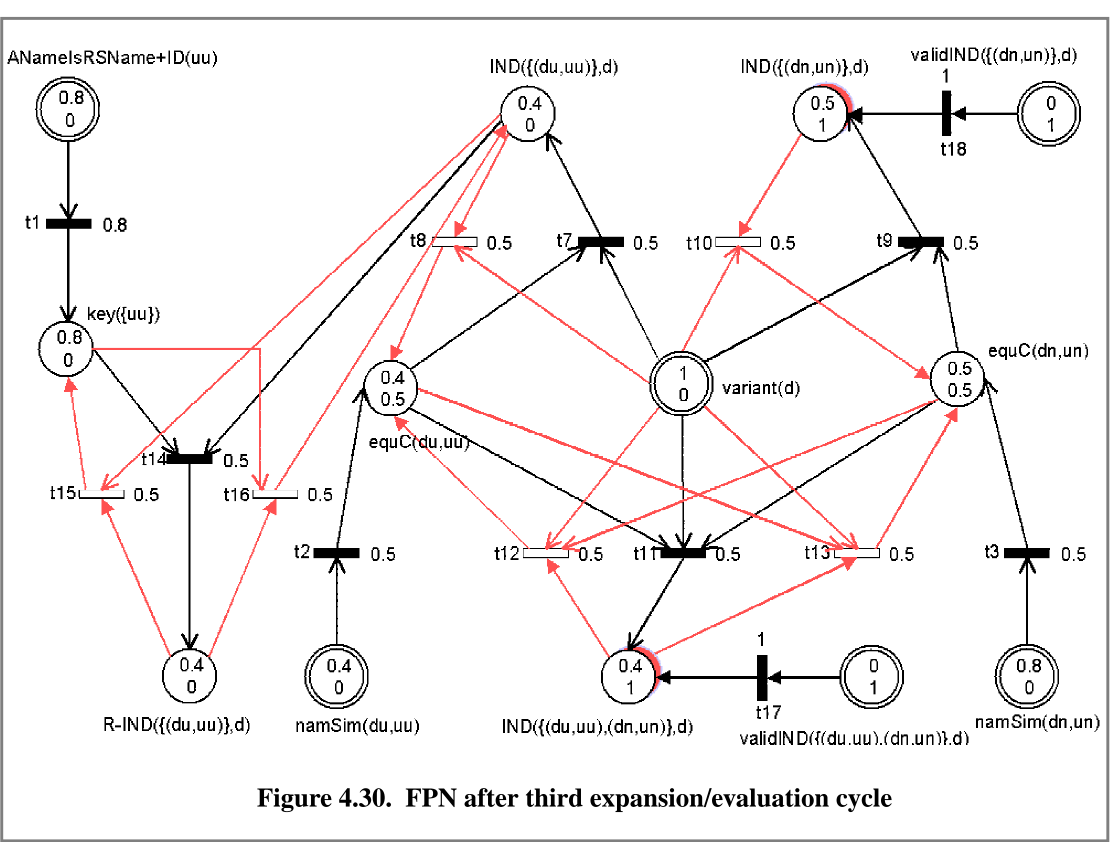


Figure 4.30. FPN after third expansion/evaluation cycle

human interaction

The FPN shown in Figure 4.30 cannot be further expanded by applying the defined expansion rules. Consequently, the automatic inference process terminates with an analysis result that is still inconsistent respectively undecided. (The R-IND between *DOCUMENT* and *USER* is only indicated with a necessity degree of 0.4). This result is presented to the reengineer in a suitable dialog. The reengineer has to use her/his domain knowledge and perform manual investigations to decide whether the hypothetical R-IND is valid. Let us assume that (s)he decides that the inferred R-IND exists: (s)he adds this definite fact, which results in another grounded place in Figure 4.32 (for proposition $R\text{-}IND^2(\{(du,uu)\},d)$). The other two uncertain propositions ($IND^2(\{(du,uu)\},d)$ and $Key^1(\{uu\})$) can be grounded likewise. However, this can also be done automatically by the inference engine if we add implications to our GFRN which specify that an IND and a key constraint is necessary for the existence of an R-IND (cf. Figure 4.31). In this case, only one interaction is necessary to arrive at the definite analysis result presented in Figure 4.33. We did not consider these additional implications in our GFRN in Figure 4.27 because it would have further increased the complexity of the FPNs displayed in this example.

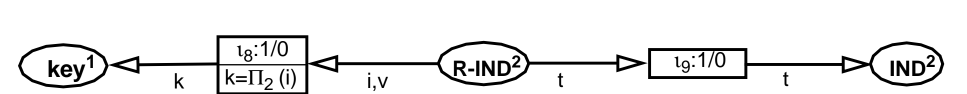


Figure 4.31. Additional implications to specify necessary conditions for R-INDs

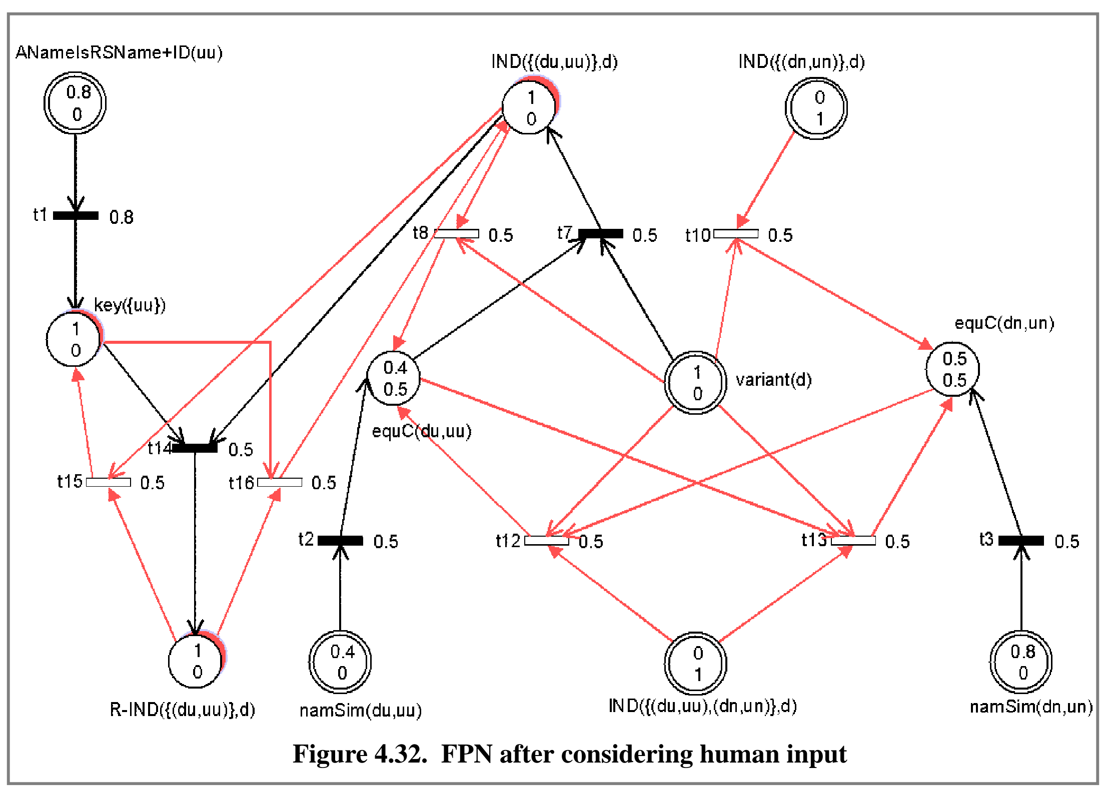


Figure 4.32. FPN after considering human input

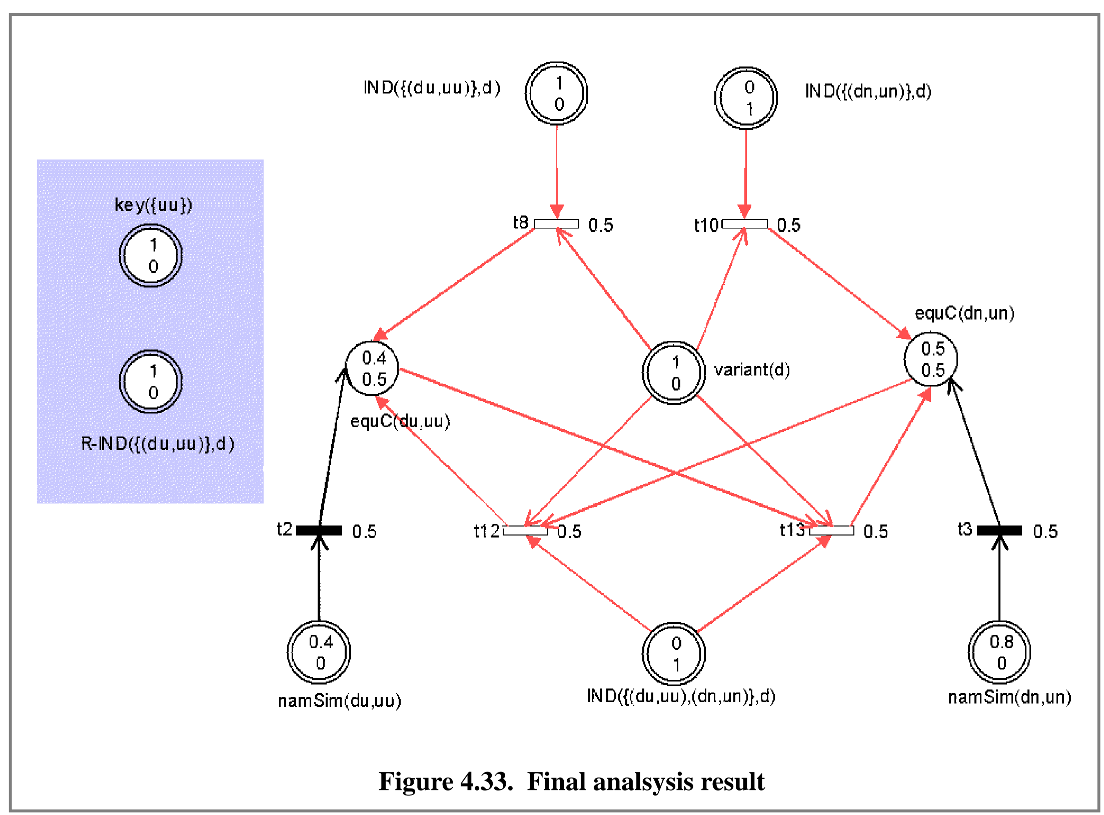


Figure 4.33. Final analsysis result

representing human assumptions

In the example presented above, human interaction is needed to validate (and support) a hypothesis that has been inferred automatically. However, in an evolutionary DBRE process, other scenarios are also possible. For example, the reengineer might annotate uncertain assumptions in order to use the IE (in combination with the knowledge provided by the GFRN) to validate these assumption and infer new hypotheses. Obviously, such uncertain assumptions cannot be represented as axiomatic instances of the corresponding predicates in the GFRN, because the FBM of axioms are immutable per definition. Therefore, we create an additional axiom for each such assumption with a transition that leads to the actual proposition. In Figure 4.34, this is illustrated for the simple case that the reengineer enters his/her subjective belief that attribute *dname* might be a key of RS *DOCUMENT*. This assumption is represented by an axiom (*SB_key({dn})*) with a transition that propagates the belief to the place that represents the actual proposition. Figure 4.34 shows that this assumption is refuted by the goal-driven analysis operation attached to predicate *validKey*[1]. Hence, the actual proposition represented by place *key({dn})* can be grounded.

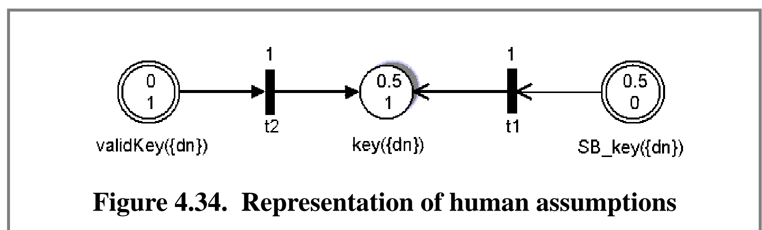


Figure 4.34. Representation of human assumptions

In the next section, we will formalize the inference mechanism that has been introduced and illustrated so far.

4.3.2.2 Formal definition

We start the algorithmic formalization of the process introduced in Figure 4.23 by discussing the main inference algorithm presented in Figure 4.35. Subsequently, we give a more detailed definition of the expansion step. This algorithm (*GFRNInference*) produces a set of definite propositions based on an input that consists of a GFRN specification and a relational DB. Two FPN variable structures are used locally to obtain this result. The first structure ($\overline{N}$) is used for the actual expansion and evaluation activities, while the second structure (N) stores the FPN that was the result of the most recent expansion/evaluation cycle. Moreover, we employ a variable X to store the set of places that are going to be axioms. Using this variable simplifies the expansion algorithm, because we do not have to distinguish between axioms and non-axioms in each situation when places and transitions are created. After each expansion/evaluation cycle, we satisfy the required structural constraints (no incoming arcs for axioms) by employing the information stored in X in a post-processing step (cf. line 39).

```
1)  algorithm GFRNInference(G, B)
2)    input  G:=((P^d,P^g,P^t),F^r,F^b,I,E,cf,th,Ω,ω)∈ Ł{GFRN}; B∈ RDB̄
3)    output R ⊂Ł{L^0}
4)    local variables  N̄:(S̄,T̄,F̄;D̄,b̄,v̄,c̄,t̄,m̄_x)∈ Ł{FPN}// current FPN
5)                     N:(S,T,F;D,b,v,c,t,m_x)∈ Ł{FPN}// result of the most recent exp./eval. cycle
6)                     X⊆S̄   // places that are going to be axioms
7)    begin
8)      let N̄:(S̄,T̄,F̄;D̄,b̄,v̄,c̄,t̄,m̄_x)=CreateEmptyFPN()
9)
10)     for each p∈ P^d do                                              // data-driven analysis
11)       for each q∈ {(w,β)∈ Ω(p)(B)/ (∃(χ,(p,i),s,d,A)∈ E)(w=s p∧β≥th(i))} do
12)         let (N̄,X)=CreatePlace(q, N̄, X, TRUE)
13)       od
14)     od
15)
16)     loop
17)       loop
18)         let D_changed={d|d∈ D̄∧(d∉ D∨m_x(b^-1(d))≠m̄_x(b^-1(d)))}      // new/changed places
19)         if D_changed≠∅
20)         then
21)           let N=N̄       // store old FPN state
22)           let N̄:(S̄,T̄,F̄;D̄,b̄,v̄,c̄,t̄,m̄_x)=ExpandFPN(G,N̄,D_changed)   // expansion
23)
24)           for each z∈ {s∈ S|b(s)=p(u)∧p(u)∈ D̄-D∧p∈ P^g} do
25)             let N̄=CreatePlace((ω(p)(B,p(u)) ,N̄, TRUE)            // goal-driven analysis
26)           od
27)
28)           let N̄=ResetMarkings(N̄)     // create axiom-based marking
29)           let N̄=EvaluateFPN(N̄)                                     // evaluation
30)
31)           for each s∈ {z∈ S̄/ grounded(z)} do                        // grounding
32)             let (a,b)=m̄_x(s)
33)             if m̄_x(s,‘’)=1 then m̄_x(s,¬)=0
34)                             else m̄_x(s,‘’)=0
35)             fi
36)             let X=X∪{s}
37)           od
38)
39)           let N̄=RemoveIncomingArcs(N̄, X) // satisfy structural constraints for axioms
40)         fi
41)       until D_changed=∅         // FPN unchanged
42)
43)       for each (w,β)∈ UserDialog(D,G) do                           // user dialog
44)         CreateOrReviseAxiom((w,β) , N̄, G)
45)       od
46)
47)     until (∀p(u)∈ D̄)(p∈ P^t→p(u)∈ X∨m̄_x(b^-1(p(u)),”)=0)
48)                                          // positive results is definite and consistent
49)   return {p(u)∈X/ p∈ P^t∧m̄_x(b^-1(p(u)),”)=1}
50) end
```

Figure 4.35. Algorithm *GFRNInference*

data-driven analysis

The inference algorithm starts by creating an empty FPN in line 8. Then, all data-driven analysis operations are executed and places are created to represent the resulting indicators in the FPN (lines 10-13). Note, that only those indicators are considered that have a credibility weakly greater than the threshold of at least one GFRN implication that has the corresponding data-driven predicate in its premise. The last parameter of the called algorithm *CreatePlace* is a boolean value that determines whether the newly created place will be an axiom (cf. Figure 4.36).

outer (interactive) inference loop

The outer (interactive) inference loop starts in line 16 and terminates in line 47 when all instances of dependent predicates with a positive lower bound of necessity are represented by axioms in the FPN. In Example 4.4, we demonstrated that such instances are converted to axioms only if they have been *grounded*, i.e., if they have a positive or negative necessity of 1. Consequently, the output of the algorithm is defined by the *classical projection* of all positive propositions, i.e., all propositions that have a positive necessity of 1 (cf. line 49). User input is considered in lines 43-45. The user might revise situation-specific knowledge by updating the corresponding FBMs and (s)he can add new propositions by creating new axioms (cf. Figure 4.34).

inner (automatic) inference loop

Lines 17-41 specifies the inner loop that automatically performs expansion/evaluation cycles until the FPN remains unchanged.The statement in line 18 computes the set of all propositions that have been added or modified in the last iteration. If this set is not empty it is used in line 22 to expand the FPN incrementally. Subsequently, goal-driven analysis operations are called for all newly added instances of goal-driven predicates (cf. lines 24-26). Then, all FBMs at non-axiom places are set to zero to obtain an FPN with an axiom-based marking that is evaluated until equilibrium state in line 29. The aforementioned activity of *grounding* is formalized in lines 31-37. In this activity all definite analysis results (i.e., propositions with a positive or negative necessity of 1) are converted to axioms and partial inconsistency is removed. (A formal definition of the notion of a grounded place is given in Definition 4.17.) Before the next iteration of the inner loop, line 39 removes all incoming arcs for places that actually represent axioms.

Definition 4.17 Grounded place

A place $s \in S$ that is part of an FPN $(S,T,F;D,b,v,c,t,m_x)$ is called ***grounded*** *in BRS x, denoted as $grounded_x(s)$, iff $Min(a,b)<Max(a,b)=1$ with $m_x(s)=(a,b)$.*

◻

Expansion process

algorithm CreatePlace

The expansion process is the process that incrementally creates and extends an FPN from a combination of a GFRN and accumulated situation-specific knowledge. We have already referred to algorithm *CreatePlace* in Figure 4.36 that is used to create instances of GFRN predicates. In lines 6 and 7 it creates a new place that is added to the set of axioms if the boolean argument *ax* is *TRUE* (cf. line 9). Then, the FBM of the new place is initialized according to the sign of the represented literal (lines 11-14). Finally, the unsigned proposition is added to the set of propositions in the FPN (lines 15 and 16).

```
algorithm CreatePlace(d, N, X, ax)
1)  input  d∈ℒ{NPL^0}, N:(S,T,F;D,X,b,v,c,t,m_x)∈ℒ{FPN}; X⊆S
2)  input  G:(P, F^r, F^b, I, E, cf, th, Ω, ω) ∈ ℒ{GFRN}; ax∈BOOL
3)  output (N,X)
4)  local variables s∈ℕ // place identifier
5)  begin
6)      let s=createID()
7)      let S=S∪{s}
8)
9)      if ax=TRUE then let X=X∪{s} fi
10)
11)     if d=(¬p(u), β)
12)         then let m_x(s)=(0,β)
13)         else let m_x(s)=(β,0)      /* d=(p(u), β) */
14)     fi
15)     let D=D∪{p(u)}
16)     let b(s)=p(u)
17)     return (N,X)
18) end
```

Figure 4.36. Algorithm *CreatePlace*

algorithm ExpandFPN

The algorithm that formalizes the forward and backward expansion rule from Figure 4.3.2.1 is presented in Figure 4.37 (*ExpandFPN*). Algorithm *ExpandFPN* starts by removing all transitions from the FPN that represent instances of implications with IQs which might be affected by the last change in the FPN (cf. lines 6-8). This is done because some of these transitions might lose their validity in presence of additional situation-specific knowledge. An alternative solution for this problem is to check the corresponding constraints for each affected transition with the new knowledge and to remove only those transitions which are no longer valid. We have chosen the first alternative because it does not increase the computational complexity of our algorithm (cf. Section 4.3.2.3) but it reduces its complicacy. The main loop in algorithm *ExpandFPN* (lines 10-35) tries to expand each implication in the GFRN that is affected by the changed situation-specific knowledge. In line 11, algorithm *ComputeBindings-ForImpl* is called to compute all valid variable bindings for the current implication. These bindings are returned in form of a relation that is assigned to the local variable *bindingset*. For a given implication $(\iota,<v_1,..,v_x>,K)\in I$, each tuple $g:(u_1,..,u_x)\in bindingset$ represents a valid binding for the variable list $<v_1,..,v_x>$. Furthermore, we define that the single elements of each such tuple can be associatively accessed by the corresponding formal variable name, i.e., $g[v_i]=u_i$ with $1\leq i\leq x$.

expansion of transitions

The loop from line 12 to line 34 extends the FPN structure for each binding in variable *bindingset*. This is done in the following steps. Firstly, all positive and negative propositions in the antecedent and consequent of the corresponding MT are stored in the local variables D_{a+}, D_{a-}, D_{c+}, and D_{c-} (lines 13-16). Then, it is checked whether the forward or backward expansion rule can be applied (lines 18-19). If this is the case, then lines 21-23 create places for all propositions that are not yet represented in the FPN. Subsequently, lines 26-29 create the MT and all CTs that are necessary to represent the propositional implication, if these transitions have not been created before (cf. line 24).

algorithm *ExpandFPN*$(G, N, X, D_{changed})$

1) **input** $G:(P, F^r, F^b, I, E, cf, th, \Omega, \omega) \in \mathcal{L}\{GFRN\}$

2) **input** $N:(S,T,F;D,b,v,c,t,m_x) \in \mathcal{L}\{FPN\}$; $D_{changed} \subseteq D$; $X \subseteq S$

3) **output** $\overline{N}:(\overline{S},\overline{T},\overline{F};\overline{D},\overline{X},\overline{b},\overline{v},\overline{c},\overline{t},\overline{m}_x) \in \mathcal{L}\{FPN\}$; $\overline{X} \subseteq \overline{S}$

4) **local variables** $bindingset \in \overline{REL}$; $g \in \mathcal{U}(B)$; $D_{a+},D_{a-},D_{c+},D_{c-} \subseteq \mathcal{L}\{L^0\}$

5) **begin**

6) **for each** $i \in \{i:(\iota,V,K) \in I / u \in \mathcal{U}(B) \wedge p(u) \in D_{changed} \wedge (\chi,(p,i),s,premise_quantified,A) \in E\}$ **do**

7) *remove all transitions from N that have been created for implication i*

8) **od**

9)

10) **for each** $i \in \{i:(\iota,V,K) \in I / u \in \mathcal{U}(B) \wedge p(u) \in D_{changed} \wedge (\chi,(p,i),s,d,A) \in E\}$ **do**

11) **let** $bindingset = ComputeBindingsForImpl(G,i,N)$

12) **for each** $g \in bindingset$ **do**

13) **let** $D_{a+}=\{p(u_1,..,u_x) / (\chi,(p,i),'',premise^*,<a_1,..,a_x>) \in E \wedge u_i=g[a_i] \wedge 1 \le i \le x\}$

14) **let** $D_{a-}=\{p(u_1,..,u_x) / (\chi,(p,i),\neg,premise^*,<a_1,..,a_x>) \in E \wedge u_i=g[a_i] \wedge 1 \le i \le x\}$

15) **let** $D_{c+}=\{p(u_1,..,u_x) / (\chi,(p,i),'',conclusion,<a_1,..,a_x>) \in E \wedge u_i=g[a_i] \wedge 1 \le i \le x\}$

16) **let** $D_{c-}=\{p(u_1,..,u_x) / (\chi,(p,i),\neg,conclusion,<a_1,..,a_x>) \in E \wedge u_i=g[a_i] \wedge 1 \le i \le x\}$

17)

18) **if** $D_{a+} \cup D_{a-} \subseteq D$ /* *forward expansion: if premise is fulfilled* */

19) $\vee\ D_{c+} \cup D \neq \emptyset$ /* *or backward expansion: if hypothesis in the conclusion*

20) **then**

21) **for each** $d \in (D_{a+} \cup D_{a-} \cup D_{c+} \cup D_{c-}) - D$ **do**

22) **let** $(N,X)=CreatePlace((d,0),N,X,G,FALSE)$

23) **od**

24) **if** $ExistsMT(N, i, D_{a+}, D_{a-}, D_{c+}, D_{c-})=FALSE$

25) **then**

26) **let** $N=CreateTransition(N, i, D_{a+}, D_{a-}, D_{c+}, D_{c-})$ // *create MT*

27) **for each** $d \in d_{a+}$ **do**

28) **let** $N=CreateTransition(N, i, (D_{a+} \backslash d) \cup D_{c-}, D_{a-} \cup D_{c+}, \emptyset, \{d\})$ **od**

29) **for each** $d \in d_{a-}$ **do**

30) **let** $N=CreateTransition(N, i, D_{a+} \cup D_{c-}, (D_{a-} \backslash d) \cup D_{c+}, \{d\}, \emptyset)$ **od**

31) // *create CTs*

32) **fi**

33) **fi**

34) **od**

35) **od**

36) **return** (N,X)

37) **end**

Figure 4.37. Algorithm *ExpandFPN*

In the following algorithm (*ComputeBindingsForImpl*), we make use of the fact that certain variables of an implication can be computed from other variables by considering the constraints specified for the implication (cf. page 83 for an example). In Definition 4.18, we formalize this concept of variable *derivability.* Furthermore, we define the notion of a *derivation sink* as a variable that can be derived from other variables but is not used to derive variables itself (cf. Definition 4.19).

Definition 4.18 Derivability

*Let $(\iota,V,K)\in I$ be an implication of a GFRN $(P,F^r,F^b,I,E,cf,th,\Omega,\omega)\in \mathcal{L}\{GFRN\}$, $v_1,v_2,..,v_n\in V$ a set of variables, and $x\in\{2,..,n\}$. We say that variable v_1 is **directly derivable** from variables $v_2,..,v_n$ w.r.t. the constraints in K, denoted as $v_1\angle_K v_2,..,v_n$, iff the following condition holds:*

$$\exists(v_1, f, W)\in K\ W\subseteq\{v_2,..,v_n\}$$

$$\vee\ \exists(v_x, f, v_1)\in K\ v_x\in\{v_2,..,v_n\}\wedge\exists f^{-1}\in\overline{FUN}\wedge\forall x\ defn(f(x))\rightarrow f^{-1}(f(x))=x$$

$$\vee\ \exists(\varepsilon,\in,v_1,v_x)\in K\vee\exists(\varepsilon,\in,v_x,v_1)\in K.$$

*We define the notion of derivability based on the transitive closure of the above relation, i.e., a variable $v_1\in V$ is **derivable** from a set of variables $v_2,..,v_n\in V$, denoted as $v_1\angle_{K^*}v_2,..,v_n$, iff*

$$v_1\angle_K v_2,..,v_n$$
$$\vee\ \exists v_{n+1},..,v_{n+m}\in V\ (v_1\angle v_{n+1},..,v_{n+m}\wedge\forall w\in\{v_{n+1},..,v_{n+m}\}\ w\angle_{K^*}v_2,..,v_n).$$

Definition 4.19 Derivation sink

*We say that a variable $v_1\in V$ is a **derivation sink** of an implication $i:(\iota,V,K)$, denoted as $dsink_K(v_1)$, iff the following condition holds:*

$$\exists v_2,..,v_n\in V\ v_1\angle_{K^*}v_2,..,v_n\wedge\neg\exists w_1,..,w_q\in V\ (w_1\angle_{K^*}w_2,..,w_q,v_1\wedge\neg w_1\angle_{K^*}w_2,..,w_q)$$

algorithm ComputeBindings-ForImpl

The algorithm that computes all possible bindings for a given implication with respect to the current propositions in the FPN is given in Figure 4.38.[a] In the first part of *ComputeBindingsForImpl* (lines 6-22), the FPN is searched for all instances of predicates that are connected to the current implication. Line 9 assures that only those propositions are considered in the search that are represented by places with an FBM weakly greater than the threshold of the current implication *i*. The relation of possible bindings (*bindingset*) is created incrementally by binding the actual parameters of found propositions to the corresponding variables and combining each variable binding with all (partial) binding tuples that have been created so far and do not violate the constraints *K* (lines 15-18). Note, that we employ the knowledge about the variable dependencies specified for *i* by excluding all variables that represent derivation sinks. The excluded variables are derived later by applying the specified functional dependencies. For example, in case of implication ι_3 in Figure 4.27 *ComputeBindingsForImpl* would only search the FPN for bindings for the variable tuple (*i, v*) because variable *k* represents a derivation sink (*k* is functionally determined by variable *i*). The bindings for derivation sinks are computed in line 35 by a call to algorithm *ComplementBindingsForImpl* according to their functional dependencies.

dealing with IQs

If there exists an IQ in the premise of the current implication, the two nested loops (at line 25 and line 27) compute bindings with all subsets of conjunctions over the corresponding propositions in the FPN that satisfy the constraints *K*. After the completion of each binding with respect to unbound derivable variables (line 35), the maximum conjunction for each IQ variable is selected in lines 36-38.

a To improve the readability of this algorithm, we consider the case of GFRN arcs with a variable vector of length one, only.

```
algorithm ComputeBindingsForImpl(G, i, N)
1)   input  G:(P,F^r,F^b,I,E,cf,th,Ω,ω) ∈ ℒ{GFRN}; i∈I; N:(S,T,F;D,X,b,v,c,t,m_χ)∈ ℒ{FPN}
2)   output bindingset∈ REL̄
3)   local variables bindingset, bindingset'∈ REL̄, g∈ 𝒰(B)
4)   begin
5)     let bindingset=∅
6)     for each {(χ,(p,i),s,d,<a>)∈E| ¬sink(a)∧¬s=premise_quantified} do
7)                // for each variable that does not represent a derivation sink
8)       let bindingset'=∅
9)       for each {p(u)∈D| m_χ(p(u),s)≥th(i)} do
10)        if bindingset=∅
11)        then
12)          let g[a]={u}
13)          if ConstraintsHold(g,K) then let bindingset=bindingset∪{g} fi
14)        else
15)          for each g∈bindingset do
16)            let g[a]={u}
17)            if ConstraintsHold(g,K) then let bindingset'=bindingset'∪{g} fi
18)          od
19)        fi
20)      od
21)      let bindingset=bindingset∪bindingset'
22)    od
23)    if ∃(χ,(p,i),s,premise_quantified,a)∈E
24)    then
25)      for each {p(u)∈D| m_χ(p(u),s)≥th(i)} do
26)        let bindingset'=∅
27)        for each g∈bindingset do
28)          let g[a]=g[a]∪{u}
29)          if ConstraintsHold(g,K) then let bindingset'=bindingset'∪{g} fi
30)        od
31)        let bindingset=bindingset∪bindingset'
32)      od
33)    fi
34)
35)    let bindingset=ComplementBindings(bindingset,i)
36)    if ∃(χ,(p,i),s,premise_quantified,a)∈E // select maximal bindings for IQ
37)    then let bindingset=bindingset-{g∈bindingset| ∃g'∈bindingset
38)                                     (g'≠g∧g[a]⊂g'[a]∧g[V\a]=g[V\a])}
39)    fi
40)    return bindingset
41)  end
```

Figure 4.38. Algorithm *ComputeBindingsForImpl*

algorithm ComplementBindings

The algorithm that performs the completion of each binding (*ComplementBindings*) is presented in Figure 4.39. The first loop (lines 6-15) considers GFRN constraints with the predefined boolean function '∈'. For each constraint of the form '$\in(w_1,w_2)$' and each binding tuple g it is checked which elements of $g[w_2]$ are valid bindings for w_1. All new valid bindings are added to relation *bindingset*. Moreover, if $\{g[w_1]\}$ is a valid binding for variable w_2 it is added to *bindingset*. The second loop (lines 18-26) uses the defined relational functions to derive bindings for all variables that have not been bound yet. All binding tuples with unbound variables that cannot be derived by bound variables are removed from relation *bindingset*.

algorithm *ComplementBindings(bindingset, G, i)*
1) **input** $G:=(P, F^r, F^b, I, E, cf, th, \Omega, \omega) \in \mathcal{L}\{GFRN\}$; $i:(\iota, V:<v_1,..,v_n>, K)\in I$; $bindingset\in \overline{SET}$
2) **output** $bindingset\in \overline{REL}$
3) **local variables** $bindingset' \in \overline{REL}$; $g\in \mathcal{U}(B)$
4) **begin**
5) **let** *bindingset'=bindingset*
6) **for each** $(\varepsilon, \in, <w_1,w_2>)\in K$ **do**
7) **for each** $g\in bindingset$ **do**
8) **for each** $u\in g[w_2]$ **do**
9) **let** $g[w_1]=u$
10) **if** *ConstraintsHold(g,K)* **then let** $bindingset'=bindingset' \cup \{g\}$ **fi**
11) **od**
12) **let** $g[w_2]=\{g[w_1]\}$
13) **if** *ConstraintsHold(g,K)* **then let** $bindingset'=bindingset' \cup \{g\}$ **fi**
14) **od**
15) **od**
16)
17) **let** *bindingset=bindingset'*
18) **for each** $g\in bindingset$ **do**
19) **let** $V_{bound}=\{v\in V / g(v)\neq\emptyset\}$
20) **if** $\forall v\in V\text{-}V_{bound}\ \exists v_2,..,v_n\in V_{bound}\ v \angle_{K*} v_2,..,v_n$
21) **then**
22) *derive bindings of all* $v\in V\text{-}V_{bound}$ *from g*
23) **else**
24) **let** $bindingset=bindingset\text{-}\{g\}$
25) **fi**
26) **od**
27) **return** *bindingset*
28) **end**

Figure 4.39. Algorithm *ComplementBindings*

4.3.2.3 Complexity and scalability

LDBs often consist of a large number of software artifacts and DBRE methods and tools have to admit to this scale in order to be of practical use. In this section, we reason about the complexity and scalability of the proposed approach to legacy schema analysis.

Worst-case complexity of the proposed algorithms

Complement-Bindings

We start with the analysis of the worst-case complexity of algorithm *ComplementBindings* (cf. Figure 4.39). The complexity of the first loop (lines 6-15) is $O(b*t)$, where b denotes the number of tuples in *bindingset* and t is the maximal cardinality of each set-valued element of such a tuple. The second loop (lines 18-26) has a complexity of $O(b)$. Together, this leads to a worst-case complexity of $O(b*t)$ for algorithm *ComplementBindings*.

ComputeBindings ForImpl

Let us assume that in the first iteration of the outer loop in *ComputeBindingsForImpl* (Figure 4.38, lines 6-22) predicate p_1 is selected. The body of the inner loop that starts in line 9 is iterated d_{p1} times, where d_{p1} denotes the number of places in the FPN that represent instances of this predicate (with the necessary FBM). Consequently, after the first iteration of the outer loop the relation *bindingset* contains d_{p1} tuples. For every further iteration of the outer loop according to an additional predicate $p_x \in p_2,..,p_n$ that does not use an IQ, the relation *bindingset* is extended by at most $d_{px}*b_{x-1}$ elements, where d_{px} denotes the number of instances of predicate p_x and b_{x-1} denotes the cardinality of relation *bindingset* after the most recent iteration. Consequently, the loop from lines 6-22 has a worst-case complexity of $O(d^a)$, where d denotes the number of propositions in the FPN and a is the number of arcs connected to the current implication. If the current implication has an IQ then the worst-case complexity of the loops in lines 25-32 is $O(d^{d*a})$, because the cardinality of *bindingset* might be doubled for each iteration of the outer loop. Given the complexity of algorithm *ComplementBindings*, the total worst-case complexity of algorithm *ComputeBindingsForImpl* is then estimated to $O(d^{d*a})$.

ExpandFPN and GFRNInference

The complexity of algorithm *ExpandFPN* is dominated by the complexity of algorithm *ComputeBindingsForImpl*. The same applies for algorithm *GFRNInference*, because Corollary 4.1 on page 81 guarantees that an axiom-based FPN can be evaluated in linear time. (Clearly, the complexity of *GFRNInference* also depends on the complexity of the employed data- and goal-driven analysis operations.) Moreover, we can only reason about the complexity of one single expansion/evaluation cycle, because the termination of the entire inference process depends on the GFRN and is undecided without the knowledge about the semantics of the functions that are used in the GFRN. For example, Figure 4.40 shows a GFRN which may or may not terminate depending on its input and the semantics of the employed functions. Let us assume that $\cup()$ denotes the union operator, *set()* is a set constructor, and $f(x)$ is true iff *'foo'*$\in x$. Then the inference process does not terminate for an input fact $p^1(\{bar\})$. On the other hand, if $f(x)$ is true iff $|x|<10$ then the inference process terminates after 10 expansion/ evaluation cycles.

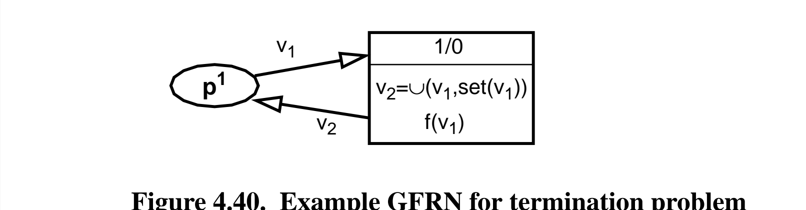


Figure 4.40. Example GFRN for termination problem

Discussion of analysis results

A worst case complexity of $O(d^{d*a})$ for the proposed inference algorithm might seem intractable. However, this exponential effort is only needed for implications that use universal quantifiers (IQ). For all other implications the inference process can be performed in polynomial time $O(t*d^a)$ w.r.t. the number of situation-specific facts in the FPN. The number of connected arcs per implication (a) is usually between 2 and 4.

In case of implications *with* IQs, our approach allows to control the computational complexity by choosing higher TVs for these implications. A higher TV reduces the number d of searched propositions and considers only the most credible ones. Hence, the reengineer can employ TVs to weigh individual GFRN implications according to their computational complexity. Consequently, this strategy allows to scale our approach to LDBs with different sizes. In the next section, we report on our practical experiences.

4.4 Implementing the *Varlet Analyst*

The algorithms described above can easily be implemented in a procedural programming language that provides a basic library of types and functions to deal with sets, tuples, and relations. We have chosen the portable programming language *Java* [GJS97] to implement and evaluate these algorithms in a CARE tool prototype named the *Varlet Analyst*. This tool supports the first phase (*schema analysis*) in the DBRE process sketched in Figure 1.3 on page 6. The following section will outline the architecture of the *Varlet Analyst*, whereas Section 4.4.2 presents the user's perspective. The *Varlet Analyst* is part of an integrated tool environment (*Varlet*) which also supports subsequent DBRE phases (*schema migration* and *data integration*). The remaining parts of the *Varlet* tool environment will be discussed in Chapter 5.

4.4.1 Architecture

The architecture of the *Varlet Analyst* is shown in Figure 4.41. The entire tool comprises approximately 30.000 lines of code. Its core component that deals with the internal GFRN representation and inference is written in *Java*. The concrete design and implementation of the inference engine (IE) including the FPN model is described by Heitbreder [Hei98]. Module *GFRN* encapsulates the logical representation of GFRNs and provides functionality to store and retrieve different specifications. Boolean and relational functions that are used in constraints of GFRN implications are implemented in module *Constraint Functions*. This module is extended during the tool customization process when additional functions are needed (cf. Section 4.4.2.1). Likewise, additional analysis operations can be added to modules *Data-Driven Operations* and *Goal-Driven Operations*.

Analysis operations use basic functionality provided by modules *Code Pattern Extraction*, *Extension Extraction*, and *Schema Extraction*. Module *Code Pattern Extraction* implements customizable detection mechanism for stereotypical code patterns (cf. page 18). Code patterns are specified on a high level of abstraction using *layered graph grammars* (LGG) [RS97]. They are stored in a pattern library that can easily be extended [Bew98]. The actual pattern recognition algorithm is implemented in the graphical programming language *Progres*

[SWZ95] and described by Bewermeyer [Bew98]. Module *Schema Extraction* provides functionality to extract information about the meta data of the LDB, while module *Extension Extraction* allows to access the available legacy data. We use an abstract interface to facilitate the adaption of the *Varlet Analyst* to different DBMS. More precisely, we employ the ODBC[a] standard [Gei95] to interface existing databases because ODBC gateways are broadly available from many DBMS vendors.

The *Varlet Analyst* provides two user interface components: the *Customization Front-End* allows to adapt the tool (i.e., to customize the GFRN specification and analysis operations), while the *Analysis Front-End* is used to apply the tool for legacy schema analysis. These user interface components have been implemented in *iTcl/Tk* [Wel97]. Internally, the logical schema is represented by an abstract syntax graph (ASG) that is initially constructed by a DDL[b] parser implemented with *lex&yacc* [LMB92].

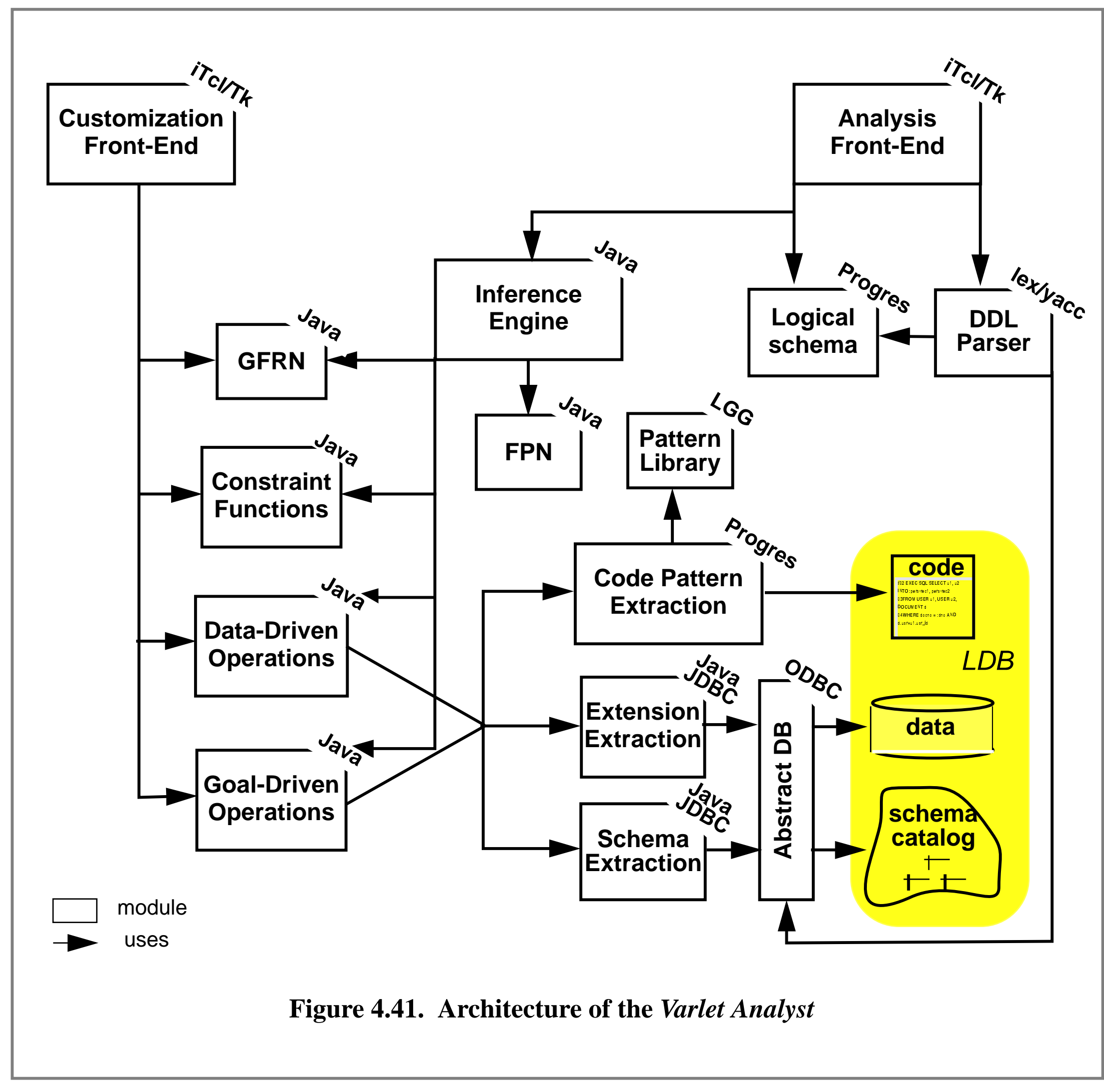


Figure 4.41. Architecture of the *Varlet Analyst*

a Open DataBase Connectivity

b Data Definition Language

4.4.2 User interface

In the following, we will present the user interface of the *Varlet Analyst* from two different perspectives: the next section describes the *Customization Front-End* which is used to *adapt* our tool to a specific application context. Subsequently, we will move to the perspective of the reengineer who uses the *Analysis Front-End* to recover a consistent logical schema of an LDB.

4.4.2.1 The *Customization Front-End*

A graphical editor for GFRN specifications represents the main component of the *Customization Front-End*. This editor can be invoked from the *Varlet Control Panel* which is also used to start all other tools in our CARE environment. Figure 4.42 shows a screenshot of the GFRN editor and the *Varlet Control Panel* (upper right corner). Note, that for technical reasons we use integer values between 0 and 100 to specify CVs and TVs in the *Customization Front-End*. Some implications in the displayed GFRN are already familiar from previous example specifications. They have been labeled by identifiers $i_1,..,i_{14}$ to make it easier to refer to them in our explanation. In order to simplify the representation, the GFRN editor skips variable names for implications which have the same variable associated with all of their in- and outgoing arcs. Note that in our GFRN editor, different types of predicates (data-driven, goal-driven, or dependent) are represented by different colors. Hence, in the grey-scale printout of our screenshot, dependent predicates are marked by black ovals while data-driven and goal-driven predicates are rendered with dark grey and light grey color, respectively.

description of sample GFRN

Implications $i_1,..,i_3$ have already been introduced in Section 4.2.1. Analogously to implications i_{10} and i_{11}, which have been discussed in Figure 4.11 on page 64, implications i_4 and i_5 specify the rule that a hypothetical key may only exist if the corresponding constraint is valid in the available data. Implications $i_{12},..,i_{14}$ represent a refinement of the heuristic given in Figure 4.8 on page 62 that also considers the type compatibility of attributes: i_{13} specifies that similar attribute names might indicate equivalent meaning, while i_{14} represents the knowledge that equivalent attributes have to be type compatible. Implication i_{12} formalizes the heuristic that a set of pairs of equivalent attributes might indicate an IND. Implication i_9 specifies that an instance of a *join* pattern is another indicator for an IND (cf. page 18). Analogously to implication i_8, which has already been known from Figure 4.4 on page 59, implication i_7 classifies an IND as inheritance relationship (I-IND) if there is a (hypothetical) key constraint for its left- *and* right-hand side. Finally, implication i_6 determines an IND to be a cardinality constraint if there exists a key constraint for its left-hand side only (cf. page 21).

multiple views to handle complexity

A typical problem of graphical languages like Petri nets, state charts, and Entity-Relationship (ER) models is that specifications soon become too complex to be visualized in a single diagram. For example, we would like to add further implications to the GFRN in Figure 4.42 representing our knowledge that an R-IND necessarily implies an IND and a key constraints in the referenced table. Other implications could express that the classification of a given IND as an R-IND or a I-IND is mutual exclusive, etc. A commonly used solution to this visualization problem is to use multiple views on a single specification. We adopt this technique in our implementation, i.e., there can be different views on the same GFRN specification. Each of these views might focus on a separate aspect of the analysis process, e.g., detection of keys, detection of INDs, and classification of INDs. Consequently, the reengineer can use another

view to add the necessary conditions for R-INDs, I-INDs, and C-INDs (*i18*, *i16*, *i15* in Figure 4.43) and to specify the mutual exclusiveness of R-INDs and I-INDs (*i17*).

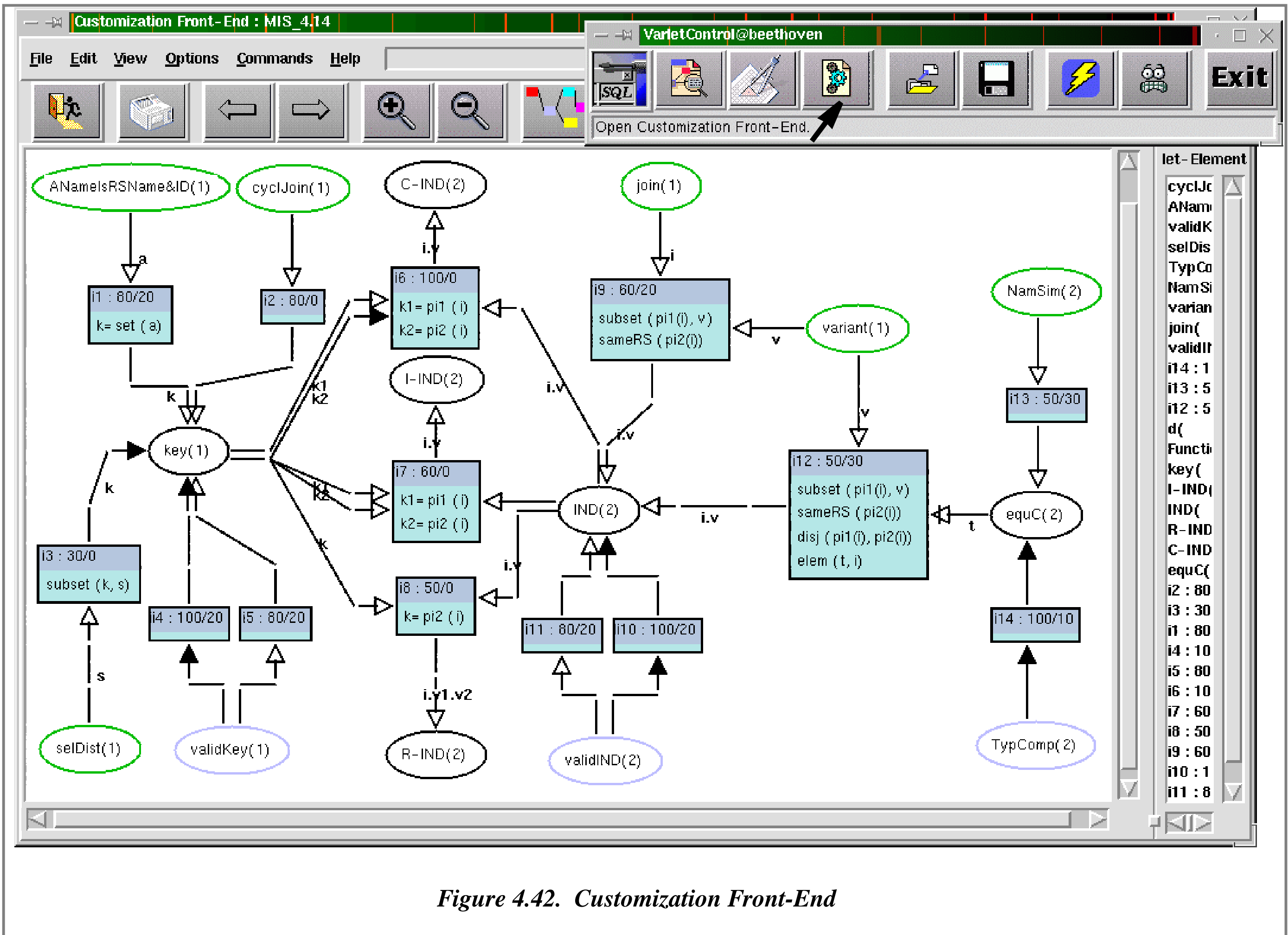


Figure 4.42. Customization Front-End

consistency check

After the reengineer has modified the GFRN specification, (s)he can invoke consistency checks which validate that the GFRN is well-formed (cf. Definition 4.3 on page 67). In addition, these consistency checks use the *Java Reflection API* [Fla97] to ensure that for each data-driven or goal-driven predicate there exists an implementation of a corresponding analysis operation in *Java*. The same applies for functions used within constraints of GFRN implications. The screenshot in Figure 4.42 shows a consistency report which indicates three missing *Java* implementations, namely the implementations for the function *disj*, the goal-driven operation *validIND*, and the data-driven operation *selDist*. The reengineer uses generated code frames to implement missing *Java* methods used in the GFRN specification. Obviously, implementing additional analysis operations is the most time-consuming activity in the customization process of the *Varlet Analyst*. However, our tool provides the reengineer with a predefined library of standard functionality that facilitates this task, e.g., DB login and access, operations on result relations, and parameterizable fuzzy membership functions. Often, it is not even necessary to add further analysis operations but the reengineer just uses the GFRN editor to change specified heuristics or modify their credibility.

implementation of analysis operations

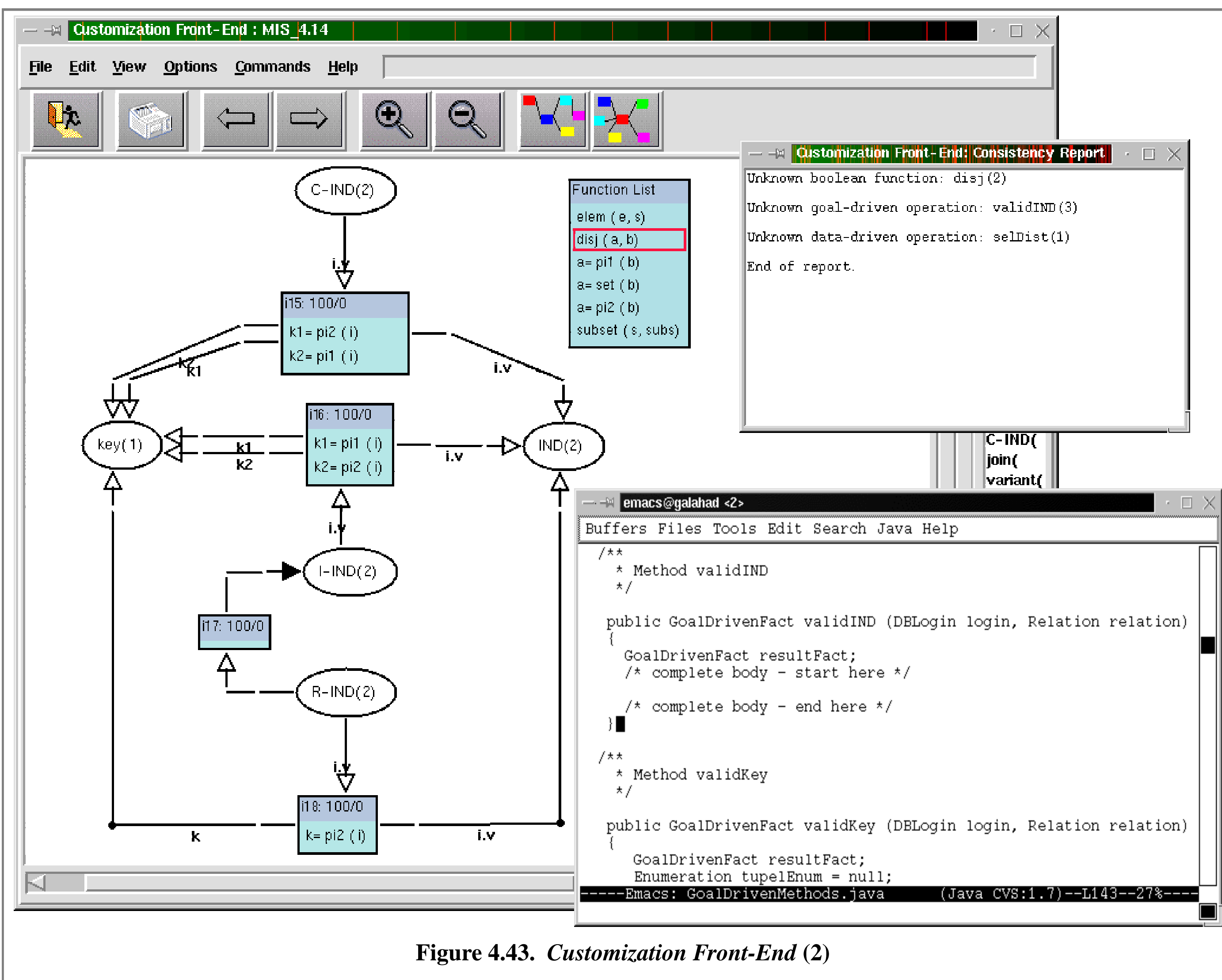


Figure 4.43. *Customization Front-End* (2)

4.4.2.2 The *Analysis Front-End*

After the *Varlet Analyst* has been customized w.r.t. to its current application context it can be applied for legacy schema analysis. The first step in this process is to extract the schema catalog from the LDB under investigation. Subsequently, all data-driven analysis operations specified in the GFRN are executed to deliver indicators for additional semantic constraints. The result of this initial automatic analysis step is graphically presented to the reengineer in the so-called *Analysis Front-End* (cf. Figure 4.44). In the *Analysis Front-End*, each box represents a table and INDs are visualized by lines. In order to cope with large schemas, the reengineer can choose from various levels of abstraction and create separate views on the same logical data structure. At the beginning of the analysis of a large LDB schema it is more appropriate to choose a view that hides details and allows to cluster groups of tables into subsections that can be analyzed separately [SdJPeA99]. The *Analysis Front-End* provides different layout algorithms to facilitate this activity.

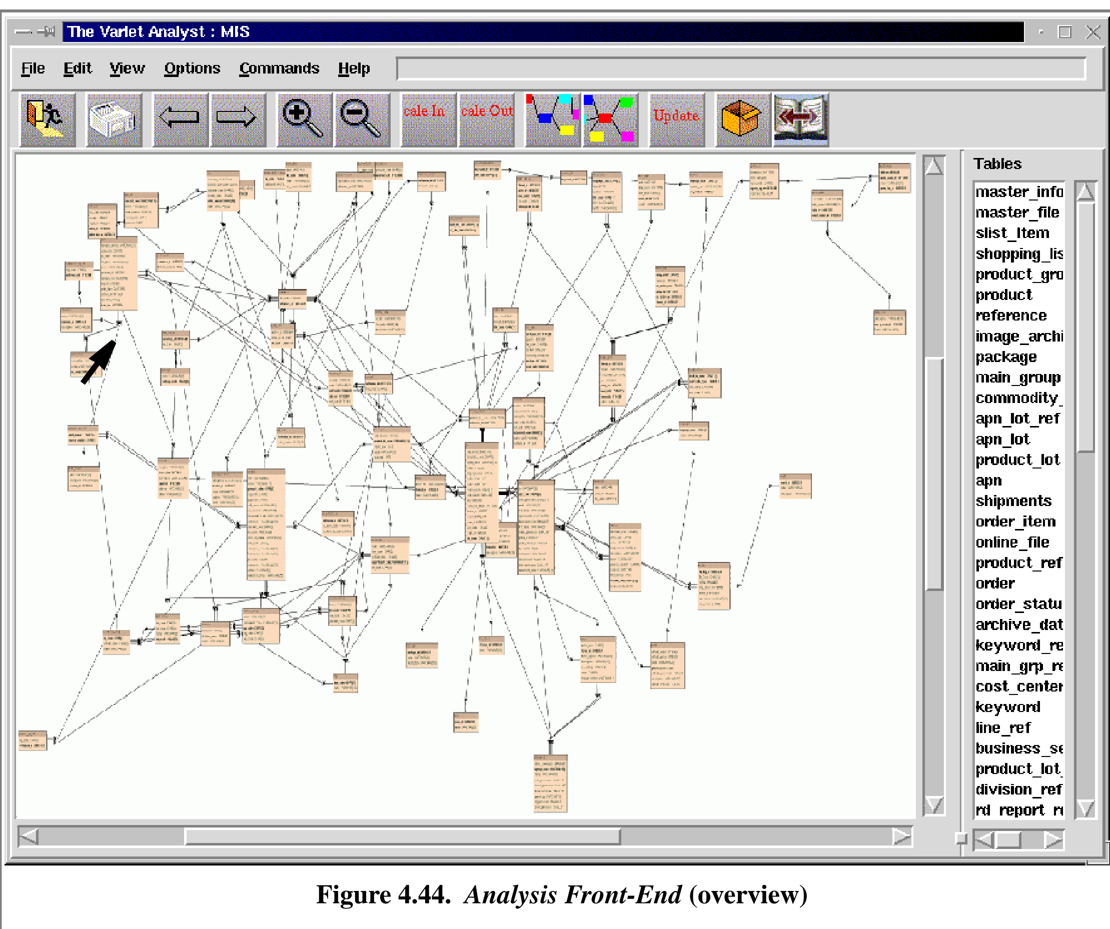


Figure 4.44. ***Analysis Front-End*** **(overview)**

detailed representation

Figure 4.45 presents another screenshot of the *Analysis Front-End* with a more detailed view on the eight sample tables from our sample scenario. It shows all attributes with their corresponding type. Key attributes are set in bold font. Additional information might be represented at the bottom of each table, e.g., the representation of table *USER* indicates the existence of another key (*2 keys*) which can be displayed using the *ShowNextKey* command from the *Commands* menu. Likewise, the representation of table *PRODREF* indicates the existence of four different variants (cf. page 20). All attributes and INDs which do not belong to the currently displayed variant are dimmed. Again the reengineer can chose from various degrees of detail. For example, most INDs are represented as single lines in Figure 4.45, but the reengineer selected a detailed representation of the IND between tables *PRODUCT* and *PRODREF*. For this IND correspondences between pairs of referencing attributes are marked by numbers. Note, that different triangular icons are used to represent INDs according to their classification:

- symbol ⯈ represents INDs without a further classification;
- an additional key symbol ⯈ marks INDs which have been classified as key-based references (R-INDs);
- symbol ⯈ marks INDs which also imply an IND (C-IND) in the reverse direction;
- again, an additional key symbol ⯈ represents R-INDs which imply an inverse C-IND;
- finally, key-based INDs which have been classified as inheritance relationships (I-INDs) are marked by white triangles ▷.

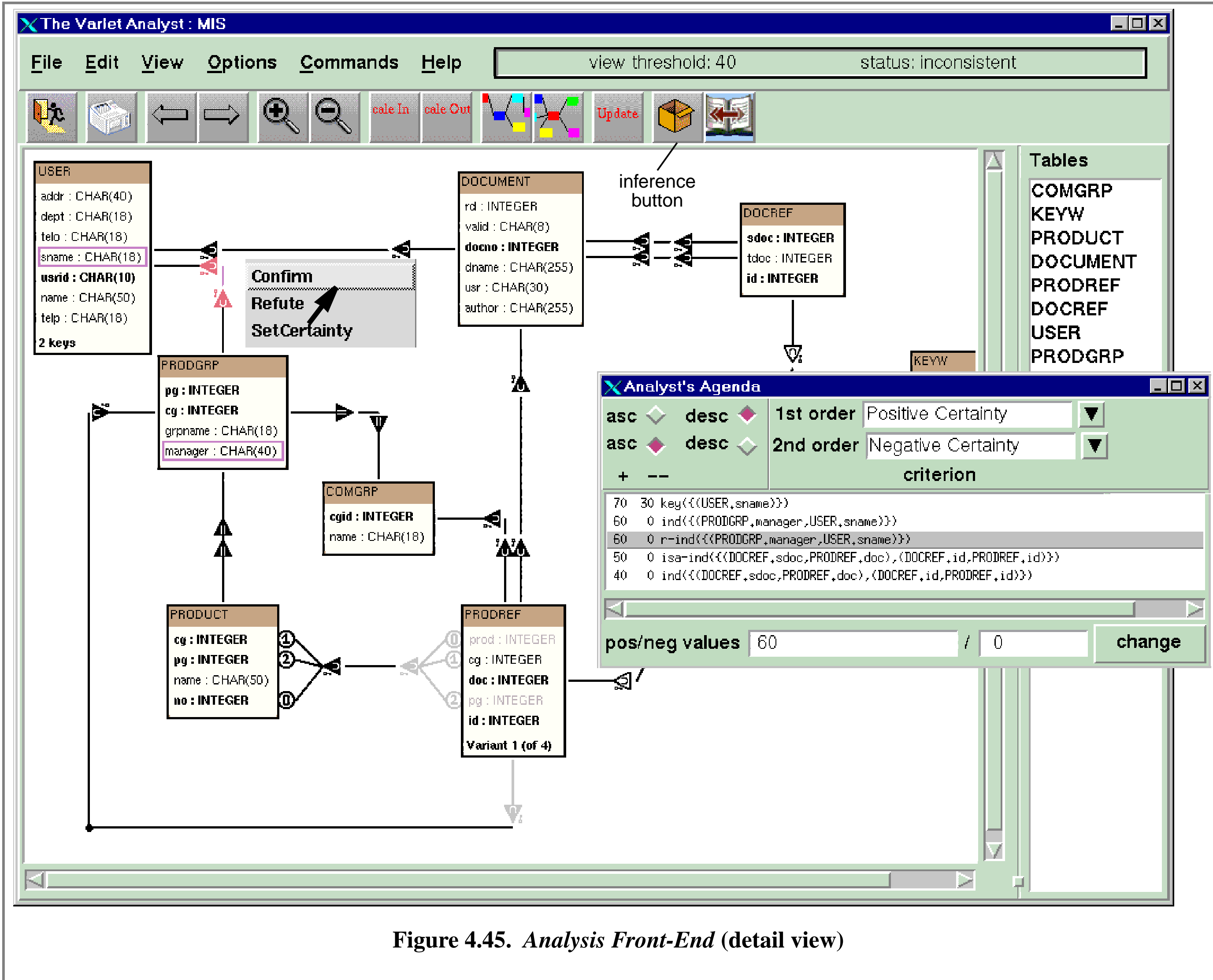


Figure 4.45. ***Analysis Front-End*** **(detail view)**

visualizing imperfect information

A key achievement of our approach is that we relax the requirement for consistency during the legacy schema analysis process. Consequently, the *Varlet Analyst* has to provide means to visualize such imperfect information about LDB schemas. A central problem that arises with such a visualization stems from using a quantitative measure to represent uncertainty. We have to avoid that the schema representation is overloaded by too many hypothetical constraints with low credibility. We solve this problem by introducing the concept of a *view threshold* which determines a lower limited of certainty for all schema artifacts displayed in the current view. Consequently, the semantics of a view threshold is an α-*cut* on the fuzzy set of all certain schema artifacts (cf. page 50). In the *Varlet Analyst*, the view threshold is displayed in the status bar over the graphical window (cf. page 105). It can be changed *on-the-fly* by the reengineer. Note, that the view threshold does only consider the certainty in favor of a hypothesis but it disregards the certainty against it. Hence, the graphical view contains also contradicting information as long as these hypotheses have not been refuted completely, i.e., as long as they have a negative certainty lower than 100 (1).

indicating imperfect information

The ultimate goal of the schema analysis process is to come up with a consistent logical schema for the LDB under investigation. In an evolutionary process, the reengineer confirms or refutes uncertain hypotheses and resolves contradictions to obtain this result. In order to do this efficiently, a CARE tool that tolerates imperfect knowledge has to provide powerful mechanisms to *indicate* imperfect information to the reengineer and *guide* him/her during the analysis. For this purpose, we have developed a dedicated dialog called the *Analyst's Agenda* which is shown on the right side of page 105. The *Analyst's Agenda* presents a list of uncertain or contradicting constraints about the current view of the logical schema. For each constraint a positive and a negative certainty is displayed. The *Analyst's Agenda* provides the functionality to sort the list items according to various criterions in ascending or descending order, e.g., positive certainty, negative certainty, degree of contradiction (absolute difference of both certainties). When the reengineer selects an entry in the agenda the corresponding graphical representation is highlighted in the main window. In Figure 4.45, the reengineer has selected the foreign key from *PRODGRP* to *USER* which has been inferred with a certainty of 60 (0.6). Let us assume that after investigating the form-based user interface of PDIS (s)he confirms that the inferred foreign key in fact represents a reference between product groups and product managers (stored in table *USER*). This can be done by selecting the *Confirm* command from the context-sensitive menu of the *Varlet Analyst* (cf. Figure 4.45). Likewise, (s)he can proceed and do further manual investigations and annotations according to the displayed agenda or additional knowledge.

automatic inference

On the other hand, (s)he can invoke the inference engine (IE) at any point in time to resume the automatic analysis process. This can be done by pressing the *inference* button on the *Varlet Analyst*'s icon bar. When invoked, the IE propagates the schema modifications and executes goal-driven operations if necessary (cf. Figure 4.23 on page 82). This automatic analysis and inference step is performed asynchronously in a separate thread. The reason for this solution is to allow the reengineer to continue his/her investigation during the inference process. At the end of each inference step the schema representation is not updated automatically but the availability of the inference result is indicated to the reengineer by changing the icon on the inference button from an empty box to a full box. We have chosen this solution to avoid confusion due to spontaneously updated representations. In the sample situation displayed in Figure 4.45, the schema update produced by the IE will remove three entries from the agenda: the selected R-IND will be removed because it has been confirmed. Moreover, the first two entries will be removed as well because, according to the GFRN in Figure 4.43, they represent necessary preconditions for the confirmed R-IND. When the agenda is empty the current view of the schema is consistent w.r.t. the defined view threshold. In this case, the reengineer can either decrease the *view threshold* and investigate hypotheses with lower certainty (if existent) or (s)he can decide to neglect all remaining hypotheses with a lower certainty, produce annotated textual and graphical documentation (cf. Figure 4.46), and continue with the conceptual schema migration process (cf. Chapter 5).

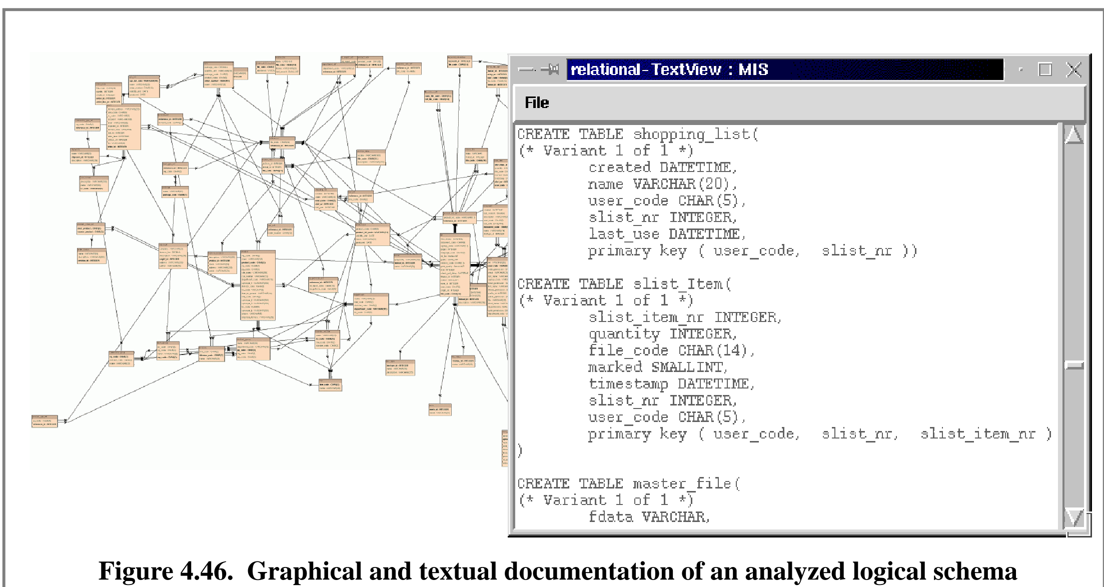


Figure 4.46. Graphical and textual documentation of an analyzed logical schema

4.5 Evaluation

first prototype

We have chosen an incremental approach to stepwise implement, evaluate, and refine our approach. We created our first implementation prototype with the high-level specification language *Progres* which has been developed at RWTH Aachen (Germany) [SWZ95]. In particular, this language has been well-suited because it is based on the notion of graphs[a] as the central implementation paradigm and GFRNs as well as FPNs are graph-oriented structures. Moreover, the *Progres* development environment includes customizable graph visualization tools which we employed as a rudimentary user interface for the *Varlet Analyst*. We used this initial implementation with small-scale schema reverse engineering problems to validate and refine our concepts. We learned that our approach is feasible in principle but the tool lacked adequate abstraction mechanisms and user dialogs to make experiments with larger case studies and attract potential users. Moreover, the performance of the tool became weak when the FPN grew larger because every data structure in *Progres* is stored persistently in a graph-oriented database with full support (and overhead) for transaction management and recovery.

second prototype

In fall 1997, we decided to (re)implement the inference engine in *Java* and create a dedicated user interface in *iTcl/Tk*. This enabled us to use *transient* FPN data structures to perform the inference process. Still, we left the internal representation of the analysis *results* (i.e., the analyzed logical schema) in the *Progres* repository to exploit the benefits of error recovery. The *Java* inference engine was about 15 times faster than the former *Progres* version [Hei98]. Moreover, the *Java Reflection API* [Fla97] allowed us to bind existing data- and goal-driven analysis operations to GFRN predicates *on-the-fly* without the need to recompile our tool. Obviously, the implementation of *new* operations with a compiler-based language like *Java* still needed recompilation. Hence, we considered using an interpretative scripting language to define analysis operations, e.g., an extension of *Tcl* [Wel97] or *Perl* [WS90]. However, the

a cf. Definition 5.1

overhead caused by the recompilation (less than one minute on a 300 MHz Sun Sparc II) was too low to justify this effort.

case study

In spring 1998, we started a collaboration with two German companies (*eps Bertelsmann* and *Merck KGA*) who provided us with a practical case study for our approach. The database schema consisted of 85 tables and 347 attributes; the database access component comprised 26.000 lines of code.[a] By implementing multiple views on the same logical schema with various levels of abstraction, we improved the usability of the *Varlet Analyst's* user interface to visualize larger application examples. After executing all automatic data-driven analysis operations, we made a first attempt to apply the GFRN inference engine to obtain initial hypotheses about schema constraints. We cancelled the inference process because it did not terminate within 30 minutes. A postmortem investigation revealed that the data-driven analysis operation $NamSim^2$ (cf. Section 4.2.1) was responsible for this undesired behavior: it produced over one thousand indicators for INDs because the example schema contained many similar column names. Most of these indicators could be falsified by the automatic analysis of the available data with the goal-driven operation $validIND^2$. However, this process was very time-consuming.

domain analysis

The experience described above emphasized the importance of the domain analysis and GFRN customization step (Figure 4.1 on page 56) before starting the actual schema analysis process. The inadequacy of the aforementioned naming heuristic could be detected with little effort by browsing the schema catalog before starting the analysis process. Within a few minutes, we discovered that in many cases the developers named columns (which seemed to be foreign keys) similarly to other tables. Consequently, we replaced the $NamSim^2$-heuristic with another heuristic which is based on similarities among column and table names to indicate INDs. The customization of the GFRN and the implementation of the new analysis operation took less than ten minutes. Subsequently, we restarted the analysis process. This time, the inference engine terminated after five minutes and indicated 46 possible INDs (out of 111 actually existing INDs). The new naming heuristic delivered 29 INDs. Another 26 INDs were indicated by instances of *join*-patterns in the database access code (cf. Section 2.4.1), but 9 of these INDs could be falsified by the automatic execution of goal-driven operation $validIND^2$. In combination with the specified and detected key constraints, 24 INDs were classified as R-INDs, 11 INDs were (primarily) classified as I-INDs, and 3 INDs were classified as C-INDs. All indicated INDs turned out to be valid. Still, we had to resolve contradictions caused by the ambiguous classification of I-INDs (cf. Section 4.4.2.1).

user guidance

We made the experience that the user needs additional guidance to detect and resolve such contradictions: so far, our tool only supported the concept of *view thresholds* and the possibility to query each schema constraint for its associated certainty (cf. Section 4.4.2.2). Using these mechanisms to find and eliminate uncertain and contradicting information about the logical schema turned out to be a tedious activity for larger examples. Hence, we introduced the *agenda* concept described in Section 4.4.2.2 with querying, sorting, and high-lighting facilities which drastically simplified this activity.

concurrent inference

The proposed iterative process of manual investigations, goal-driven analysis, and automatic inference and propagation of results proved to be of great benefit for our application examples.

a The entire system had a size of several hundred lines of code.

Still, the experimental users of the *Varlet Analyst* complained that they had to wait for the inference engine to terminate each time they resumed the automatic analysis process. This was disturbing because the entire analysis/inference cycle took up to several minutes for the example schema. Therefore, we implemented the inference engine in a separate process that ran in parallel to the *Analysis Front-End*. This solution allowed the users to invoke the inference engine and proceed with their manual investigations. Still, it had the spurious side-effect that sometimes spontaneous screen updates caused by the inference results interfered with manual analysis activities. Hence, we decided to *indicate* the availability of new inference results in the *Analysis Front-End* and let the user decide when the graphical representation should be updated accordingly.

experiences with the current tool

Our experiences with the current version of the *Varlet Analyst* are positive. By incorporating imperfect DBRE knowledge, our tool provides significantly better support for schema analysis and completion than existing approaches. With little effort, it can be customized to the characteristics of different legacy schemas. We learned that it is especially important to adapt heuristics that deal with naming conventions in this customization step. Even though the current prototype still has a number of technical problems, which mainly stem from combining multiple languages (*Java, Progres, C, and iTcl/Tk*), we are confident that many of our implemented concepts have the maturity necessary to find their way into commercial DBRE tools. Still, a frequently mentioned point of criticism with our approach is that confidences of heuristics are hard to estimate in terms of real numbers. Introducing a limited set of *symbolic* confidences to choose from (e.g., *certain*, *more or less certain*, *weakly certain*) could ease the specification of heuristics.

4.6 Related work

Blaha and Premerlani

Most existing approaches to legacy schema analysis aim to recover a complete logical schema by following a predefined process of subsequent reverse engineering activities. Some approaches suggest loosely coupled tools to support certain activities. In [PB94, BP95, Bla98], Premerlani and Blaha report on their experience in schema analysis using simple tool sets which mainly contain UNIX tools [RRF90] like *grep* and *awk* and predefined SQL queries. They argue that a flexible, interactive approach to DBRE is more likely to succeed than batch-oriented compilers. The proposed DBRE process is based on the *Object Modeling Technique* (*OMT*) [RBP+91] and starts with an initial object model where each RS represents a candidate class. Subsequently, the reengineer has to detect abstract design concepts based on a set of informal heuristics, guidelines, and clues [BP98]. The main drawback of their approach is that loosely coupled tools provide little support for exchanging and combining analysis results, automatically. They lack the ability to control, propagate, and indicate inconsistencies. Moreover, they play a mostly *passive* role in the DBRE process. This means that the reengineer is responsible to invoke analysis operations for code, data, and schema inspection, manually. Our approach overcomes this limitation and allows to integrate such existing analysis operations in a GFRN as a common framework (cf. Section 4.2.2).

Petit et al.

Petit et al. present an approach to analyze queries in existing application code to derive semantic constraints about legacy schemas, e.g, INDs and inheritance relationships [PKBT94, PTBK96]. They search the application code for stereotypical patterns like *equi-joins*, *auto-joins*, *set operations*, and *group-by clauses* which serve as semantic indicators. Once such

Andersson

semantic indicators have been detected, Petit et al. use additional queries to the available data in order to determine further information about the hypothetical constraints, e.g., the cardinality of associations or the direction of inheritance relationships. Similar to Petit's approaches, Andersson employs stereotypical code patterns as semantic indicators for key and foreign key constraints [And94]. Both methods can be integrated in our approach in form of analysis operations which are bound to GFRN predicates. For example, *data-driven* operations can be used to search for initial indicators while *goal-driven* operations perform further cardinality analysis and validation for all indicators found.

Signore et al.

In [SLGC94], Signore et al. present a knowledge-based approach to DBRE that uses *Prolog* clauses [Wil86] to infer schema constraints from detected semantic indicators. In a first step, indicators for primary keys, candidate keys, and foreign keys are collected by comparing names and types of attributes and investigating their usage in the application code. Each indicator is stored as a *Prolog* clause in the fact base of the DBRE tool. The second step is called *conceptualization*. In this step, predefined heuristics modeled as *Prolog* rules are used to infer abstract modeling concepts like *many-to-many* relationships, complex attributes, and generalizations. These pattern recognition rules can easily be adapted by the reengineer, which is similar to our approach. Still, a significant drawback of Signore's tool is that the employed *Prolog* interpreter supports only backward reasoning, i.e., it validates hypotheses of the reengineer but it does not create new hypotheses. Hence, it restricts the reengineer to a top-down analysis process. Moreover, there is no tool support to detect indicators (e.g., instances of code patterns, naming conventions, variant structures). All indicators have to be present before the inference process and there is no mechanism to execute goal-driven analysis operations on-demand. Another limitation of Signore's approach is that heuristics and uncertain results are represented as definite clauses without a valuation for their credibility or their contradiction.

Hainaut et al.

A comprehensive CARE environment for schema analysis and migration (*DB-Main*) has been developed since 1993 at the University of Namur, Belgium [HEH+96, HHHR96]. It provides the reengineer with a powerful scripting language called *Voyager 2* [Eng98]. *DB-Main* includes several predefined scripts for extracting data structures declared in catalog tables and source code [HEH+98]. *Voyager 2* allows the reengineer to extend the set of available extractors by new analysis operations. Even though this approach is very powerful its disadvantage is the low level of abstraction: DBRE heuristics and processes are coded in procedural scripts whereas declarative formalism would be more appropriate. It takes a significant amount of training to learn how to use *Voyager 2* to customize the analysis process. Moreover, extractor scripts have a passive nature, i.e., they have to be invoked explicitly by the reengineer. In our opinion, a combination of *Voyager 2* scripts to develop data- and goal-driven analysis operations with the declarative GFRN approach to specify heuristics and active processes seems most promising and beneficial.

Hodges and Ramanathan

In [RH97, RH96], Hodges and Ramanathan describe a method to identify abstract concepts like associations, aggregations, and inheritance structures in relational schemas. Their approach is based on the assumption that the relational schema description is structurally complete, i.e., the reengineer has complete information about key and foreign key constraints. This assumption is too idealistic for many existing LDB systems [HCTJ93]. Vossen and Fahrner describe similar techniques to annotate relational schemas semantically [FV95]. However, their approach also covers the phase of structural schema completion (cf. Section 2.4.1): they propose an algorithm to infer INDs based on equivalence classes of

Vossen and Fahrner

relational attributes. We specified central ideas of their method in the GFRN specification described in this chapter.

Soutou

Several other methods and algorithms have been proposed to detect indicators for structural or semantical information about relational LDB schemas. Soutou presents an algorithm to recover *n-ary* associations [Sou98a]. This analysis is performed in two steps: firstly, information about key and foreign key dependencies are used to identify candidates for RS that represent *n-ary* associations. Secondly, an algorithm generates tentative queries to determine cardinality constraints for these associations. Based on the analysis results, Soutou proposed a method to recover *aggregate* relationships in relational schemas [Sou98b]. Blockeel and De Raedt adopt methods known from the domain of *inductive logic programming* (ILP) to detect constraints in relational DBs [BR97]. They propose an algorithm to find relationships among different RS that can be implemented in SQL. The GFRN approach described in this dissertation allows to integrate such algorithms in terms of data- and goal-driven analysis operations.

Blockeel and De Raedt

DBInformer, ERwin, SeeData

Some tools have their primary focus on *visualizing* existing LDB structures. Most of these approaches generate graphical networks of entities and relationships which can be browsed and annotated interactively by the reengineer, e.g., *DBInformer* [Him97] and *ERwin* [Log97]. The problem of such graph-oriented representations is that they tend to clutter for larger database schemas (several hundreds of tables). A more scalable approach to schema visualization has been developed at AT&T Bell Labs [AEP96]. Their tool (called *SeeData*) provides several different views which display separate aspects of an LDB on various levels of abstraction. These views also cover the relationship between the LDB schema and the corresponding application code. This allows the reengineer to determine those parts of the code which are affected by a given schema modification. More powerful visualization techniques and the ability to browse the source code which is associated to certain schema artifacts would further increase the usability if the *Varlet Analyzer Front-End*.

Sousa

An approach to reverse engineer large LDB schemas that follows the divide-and-conquer paradigm has been developed by Sousa et al. [SdJPeA99]. Their idea is to use information about primary keys to cluster relations into so-called *abstract entities* and *relationships*. Each abstract entity (and relationship) represents an excerpt of the entire LDB schema which is reverse engineered separately. In a final step, the resulting reverse engineered subschemas are integrated to a common schema and completed with missing elements. Sousa's approach can be viewed as a *meta process* as they do not make any assumption on the actual method which is used for schema analysis. An integration of similar clustering techniques as pre- and postprocessing steps in our schema analysis process could further increase the scalability of our approach. However, one limitation of Sousa's original method is the lack of control about the granularity of clusters: some clusters are composed by a large numbers of relations while others consists of a single relation.

4.7 Summary

In this chapter, we have elaborated an approach to incorporate and exploit imperfect knowledge in human-centered DBRE processes. Our research was driven by the observation that imperfect knowledge plays an important role in database schema analysis activities. Currently existing approaches to DBRE do not consider imperfect knowledge. They presume a

mostly *monotonic* schema analysis process that consists of accumulating definite (and consistent) knowledge about an LDB until the structural and semantic information about the schema is complete. We set up the hypothesis that by temporarily relaxing this requirement for consistency and precision, we would be able to develop a DBRE tool that considers the human reasoning process of reengineers more adequately.

To solve this problem, we proposed an evolutionary analysis process controlled by a *non-monotonic* inference engine that propagates intermediate results and automatically invokes analysis operations. We introduced *Generic Fuzzy Reasoning Nets* (GFRNs) as a dedicated, abstract formalism to specify domain-specific heuristics and integrate automatic analysis operations. A major concern with the development of the GFRN language was that GFRN specifications can be customized with little effort to changing application contexts. The motivation for this requirement was our observation that the heuristics and operations applied in a schema analysis process depend on the specific characteristics of the LDB system under investigation. Syntax and semantics of the GFRN language have been defined in the formal framework of necessity-valued possibilistic logic.

Based on the notion of *fuzzy Petri nets*, we have developed an inference algorithm to operationalize GFRN specifications in human-centered DBRE processes. The implementation of this inference algorithm in a procedural programming language is straight-forward. We experimented with implementations in *Progres* and *Java*. Early experiences with practical application examples showed the feasibility of our approach. However, they also emphasized the importance of dedicated user interface concepts to communicate imperfect information to the reengineer and efficiently guide him/her to a complete and consistent analysis result. We implemented and refined such concepts in the current version of our DBRE environment (*Varlet Analyst*). An evaluation of the *Varlet Analyst* in an industrial project clearly showed the benefits of our approach over existing DBRE tools and validated the hypothesis stated at the beginning of this section.

CHAPTER 5 CONCEPTUAL SCHEMA MIGRATION AND DATA INTEGRATION

Someone must maintain the mapping between the entity-relationship diagram and the relations in the database as the database evolves. This can be a difficult task.

Antis et al. [AEP96]

In the previous chapter, we developed concepts, techniques, and tools to support reengineers in analyzing legacy database (LDB) schemas. The output of such an analysis activity is a logical schema that has been annotated structurally and semantically as far as possible (cf. Figure 4.46). Based on this intermediate result, the present chapter focusses on two important subsequent database reengineering (DBRE) activities, namely *conceptual schema migration* and *data integration*.

schema migration

As exemplified in Chapter 2 (Section 2.4.2), conceptual schema migration aims to produce an abstract design for an LDB schema. High-level modeling concepts like objects, aggregation, and inheritance are employed in this human-intensive activity that cannot be performed fully automatically [ALV93]. The resulting conceptual schema provides a level of abstraction that is suitable to facilitate understanding and assessment of an LDB's static structure. Furthermore, it is a prerequisite to achieve a large variety of maintenance goals, e.g., the integration with enabling technologies like object-orientation, the Internet, and Client-Server architectures [Uma97].

problem of iterations

Most currently existing computer-aided reengineering (CARE) tools that support schema abstraction and migration generate an initial conceptual schema based on a given logical schema (e.g., [BGD97, MCAH95, RH97, Fon97, MAJ94]). Subsequently, the reengineer uses another tool to restructure, enhance, and annotate this initial conceptual schema (e.g., [Log97, Rat98]). Even though, these approaches allow to validate the consistency of the created conceptual schema itself, they hardly provide any support to check the consistency among different documents in the entire DBRE project. This is a severe limitation because the DBRE process has an explorative and iterative character (cf. Chapter 2). Whenever the information about the logical schema is revised, the consistency with the conceptual schema that has been created so far is lost. Using such loosely integrated approaches, the only possibility to re-establish consistency automatically is to generate the conceptual schema anew. In this case, interactive enhancement and redesign operations performed by the reengineer are lost and have to be repeated manually.

problem of data integration

Like in our case study, many DBRE projects focus on *integrating* LDBs with new technologies rather than aiming on their complete replacement. Often, the conceptual schema is used as a basis to define the class structure of object-oriented applications that access the LDB. Frequently used programming languages for such applications are *Java* [WM97] and *C++* [Str97]. In such scenarios, a programmer has to develop a *middleware* component for data integration that implements the data dependencies between the logical schema and the migrated conceptual schema. Middleware generators that have been developed to forward

engineer *new* information systems prescribe a canonical (mostly object-relational) mapping that generally lacks the flexibility to integrate arbitrary pre-existing LDB schemas. Even with approaches which focus on integrating pre-existing systems it is still the responsibility of the reengineer to define a consistent schema mapping description.

approach: tight integration

To overcome this limitation, we adapt techniques described by Nagl et al. [Nag96] to the DBRE domain. This means that we propose a fine-grained integration of tools used in the different phases of the DBRE process (i.e., schema analysis and migration) by a common *migration graph* structure. This approach enables incremental change propagation and consistency preservation and, thus, supports process iterations. In addition, the migration graph is used to map changes in the conceptual schema back to the implementation model, i.e., the logical schema. Another benefit of this tight integration is that it allows to generate middleware components for data integration based on the schema mapping information that is maintained implicitly. This is a significant progress over existing approaches to middleware generation where it is the responsibility of the reengineer to define a consistent schema mapping description manually [CER99, Hüs97, ONT96, Rad95].

The described approach to incremental schema migration and generative data integration is illustrated in Figure 5.1. The migration process starts with a canonical translation of the analyzed logical schema into a conceptual data model. Then, the resulting conceptual schema is redesigned and extended interactively by the reengineer. The grey parts in Figure 5.1 actually belong to the schema analysis process described in Chapter 4. They are shown to emphasize the fact that *incremental* schema migration in a tightly integrated DBRE environment enables iterative and intertwined execution of analysis and migration activities. Internally, the logical schema and the conceptual schema are represented by their *abstract syntax graphs* (ASG). The dependencies between both schemas are represented by an intermediate graph called the *schema mapping graph* (SMG). In case of process iterations, the information maintained in the SMG is employed to control incremental change propagation operations that aim to re-establish project consistency. Moreover, the SMG is taken as the basis to generate an object-relational middleware layer without the need for the user to define schema dependencies explicitly.

The approach outlined above is described in detail in the following subsections: Section 5.1 introduces and formalizes the *migration graph model* which covers both ASG representations and the SMG model. Based on this formalization, in Section 5.2, we employ *triple graph grammars* [LS96] to specify a mapping between the relational and the conceptual data model. This mapping is used to perform an automatic translation of a relational schema to an *initial* conceptual schema. In most cases, such an automatic translation is unsatisfactory and has to be redesigned or extended to meet new requirements. Hence, in Section 5.3, we define a catalog of conceptual schema redesign transformations that can be applied interactively by the reengineer. Section 5.4 is dedicated to the problem of re-establishing the consistency *and* preserving as many of these interactive redesign transformations as possible in case of process iterations. An implementation of the described concepts and techniques is presented in Section 5.5. In Section 5.6, we describe a generative approach to object-relational data integration based on mapping information that has been created and maintained implicitly during the schema migration process. Section 5.7 evaluates our approach and reports on our practical experiences

with application examples. A discussion of related work in this domain is presented in Section 5.8. Finally, Section 5.9 gives a summary of the main contributions of this chapter.

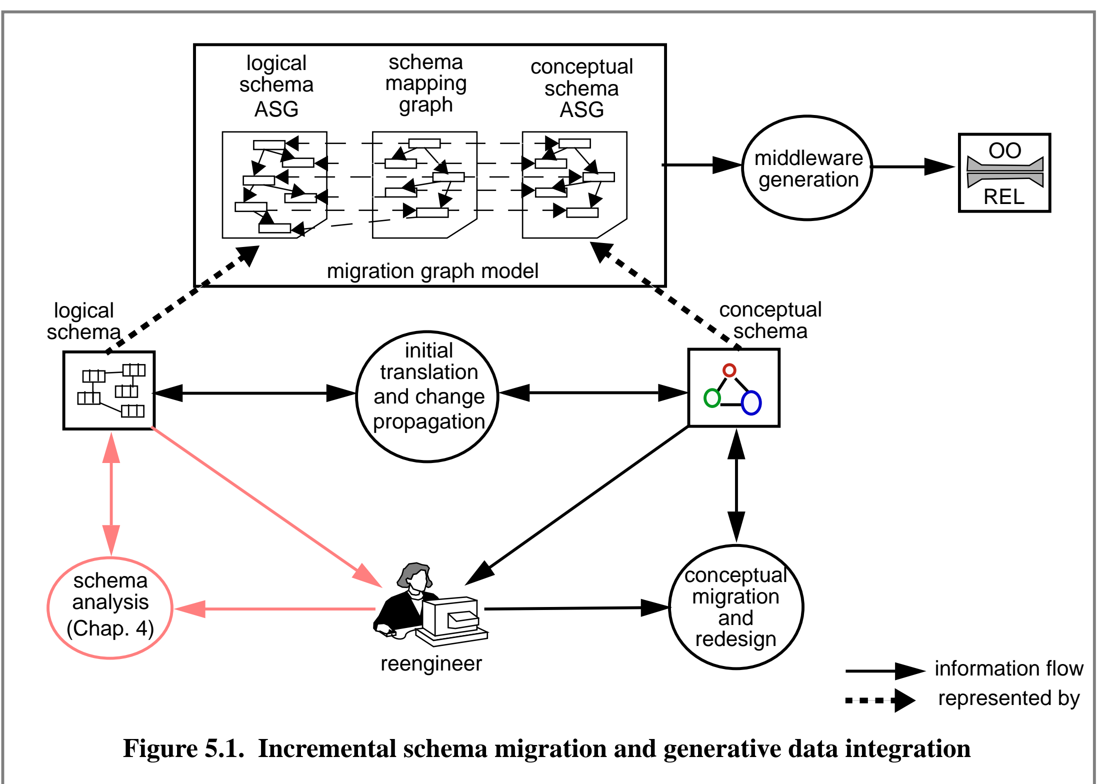


Figure 5.1. Incremental schema migration and generative data integration

5.1 The migration graph model

The formal basis for the migration graph is the concept of a *directed, attributed graph with node and edge types* [Eng86]. In the following, we use the term *graph* for abbreviation whenever we refer to a directed, attributed graph with node and edge types. Such a graph can be defined as shown in Definition 5.1.

graph

Definition 5.1 Graph

$G := (N, E, y_N, A)$ is a graph *over two given type label sets L_N, L_E with:*

- *$N(G):=N$ is a finite set of **nodes**;*
- *$E(G):=E \subseteq N \times L_E \times N$ is a finite set of **edges**;*
- *$y_N(G):N \rightarrow L_N$ is a **typing function** for nodes;*
- *A is a finite set of **node attributes**, each $a \in A$ is a partial function $a:N \rightarrow dom(a)$, where 'dom(a)' denotes the domain of attribute 'a'.*

Moreover, we define the following auxiliary functions:

- *$s(G):E \rightarrow N$ with and $t(G):E \rightarrow N$ with $s((n_1,l,n_2)):=n_1$ and $t((n_1,l,n_2)):=n_2$, return for each edge $(n_1,l,n_2) \in E$ its **source** and **target**;*
- *$y_E(G):E \rightarrow L_E$ returns for each edge $(n_1,l,n_2) \in E$ its **label**.*

graph model in Progres

In the following, we are not interested in defining particular instances of migration graphs but we aim on defining a schema for a graph class that contains all valid migration graphs. We call such a schema a *graph model*. We have used the formal specification language *Progres (PROgrammed Graph REplacement Systems)* [Sch91, SWZ95] to define and implement the graph models discussed in this dissertation.

migration graph model

The migration graph model mainly consists of two ASG models, one for the logical data model and the other one for the conceptual model. Both ASG models are connected by an intermediate graph model, the SMG model. Figure 5.2 shows the most important parts of this graph model in a diagrammatic *Progres* notation that is similar to UML [UML97]. To avoid confusion with classes and associations which are modeled within a conceptual DB schema, we keep on using the graph-oriented terms *node type* and *edge type* instead of *class* and *association* like in UML. Note, that cardinalities of edge types are denoted in form of intervals. If no cardinality is specified in the diagram its default value is defined as [1:1]. A formalization of the complete migration graph model in form of a textual *Progres* graph schema has been included in Appendix A.

5.1.1 Graph-based representation of logical and conceptual schema

logical schema

The left-hand side of Figure 5.2 represents the ASG model for the analyzed logical schema which is derived directly from Definition 4.1 on page 58. Names of edge types that begin with *c_* represent syntactical *containment* relationships in the ASG model. A node of type *LSchema* represents the root of the ASG model for a logical schema. This syntactical root contains a set of nodes of type *RS* and *LType*, which represent relation schemas and column types, respectively. Each *RS* node has an attribute *rsname* that stores the name of the represented RS. An RS is composed by a non-empty set of *Variant* nodes, a primary key (*LKey*) that is referenced by an *c_pk* edge, and a set of alternative keys which are referenced by edges of type *c_ak*. Each *Variant* node contains a set of foreign-keys (*ForKey*) and a non-empty set of columns (*Column*). A column has an *lt* edge to point to its type. An IND is represented by one of three node types *I-IND*, *C-IND*, and *R-IND* with respect to its semantic classification (cf. page 58 and [FV95]). These nodes types are derived from an abstract[a] node type *IND* which has two out-going edge types *c_f* and *c_k* that point to a key and a foreign key node.

rational for selecting the conceptual schema

Since the introduction of the *Entity-Relationship* (ER) model in 1976 by Chen [Che76], many variations and extensions of this conceptual data model have been proposed to facilitate the description of data structures. The most common extensions are concepts for abstraction by *aggregation* and *inheritance* [BCN92]. Such extended ER models have had a major influence on the development of the key concepts for modern, object-oriented programming languages. In the context of our application domain (DBRE), we approach the problem of choosing a specific conceptual model from the opposite direction: the expressiveness of our conceptual data model is mainly determined by the distributed programming language *Java* and its object-oriented database binding *ODMG-2.0* [CBB+97] because, currently, *Java*-based technology is the migration platform that provides the greatest potential to leverage existing information systems. The type system of *Java* does not allow for *multiple inheritance* [SCC+93]. Hence, we have chosen a conceptual data model that restricts classes to have at most one

---

a In *Progres* schema diagrams, abstract node types are represented as boxes with sharp corners.

generalization. As a consequence, we do not have to deal with typical inheritance conflicts like repeated inheritance and name collisions. The object-oriented data model proposed by the OMG (*Object Management Group*) defines further concepts for ordered list structures and complex attributes [CBB⁺97]. In our conceptual data model, we have not defined an explicit notion of complex attributes for the sake of simplicity. This is not a severe limitation as complex attributes can always be represented by aggregated objects. Moreover, we decided to consider only set-valued relationships to reduce the complexity of our graph model.

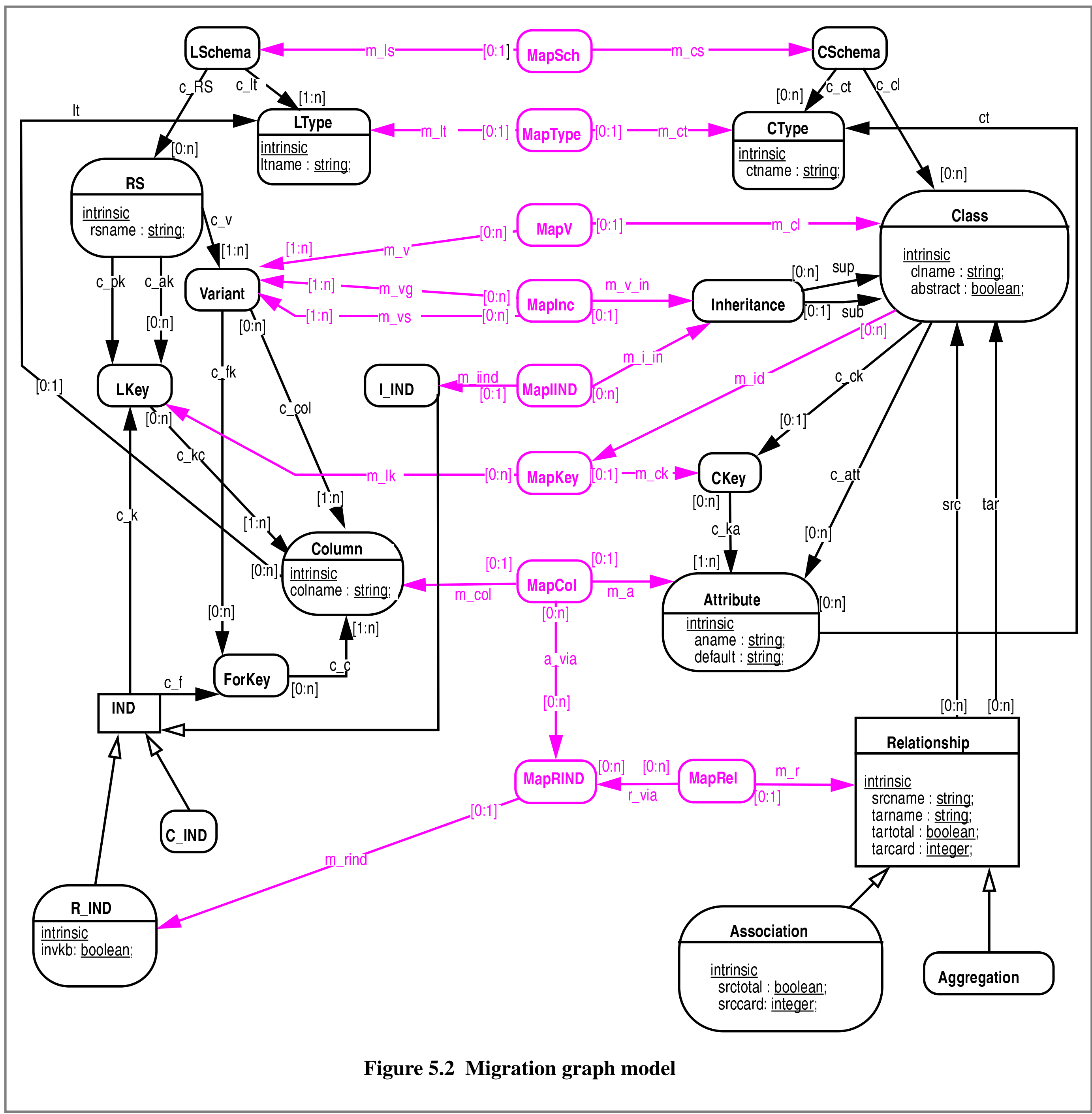


Figure 5.2 Migration graph model

conceptual schema The right-hand side of Figure 5.2 depicts the ASG model that specifies the chosen conceptual model. A node of class *CSchema* is the syntactic root of this ASG. Analogously to the logical schema, this root contains a set of attribute types (*CType*) and a set of classes (*Class*). A boolean attribute (*abstract*) is used to store the information whether a class is abstract or concrete, i.e., whether a class can be instantiated. The name of a class is stored in attribute *clname*. Inheritance relationships are represented by nodes of type *Inheritance* with two edge types *sub* and *sup* which point to the participating subclass and its generalization, respectively. Classes are composed by a set of *Attribute* nodes and an optional key (*CKey*). A *CKey* node itself is composed by a non-empty set of *Attribute* nodes. An *Attribute* node stores its name (*aname*) and a default value (*default*). Associations and aggregations are represented by node types *Association* and *Aggregation* which are generalized to an abstract node type *Relationship*. For each *Relationship* node, attributes *srcname* and *tarname* store the role names of the classes that participate as source and target of the relationship, respectively. Attributes *tarcard* and *tartotal* represent the information about the cardinality of the target class. The value of attribute *tarcard* defines the maximum cardinality for the target of the relationship.[a] If attribute *tartotal* is *true* the relationship is total w.r.t. to its target. The same information is represented for the other side of an association by attributes *srccard* and *srctotal*. Note, that these attributes are not needed for node type *Aggregation* because we restrict the source of an aggregation to represent a total, single instance.

graph constraints The migration graph model in Figure 5.2 contains the specification of a number of simple constraints by means of cardinalities of edge types, e.g., the restriction to single inheritance. Still, these mechanisms are not sufficient to express more complex constraints of correctness that consider a larger graph context and attribute values. Examples for such constraints are scoping rules like "*attribute and reference names have to be unique per class*" and "*class names have to be unique per schema*", etc. In *Progres*, it is possible to denote complex constraints by so-called *graph constraints* which are enforced by the graph repository on the occurrence of predefined events (cf. [Tea99, p. 15]). In the case of constraint violations, automatic *repair* operations re-establish the consistency of the graph. However, this strategy is not suitable for our application. In evolutionary and iterative DBRE processes, the reengineer needs a mechanism that validates correctness constraints on demand but violations should be *indicated* rather than *eliminated* automatically. Hence, in contrast of using *graph constraints*, we employ so-called *graph tests* to check the migration graph for violations of constraints (cf. [Tea99, p. 20]). Graph tests allow to specify conditions for constraint violations on a high level of abstraction. They can be performed in predefined situations to report about the correctness of the conceptual schema. This provides the reengineer with the necessary flexibility to react on indicated constraint violations.

graph tests

Figure 5.3 shows an example for a graph test that checks for duplicate class names in the conceptual schema. When a graph test is applied to a given migration graph it searches for a subgraph that is an isomorphic match for the graph specified in the graphical body of the test.[b] The test evaluates to true if and only if such a subgraph can be found in the migration graph. In addition, this match has to fulfill the attribute conditions specified below the graphical body.

a A zero value means infinity.

b Even though the general problem of finding such a match is NP-complete [Chr75], Zündorf provided the *Progres* compiler with an efficient algorithm that solves this problem for most practical applications [Zün95]. The central idea of this algorithm is to employ typing and cardinality information provided by the graph model.

Unique node numbers are used to refer to particular nodes in the condition part of the test. The graph test in Figure 5.3 searches for two *Class* nodes that belong to the same conceptual schema and have the same value in attribute *clname*. Likewise, the *Progres* specification of the migration graph model in Appendix A includes the usual scoping and correctness constraints for relational and object-oriented schemas as further (negative) graph tests.

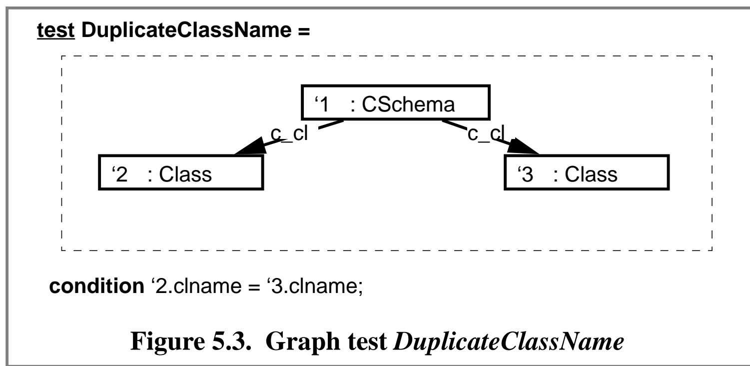


Figure 5.3. Graph test *DuplicateClassName*

5.1.2 The schema mapping graph model

The schema mapping graph (SMG) connects the ASGs of the logical and the conceptual schema and represents their interdependencies. The graph elements of the SMG model are displayed in grey color in Figure 5.2.[a] The information maintained in the SMG serves two separate purposes: (1) it is the basis for the initial schema translation and (2) it enables the generation of schema mapping descriptions for middleware components that facilitate data integration. The SMG model is rather complex because it has to provide suitable flexibility to allow for alternative schema mappings. In the following, we will give a brief overview on the graph elements involved. Their purpose is motivated and described in more detail in the following sections.

mapping types and classes

A node of type *MapSch* is used to connect the syntactic roots of both ASGs. *MapType* nodes are used to map column types to attribute types. Each variant in the logical schema is represented by a concrete class in the conceptual schema. However, if an RS has more than one variant, they usually comprise common columns which implies an inheritance hierarchy with *abstract* classes in the conceptual schema. Consequently, an abstract class is mapped to more than one variant, namely all variants which are represented by its *concrete* subclasses. In the SMG, correspondences among classes and variants are represented by nodes of type *MapV* (cf. Figure 5.2).

mapping inheritance relationships

Inheritance relationships in the conceptual model can be mapped in two different ways to constructs in the logical schema. Firstly, they can be mapped to the *inclusion* of more specific variants in less specific variants that belong to the same RS. Consider Figure 2.16 on page 23 as an example for such a situation. In this example, Variant 4 of table *PRODREF* is less specific than Variant 3, i.e., Variant 4 is included in Variant 3. This situation is represented by an inheritance relationship in the conceptual model which is mapped by a node of type *MapInc* to the two variants (cf. Figure 5.4). An edge of type *m_vs* is used to reference the variant which is

a Note, that all names of edge types that belong to the SMG start with *m_*.

more specific, while an edge of type *m_vg* references the variant which is more general. The second possibility is to map inheritance relationships to INDs in the logical schema that have been classified as inheritance relationships (I-INDs) in the analysis process (cf. page 20). In this case, the mapping is represented by a node of type *MapIIND*.

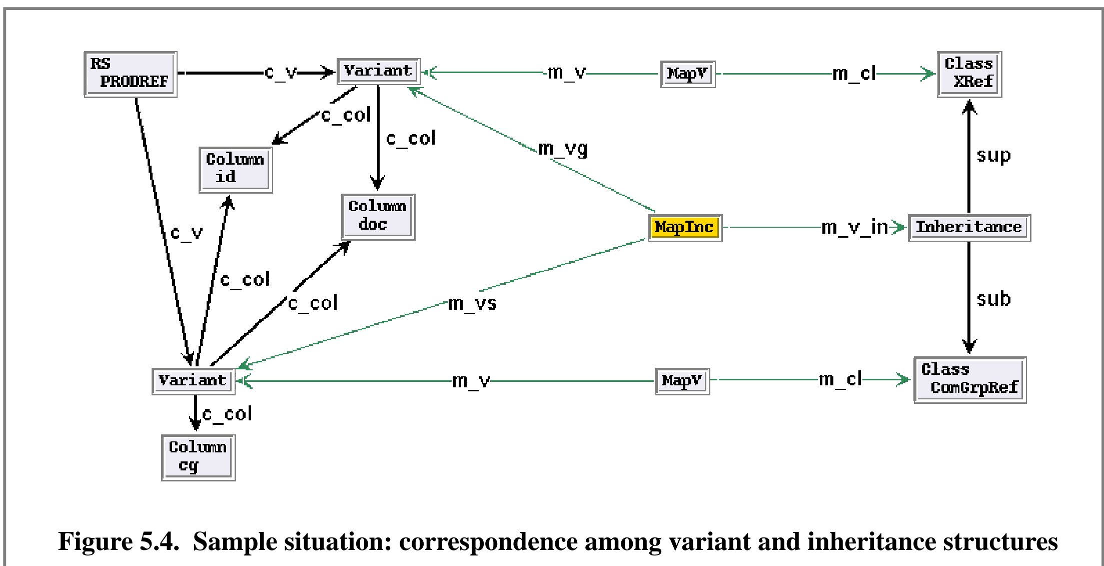


Figure 5.4. Sample situation: correspondence among variant and inheritance structures

mapping keys

Nodes of type *MapKey* are used to map primary keys in the logical schema to keys in the conceptual schema. According to the ODMG data model, our conceptual model includes the notion of unique object identifiers (OIDs) for instances of classes [CBB+97]. Hence, it is not required that every class contains a value-based key. Still, if we aim for object-relational data integration, OIDs have to be resolved to value-based keys in the logical data model. For this purpose, every class has an edge of type *m_id* that references a *MapKey* node in the schema mapping graph.

mapping attributes and relationships

Attributes are mapped to columns by nodes of type *MapCol*. To provide the flexibility to allow for different alternative schema mappings, we admit that attributes of a single class can be mapped to columns in separate RS. For such *remote* columns, the SMG has to maintain the *access path* from the RS that includes the value-based key associated to the class and the RS which includes the remote column. This information is represented by edges of type *a_via*: if a *MapCol* node does not have an *a_via* edge the mapped column belongs to the RS that contains the key referenced by the *m_id* edge of the class that contains the mapped attribute. Otherwise, the mapped column belongs to a different RS and the *a_via* edge of the corresponding *MapCol* node points to a set of *MapRIND* nodes. These nodes represents the *access path* from the RS that contains the key referenced by the *m_id* edge to the RS that contains the mapped column. Each *MapRIND* node is connected to an *R-IND* node which logically represents a foreign key that has to be dereferenced to access the mapped column. Analogously to columns, *MapRel* nodes and *r_via* edges are used to map associations and aggregations to sets of foreign keys (represented by nodes of type *R-IND*).

5.2 A graphical formalism to implement schema translators

Most existing approaches to conceptual schema translation employ *rule-based* transformation systems. Such transformation rules are often specified in a textual pattern language [MCAH95] or in a calculus based on first-order logic and set theory [BGD97, HHHR96]. However, despite their precise semantics such transformation rules are difficult to understand. Therefore, researchers typically employ diagrams to explain the meaning of transformation rules. Furthermore, some formal specifications cannot be executed directly but have to be implemented in a programming language on a lower level of abstraction. In our approach, we employ *graph grammars* to specify schema transformations because they are executable and have the expressiveness of diagrams.

graph grammars

A number of graph grammar formalisms have been proposed based on different theories with their specific advantages and drawbacks. A comprehensive overview on these approaches is given in [Roz97]. The approach used in this chapter has been known as the *logic-based* approach [Sch95]. It is the basis for the specification language *Progres* which has been introduced and formally defined by Schürr [Sch91]. In the following, we will give an example-driven, semi-formal introduction to the essential concepts of this graph grammar formalism which are necessary to understand our application. Analogously to classical (textual) grammars, a graph grammar consist of a *start graph* and a set of *(graph) productions*.

graph production

In general, a graph production can be defined as a pair of graphs, a set of *application conditions*, and a set of attribute *transfer clauses* (cf. Definition 5.2). The two graphs are called the *left-hand side* and the *right-hand side* of the production, respectively. The application of a production to a given graph is described in the following Definition 5.3. Note, that *Progres* productions allow for extended concepts like optional nodes, node sets, path expressions, etc. [Sch91]. However, the semantics of these extended concepts can be defined based on the primitive concepts described below [Zün99].

Definition 5.2 Graph production

A ***graph production*** *is a tuple r:(P, Q, C, T), where*

- *P(r)=P and Q(r)=Q are two graphs over the same sets of node and edge type labels;*
  P(r) is called the ***left-hand side*** *and Q(r) is called the* ***right-hand side*** *of r.*
- *C is a set of* ***application conditions****.*
- *T is a set of attribute* ***transfer clauses****.*

Definition 5.3 Application of a production

A production r:(P,Q,C,T) is ***applied*** *to graph G in the following five steps:*

- ***CHOOSE*** *an* ***occurrence*** *of the left-hand side P in G. P has an occurrence in graph G if there is a morphism m:P→G which preserves source and target and labelling mappings. Furthermore, the occurrence has to fulfill the so-called identification condition which prescribes that elements on the left-hand side which do not occur on the right-hand side can uniquely be identified in G, i.e.,* $\forall x \in P \backslash Q,\ x' \in P: m(x)=m(x') \Rightarrow x = x'$.
- ***CHECK*** *the application conditions according to C. If they are fulfilled the occurrence of P in G is called a* ***match*** *for P.*

- ***REMOVE*** *all elements in G which have been matched to elements in P that do not occur in Q, i.e., remove m(P\Q) from G. If the removal of nodes causes dangling edges in G these dangling edges are removed as well.*
- ***ADD*** *all elements to G which are new in Q, i.e., which do not occur in P. These new elements are glued to G in the preserved graph elements identified by m(P∩Q). We denote the morphism* $\overline{m}$*:Q→G that identifies the (newly created) occurrence of Q in G as* **comatch**.
- ***TRANSFER*** *attribute values to nodes in G that match nodes in Q according to the attribute transfer clauses specified in T.*

In the following, we denote $G^{\downarrow(r,m)}$ *for the graph that is produced by the application of a production r to another graph G (in a match m).*

Figure 5.5 shows a simple *Progres* production *AddRSToLSchema* which specifies the extension of a logical schema by a new RS.[a] The left-hand side of production *AddRSToLSchema* contains only a single node of type *LSchema*. If the production is applied this node is preserved because it occurs on the right-hand side with an identical node number. Furthermore, *G* is extended by three new nodes and three new edges which represent a new RS with one variant and a primary key.

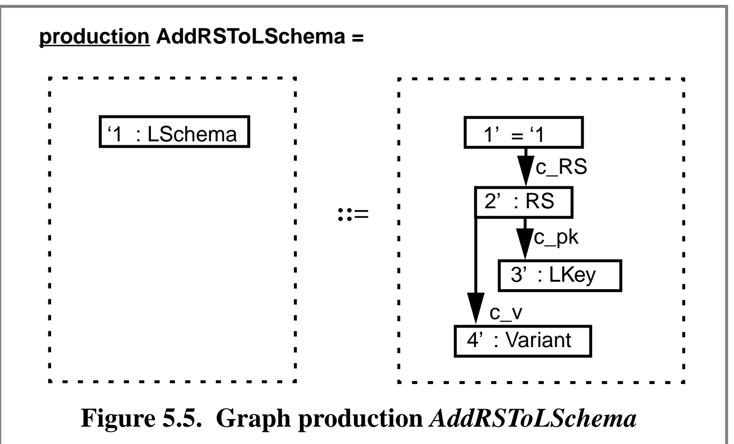


Figure 5.5. Graph production *AddRSToLSchema*

5.2.1 Triple graph grammars

Usually, a graph grammar is used to define a *single* graph model in terms of all possible graphs that can be derived by applying the productions to a given start graph. They are less suitable to specify the mapping between two different ASG models as needed in our application. In [LS96, Lef95], Lefering and Schürr propose an extended formalism called *triple graph grammars* that is dedicated to this problem. A triple graph grammar consists of a set of *mapping rules*. Basically, each mapping rule consists of a production triple, i.e., it contains three productions. Two of these productions specify equivalent extensions of the first and the second ASG, while another production is used to extend a mapping graph that represents the correspondences between both ASGs.

a For layout reasons, the right-hand side of a production might also be *below* its left-hand side.

Figure 5.6 shows an example for such a mapping rule. In this notation which has been proposed in [JSZ96] the three productions are separated by vertical, grey bars. Triple graph grammars deal with *extending* productions only, i.e., no graph elements are removed. Hence, a single graphical diagram can be used to represent both sides of an extending production in a condensed way. For example the left production of the mapping rule in Figure 5.6 is a condensed notation for production *AddRSToLSchema* in Figure 5.5. The entire mapping rule *MapRSToClass* in Figure 5.6 specifies that an extension of a logical schema by a new RS corresponds to the extension of the conceptual schema by a new class. The production in the middle part of the mapping rule is used to update the SMG that represents the correspondence between both ASGs.

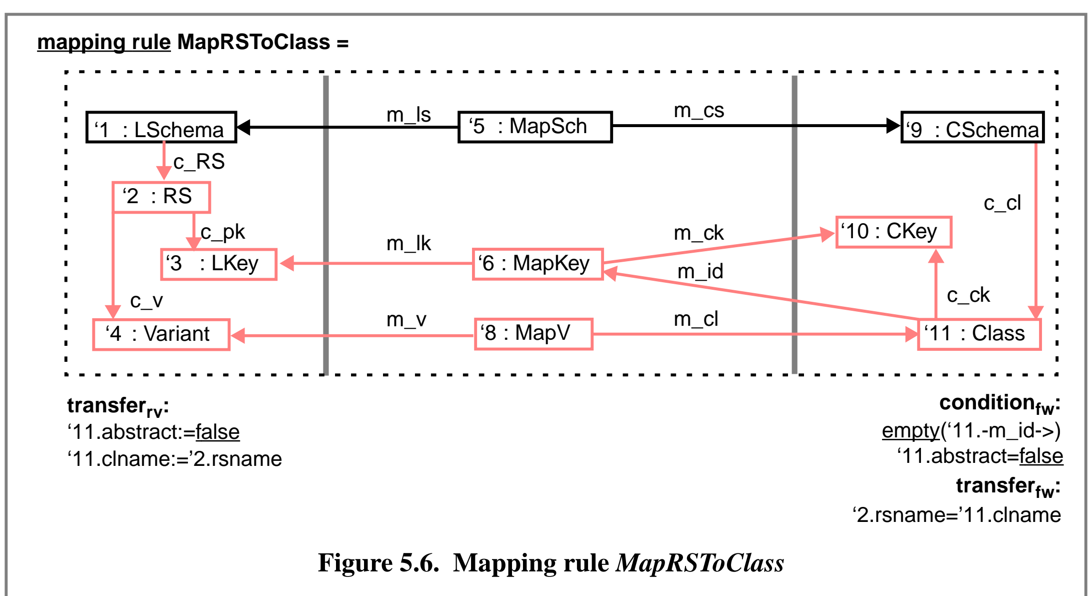


Figure 5.6. Mapping rule *MapRSToClass*

generation of reverse and forward translators

A triple graph grammar allows to generate automatic translators that create conceptual schemas from logical schemas (*reverse mapping*) and vice-versa (*forward mapping*). Such an automatic translator consists of a set of conventional graph grammar productions. Each such production is derived from one mapping rule specified in the triple graph grammar. A *reverse production* p_{rv} is derived from a mapping rule by choosing its black parts and its left side as the left-hand side of p_{rv} and the elements in the entire mapping rule as the right-hand side of p_{rv} (cf. Figure 5.7). Analogously, the *forward production* p_{fw} is derived by choosing the black parts and the right-hand side of the mapping rule as the left-hand side of p_{fw} and the elements in the entire mapping rule as the right-hand side of p_{fw} (cf. Figure 5.8).

attribute transfer clauses

As defined in Definition 5.2, *Progres* productions might include *attribute transfer clauses*. They are added in textual form under the graphical part of the production. The first attribute transfer clause in Figure 5.7 assigns the boolean value *false* to attribute *abstract* of the new *Class* node '11. The second transfer clause transfers the name of the mapped RS to this new node. In a triple graph grammar, we add transfer clauses (and application conditions) for both derivable productions to each mapping rule. This is exemplified in Figure 5.6 where we use the suffixes *rv* and *fw* to denote whether the clauses belong to the reverse or the forward production.

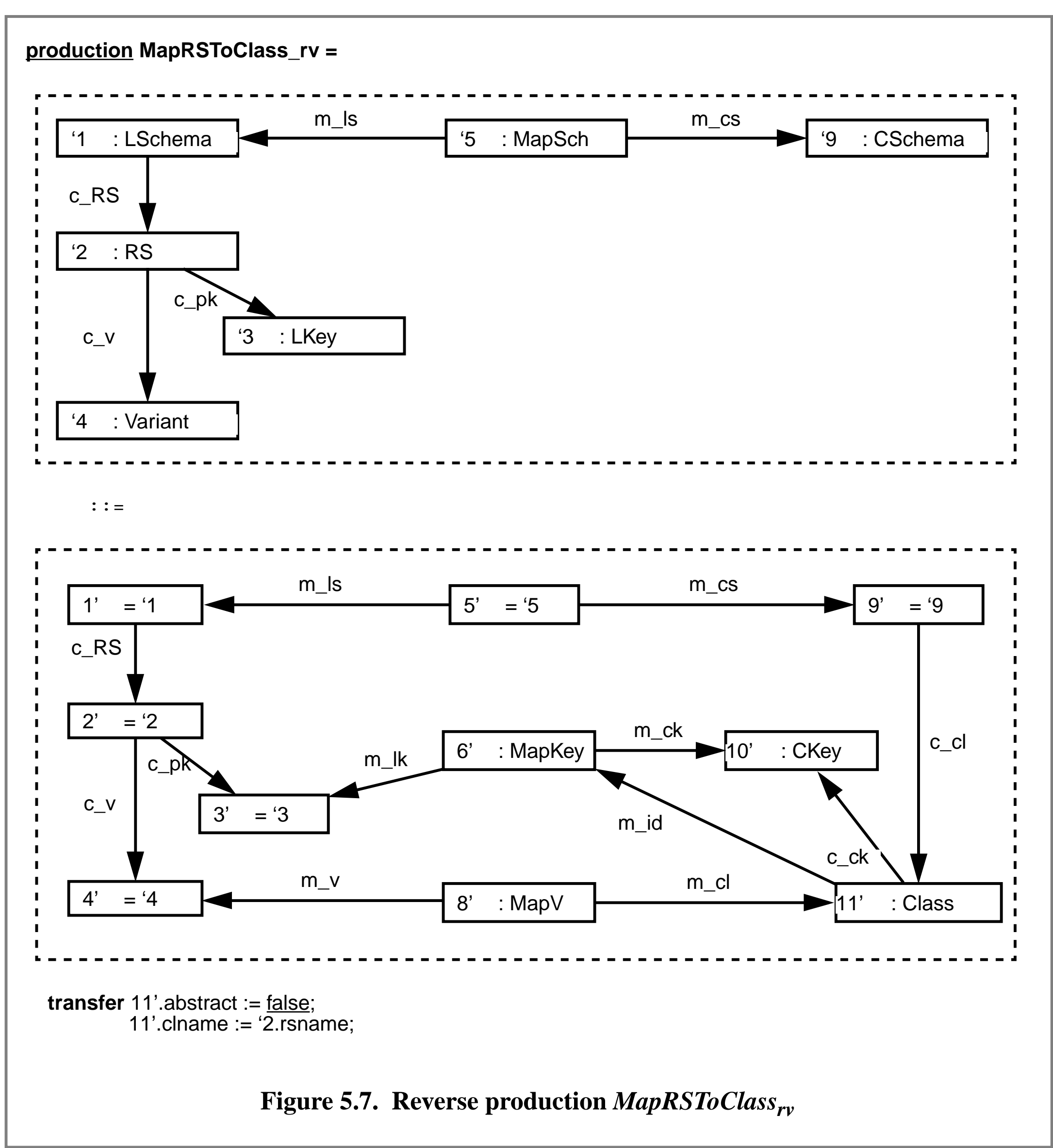


Figure 5.7. Reverse production *MapRSToClass$_{rv}$*

application conditions

In *Progres*, *application conditions* often contain so-called *path expressions* [Tea99, p. 33]. Path expressions consist of a sequence of edge traversals separated by dots or the ampersand symbol, e.g., *-e1-> & <-e2-* defines a path over an outgoing edge of type *e1* and an ingoing *e2* edge. When a path expression is applied to a node *n* (or a set of nodes) its application returns all nodes that can be reached from *n* by traversing the specified path. For example, the expression *'11.-m_id->* in the condition part of *MapRSToClass$_{fw}$* (Figure 5.8) returns all variant nodes that can be reached from node '11 over an outgoing edge of type *m_id*. The boolean predicate *empty* returns *true* if and only if its argument is an empty set. This condition is necessary to enable that *several* classes in an inheritance hierarchy can be mapped to variants of the *same* RS: new RS nodes are created for classes only if they do not have the same value-based key (referenced by edge *m_id*) as another class which has been mapped before. Moreover, the attribute condition "*'11.abstract=false*" ensures that only concrete classes are mapped to RS' in the logical schema.

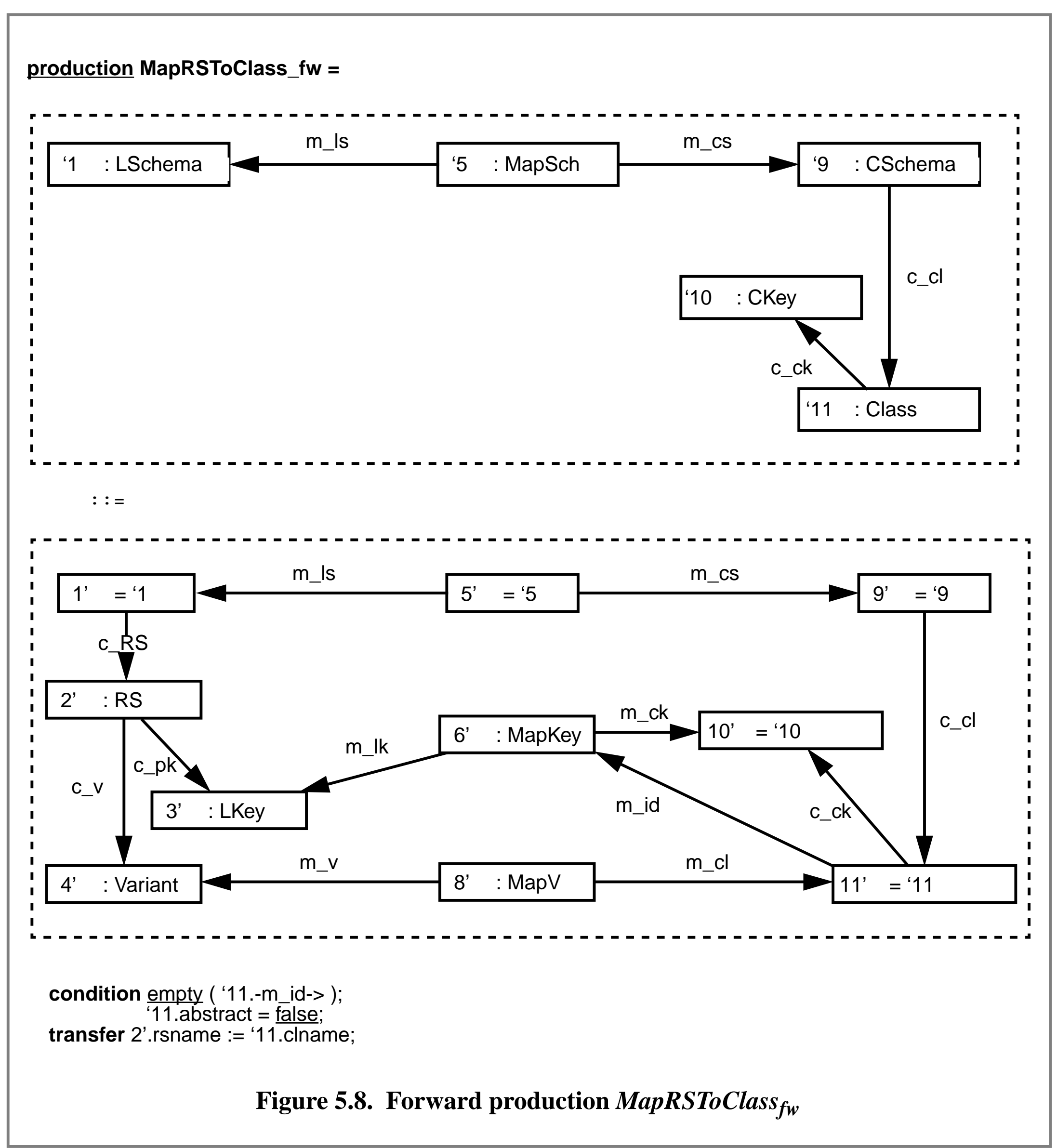


Figure 5.8. Forward production ***$MapRSToClass_{fw}$***

start graph

Similar to start symbols of conventional textual grammars, graph grammars are applied to an initial graph that is called *start graph*. In our application, the minimal start graph consists of the syntactic root nodes for the ASGs of both schemas and graph elements that represent all attribute and column types (cf. Figure 5.9). Pairs of equivalent atomic data types are mapped by nodes of type *MapType*. The correspondences among atomic types in the logical and the conceptual schema, respectively, depends on the concrete application context of the DBRE tool. Different DBMS provide different data types. Hence, in our approach, the reengineer has to enter atomic type correspondences in an initial customization dialog of our DBRE tool.[a]

a In some cases, it might also be necessary to implement type conversion functions. In principle, such functions can be stored in further attribute of *MapType* nodes. However, we abstract from this detail in the following discussion.

In typical DBRE scenarios, the start graph contains further parts of an analyzed logical schema ASG which are going to be translated to conceptual schema constructs. Moreover, during the migration process it often occurs that modifications in conceptual schemas have to be remapped to the original logical schema. In this case, the mapping algorithm is applied to a start graph that contains ASG elements from the logical schema as well as from the conceptual schema (illustrated by the grey subgraph in Figure 5.9).

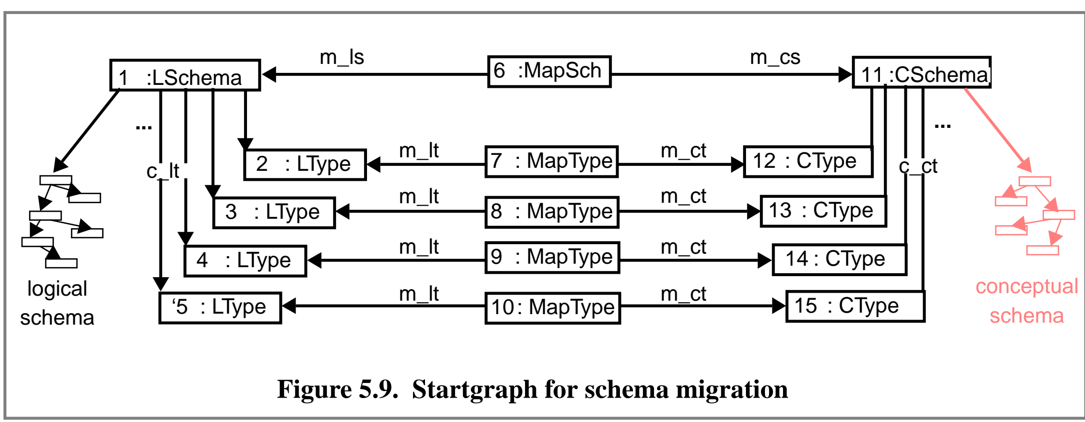


Figure 5.9. Startgraph for schema migration

translation algorithm

In sections 5.2.2-5.2.4, we complement the triple graph grammar specification that defines a mapping among logical and conceptual DB schemas. The translation process is based on the execution of forward and reverse productions that are derived from each mapping rule. The corresponding translation algorithm is described in Figure 5.10. It iteratively chooses a production r from the set of all derived productions R that has a match in the current migration graph G. Furthermore, it is validated that this match cannot be extended to a match that includes all SMG elements on the right-hand side of r. This is to avoid multiple applications of the same production in the same match. If such a match can be found, the corresponding production is applied to the host graph. These steps are iteratively performed until no production in R fulfills the condition in lines 8 and 9.

algorithm *MapSchema*(R, S)
1) **input** R, a set of forward and reverse productions derived from a triple graph grammar
2) **input** S, a start graph (according to Figure 5.9)
3) **output** G, a migration graph (according to Figure 5.2)
4) **begin**
5) **let** $G=S$
6) **repeat**
7) **let** $r{:}(P,Q,C,T) \in R$ be a production that fulfills the following conditions
8) - P has a match in G represented by a morphism $m{:}P \rightarrow G$
9) - this match cannot be extended in G by a match for the SMG elements in Q
10) **let** $G = G^{\downarrow(r,m)}$
11) **until** no production $p \in P$ fulfills the conditions in lines 8 and 9
12) **return** G
13) **end**

Figure 5.10. Algorithm MapSchema

The described algorithm defines how triple graph grammars can be employed for bi-directional schema translation. Note, that the productions are not tested and applied in a predefined order. (The specification of the schema mapping rules ensures *confluence* [Roz97, p. 105] for all production applications.) For larger schemas this simple algorithm lacks efficiency. This problem can be solved by implementing a procedural framework that defines an order for the application of the derived graph productions. The procedural framework that has been implemented in our DBRE environment is described by Holle [Hol97].

5.2.2 Mapping variants to class hierarchies

In our approach, RS in the logical schema are initially mapped to classes in the conceptual schema. However, in contrast to other tool-based approaches to schema translation, we consider the fact that relational DBs often comprise hidden inheritance structures in form of different variants of tuples in RS (cf. page 20). Consequently, RS with more than one variant are mapped to several classes which participate in an inheritance hierarchy.[a] In Figure 5.6, we presented a mapping rule which maps an RS to a class. This rule is sufficient for the standard case where an RS has only one variant of tuples.

MapVariantTo-ConcreteClass

If an RS has more than one variant each additional variant has to be mapped to a concrete class which participates in the same inheritance hierarchy like the class mapped in rule *MapRSToClass*. Since the relational data model has no explicit concept for the corresponding inheritance relationship, it is not considered in the (bi-directional) mapping rule *MapVariantToConcreteClass* in Figure 5.11. (Note, that class node ’9 has been mapped by rule *MapRSToClass* in Figure 5.6.)

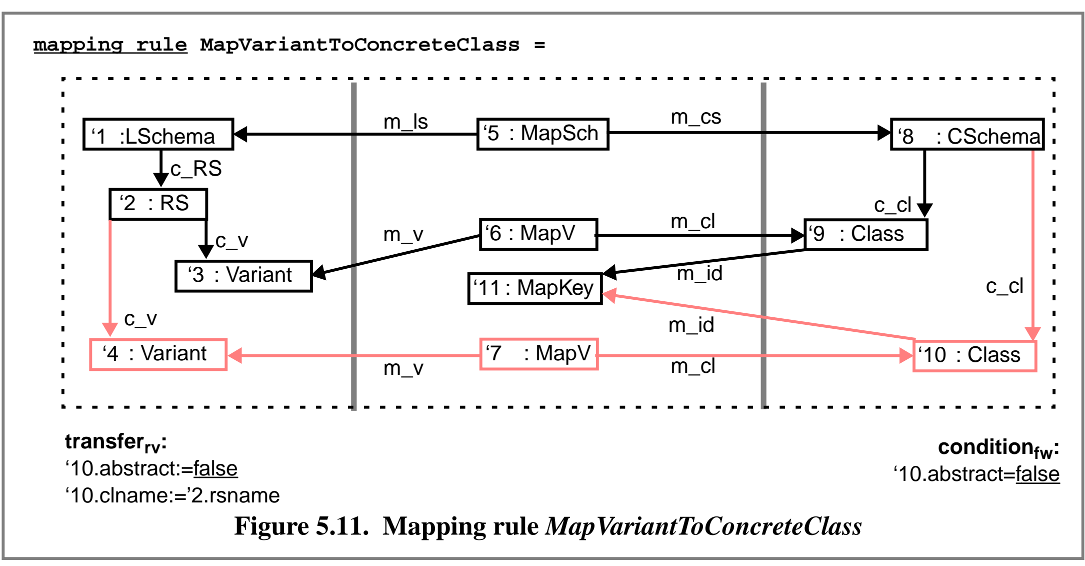


Figure 5.11. Mapping rule *MapVariantToConcreteClass*

a Due to the restriction to single inheritance, there might be variant structures that cannot be mapped to inheritance hierarchies in our conceptual model. The reengineer has to resolve such conflicts by adding or removing variants.

Example 5.1 Application of rules *MapRSToClass* and *MapVariantToConcreteClass*

Let us illustrate the correspondences among logical variants and concrete classes with a sample RS (*Tenant*) that has two variants (cf. Figure 5.12). Note, that we use an example different from our case study (Figure 2.13) to improve the readability of the graph representation and include an abstract class in our consideration. Tuples belonging to the first variant of RS *Tenant* have *null* values in column *mtenant*, while all remaining tuples have *null* values in column *rent*. Conceptually, the first variant stores main tenants while the second variant represents sub tenants. Both concrete variants share a common attribute *name* which gives rise to an abstract generalization in the conceptual schema.

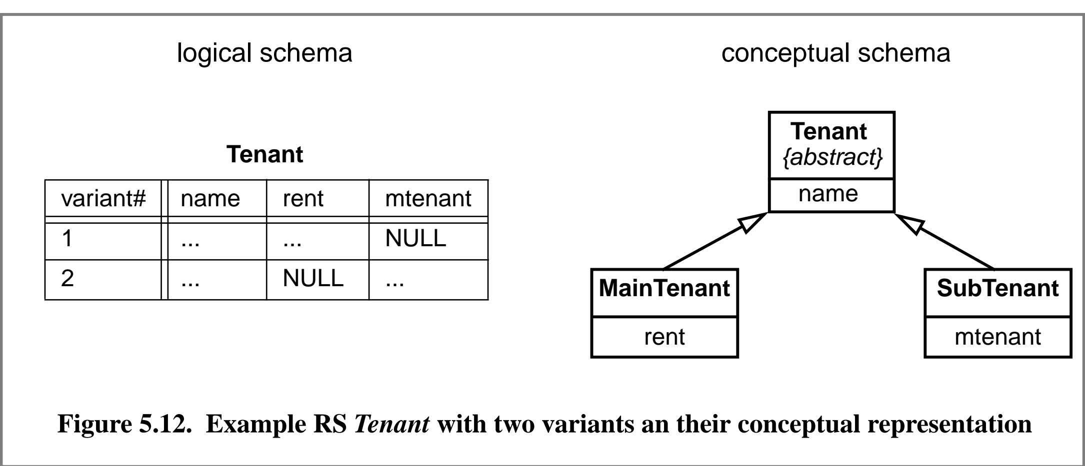


Figure 5.12. Example RS *Tenant* with two variants an their conceptual representation

Figure 5.13 shows the graph representation for our example after applying rule *MapRSToClass* followed by an application of rule *MapVariantToConcreteClass*. We skipped all nodes representing schema, key, and type mappings in order to simplify the graph layout. Note, that class nodes with a label *"false"* indicate concrete classes because this label indicates the current value of the boolean attribute *abstract*.

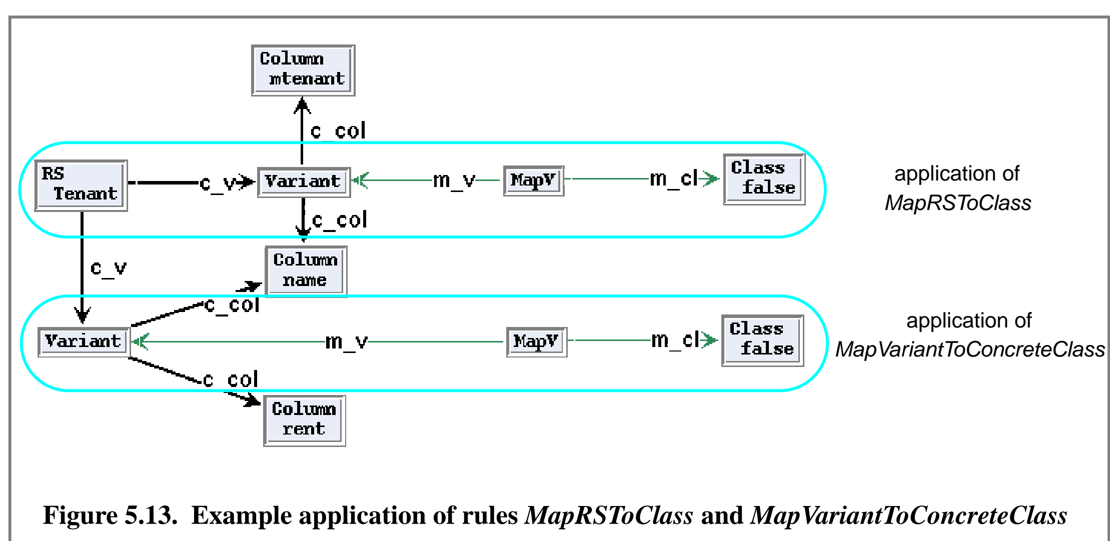


Figure 5.13. Example application of rules *MapRSToClass* and *MapVariantToConcreteClass*

MapVariantsTo-AbstractClass$_{rv}$

Recovering inheritance hierarchies from variant structures might require the creation of *abstract* classes. Abstract classes do not have corresponding constructs in the logical schema. Consequently, we employ a unidirectional (reverse) production (*MapVariantsToAbstract-Class$_{rv}$*) to recover abstract classes from variant structures (cf. Figure 5.14).

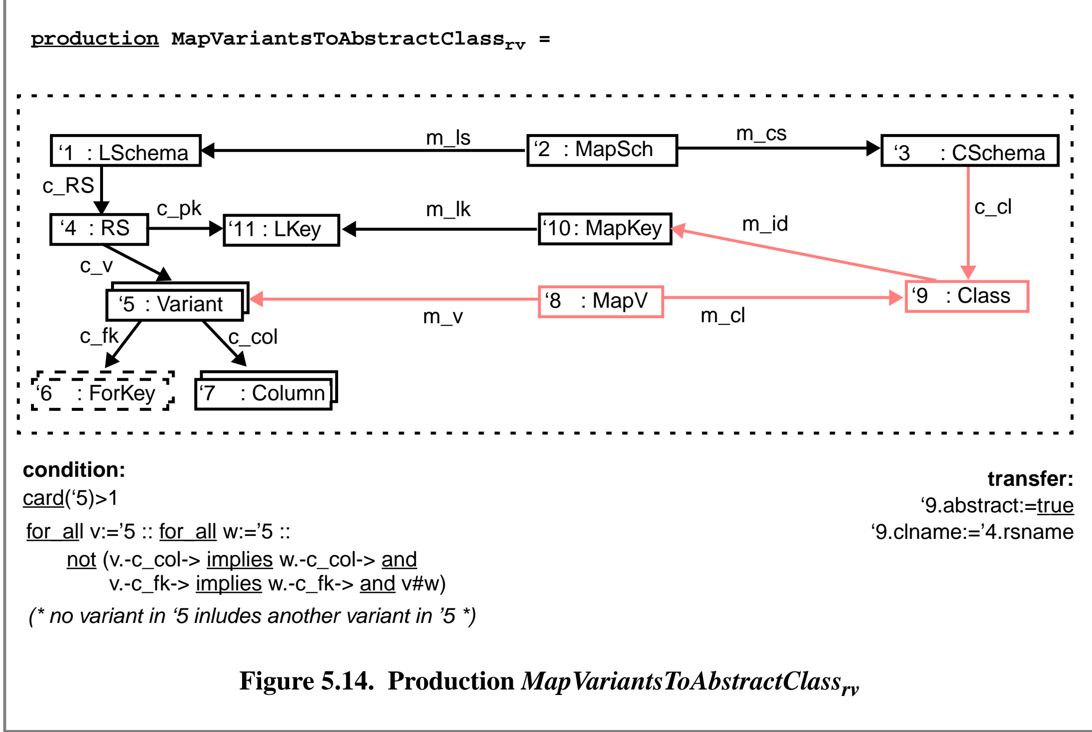


Figure 5.14. Production *MapVariantsToAbstractClass$_{rv}$*

Production *MapVariantsToAbstractClass$_{rv}$* in Figure 5.14 uses a node of type *MapV* to map a set of two or more variants to an abstract class if all variants in this set (5') comprise a common sets of columns ('7) and foreign keys ('6), respectively. In *Progres*, boxes with shadows ('5, '6, and '7) represent node *sets* while a dashed shape is used to mark *optional* graph elements, i.e., the set of foreign keys ('6) is allowed to be empty.[a] The first application condition *"card('5>1)"* of *MapVariantsToAbstractClass$_{rv}$* specifies that a set of more than one variant is needed to be mapped to an abstract class. The second condition specifies that '5 may not contain two distinct variants *v, w* where *w* includes *v*, i.e., the set of variants in '5 has to be minimal. Note, that the *Progres* set operator *implies* returns *true* if and only if its first argument is a subset of its second argument. Furthermore, the sign # represents the inequality operator.

a For computational difficulties, the current *Progres* compiler (Version 9.2) does not allow the user to specify edges among node sets. In this dissertation, we use this notation because it is easier to understand than equivalent textual circumscriptions: whenever, we use an edge between two node sets we require the existence of an edge of this type between each node in the first set and each node in the second set. (An implementation of the above rules which is compliant with the current *Progres* compiler is described by Wadsack [Wad98]).

Example 5.2 Application of production *MapVariantsToAbstractClass*$_{rv}$

Figure 5.15 illustrates the application of production *MapVariantsToAbstractClass*$_{rv}$ to the example graph in Figure 5.13. Node set ’5 has been matched to both variant nodes because they share a common column (*name*) and do not include each other.

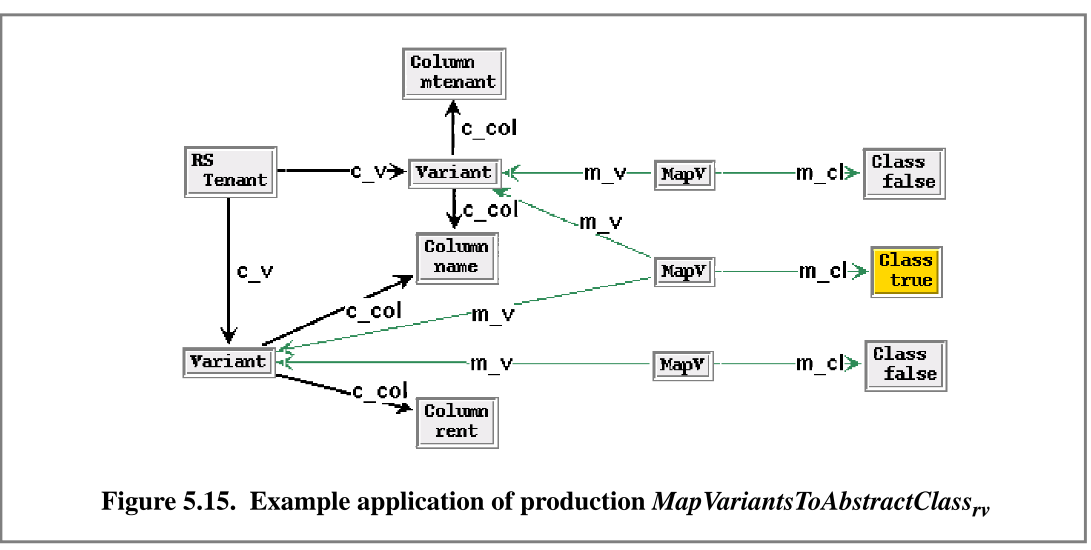


Figure 5.15. Example application of production *MapVariantsToAbstractClass*$_{rv}$

***MapVariantsTo-Inheritance*$_{rv}$**

Now, that we have mapped variants to concrete and abstract classes in the conceptual schema, we complement the inheritance hierarchy by adding *Inheritance* nodes to represent generalization relationships. For this purpose, we employ a second reverse production (*MapVariantsToInheritance*$_{rv}$) in Figure 5.16. This production specifies that a class (’10) is a *direct* generalization of class (‘7) if

- (**C 1**) the common properties (attributes and foreign keys) of all variants (‘6) that have been mapped to superclass ‘7 are included in the set of properties common for all variants (‘5) which have been mapped to the new subclass ‘10; and
- (**C 2**) class ‘7 is a *direct* superclass, i.e., there is no other class (‘11) which has been mapped to a set of variants that includes the properties common for all variants in ‘6 but has a subset of those properties common to all variants in ‘5.

Condition **C1** is ensured by the textual application condition on the bottom-left corner of Figure 5.16. Condition **C2** is necessary to avoid the creation of *transitive* inheritance relationships. It is specified in form of a *negative application condition* with an annotated *restriction* (cf. [Tea99, p. 26]). The negative application condition is represented by a cancelled node (‘11) which inhibits the application of the production if a match for this node can be found that complies to the specified restriction. The textual restriction employs the *use* statement to define three local variables, namely

- *vars*, the set of variants mapped to class ‘11,
- *fks*, the set of foreign keys common to all variants in *vars*, and
- *cols*, the set of columns common to all variants in *vars*.

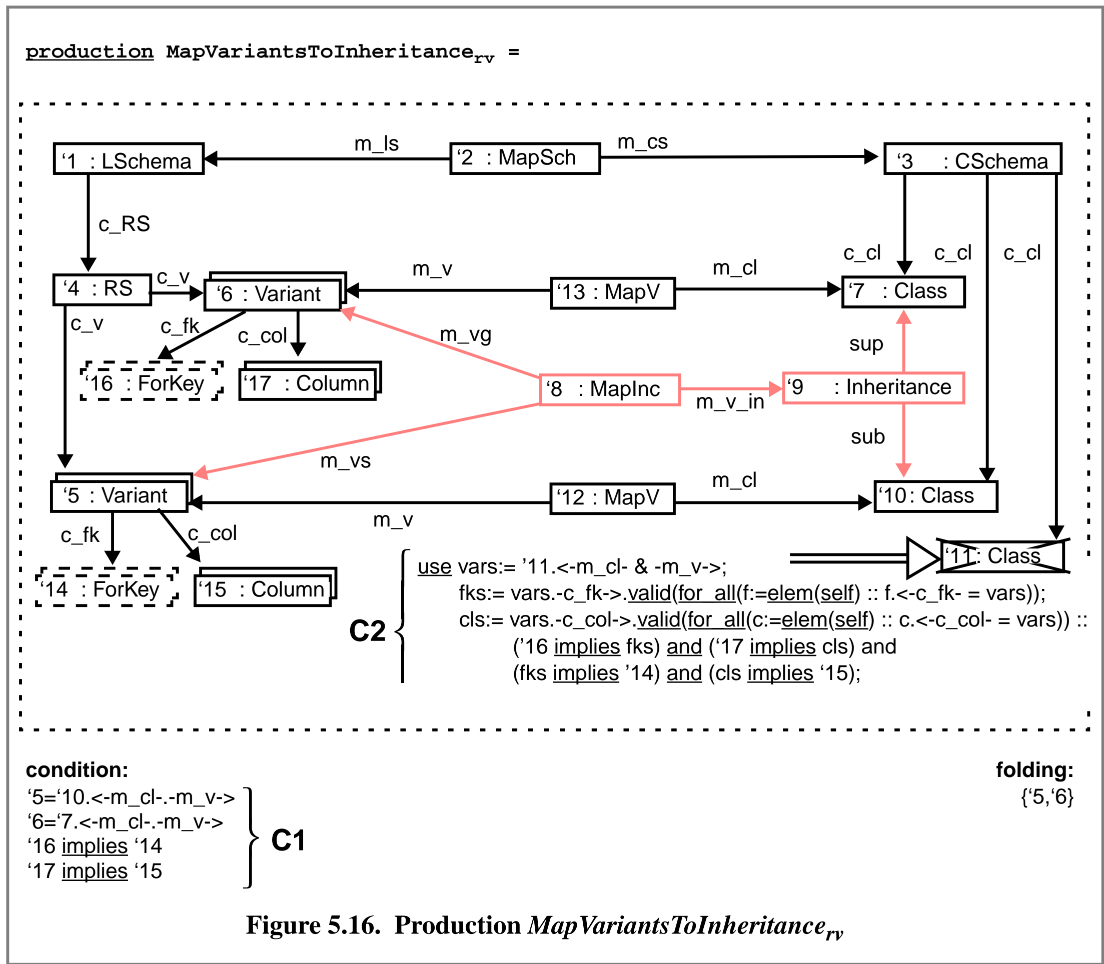


Figure 5.16. Production *MapVariantsToInheritance$_{rv}$*

Note that the assignment "*fks:= vars.-fk->*" delivers the set of all foreign key nodes which belong to *any* variant in *vars*. Hence, we used the *Progres* operator *valid* [Tea99, p. 27] to further restrict this set to those foreign key nodes which belong to *all* variants in *vars*. In addition, we have to admit common elements in node sets '5 and '6. This is specified by a so-called *folding clause* in the bottom-right corner of Figure 5.16. If no folding clause was specified the match would have to be *isomorphic* (cf. [Tea99, p. 25]).

If production *MapVariantsToInheritance$_{rv}$* is applicable it creates a new *Inheritance* node ('9) which is mapped over a *MapInc* node ('8) to node sets '5 and '6. All variants which have been mapped to the subclass of the new inheritance relationship are referenced by edges of type *m_vs*, while all variants that correspond to the generalization are referenced by *m_vg* edges.

Example 5.3 Application of production *MapVariantsToInheritance$_{rv}$*

Production *MapVariantsToInheritance$_{rv}$* can be applied twice to the example graph in Figure 5.15 The resulting graph, which completes the reverse mapping of RS *Tenant* to the corresponding class hierarchy in the conceptual model is displayed in Figure 5.17. In this representation, bold lines have been used to mark all additional edges. Note, that bold lines with two labels (*m_vg / m_vs*) represent the existence of two separate edges with these labels between the corresponding source and target nodes.

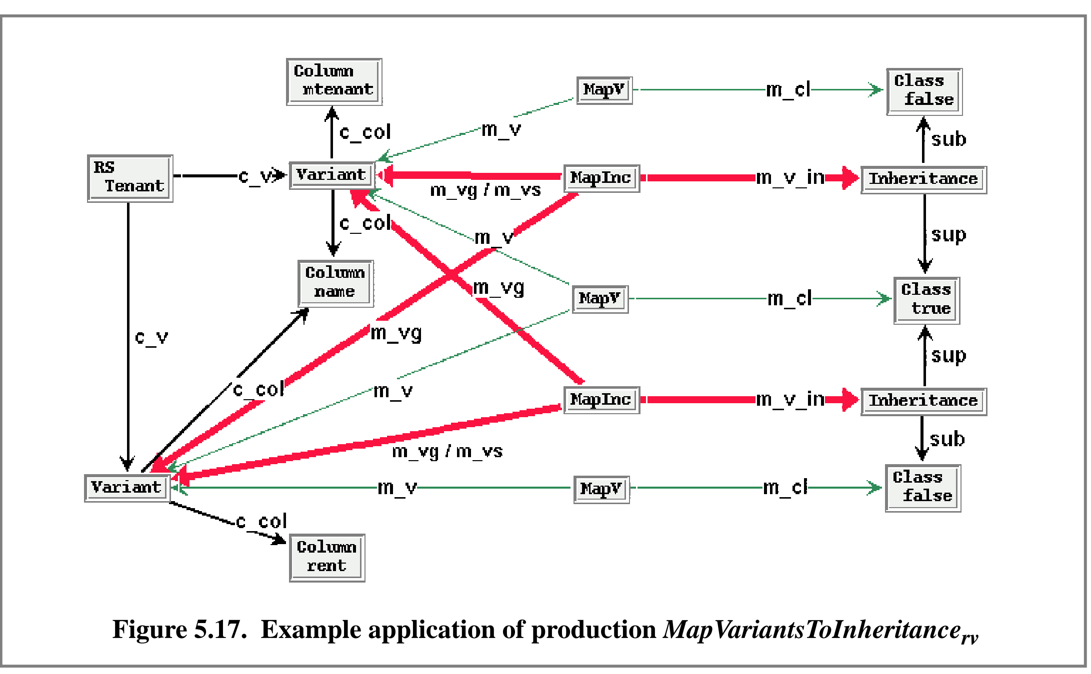


Figure 5.17. Example application of production *MapVariantsToInheritance$_{rv}$*

5.2.3 Mapping columns to class attributes

In the relational data model, the representation of logical entities and their relationships is based on the simple mathematical concept of relations. Hence, columns are basically used for two purposes: they might represent actual data values of entities or they might represent references implemented as redundant copies of such data values in other relations (foreign keys). Only columns that do not represent foreign keys should be mapped to attributes in the conceptual model because it includes explicit concepts for relationships (associations and aggregations). If we admit the existence of different variants of tuples in an RS, we have to generalize this restriction such that only those columns are mapped to attributes which do not belong to foreign keys in *all* of these variants. This restriction is considered within the first part of the reverse application condition of mapping rule *MapColToAttr* (cf. the comment in Figure 5.18).

Even though an RS with multiple variants is mapped to an inheritance hierarchy of classes, each of its columns is mapped to only one class attribute in this hierarchy. This attribute is then inherited by all subclasses in the hierarchy. The second part of the reverse application condition ensures that the column is mapped to the most general class (‘8) in the inheritance hierarchy. This requirement is represented by a *conditional boolean expression* [Tea99, p. 44] which returns *true* if there exists no such generalization. Otherwise, it ensures that at least one variant that has been mapped to the generalization of class ‘8 does not include column ‘2. Note, that the operator *in* tests the membership of its first argument in the set represented by its second argument.

key columns

Nodes ‘4, ‘5, and ‘9 have been declared as *optional* graph elements (cf. page 129) to consider the two possible cases of mapping key columns or non-key columns. If the column (respectively the attribute) belongs to a key this information is reflected by adding the corresponding syntactical edges in both ASGs. The outlined arrow between nodes ‘1 and ‘4 marks a *graphical path expression*.

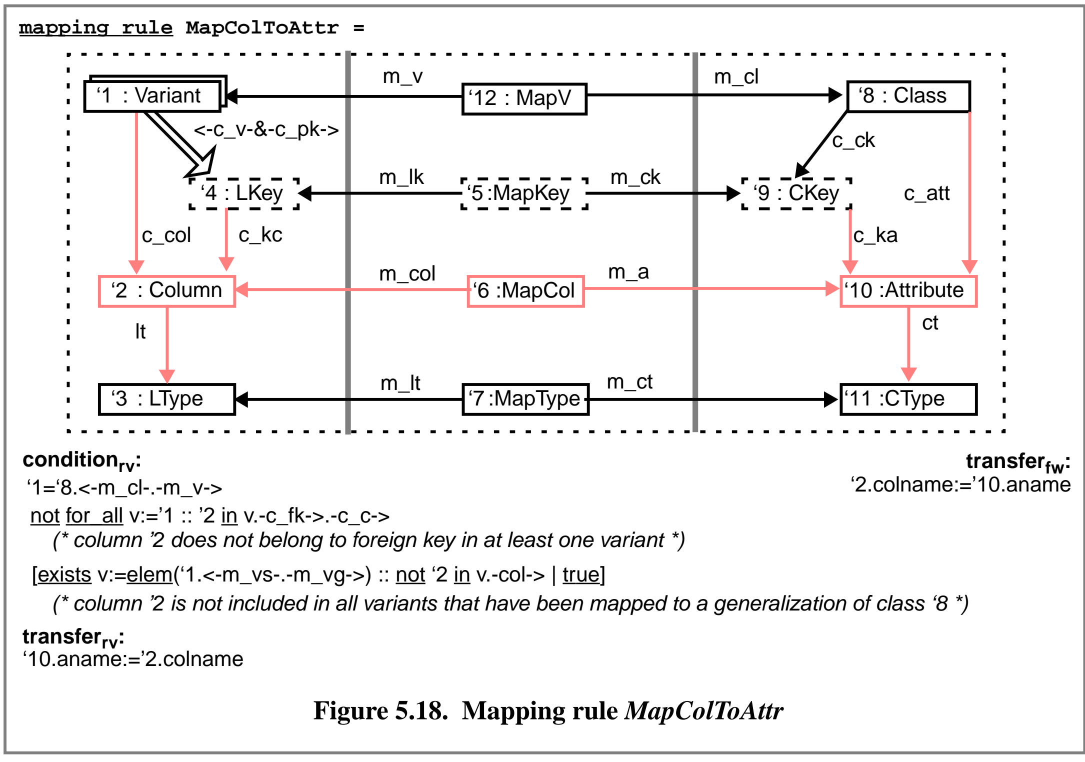


Figure 5.18. Mapping rule *MapColToAttr*

5.2.4 Mapping inclusion dependencies to relationships

In contrast to the variety of concepts for relationships in the conceptual model (inheritance, association, and aggregation with different cardinalities), INDs are the only means to implement references among different RS in the relational model. The schema analysis activities described in Chapter 4 aim to narrow this semantical gap by classifying INDs either as normal references (R-IND), cardinality constraints (C-IND), or as inheritance relationships (I-IND) (cf. Definition 4.1 on page 58). Based on this classification, we present four mapping rules that translate INDs to relationships in the conceptual model and vice-versa. The first three rules map R-INDs (in combination with C-INDs) to associations with different cardinalities, while the fourth rule maps I-INDs to inheritance relationships.

MapRINDToAssoc [1:1]

Rule *MapRINDToAssoc[1:1]* in Figure 5.19 maps an R-IND which is inversely key-based to a total *one-to-one* association in the conceptual model (cf. Figure 2.15 on page 22). The restriction to inversely key-based INDs is specified by testing attribute *invkb* in the textual condition part of rule *MapRINDToAssoc[1:1]*. Analogously to the previous mapping rules, the rest of this condition block ensures that the new association is created among the most general classes in the corresponding inheritance hierarchy.

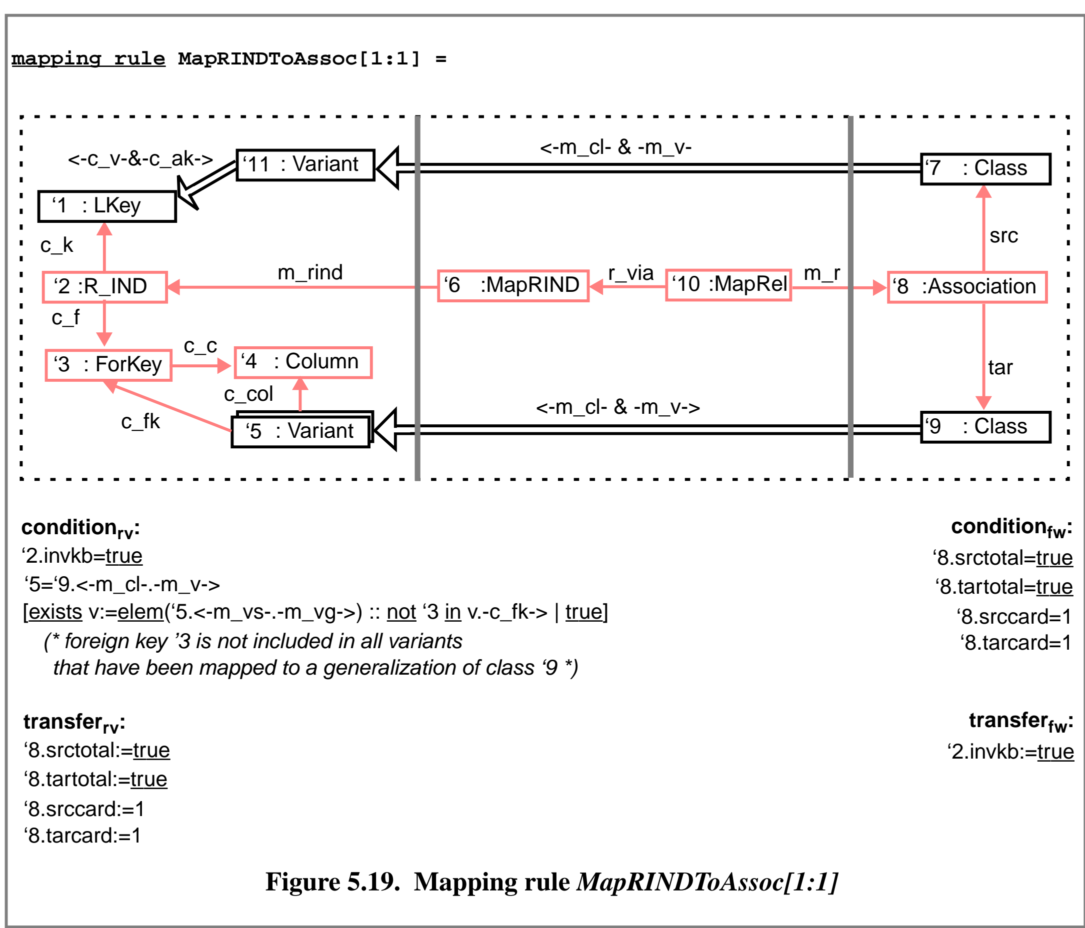


Figure 5.19. Mapping rule *MapRINDToAssoc[1:1]*

MapRINDToAssoc [N:0,1] Similar to *MapRINDToAssoc[1:1]* the next rule (*MapRINDToAssoc[N:0,1]*) in Figure 5.20 maps an R-IND that is not inversely key-based but has an inverse C-IND to a left-total *one-to-many* association. This rule contains a folding clause to enable that nodes '9 and '10 might represent the same class.

MapRINDToAssoc [0,N:0,1] All remaining R-INDs (which are not inversely key-based and do not have inverse C-INDs) are mapped to partial *one-to-many* associations (cf. Figure 5.21). Again, we employ a negative graphical application condition (node '5) to require the absence of the inverse C-IND.

MapIINDTo Inheritance Finally, rule *MapIINDToInheritance* in Figure 5.22 specifies the correspondence of I-INDs with inheritance relationships. The condition specified for the reverse production ensures that each class has only one generalization. Analogously to the reverse translation of variants to class hierarchies, it might occur that an I-IND cannot be mapped because this would violate the single inheritance condition. The reengineer has to resolve such a conflict, e.g., by changing the classification of the IND from I-IND to R-IND.

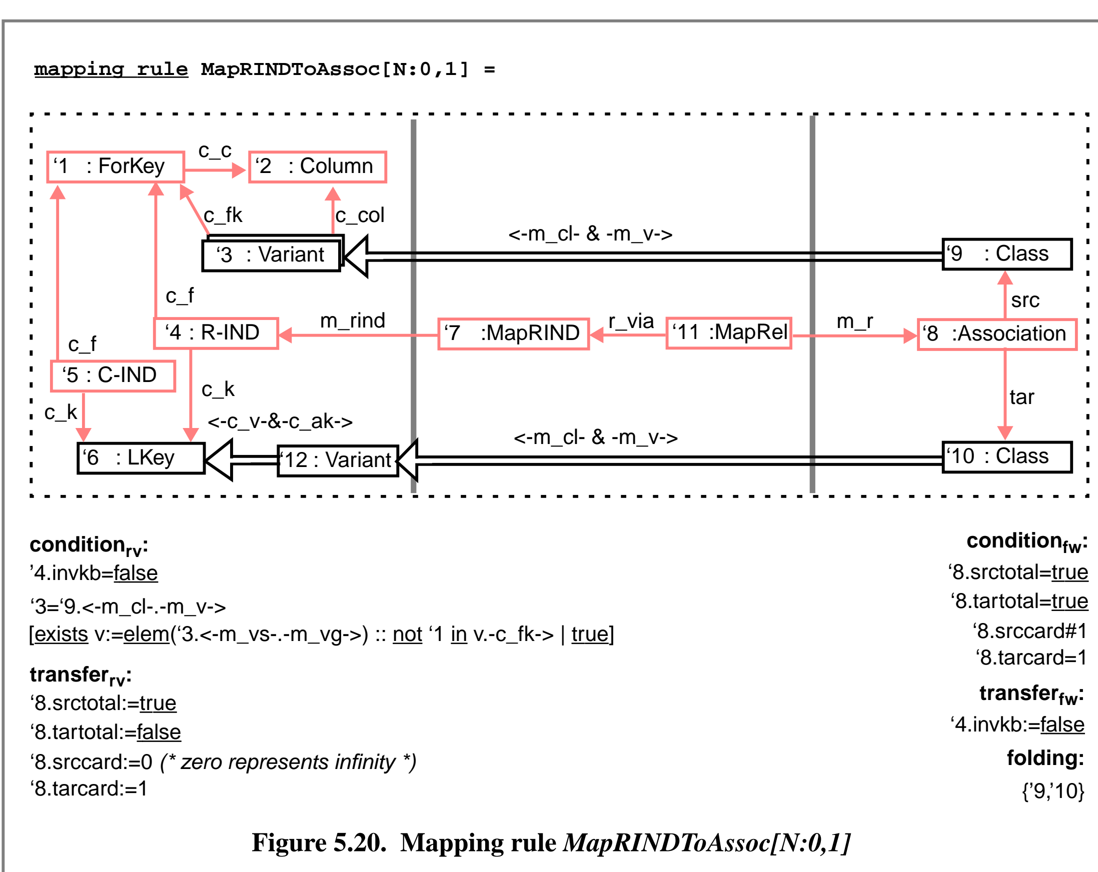


Figure 5.20. Mapping rule *MapRINDToAssoc[N:0,1]*

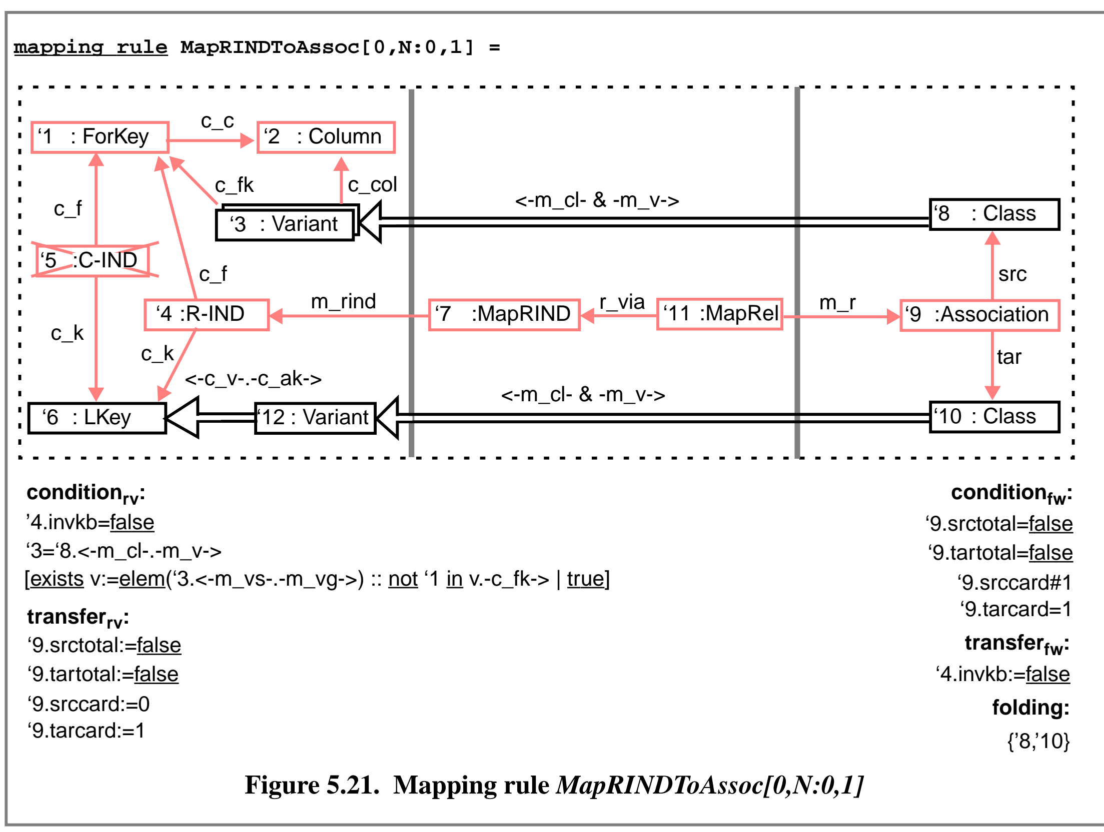


Figure 5.21. Mapping rule *MapRINDToAssoc[0,N:0,1]*

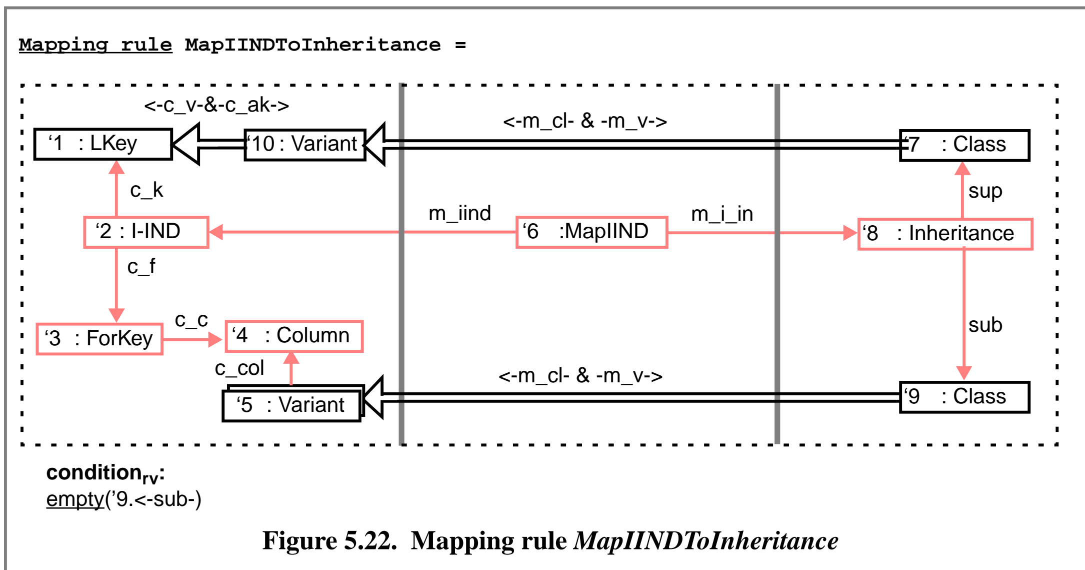


Figure 5.22. Mapping rule *MapIINDToInheritance*

5.2.5 Discussion

The main advantage of using triple graph grammars to specify and implement schema translators is their high level of abstraction. Graph-oriented specifications are much easier to define, comprehend, and extend than textual formalisms. Another benefit of this approach is that it enables the generation of bi-directional translators, because it defines correspondences among increments in both data models. Hence, triple graph grammars are best suited to integrate two document types with similar concepts and granularity. The previous section demonstrates the elegance of using triple graph grammars to define correspondences among similar concepts like INDs and relationships. Still, the triple graph grammar approach reaches its limit when there is a significant divergence between the expressiveness of both data models to be integrated. This was exemplified in Section 5.2.2 where we used two additional reverse productions to recover inheritance relationships with abstract classes from variant structures in the logical schema.

Even though the presented mapping rules define a bi-directional mapping among logical and conceptual schemas, it is important to note that this mapping is partial: further mapping rules are needed to define correspondences among additional conceptual constructs like aggregations and *many-to-many* relationships. These mapping rules can be defined analogously to the rules described before (cf. [Wad98]). Typically, their definition leads to ambiguities in the (reverse) translation process from the logical to the conceptual schema. For example, a given R-IND can be mapped to an association or an aggregation, and an RS with two foreign keys can be mapped to a class or a *many-to-many* relationship (association or aggregation). Such ambiguities can be solved by adding priorities to mapping rules [JSZ96] or extending the logical schema by further semantic annotations, e.g., to mark an aggregation relationship. Still, we made the experience that the number of mapping rules grows very large if we strive to consider all possible (and reasonable) correspondences among logical and conceptual schema constructs. We tackle this problem by combining a fully automatic schema translator generated from a limited set of mapping rules with a set of conceptual *redesign transformations*. The reengineer can use these redesign transformations to choose from alternative conceptual constructs while the correspondences to the logical schema are kept automatically.

5.3 Conceptual schema redesign

In the previous section, we described and specified a canonical translation from an analyzed logical schema to a conceptual schema (and vice-versa). This canonical translation allows to represent and assess the persistent data structure of LDBs on a higher level of abstraction by employing object-oriented modeling concepts. Still, in most DBRE scenarios such a canonical translation is just a first step in the schema migration process: typically, the initial conceptual schema is restructured and extended in order to meet new requirements and fully exploit abstract modeling concepts, e.g., aggregations and cardinality constraints. Most DBRE tools applied in the activity of conceptual schema restructuring provide little support beyond the functionality of conventional DB schema design tools: they just provide editor operations to create or remove schema artifacts like entities, attributes, relationships. Most of these schema editors are also capable of generating (new) DB schema catalogs from the conceptual model. However, these approaches do not maintain information about the dependencies of the restructured conceptual schema with the original LDB schema. This is a severe limitation in case of iterations in the DBRE process because this information is needed to propagate changes of the analyzed schema to the conceptual schema and re-establish consistency (cf. page 22). Likewise, it is not possible to modify the original logical schema *incrementally* according to extensions made in the conceptual model. Incremental schema changes are especially important in the DBRE domain because they are local (e.g., insertion of new attributes or RS), i.e., they allow to preserve a large amount of the legacy data. Finally, dependency information between the logical and the conceptual schema is needed to generate middleware components that facilitate data integration.

5.3.1 Schema redesign transformations

In our approach, we employ the notion of *schema (redesign) transformations* instead of simple editing operations to overcome the described limitations. Redesign transformations have traditionally been applied in logical DB design [BCN92, p. 424]. For example, they are used as decomposition operations in algorithms to obtain a normalized relational DB schema [EN94]. In contrast to simple editor operations, schema transformations include a definition of the *semantics* of the schema change. This semantics is declared by a definition how instances of the source schema are translated to instances of the target schema of the transformation. Hence, a schema transformation is often defined as a tuple (T,I), where T denotes the so-called *structure transformation* and I is the *instance mapping* [Hai91]. The structure transformation represents a function $T{:}S^T \rightarrow S^*$ that is defined on the subset $S^T \subset S^*$ of all schemas S^* that satisfy the precondition of T. It replaces a given source schema $S \in S^T$ by a target schema $S' = T(S)$. Consequently, the instance mapping $I{:}\mu(S) \rightarrow \mu(S')$ converts valid database extensions of the original schema into valid extensions of the target schema S'. $\mu(S)$ denotes the *information capacity* of a given schema S which is defined as the set of all valid database states (or instances) of S. According to [BCN92], a given schema transformation can be classified as

- *information-preserving (IP)* if its instance mapping I is bijective or
- *information-changing (IC)*, otherwise, namely,
  - *information-augmenting (IA)* if I is injective but not surjective or
  - *information-reducing (IR)* if I is surjective but not injective.

Many approaches in the domain of DB evolution allow to reorganize the data after a redesign transformation has been applied to the schema [Sch93, Tre95]. In our application, we focus on integrating legacy DB schemas with distributed, object-oriented technology by generating a middleware component that provides data integration. The necessary schema dependency information is represented by the schema mapping graph (SMG) (cf. Section 5.6). Consequently, we describe the semantics of schema transformations by defining the modification to the SMG in correspondence to the structural transformation of the conceptual schema. Using a data integration middleware, the conceptual schema represents an object-oriented view on the implemented logical schema. Redesign transformations that are performed to this view do not necessarily change the implemented data model. In fact, we are interested in keeping the modifications of the legacy schema to a minimum to preserve compatibility with existing legacy application code. Only IA transformations require actual changes to the implementation of the schema.

insufficiency of predefined transformations

Several researchers have proposed catalogs of redesign transformations for different conceptual schemas, e.g., [BKKK87, Hai91, Sch93, Tre95, BP96]. Typically, these catalogs consist of so-called *primitive* transformations which serve as the basic building blocks of more *complex* transformations. Banerjee et al. argue that their catalog of transformations is complete [BKKK87]. Still, Schiefer shows examples for important schema transformations that cannot be performed with this catalog [Sch93]. Especially, in the context of DBRE, we doubt the feasibility of defining a complete catalog of schema redesign transformations. This is because LDB schemas often comprise complex idiosyncratic optimization patterns and unforeseen design structures [BP95]. An example for such complex optimization patterns is described in Chapter 2 on page 21. In most cases, it is not sufficient to apply primitive transformations to the building blocks of such a pattern. On the contrary, a transformation that is suitable to normalize such a complex structure has to deal with the entire pattern. Hence, our special focus is on providing a catalog of transformations that is easily *extensible* rather than trying to create a catalog that is *complete*. The combination of the expressive power of graph grammar productions with the *Progres* code generation mechanism [SWZ95] enables us to achieve this goal: the catalog of redesign transformations that are provided by our schema migration tool can easily be extended or customized on a high level of abstraction.

5.3.2 An extensible catalog of schema redesign transformations

Figure 5.23 shows an initial catalog of schema transformations which are specified and implemented in this dissertation. A semi-formal proof of their classification as IP, IC, IR, or IA transformations is given by Rummel [Rum98]. In the following, we will discuss four of these transformations in more detail to illustrate our approach. The specifications for all other transformations are presented in Appendix B.

SplitClass

As a first example, we have chosen the IP redesign transformation *SplitClass* which is specified as a graph production in Figure 5.24. Redesign transformations are performed interactively by the reengineer who provides the parameters included in the signature of the graph production. *SplitClass* creates a new class with name *clName* which is connected by a total *one-to-one* association to a given class *cl*. Parameters *oldRole* and *newRole* contain the role names of the pre-existing class and the new class in the created association, respectively.

Transformation	Informal description	Type
Aggregate	Transforms an association into an aggregation	IP
AssociationToClass	Transforms an association between two classes to an intermediate class with two associations	IP
ChangeAssocCardinality	Modifies the cardinality of a given association	IC
ChangeAttributeType	Changes the type of an attribute	IC
ClassToAssociation	Transforms a class that participates in two *one-to-many* associations to a *many-to-many* association	IP
CreateAssociation	Creates an association between two given classes	IA
CreateAttribute	Creates an attribute in a given class	IA
CreateClass	Creates a new class	IA
CreateInheritance	Creates an inheritance relationship between two given classes	IA
CreateKey	Creates a key for a given class	IR
DisAggregate	Transforms an aggregation into an association	IP
Generalize	Creates a generalization for a given class	IA
ConvertAbstract	Converts a concrete class into an abstract class	IR
ConvertConcrete	Converts an abstract class into a concrete class	IA
MergeClasses	Merges two classes which are associated by a *one-to-one* relationship into a single class	IP
MoveAttribute	Moves an attribute from one class to an associated class via a given *one-to-one* relationship	IP
PushDownAttribute	Moves an attribute of a given class to its specialization	IR
PushDownAssociation	Moves a relationship of a given class to its specialization	IR
PushUpAttribute	Moves an attribute of a given class to its generalization	IA
PushUpAssociation	Moves a relationship of a given class to its generalization	IA
Remove	Removes an increment from the conceptual schema	IC
RenameAttribute	Changes the name of an attribute	IP
RenameClass	Changes the name of a class	IP
RenameRelationship	Changes the role names of a relationship	IP
Specialize	Creates a specialization for a given class	IA
SplitClass	Splits a class in two classes connected by a *one-to-one* relationship	IP
SwapAssocDirection	Swaps source and target of a given association	IP

Figure 5.23. Catalog of conceptual redesign transformations

In Figure 5.24 and the following graph productions, we use bold nodes and edges to make it easier to identify the part of the production that specifies the actual change in the conceptual schema. Thin nodes and edges represent the remaining part that specifies the corresponding modification in the mapping graph. Production *SplitClass* specifies that the newly created class (node 6') is mapped to the same variants that have been mapped to the pre-existing class (node 1'). A new edge of type *m_id* represents the information that OIDs of the new class are translated to the same value-based key like OIDs of the old class. The new association is not mapped to any foreign key (R-IND) in the relational schema. However, it is connected to a new node of type *MapRel* to indicate that the association has already a corresponding representation in the logical schema (cf. the mapping algorithm on page 126).

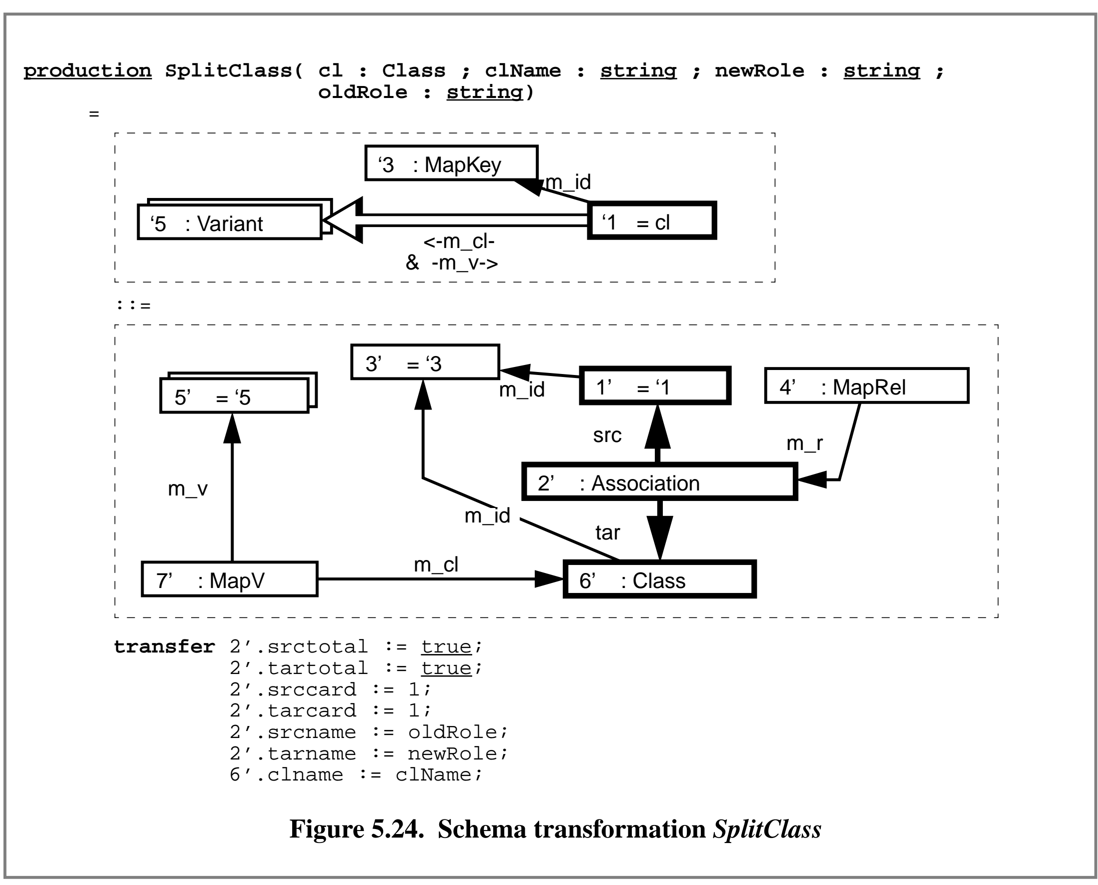


Figure 5.24. Schema transformation *SplitClass*

MoveAttribute

Classes that are newly created by applying transformation *SplitClass* do not contain attributes or participate in any relationship other than the newly created association. The reengineer can use IA transformations like *CreateAttribute* or *CreateAssociation* to create new class properties. In this case, the mapping rules defined in Section 5.2.2 are used to translate these properties to columns and foreign keys which extend the original logical schema. Besides the possibility to add new properties, the catalog in Figure 5.23 contains two transformations (*MoveAttribute* and *MoveAssociation*) that allow to move class properties from one class over an *one-to-one* association to another class. These transformations do not augment the information capacity of the schema. Hence, they do not imply changes in its implementation.

The graph production for transformation *MoveAttribute* is presented in Figure 5.25. The two parameters *attr* and *assoc* represent the attribute that has to be moved and the association that connects source and target of this relocation operation. The right-hand side of production *MoveAttribute* shows that the attribute which was initially aggregated in class ‘1 by a *c_att* edge has been relocated to class 3’ after the transformation has been applied. The information about the relocation is reflected in the mapping graph by adding the set of all *MapRIND* nodes (‘5) to the access path of the relocated attribute which have been mapped to the association. This is done by adding *a_via* edges from the attribute mapping node ‘6 to all nodes in set ‘5.[a]

a In Section 5.6, we use this information for generating middleware components for data integration.

Still, it is also possible that association *assoc* is not mapped to any *MapRIND* node, e.g., if it has been created by applying the *SplitClass* transformation. Hence, node set ‘5 is defined to be optional. In the case that no match can be found for node set ‘5, the mapping information of the relocated attribute remains unchanged.

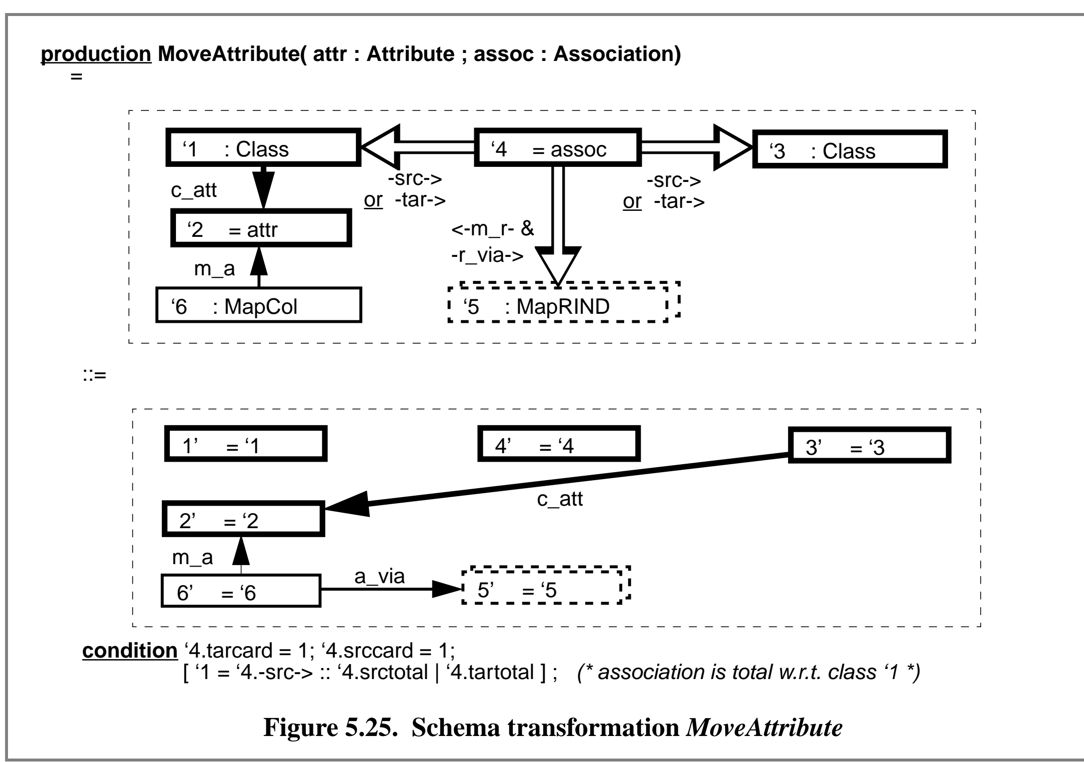


Figure 5.25. Schema transformation *MoveAttribute*

The application condition of production *MoveAttribute* restricts its applicability to *one-to-one* associations only. The relocation of class properties over *many-to-one* associations is ambiguous w.r.t. to the instance conversion and, thus, has to be prohibited. On the other hand, relocating class properties over a *one-to-many* association would represent an IA transformation. In the case that a relocation operation aims at an augmentation of the information capacity, the corresponding properties have to be deleted from the variants mapped to class ‘1 and added to the variants mapped to class ‘3. This can be done by a concatenation of *remove* and *create* transformations (cf. Figure). Strategies to reorganize the available data after IA transformations have been developed in the domain of DB evolution [Sch93, Tre95]. One typical solution is to insert default values for undefined attribute values.

Association ‘4 has to be total w.r.t. class ‘1 to avoid information augmentation. This requirement is represented by a *conditional boolean expression* to cover the case that class ‘1 is the source of the association or its target, respectively. The semantics of this *conditional expression* is that if nodes ‘1 and ‘4 are connected by an edge of type *src*, attribute *‘4.srctotal* is evaluated as the result of the expression. Otherwise, the result is defined as the value of attribute *‘4.tartotal* (cf. [Tea99]).

Generalize

The transformations described so far employ relationship concepts like association and aggregation to redesign the structure of conceptual schemas. We propose additional transformations to modify inheritance structures. Two important examples are transformations *Generalize* and *Specialize*. The purpose of transformation *Generalize* is to create a new generalization for the root class of an inheritance hierarchy, while transformation *Specialize* is used to insert a new subclass of a given class. New classes which are created by these two transformations are mapped to additional variants in the logical schema. We have selected this implementation alternative because it does not entail modifications of the logical schema and a reorganization of the legacy data. Other possible implementations of inheritance relationships are described, e.g., by Hainaut et al. [HHEH96] and Fussell [Fus97]. Note, that transformation *Generalize* creates a concrete class per default which can be converted to an abstract class using transformation *ConvertAbstract* from our catalog.

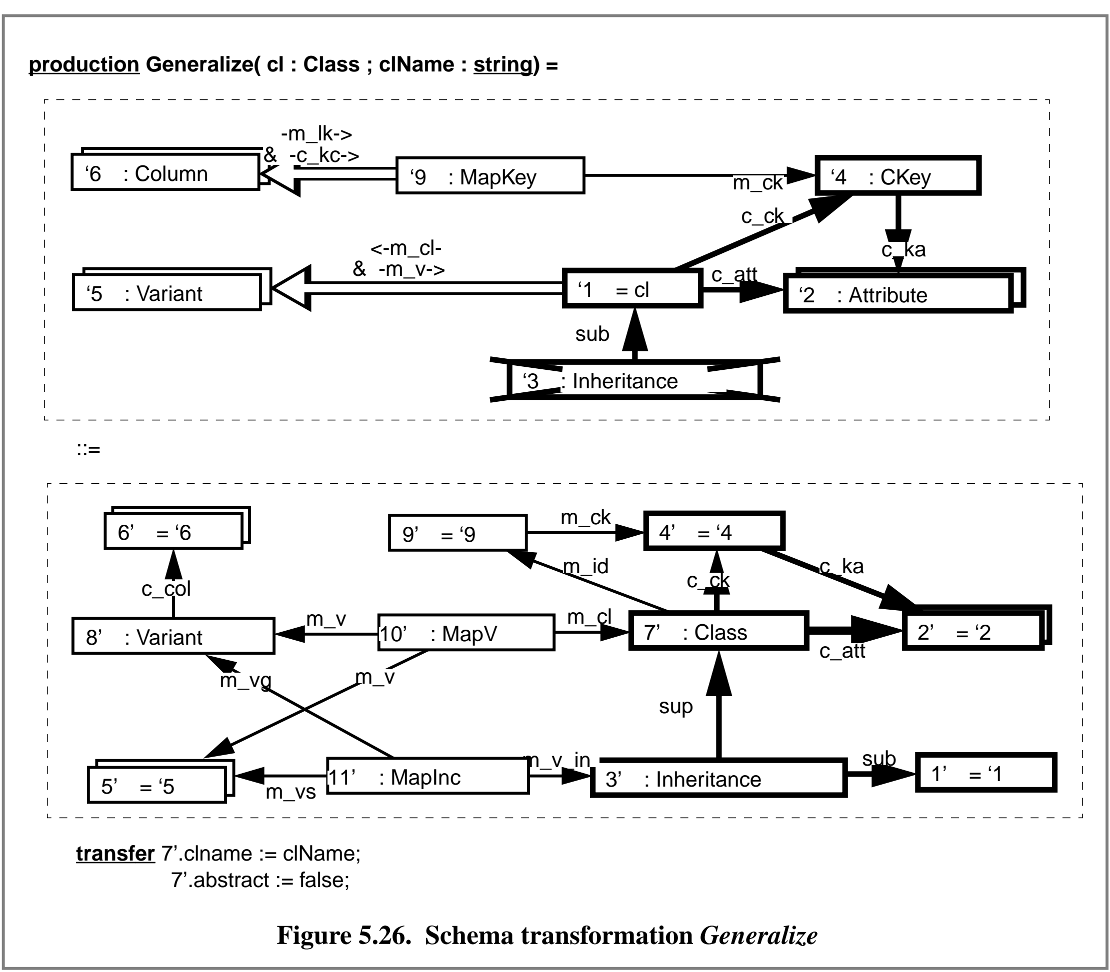


Figure 5.26. Schema transformation *Generalize*

The specification for transformation *Generalize* is presented in Figure 5.26. Its signature has two parameters, namely the class that has to be generalized (*cl*) and the name of the new superclass (*clName*). The bold graph elements of the corresponding production show that the key attributes of class *cl* (‘2) are relocated to the new class (7’). This is because the new class is represented by a new variant (8’) in the logical schema and each variant has to include the primary key of its RS. The inheritance relationship itself is mapped to the inclusion of the new variant (8’) in the existing variant (5’) by a node of type *MapInc* (11’).

PushUpAttr

Similar to the relocation of class properties via associations (*MoveAttribute*, *MoveAssociation*), we define redesign transformations to relocate class properties in inheritance hierarchies. According to the common practice to denote inheritance hierarchies as inverse vertical trees we have named these transformations *PushUpAttribute*, *PushUpAssociation*, *PushDownAttribute*, and *PushDownAssociation*. The first two transformations are information-augmenting while the latter two transformations are information-reducing. As an example, Figure 5.27 shows the specification of transformation *PushUpAttribute* which relocates a given attribute from one class to its generalization.

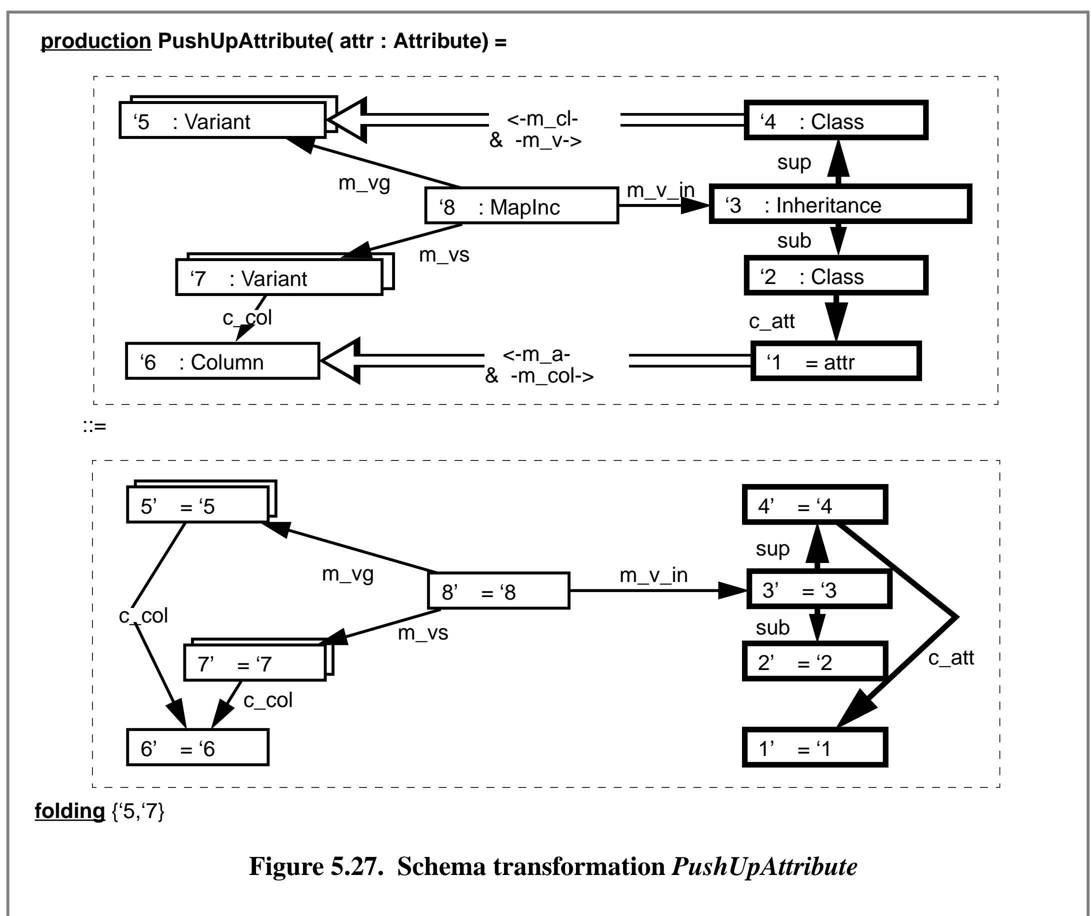


Figure 5.27. Schema transformation *PushUpAttribute*

Note, that we restrict the application of relocation transformations to inheritance relationships that have been mapped to variants of a single RS. The reason for this restriction is that otherwise we would have to relocate the corresponding column in the logical schema to a different RS and reorganize the data. Consequently, *PushUp* and *PushDown* transformations cannot be applied to inheritance relationships that are mapped to I-INDs. If such a schema modification is desired the corresponding attribute has to be removed from the subclass and added to its generalization. Again, DB evolution strategies elaborated for example by Schiefer [Sch93] and Tresch [Tre95] can be used to reorganize the data accordingly.

5.3.3 Complex schema redesign transformations

In the previous section, we employed graph productions to specify a catalog of primitive schema redesign transformations. In order to facilitate maintainability of this catalog it should be minimal, i.e., it should not contain transformations that can be simulated by executing a sequence of other transformations in this catalog. Still, from the reengineer's point of view it is more convenient and efficient to use more powerful redesign transformations. For example, a reengineer might want to relocate several attributes over an aggregation. In this case, (s)he would prefer to select a single operation (e.g., *MoveOverAggregation*) instead of transforming the aggregation into an association (primitive transformation *DisAggregate*), moving each attribute separately (primitive transformation *MoveAttribute*), and transforming the association back to an aggregation (primitive transformation *Aggregate*).

The obvious solution to meet this requirement is to provide some kind of macro mechanism that allows to concatenate primitive transformations to more *complex* transformations. However, we have to be aware of the fact that each primitive transformation has its own application precondition. Hence, it is possible that the precondition of some intermediate transformation is not fulfilled. Let us assume that in the above scenario the reengineer wants to relocate attributes over a *one-to-many* aggregation. If the complex transformation *MoveOverAggregation* is implemented as a script that calls the different primitive transformations it will fail with the first call to *MoveAttribute* (because it requires a *one-to-one* association). Still, the precondition of the first primitive transformation (*DisAggregate*) was valid and it has been applied to the migration graph. Obviously, the result of such an aborted complex transformation is not what the reengineer intended.

The described example motivates the need for some mechanism which guarantees that complex transformations are executed either completely or not at all. This problem is well-known from the domain of transaction processing in database management systems [EN94]. Hence, one solution is to use a transaction monitor that, in case of a violated precondition, allows to recover the state of the migration graph before the execution of the complex transformation. An alternative solution is to check all preconditions of involved primitive transformations at the beginning of a complex transformation. However, this would involve additional effort to rewrite those preconditions which actually depend on the output of other primitive transformations in the complex sequence.

In our approach, we have selected the former alternative. We employ the transaction concept which is provided by the graph-oriented database *GRAS* [KSW95]. The *Progres* language provides control structures to specify such transactions. This is exemplified in Figure 5.28. In this example, *assoc* is declared as a local variable of type *Association*. Primitive transformations are invoked like simple method calls. Further complex redesign transformations can be defined analogously, e.g., a concatenation of transformations *Generalize* and *PushUpAttribute*.

```
transaction MoveOverAggregation( aggr : Aggregation ; attrs : Attribute [1:n])
=
  use assoc : Association
  do
     DisAggregate ( aggr, out assoc )
   & for all attr := attrs
     do
       MoveAttribute ( attr, assoc )
     end
   & Aggregate ( assoc, out aggr )
  end
end;
```

Figure 5.28. Complex transformation *MoveOverAggregation*

5.4 Incremental change propagation

Inconsistencies among different representations on various levels of abstraction often cause update problems in DBRE projects. In Chapter 2, we exemplified that such inconsistencies might be caused by process iterations (cf. page 22): whenever the reengineer discovers new information about the real semantics of (low-level) implementation constructs all (high-level) representations of the LDB that have been created so far must be updated accordingly. A further typical source of inconsistencies are *on-the-fly* modifications to the implementation of the LDB due to urgent requirements while the DBRE project is in progress. Detecting and eliminating such inconsistencies manually is a time-consuming and error-prone activity. Hence, a commonly used approach is to discard all created high-level views of the LDB and generate default representations anew. In this case, the redesign work that has been performed manually by the reengineer is lost and has to be repeated. Obviously, both alternatives are unsatisfactory. Therefore, we have developed an *incremental* approach to consistency management in DBRE environments. In this section, we describe an automatic mechanism to propagate changes of an LDB's implementation to its conceptual representation without discarding manually performed redesign operations that remain valid.

The developed consistency management mechanism is based on the fact that our approach to schema migration employs transformations as the fundamental concept. In Section 5.2.1, we have shown how to derive an automatic transformation system from a triple graph grammar to translate a logical LDB schema into an initial conceptual representation. Subsequently, we have proposed a catalog of redesign transformations that can be applied to this conceptual representation, interactively. The main idea of our consistency management concept is to keep track of input/output dependencies among all transformations that have been applied to the implemented logical schema. In the case of implementation changes or modified semantic annotations, this dependency information is employed to detect all transformations which are affected by the change. Each of these transformations is re-evaluated automatically to determine if their preconditions are still applicable. Only those transformations which have lost their applicability are discarded.

5.4.1 The history graph

history graph

In this dissertation, we have used graph productions to formalize and implement transformations. In this sense, the left-hand side of a graph production represents the input of the corresponding transformation, while its output is represented by the right-hand side. If we want to maintain input/output dependencies of applied transformations, we have to store information about the matches for the corresponding graph productions. A graph-based structure is most suitable to maintain these dependencies. We call the corresponding graph *history graph* because it reflects the migration history of an LDB schema. Figure 5.29 illustrates the basic structure of a history graph: applied transformations are explicitly represented by *T*-nodes with corresponding input and output parameters. Input parameters which have actually been removed by an applied transformation remain as place holders in the history graph to represent the necessary dependency information (cf. *C*-nodes with grey shape in Figure 5.29).

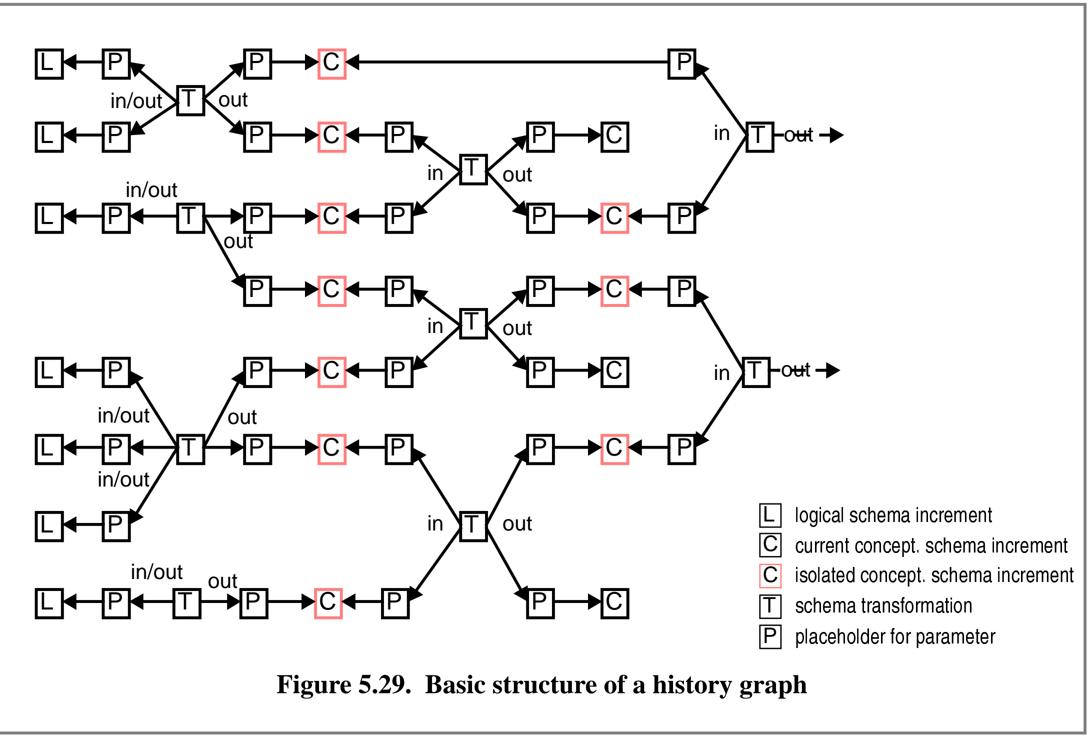


Figure 5.29. Basic structure of a history graph

transformation templates

In order to maintain the application contexts of transformations we have to identify and represent the graph elements on their left- and right-hand sides explicitly in the history graph. In the *Progres* graph model, it is sufficient to consider node parameters only, because they uniquely determine the application context of productions (cf. Definition 5.1 on page 115). For example, let us consider transformation *Generalize* in Figure 5.26 on page 142. It has six input node parameters and eleven output node parameters. Each parameter has a unique node number and some of the output parameters also serve as input. Figure 5.30 shows this input/output structure for transformation *Generalize*. The *Parameter* nodes serve as place holders for the actual parameters of a transformation application. Hence, we call this structure a *transformation template*. The parameter numbering is based on the node numbers of the corresponding *Progres* production.

dependencies among edges

Even though node parameters are sufficient to determine the application context of a *Progres* production, its application itself obviously depends also on edge parameters. These dependencies cannot be represented directly in the history graph because the underlying graph model does not allow for *higher-order* edges, i.e., edges that have edges as their source or target (cf. Definition 5.1 on page 115). However, this dependency information can be disregarded if all graph productions comply to the requirement that whenever an edge is modified its source and target nodes have to occur on the left-hand sides. This requirement is satisfied in all graph productions included in this dissertation. Still, *Progres* provides other means to modify edges in terms of so-called *redirection, embedding,* and *copy clauses* which can be added to productions (cf. [Tea99]). Such clauses may not be used in our approach.

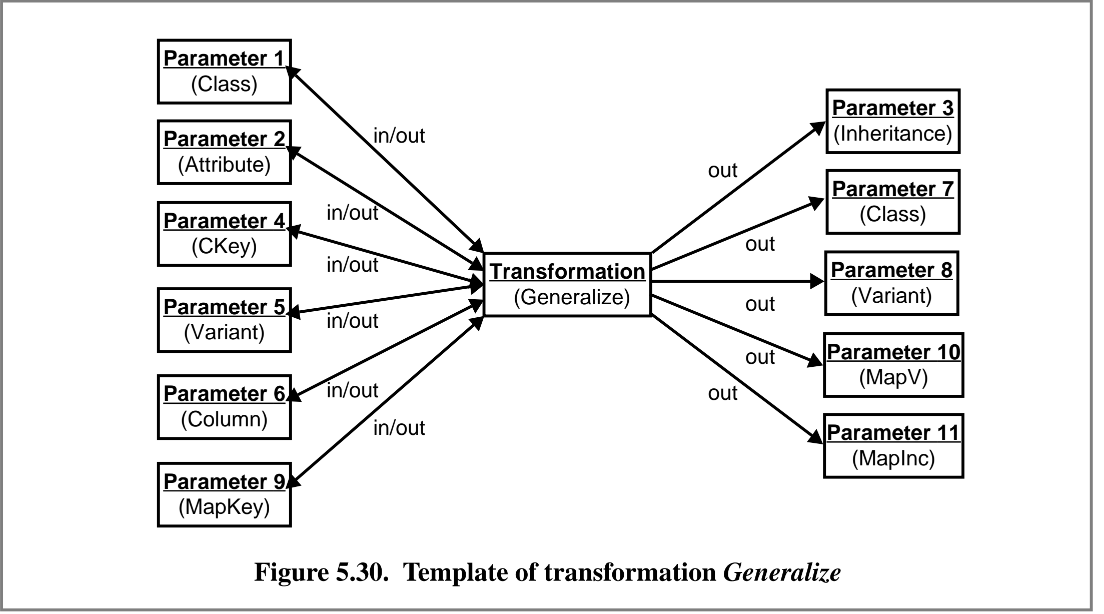


Figure 5.30. Template of transformation *Generalize*

restriction: path expressions

Another problem arises with *Progres* productions that employ path expressions. For example, production *Generalize* has two path expressions on its left-hand side (cf. Figure 5.26 on page 142). Although path expressions represent a powerful means to specify graph traversals they are problematic for our consistency management mechanism because they imply additional input dependencies. In principle, it is necessary to add input dependencies to each node that has been visited in an application of such a path expression. However, collecting all visited nodes would imply modifications to the internal implementation of the *Progres* compiler. On the other hand, prohibiting the usage of path expression completely would entail a severe restriction for the expressiveness of our formalism. Therefore, we decided to restrict our formalism to path expressions that have a maximum path length of two edge traversals. This restriction allows to combine the main benefits of path expressions with a simple (conservative) approach to consider the additional input dependencies. The idea is simply to add input dependencies to all direct neighbors of nodes matched to the left-hand side of an applied transformation. These additional nodes are called *1-context* of the actual input parameters of the transformation. Formal definitions for the 1-context and the entire application context of transformations are given in Definition 5.4 and Definition 5.5.

transitive closure in path expressions

The restriction to short path expressions does also prohibit the use of *Progres* operators for transitive closure (*,+) in such expressions. However, due to our experience this is no real limitation because transitive path expressions are usually employed in transformations to check for violations of invariant graph constraints. An example for such an invariant constraint is that there might not be two classes with the same name. Such invariant constraints do not depend on the actual transformation context and, thus, can be specified separately as described on page 118. They can be validated before a transformation is finally committed. In addition, this strategy reduces the redundancy in transformation specifications because otherwise the corresponding condition had to be specified in all transformations that may violate it.

Definition 5.4 1-context of a set of nodes

*The **1-context** of a set of nodes S in a graph G is defined as the set of nodes S' which contain all direct neighbors of nodes in S which do not belong to S, i.e.,*
$S'=1\text{-}context(G,S):=\{n \mid n \in N(G) \setminus S \wedge \exists e \in E(G) : (s(e) \in S \wedge t(e)=n) \vee (t(e) \in S \wedge s(e)=n)\}$

Definition 5.5 Context of a transformation application

*The **context** of an application of a transformation (represented by a production) r:(P,Q,C,T) to a graph G in a match m:P→G is defined by a tuple* $\bar{r}$*:(in, out, con1) of two mappings and a set:*

- $in:N(P) \rightarrow N(G)$ *with* $in(n)=m(n)$ *for* $n \in N(P)$,
- $out:N(Q) \rightarrow N(G^{\downarrow(r,m)})$ *with* $out(n)=\bar{m}(n)$ *for* $n \in N(Q)$, *where* $\bar{m}(n)$ *is the comatch of the production application (cf. Definition 5.3 on page 121).*
- $con1:=1\text{-}context(G,in(N(P)))$.

negative conditions

Negative application conditions in graph productions cause another problem because they specify the necessity for the absence of certain graph elements. Still, negative conditions are frequently needed to select the right transformation. An example is given in Figure 5.21 on page 135. Here, the absence of a C-IND node is required in order to map an R-IND to a partial *many-to-one* association. If the reengineer finds out, at a later point in time, that such a C-IND in fact exists, the transformation has to be undone. We solve the problem of negative application conditions as follows: we require that negative nodes have to be in the *1-context* of at least one other node on the left-hand side of the production. Whenever a new node *n* has been created that is used in a negative application condition, the nodes in the *1-context* of *n* are marked changed.

history graph model

Figure 5.31 shows a graphical *Progres* specification for the history graph model. According to Definition 5.5, input and output dependencies are represented by edges of type *In* and *Out*, whereas the nodes in the 1-context of a transformation application are referenced by *con1* edges. Figure 5.31 also shows that the history graph model is an extension of the migration graph model, i.e., the history graph contains the migration graph as a subgraph. Node type *Increment* represents a generalization if all node types in the migration graph model represented in Figure 5.2 on page 117. Edges of type *actual* connect parameter place holders of transformation templates with their actual input and output parameters in the migration graph.

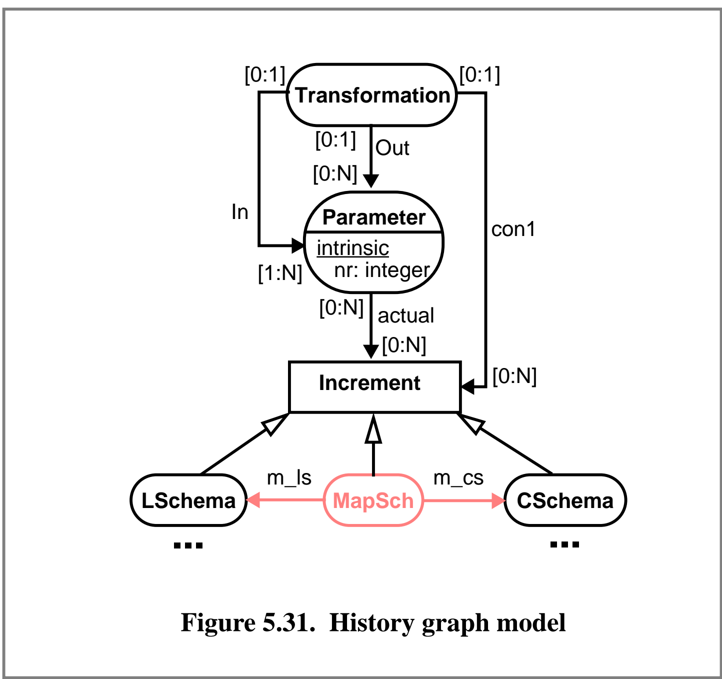


Figure 5.31. History graph model

Definition 5.6 History graph

*The **history graph** is a graph that includes the migration graph as a subgraph. Moreover, it contains nodes and edges that represent all application contexts of (mapping and redesign) productions in the entire editing history. The corresponding extension of the migration graph model (Figure 5.2 on page 117) is given in Figure 5.31. The projection of a history graph* $H:(N,E,y_N,A)$ *on the current migration graph* $MG(H):(N',E',y'_N, A')$ *includes all increments which do not occur as in-parameters of a transformation without occuring as out-parameters of the same transformation, i.e,*

- $N':=\{n\in N \mid y_N(n)\notin \{\text{'Transformation', 'Parameter'}\} \wedge$ $(\forall n_p,n_t\in N, \forall e_a,e_i\in E: t(e_a)=n \wedge s(e_a)=n_p \wedge t(e_i)=n_p \wedge s(e_i)=n_t \wedge y_E(H)(e_a)=\text{'actual'} \wedge$ $y_E(H)(e_i)=\text{'In'} \Rightarrow \exists e_o \in E: t(e_o)=n_p \wedge s(e_o)=n_t \wedge y_E(H)(e_o)=\text{'Out'})\}$
- $E':=\{e\in E \mid s(e), t(e) \in N'\}$
- $y'_N:=y_N \backslash \{\text{'Transformation', 'Parameter'}\}$
- $A'=A$

The history graph defined above is a specific implementation of the general concept of a *graph process* as introduced by Corradini et al. [CMR96]. A graph process is a *partially ordered structure*, plus suitable *mappings* which relate the elements of this structure to those of a given typed graph grammar. According to this terminology, the *Transformation* and *Parameter* nodes with their *In*, *Out*, and *con1* edges represent the above mentioned partially ordered structure; edges of type *actual* represent the mapping between this partially ordered structure and the typed graph elements representing the logical schema, the conceptual schema, and the SMG, respectively.

5.4.2 The propagation mechanism

application of transformations to the history graph

In order to log the application of transformations in the history graph, we have to redefine the way how transformations (graph productions) are applied (cf. Definition 5.7). The main difference of this definition w.r.t. Definition 5.3 on page 121 is that nodes which are deleted on the right-hand side of production are not removed from the history graph but they are isolated, i.e., all their in- and out-going edges in the corresponding migration graph are deleted.

Definition 5.7 Application of transformations to a history graph

A transformation that is represented by a production t:(P,Q,C,T) is ***applied*** *to a history graph H in the following five steps.*

- ***CHOOSE*** *an occurrence of the left-hand side P in MG(H) (analogously to Definition 5.3 on page 121).*
- ***CHECK*** *the application conditions according to C.*
- ***REMOVE*** *all edges from H that have been matched to edges in E(P\Q) .*
- ***ADD*** *all elements to G which are new in Q, i.e., which do not occur in P. These new elements are glued to G in the preserved graph elements identified by m(P∩Q).*
- ***LOG*** *the context of the applied transformation t:*
  - ***EXTEND*** *H by the corresponding template for t (cf. Figure 5.30)*
  - ***EMBED*** *the new template according to the context information, i.e.,*
    • *create an* actual *edge from each parameter to the corresponding node in H, and*
    • *create a* con1 *edge from the new* Transformation *node to each node in 1-context(MG(H), N(m(P)).*
- ***ISOLATE*** *all nodes in MG(H) that have been matched to nodes in N(P\Q), i.e., remove all edges from H which belong to MG(H) and are connected to nodes in m(P\Q).*
- ***TRANSFER*** *attribute values to nodes in G that match nodes in Q.*

change propagation

In the remainder of this section, we describe how the information stored in the history graph can be used for incremental change propagation. Let us assume a scenario where an analyzed logical LDB schema has been translated to a conceptual representation which subsequently has been redesigned and extended. Our case study describes a sample situation for a change in the logical schema during such an ongoing conceptual migration process (cf. page 22). Using the history graph that has been created during the translation and editing history, the change propagation process has four major phases, namely *forward propagation*, *backward propagation*, *reevaluation*, and *translation*.

Phase I: forward propagation

In the first phase, the input/output dependencies in the history graph are used to detect all transformation applications (and increments in the conceptual schema) which are affected by the modifications in the logical schema. This step is illustrated in Figure 5.32 where *L*-nodes with a pencil mark the modifications and extension of the logical schema, respectively. Note, that in this phase, *con1* edges are used in the same way like *in* edges to find (potentially) affected transformation applications. However, we do not represent the 1-context of transformation applications in Figure 5.32 (and the following diagrams) for reasons of simplification.

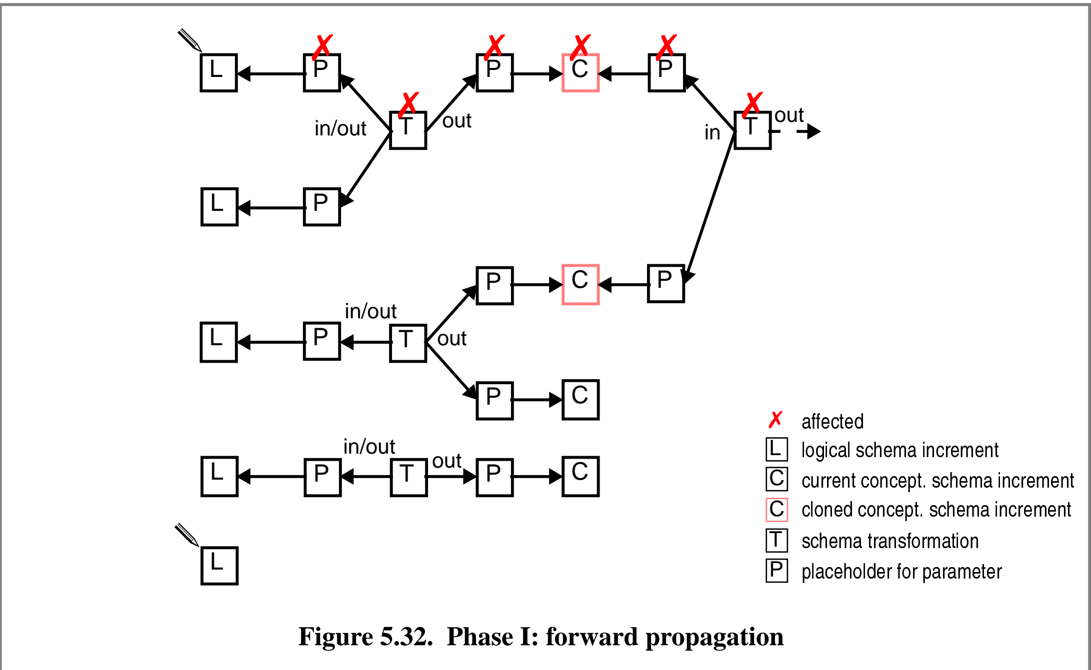


Figure 5.32. Phase I: forward propagation

Phase II: backward propagation

Obviously, all transformation applications that have been marked in the forward propagation step have to be validated. However, some of these transformation applications depend on input parameters which have been consumed by a transformation. These parameters, which are only represented by isolated place holders, have to be reproduced before the dependent transformation can be re-evaluated. Reproducing these parameters means to re-evaluate all transformations that have been applied to produce them. Some of the transformation applications that have to be re-evaluated might not have been marked in the forward propagation phase because they are not directly affected by the modification in the logical schema. Hence, we need a further *backward propagation* phase to mark such indirectly affected transformation applications in the history graph (cf. Figure 5.33).

Phase III: reevaluation

In the third phase, the marked transformation applications are re-evaluated in the predefined order of their input/output dependencies. Reevaluating a transformation application means to apply the corresponding transformation anew to the current (maybe changed) parameters. Each transformation that remains applicable remains in the history graph. Figure 5.34 shows that the output parameters of such a transformation and the input parameters of a dependent transformation application are actualized to the newly created conceptual schema increments. All old parameter place holders are deleted from the history graph. Likewise, all transformations which are no longer applicable are deleted as well. In Figure 5.34, this is illustrated for the right-most transformation template.

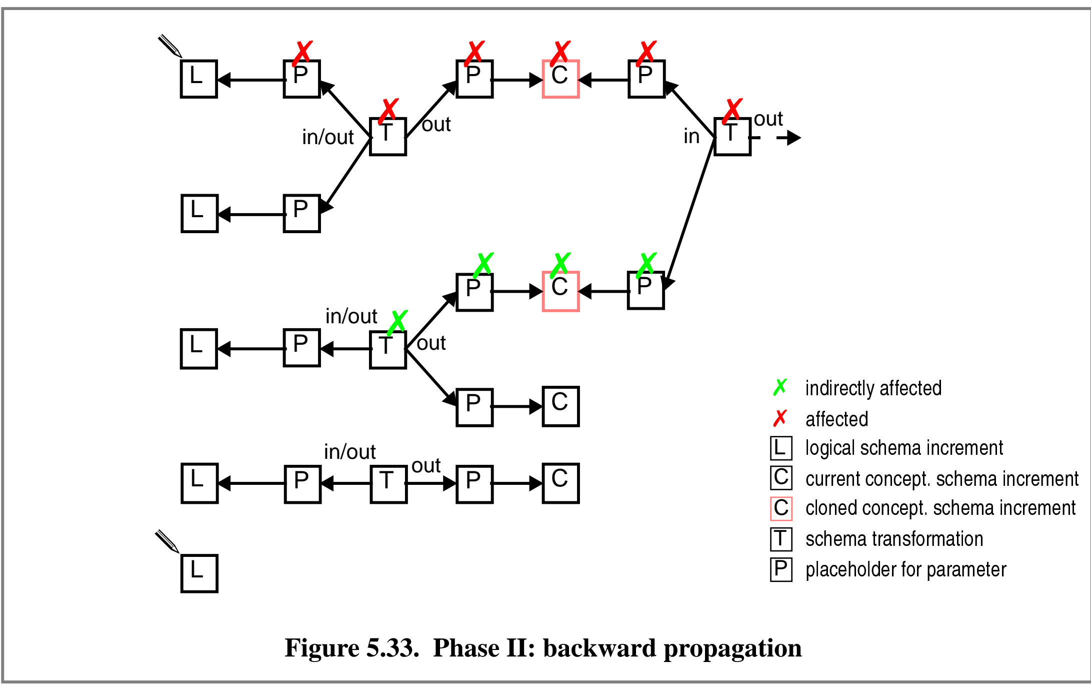


Figure 5.33. Phase II: backward propagation

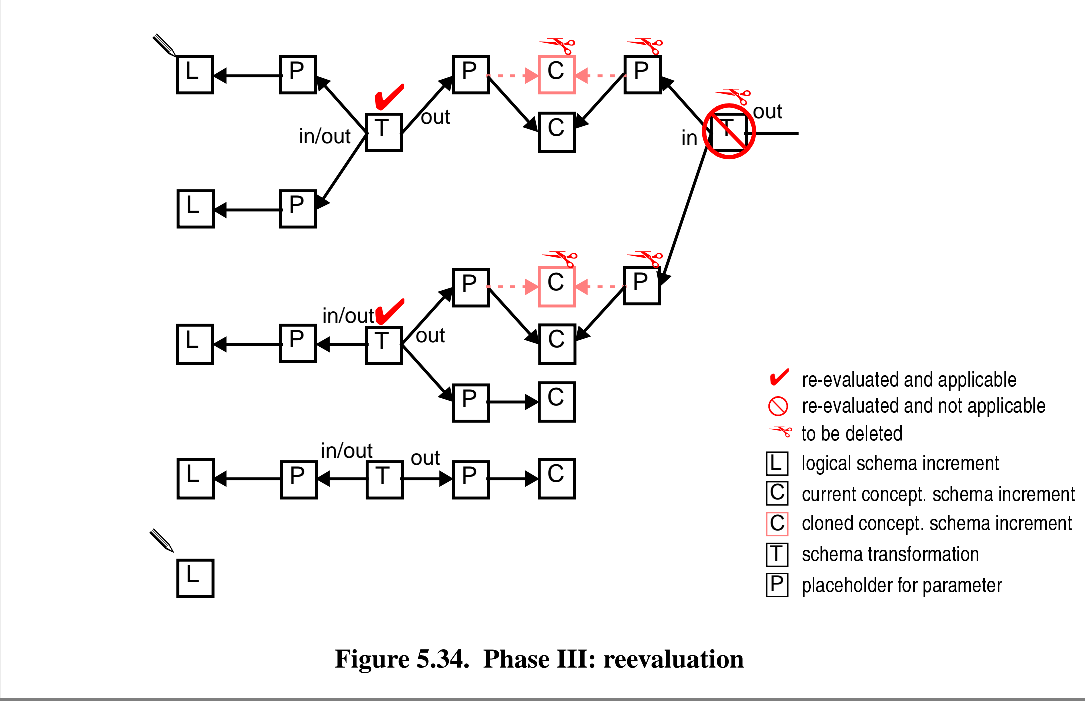


Figure 5.34. Phase III: reevaluation

Phase IV: translation

The purpose of the final phase in the change propagation process is to translate logical schema increments which do not have a current representation in the conceptual schema (cf. Figure 5.35). This is necessary for logical schema increments which have been added during the last modification. Furthermore, translations of existing logical schema increments might have been deleted during the reevaluation phase because the corresponding transformation rules are no longer applicable. At the end of this translation phase, the consistency of the logical schema with its conceptual representation has been reestablished.

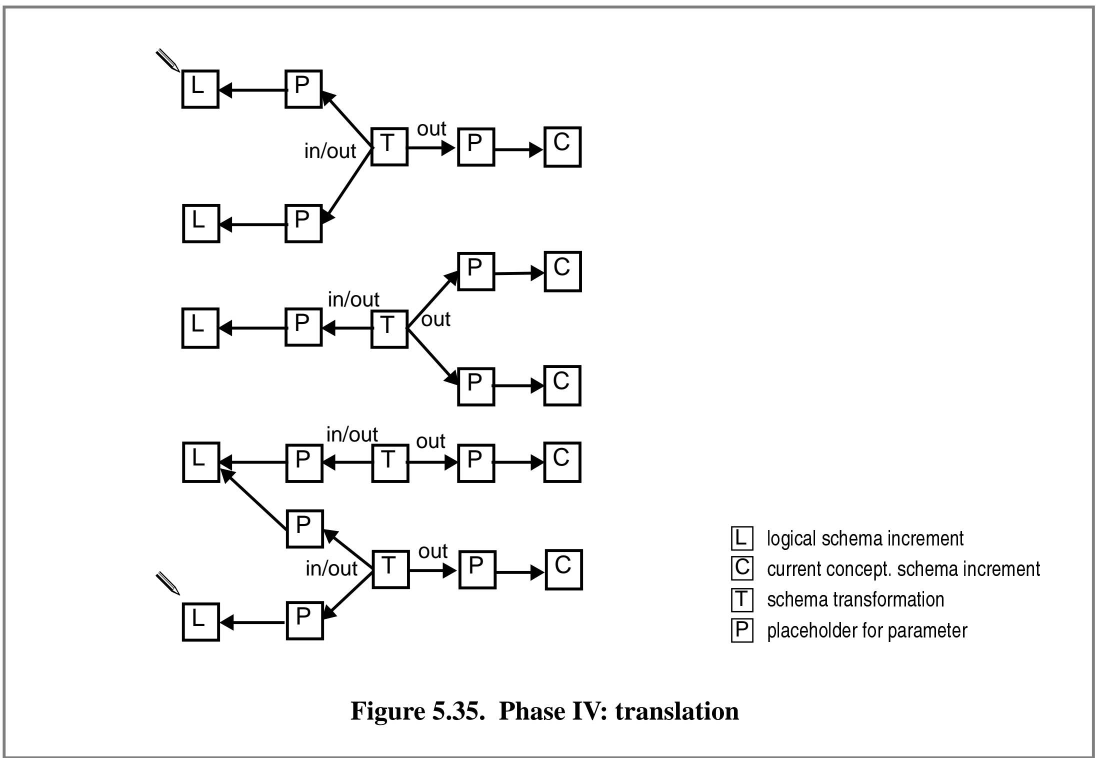


Figure 5.35. Phase IV: translation

realization in Progres

The described incremental change propagation algorithm has been implemented in *Progres*. This implementation is described in detail by Wadsack [Wad98]. Figure 5.36 shows the transaction *PropagateChange* which formalizes the propagation process. It requires an argument *changeSet* which represents the set of all logical schema increments that have been added or modified.[a] In the first phase, (transitive) path expressions are used to collect all directly affected transformation applications in the local variable *affectedTrafoAppls*. In the backward propagation phase all transformation applications are added to this variable which are needed to reproduce consumed parameters. Phase III is performed in a loop that repeatedly chooses one transformation application (*oldTrafoAppl*) that does not depend on any other transformation application in *affectedTrafoAppls*. Note, that the *Progres* operator *and* computes the intersection of two sets. The following *choose* statement tries to reapply the transformation in *oldTrafoAppl*. If this is possible and the specified invariant graph constraints are fulfilled it actualizes the output parameters of the new transformation application. Subsequently, the re-evaluated transformation application *oldTrafoAppl* is removed from the set *affectedTrafoAppls*. This is done by using the *Progres* operator *but_not* which computes the difference of two sets. In the case that the transformation in *oldTrafoAppl* has lost its applicability, the *else* block of the *choose* statement in Figure 5.36 collects all dependent transformation applications in variable *depTrafoAppls*. Subsequently, these transformation applications are removed from the history graph.

a These increments are collected by the *Varlet Analyst* during interactive schema analysis activities.

```
 transaction PropagateChange( changeSet : Increment [1:n]) =
    use
      affectedTrafoAppls, depTrafoAppls : Transformation [0:n];
      oldTrafoAppl, newTrafoAppl : Transformation
    do

(* Phase I: forward propagation *)

        affectedTrafoAppls := changeSet.(     (    <-actual-
                                                &  <-In-     )
                                          or  <-con1-         )
      & affectedTrafoAppls :=
          affectedTrafoAppls.( (    -Out->
                                 &  -actual->
                                 &  (     (    <-actual-
                                            &  <-In-     )
                                      or  <-con1-         ) ) * )

(* Phase II: backward propagation *)

& affectedTrafoAppls :=
          affectedTrafoAppls.( (    (     (    -In->
                                            &  -actual-> )
                                      or  -con1->         )
                                 &  <-actual-
                                 &  <-Out-                   ) * )

(* Phase III: reevaluation *)

     & loop
        oldTrafoAppl :=
          affectedTrafoAppls.valid (empty ( (self.-In->.-actual->.<-actual-.<-Out-)
                                            and affectedTrafoAppls
        & choose
              Reevaluate ( oldTrafoAppl, out newTrafoAppl )
            & CheckGraphConstraints (* cf. page 118 *)
            & ActualizeOutParams ( oldTrafoAppl, newTrafoAppl )
            & affectedTrafoAppls :=
                (affectedTrafoAppls but not oldTrafoAppl)
          else
              depTrafoAppls :=
                oldTrafoAppl.( (    (          -Out->
                                      but not  -In->  )
                                 &  -actual->
                                 &  <-actual-
                                 &  <-In-               ) * )
            & RemoveTrafoAppls ( depTrafoAppls )
            & affectedTrafoAppls :=
                (affectedTrafoAppls but not depTrafoAppls)
          end
      end

(* Phase IV: remapping *)

     & MapSchema (* cf. Figure 5.10 on page 126 *)

    end
  end;
```

Figure 5.36. Transaction *PropagateChange*

adaption of productions

In order to retrieve the necessary information about the context of transformation applications, we have to modify the corresponding *Progres* productions in a way such that the matched nodes are returned as parameters. Moreover, the described propagation algorithm requires the possibility to re-evaluate transformations with a predetermined application context. Therefore, we split each production that specifies a schema transformation into two separate parts, namely a *graph test* and a *parameterizable production* that accepts a predetermined application context. This is exemplified for transformation *Generalize* in Figure 5.37 and Figure 5.38.

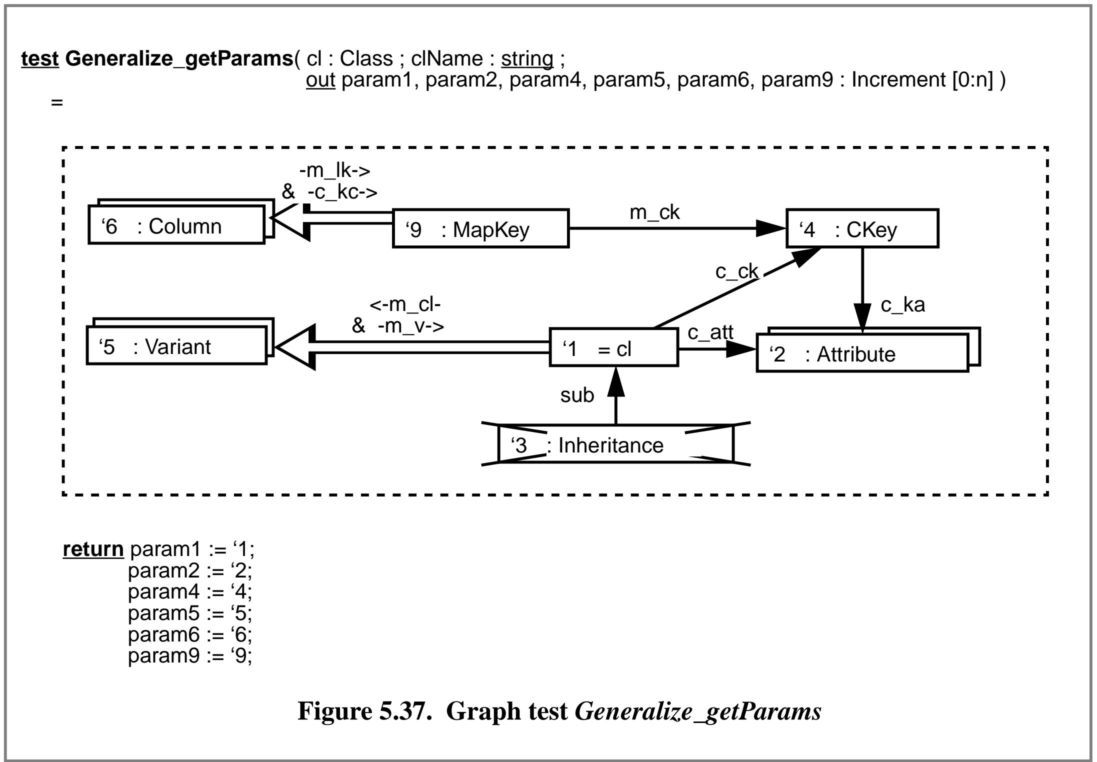


Figure 5.37. Graph test *Generalize_getParams*

Whenever a transformation is applied the graph test is used to deliver the input parameters for the application context of this transformation. The 1-context can be easily computed from these nodes (cf. Definition 5.4). If this test succeeds the corresponding parameterizable production is invoked with the delivered input parameters. Subsequently, this production returns the output parameters which are needed to complete the information about the application context. All nodes which are actually deleted by a production have to be added to its right-hand side, because they serve as isolated place holders in the history graph. During the change propagation process the parameterizable production is re-evaluated with the actualized application context. Note, that the described adaption of productions can be performed automatically by a canonical pre-compilation step and does not have to be done manually.

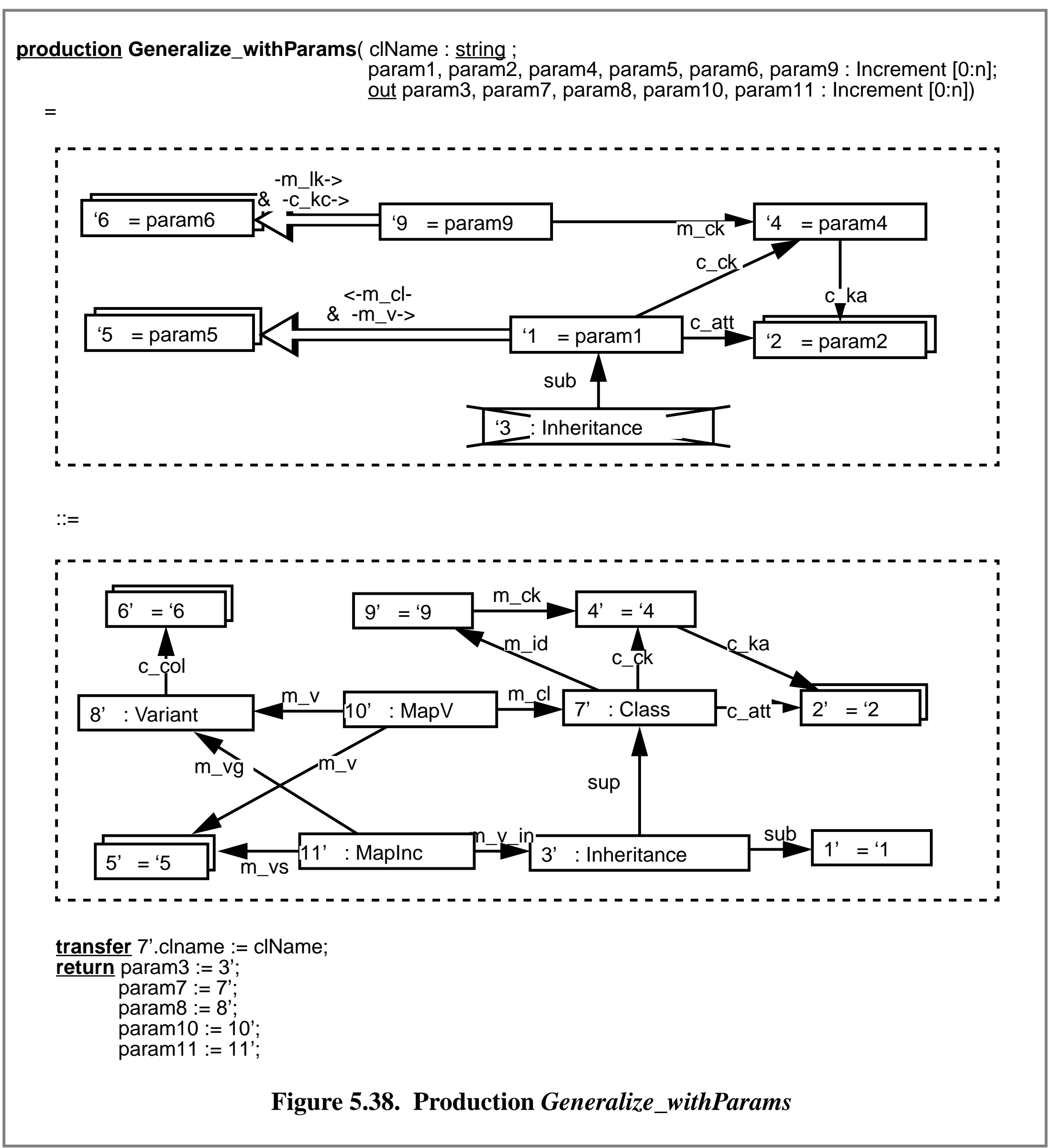


Figure 5.38. Production *Generalize_withParams*

scalability

The change propagation mechanism described above can efficiently be executed. Maintaining the history graph does not add to the run-time complexity of applying schema transformations. Each applied transformation extends the history graph by one instance of a transformation template (cf. Definition 5.7). Hence, the space complexity of the history graph is $O(n)$ where n is the number of transformations applied during the conceptual schema migration process. If we make the simplifying assumption that application conditions of graph productions can be validated in constant time then the time complexity of algorithm *PropagateChange* in Figure 5.36 is also $O(n)$.

5.5 Implementing the *Varlet Migrator*

Our approach to conceptual schema migration has been implemented in a tool called the *Varlet Migrator*. In order to achieve the incremental and iterative DBRE process described above, the *Varlet Migrator* is tightly integrated with the *Varlet Analyst* presented in Section 4.4. The following section describes this integrated architecture in more detail, while Section 5.5.2 is illustrates to the user's perspective.

5.5.1 Architecture

The central component of the *Varlet Migrator* is a repository that maintains the migration graph in the dedicated software engineering database *GRAS* [KSW95]. GRAS provides the possibility to access large graphs efficiently with full support for transaction management, recovery, and operation undo/redo. Figure 5.39 shows that the schema for this repository is devided into logical subsections for the ASG models of logical and conceptual LDB schemas, the mapping graph model, and the history graph model. The grey components in Figure 5.39 illustrate that GRAS is also used as repository for the *Varlet Analyst*.[a] This architecture enables the desired tight integration among both tools.

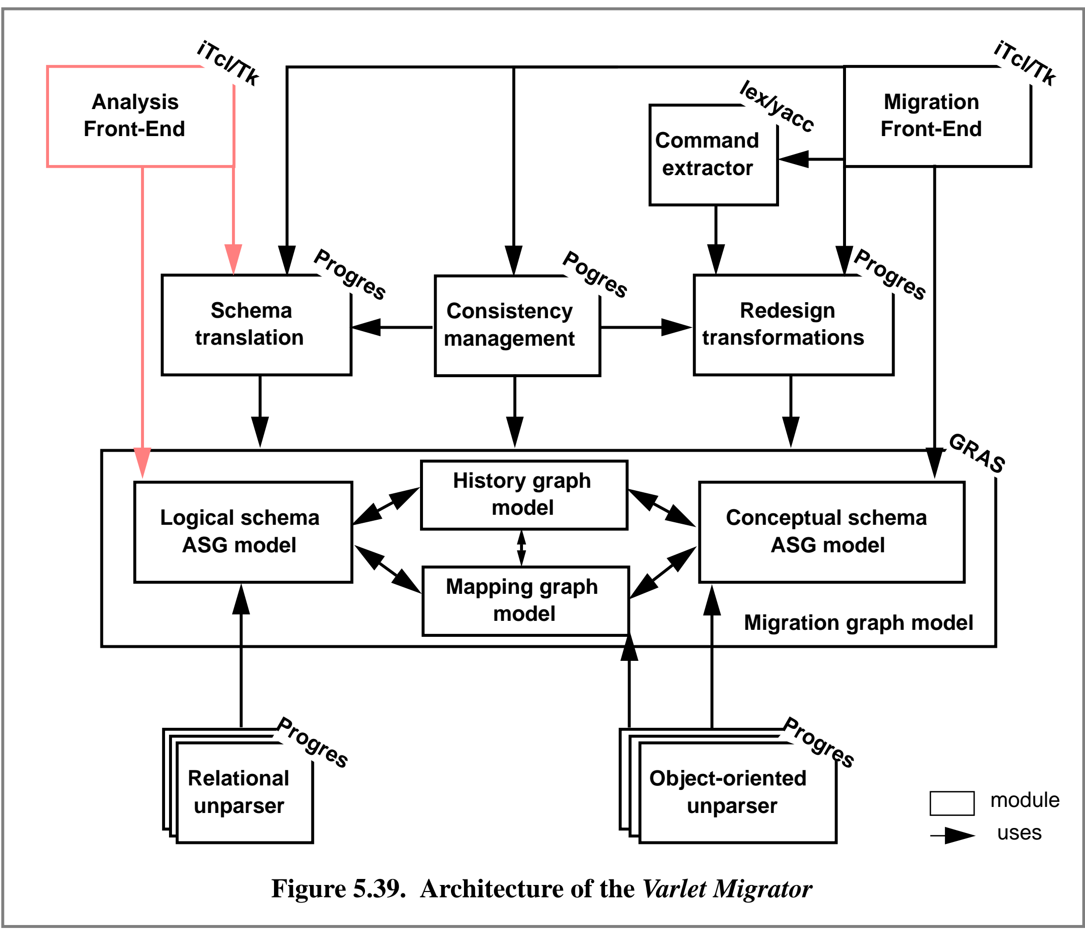


Figure 5.39. Architecture of the *Varlet Migrator*

a More precisely, the *Varlet Analyst* is based on an extended version of the logical schema ASG model depicted in Figure 5.2 which allows to represent certainty measures for constraints like keys, INDs, etc.

The internal functionality of the *Varlet Migrator* is entirely implemented in *Progres*. Module *Schema Translation* implements the triple graph grammar based bidirectional schema mapping mechanism described in Section 5.2. This module and the compiler that derives conventional *Progres* productions from triple graph grammar rules is described by Holle [Hol97]. The change propagation mechanism described in the previous section is implemented in module *Consistency Management*. Module *Redesign Transformations* implements the extensible catalog of primitive and complex redesign transformations discussed in Section 5.3. The *Progres* development environment [SWZ95] provides a visual editor for graph productions and transactions which has proven very useful to add redesign transformations to our catalog.

Figure 5.40 depicts a screenshot of the *Progres* editor that shows an implementation of a new redesign transformation (*MergeParallelAssociations*) in order to deal with the optimization structure detected in our case study (cf. page 21). Such specific transformation can be added "*on-the-fly*" to the catalog of available transformations. However, one problem that remains is that in many cases the resulting idiosyncratic schema dependencies cannot be represented by the SMG model. This issue does not affect the conceptual schema migration process but it disables the generation of data integration middleware components for these idiosyncratic parts of the schema. In these cases the reengineer has the choice of extending the SMG model or implementing the data integration components for these optimization structures manually.

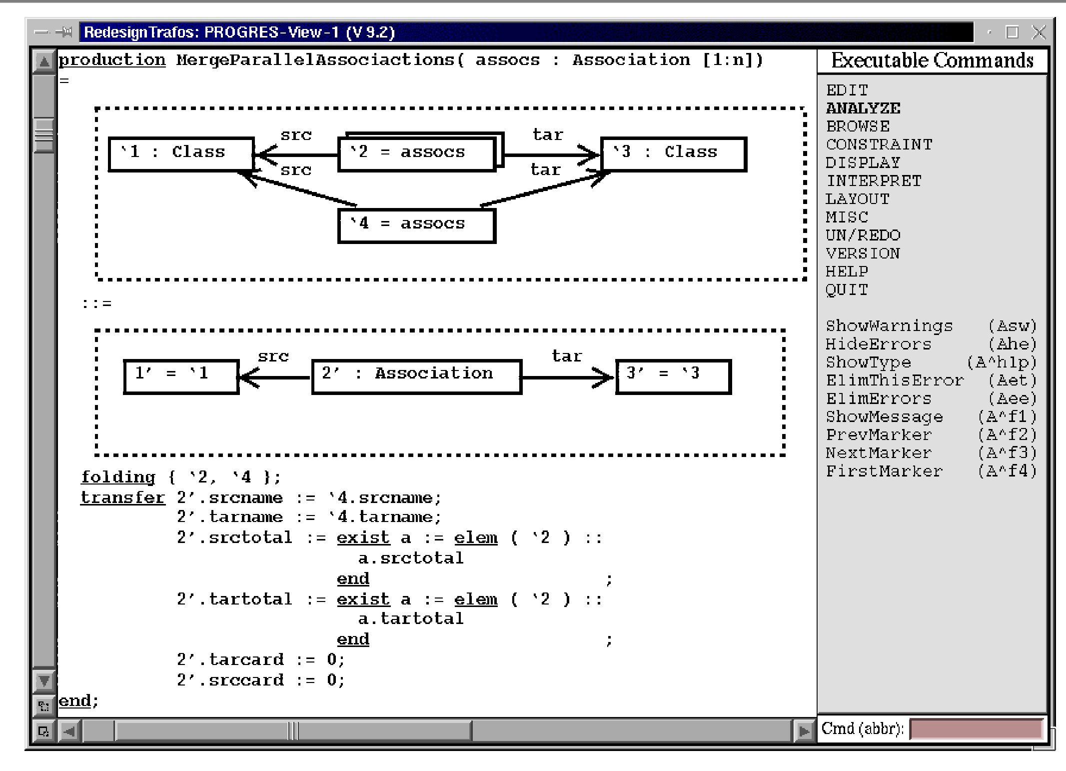


Figure 5.40. **Using the *Progres* environment to extend module *Redesign Transformation***

command extraction

Obviously, whenever new redesign transformations have been added, they should be made available at the user interface (the so-called *Migration Front-End*). In order to avoid manual changes to the *Migration Front-End* due to changes in the transformation catalog, we have implemented a generic command generation mechanism that parses module *Redesign transformations* and extracts signatures for all implemented transformations. These signatures are stored in a text file which is read during start-up of the *Migration Front-End* to build the list of available commands. However, a problem of this generic solution is that the generated list of menu commands soon becomes rather huge and confusing to the user. We solved this problem by offering *context sensitive* menus: whenever the user has selected a number of schema artifacts on the screen we exploit the extracted information about the signatures of commands to offer only those commands which accept the selected artifacts as parameters.

textual unparsers

The *Varlet Migrator* includes several unparsers to generate textual representations of different parts of the migration graph. We have implemented unparsers for language standards like SQL [BED94] and ODL [CBB^{+}97] as well as for proprietary formats like object-oriented schema descriptions for O_2 [LR89] and *ObjectDRIVER* [CER99]. The extraction of textual schema descriptions from the migration graph is performed by traversing and unparsing the ASGs for the logical and the conceptual schema, respectively. For this purpose, we employ a *Progres* mechanism called *derived attributes* [Tea99] which is similar to the well-known *semantic rules* in *attribute grammars* [Knu68, Kas80]. The concrete implementation of the derived attributes for the textual schema descriptions is given by Holle [Hol97].

5.5.2 User interface

Let us revisit the schema migration sample scenario from Section 2.4.2, on page 24 to illustrate the user interface of the *Varlet Migrator*. This scenario deals with two iterations among legacy schema analysis and conceptual schema migration activities. The top section of Figure 5.41 shows the logical schema that is the result of the first analysis activity. This schema contains our familiar excerpt from the PDIS case study shown in Figure 2.7.

initial translation

When the user invokes the *Varlet Migrator* for the first time the current logical schema is translated into an initial conceptual schema. This is performed according to the translation algorithm *MapSchema* (cf. Figure 5.10 on page 126). The screenshot of the *Migration Front-End* in the bottom section of Figure 5.41 shows that the product of this initial conceptual translation still looks similar to the logical schema: basically, each table has been mapped to a class and each foreign key has been mapped to an association with corresponding cardinality constraints.

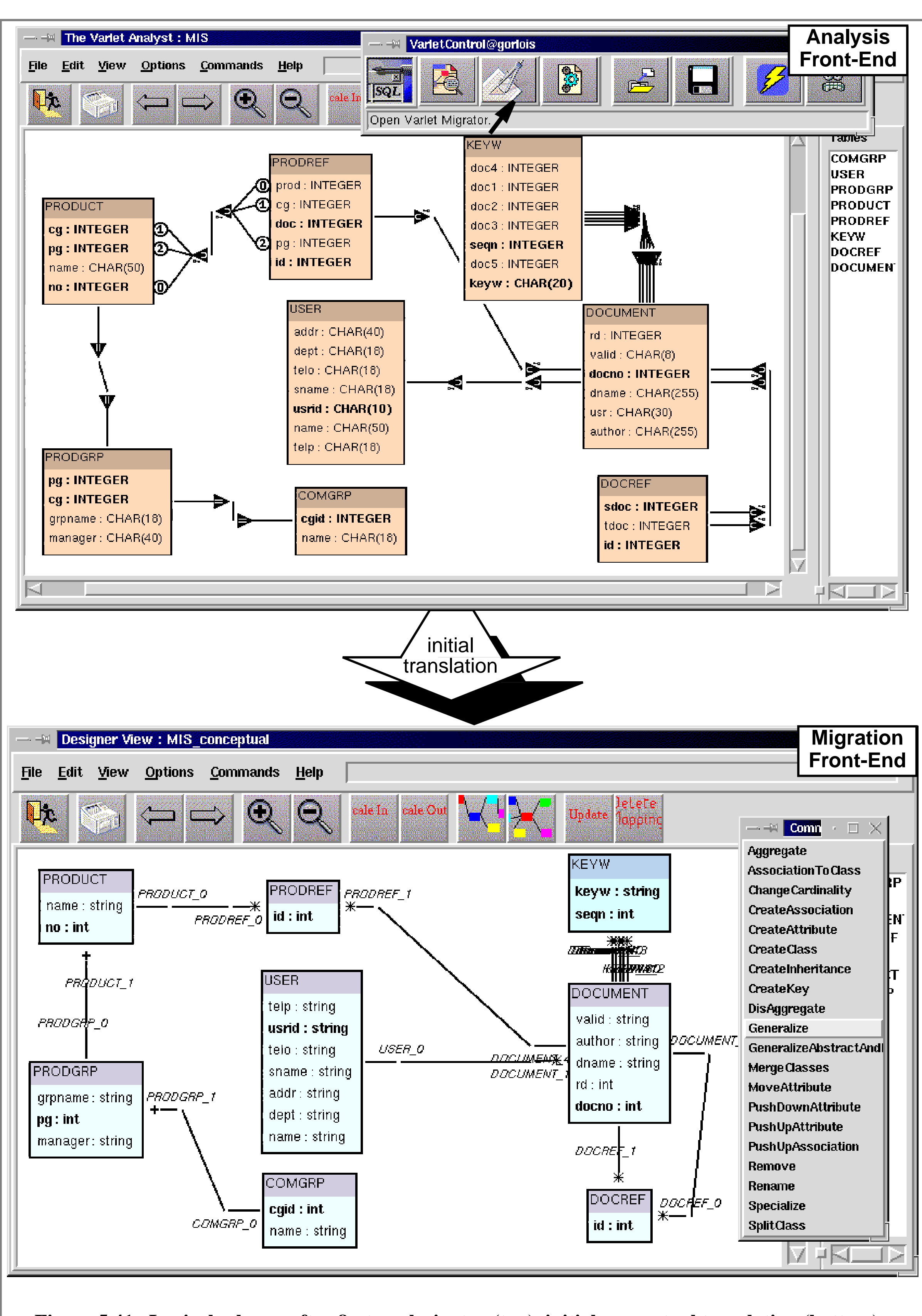


Figure 5.41. Logical schema after first analysis step (top), initial conceptual translation (bottom)

conceptual redesign

Now, the reengineer can use the catalog of available transformations to redesign and extend the conceptual schema according to the new requirements. In our sample scenario in Chapter 2, the reengineer extended the schema by additional classes and associations to store information about customers and on-line documents (cf. Figure 2.18 on page 26). Figure 5.42 illustrates how the *Varlet Migrator* is used to perform these schema modifications. In this picture, we use grey arrows to indicate some of the redesign transformations performed to the conceptual schema. The dialog box entitled *Execute Command* shows that the reengineer is about to transform class *DOCREF* into a *many-to-many* association. Note, that in contrast to our sample scenario (Figure 2.18) our conceptual data model is restricted to unordered associations only (cf. page 116).

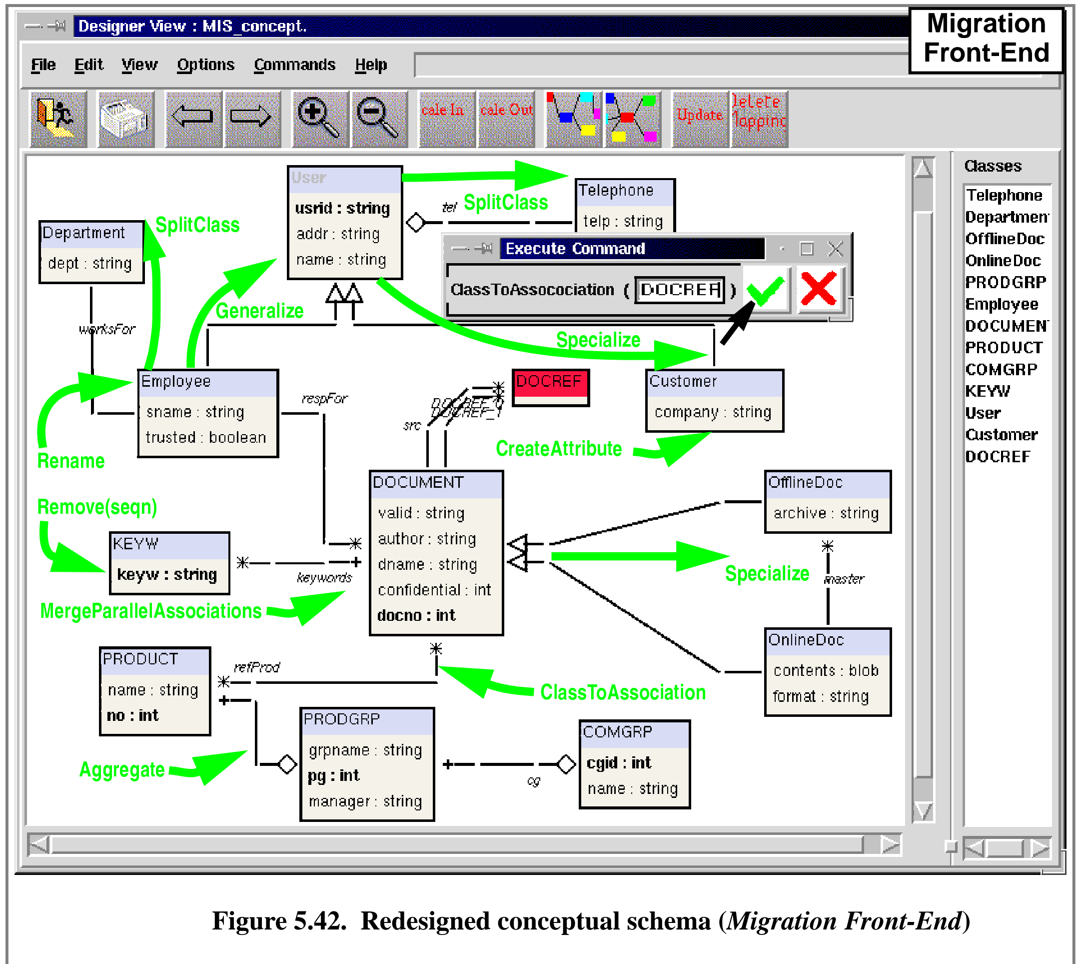


Figure 5.42. Redesigned conceptual schema (*Migration Front-End*)

iteration

In our sample scenario, we assumed that by talking to operators and investigating legacy data, the reengineer detects four different variants in table *PRODREF*. Moreover, (s)he finds out that column *manager* in table *PRODGRP* represents a foreign key referencing an alternative key (*sname*) of table *USER* (cf. page 22). Using the *Varlet Analyst* (s)he can add this information to the logical schema of PDIS. In the top part of Figure 5.43, we used ovals to mark the differences between the completed logical schema and the first analysis result in Figure 5.41. Note, that the reengineer used the filter mechanisms provided by the *Analysis Front-End* to hide columns *doc2,..,doc5* of the optimization structure in table *KEYW*.

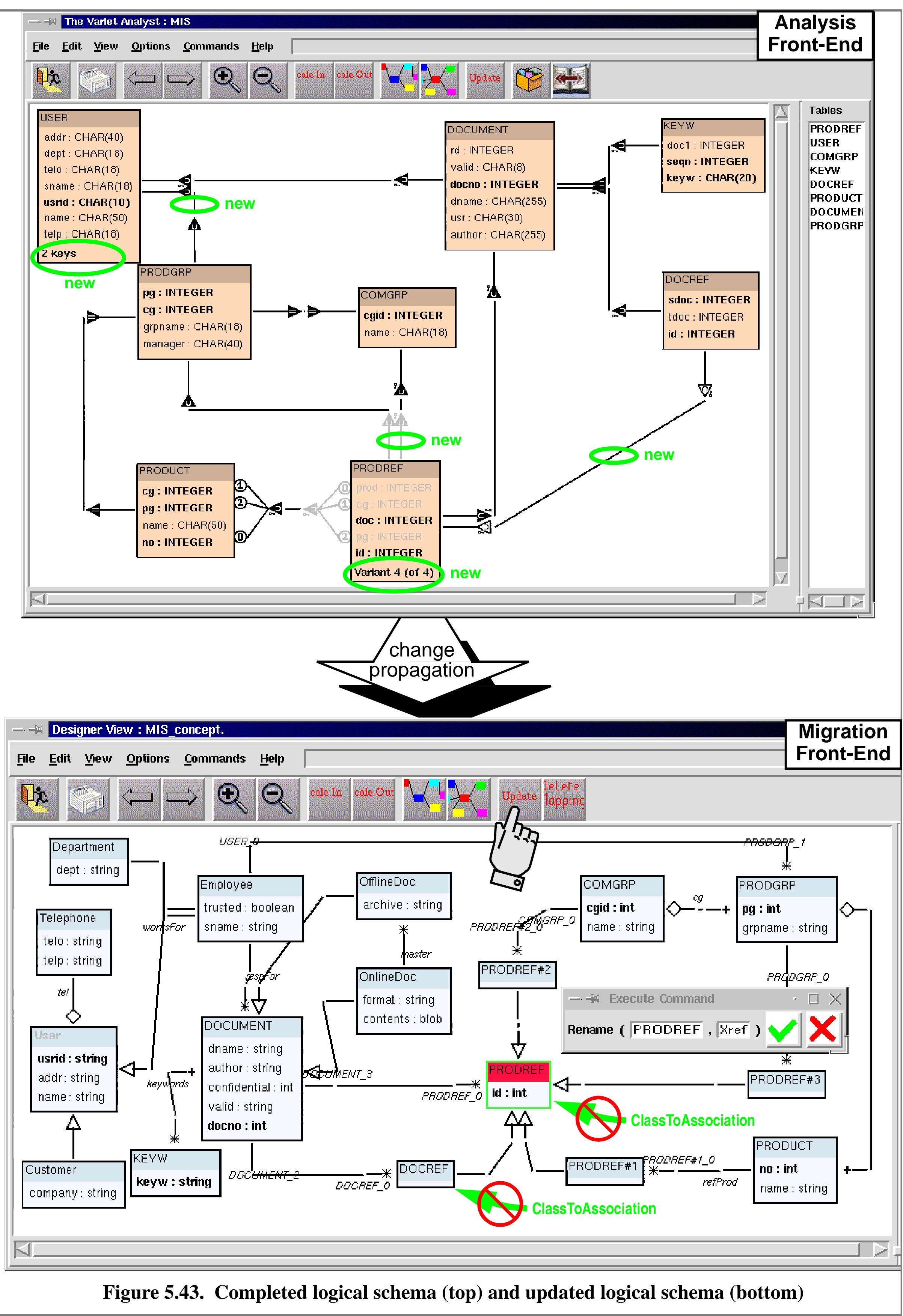


Figure 5.43. Completed logical schema (top) and updated logical schema (bottom)

change propagation

After modifying the logical schema, the reengineer uses the incremental consistency management mechanism described in Section 5.4 to propagate the changes into the redesigned conceptual schema. In *Varlet*, this is done by pressing the *Update* button in the *Migration Front-End*. The bottom section of Figure 5.43 shows that the four variants in table *PRODREF* have been mapped to an inheritance structure with superclass *PRODREF* and three subclasses *PRODREF#1-3*. It is the task of the reengineer to determine reasonable names for these classes. For example, (s)he might rename the superclass to *XRef* and the subclasses to *ProdRef*, *ProdGrpRef*, and *ComGrpRef* like in Figure 2.19. In addition, the updated conceptual schema contains several other changes:

- class *DOCREF* represents a further subclass of class *PRODREF* because of the I-IND between the logical representations of these two classes,
- attribute *manager* has been removed from class *PRODGRP* because it only represents a borrowed key in the logical schema,
- there is a new *one-to-many* association among classes *PRODGRP* and *Employee* because of the newly detected foreign key manager in table *PRODGRP*.

Most applied redesign transformations are still valid in the updated conceptual schema. Still, Figure 5.43 shows that the two applications of transformation *ClassToAssociation* to classes *DOCREF* and *PRODREF* have been undone. This is because their application condition is violated for classes that participate in inheritance hierarchies (cf. Figure B.5 on page 199). Note that the grey arrows which indicate the cancelled transformations do not (yet) belong to the user interface of the *Varlet Migrator*. Still, our tool provides the user with a textual update report including information about all cancelled transformations.

implementation of extensions

During the conceptual migration activity, the reengineer has made several modifications which extend the information capacity of the original PDIS schema, e.g., (s)he added new subclasses, class attributes, and associations. These changes do not have to be implemented manually but the schema mapping mechanism described in Section 5.2 can be used to extend the logical schema, automatically. For this purpose, the *Analysis Front-End* contains an *Update* button similar to the *Migration Front-End*. Figure 5.44 shows the result of this logical schema update. As specified in mapping rule *MapVariantToConcreteClass* (on page 127), the new classes have been mapped to new variants in tables *USER* and *DOCUMENT*. All new attributes have been mapped to columns in these tables and the association *master* among on-line and off-line documents has been mapped to a cyclic foreign key *master* in table *DOCUMENT*. The SQL unparser allows the reengineer to retrieve a textual representation of the schema modifications which can be used to update the LDB schema catalog.

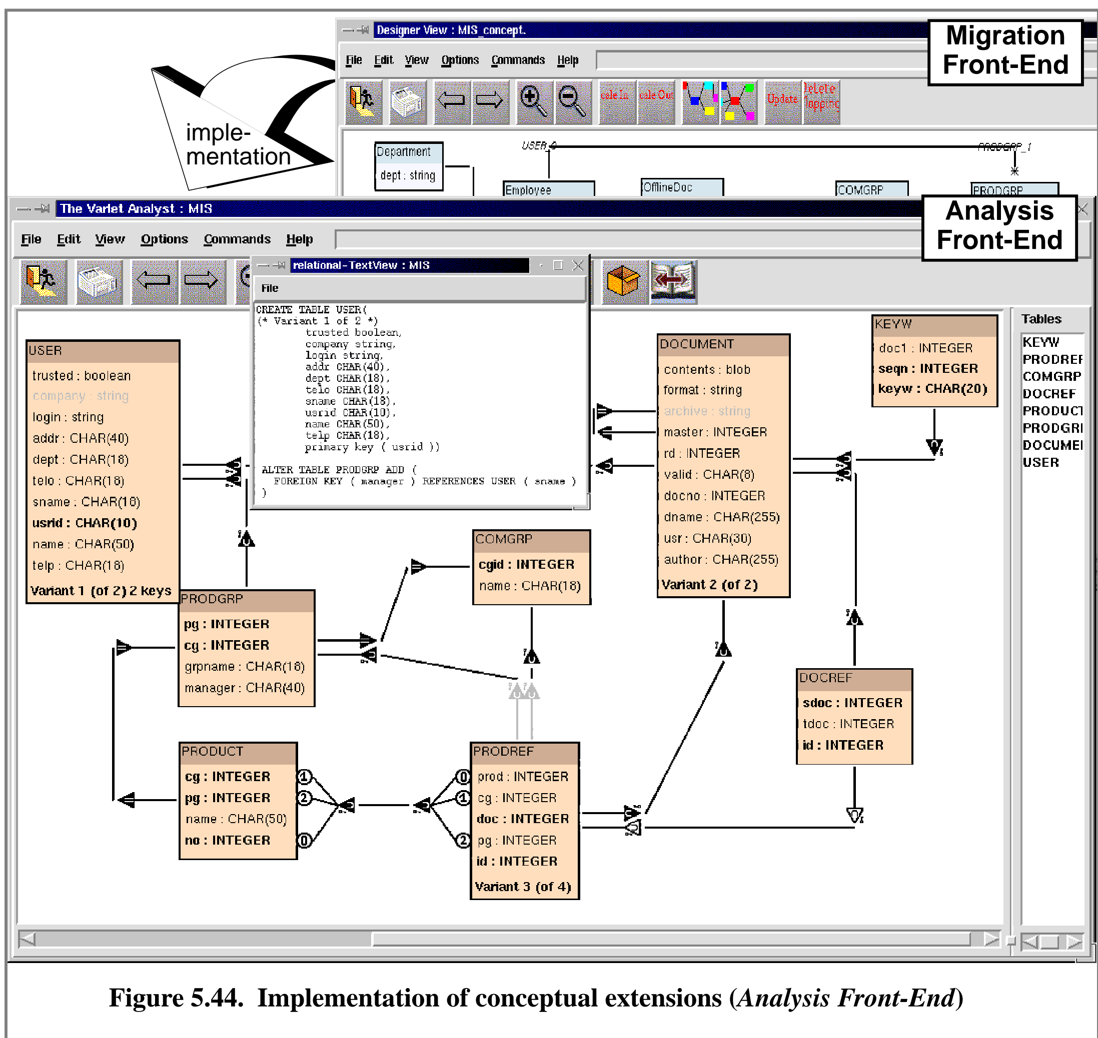


Figure 5.44. Implementation of conceptual extensions (*Analysis Front-End*)

5.6 Data integration

In general, the output of a conceptual schema migration activity is an abstract design document for an LDB schema. This documentation facilitates understanding, assessment, and maintenance of the LDB. The techniques described in the previous sections allow to integrate schema migration and maintenance activities in an evolutionary and intertwined process. This helps to solve the well-known problem of keeping the conceptual design up-to-date and consistent with the current implementation. The conceptual design gains even greater importance in DBRE projects that aim on migrating LDB applications to new technologies, programming languages, or architectures. Object-oriented technology is a common standard for the development of modern cooperative information system infrastructures [Vin97, CBB+97]. In such projects, the conceptual design is not only used as abstract documentation but also as an object-oriented view to access the information maintained in the LDB. Such object-oriented access layers allow to create unified views on heterogeneous component databases and abstract from low-level implementation details like idiosyncratic data formats and optimization constructs. By encapsulating the concrete structure of the LDB, they improve

the robustness of the entire information system infrastructure w.r.t. the evolution of single component schemas. Several so-called *middleware* components and libraries have been developed to facilitate the development of object-oriented access layers, e.g., [CER99, Obj99b, Hüs97, ONT96, Rad95]. Most approaches employ proprietary programming languages and APIs to specify the dependencies among the component schemas and their object-oriented representations [CER99, Obj99b, Hüs97, Rad95] while other products provide menu-driven dialog interfaces [Obj99b, ONT96]. However, the problem that prevails with these approaches is that the reengineer has to specify and maintain these dependencies manually.

ObjectDRIVER

The integrated approach to schema migration developed in this dissertation allows to overcome this problem. The correspondences implicitly stored in the *migration graph* during the schema migration process enable automatic generation of the dependency information necessary for middleware components. We have chosen the middleware product *ObjectDRIVER* [CER99] which has been developed by the CERMICS Database Team at Sophia Antipolis Cedex, France, to evaluate this approach. *ObjectDRIVER* provides seamless integration of object-oriented applications written in *Java* or *C++* with legacy data sources (cf. Figure 5.45).[a] It allows to create a ODMG-compliant [CBB+97] object-oriented interface that hides the concrete database implementation. An *OQL* (*Object Query Language*) [CBB+97] interpreter supports the formulation of adhoc-queries based on the abstract, object-oriented schema.

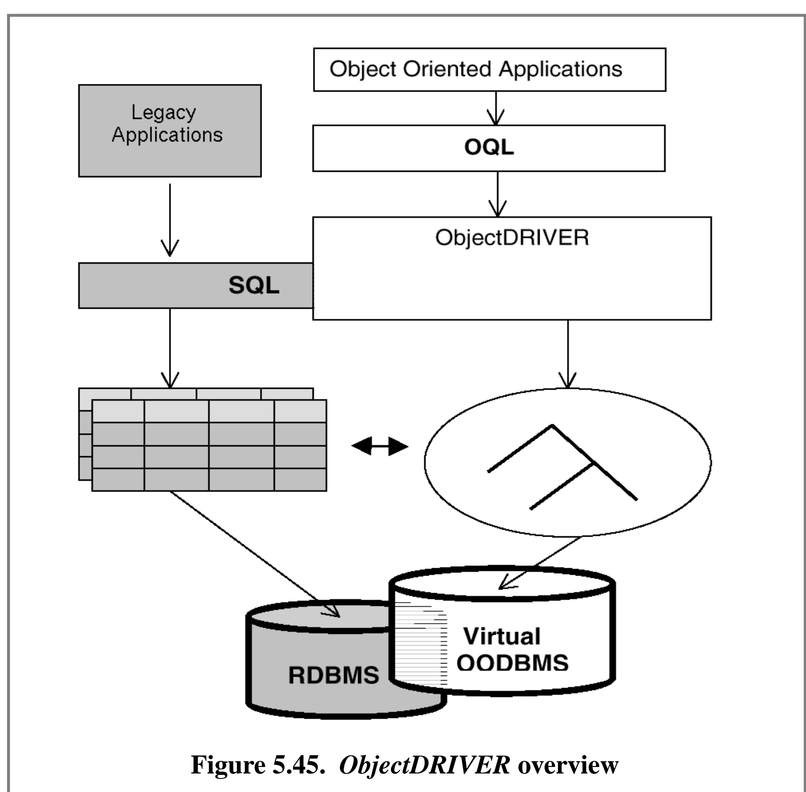


Figure 5.45. ***ObjectDRIVER*** **overview**

a Figure 5.45 has been adopted from [CER99] under permission of the CERMICS Database Team.

Integration of ObjectDRIVER and Varlet

The integration of the *ObjectDRIVER* middleware with the *Varlet* schema migration environment is illustrated in Figure 5.46. Based on the information maintained in the migration graph, *Varlet* generates textual descriptions for both schemas and their interdependencies which are required as the input for *ObjectDRIVER*. In the following, we will use our DBRE sample scenario to exemplify the structure of these textual descriptions and to describe how the necessary information is extracted from the migration graph.

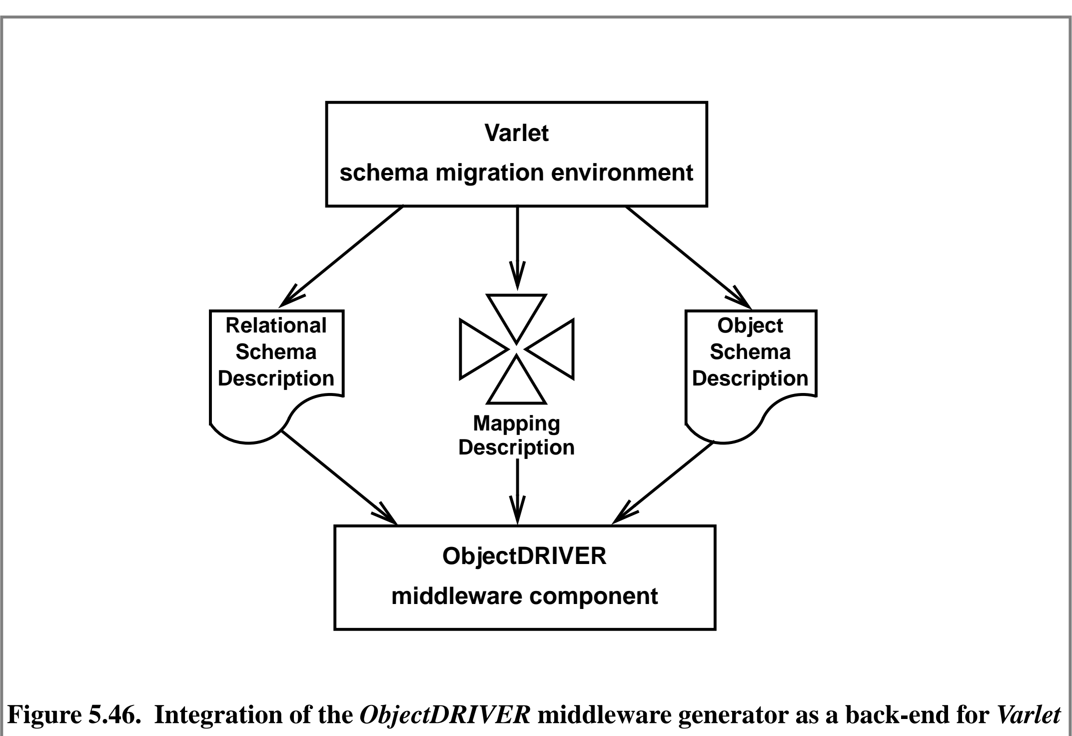


Figure 5.46. Integration of the *ObjectDRIVER* middleware generator as a back-end for *Varlet*

5.6.1 Generating descriptions for relational and object-oriented schemas

The textual schema descriptions for *ObjectDRIVER* have to be in a proprietary format that does not comply to any common standard like SQL DDL [BED94] or ODL [CBB+97]. Still, the format for the relational schema description is similar to data definitions in standard SQL. Figure 5.47 illustrates this format for the eight RS considered in our sample scenario. The specification of a primary key for each RS is mandatory. Those columns of an RS which belong to such a key are marked by the suffix *keyPart*. Note, that we had to eliminate the optimization structure in table *KEYW* because the mapping mechanism provided by *ObjectDRIVER* lacks the necessary flexibility to access it (cf. page 21). An alternative solution that allows to keep (more important) optimization structures is to disregard the corresponding RS during the middleware generation and program the necessary data access functionality manually afterwards.

```
define schema MIS {
    relationalDbms DB2;

define table COMGRP {
    cgid        integer keyPart,
    name        string(18)
};
define table PRODGRP {
    cg          integer keyPart,
    manager     string(40),
    pg          integer keyPart,
    grpname     string(18)
};
define table PRODREF {
    id      integer keyPart,
    pg      integer,
    prod    integer,
    cg      integer,
    doc     integer keyPart
};

define table DOCUMENT {
    docno       integer keyPart,
    dname       string(255),
    valid       string(8),
    rd          integer,
    archive     string(80),
    master      integer,
    author      string(255),
    usr         string(30),
    format      integer,
    contents    octet
};
define table DOCREF {
    id      integer keyPart,
    sdoc    integer keyPart,
    tdoc    integer
};
define table PRODUCT {
    name    string(50),
    no      integer keyPart,
    pg      integer keyPart,
    cg      integer keyPart
};

define table USER {
    usrid       string(10) keyPart,
    name        string(50),
    login       string(10),
    trusted     boolean,
    dpt         string(18),
    company     string(255),
    sname       string(18),
    addr        string(40),
    telo        string(18),
    telp        string(18)
};
define table KEYW {
    keyw        string keyPart,
    doc         integer
};
};
```

Figure 5.47. Relational schema description for *ObjectDRIVER*

Figure 5.48 presents the *ObjectDRIVER* schema description for the conceptual view on the MIS sample schema. The notation is very similar to schema definitions for the purely object-oriented database O_2 [O2 93]. Associations among classes can be implemented either as single references or as pairs of references using the keyword *inverse* to specify their correspondences. As described in Section 5.5.1, we employ derived text attributes to extract both textual schema descriptions from the *Varlet* migration graph. In contrast to the generation of the schema mapping description, this unparsing mechanism is simple and straight forward because we only need to consider the syntactical structure of the logical and the conceptual representation.

5.6.2 Generating object-relational mapping descriptions

The schema mapping description for *ObjectDRIVER* is not represented in a separate file but it is an extension of the object-oriented schema description by additional mapping directives. The *Varlet* schema mapping graph stores the information needed to generate these mapping directives (cf. Section 5.1.2, on page 119). Analogously to the generation of schema descriptions, we use derived text attributes to unparse this textual information. However, the derivation rules of such text attributes are less suited to facilitate understanding of our approach, because they include many conditionals. Therefore, in the following, we specify the extraction of the mapping description with *Progres* graph tests (cf. page 118) and we use our DBRE case study to exemplify the generation of each different mapping construct for *ObjectDRIVER*.

```
class User
type Tuple (
    name        String,
    login       String,
    addr        String,
    telephone   Telephone
)
class Employee inherit User
type Tuple (
    shortName   String,
    trusted     boolean,
    worksFor    Department
)
class Customer inherit User
type Tuple (
    company     String
)
class Telephone
type Tuple (
    office      String,
    private     String
)
class XRef
type Tuple (
    no          integer,
    refBy       Document
)
class DocRef inherit XRef
type Tuple (
    ref         Document
)
class Department
type Tuple (
    deptName    String
)

class ComGrpRef inherit XRef
type Tuple (
    ref         CommodityGroup
)
class ProdGrpRef inherit XRef
type Tuple (
    ref         ProductGroup
)
class ProdRef inherit XRef
type Tuple (
    ref         Product
)
class Document
type Tuple (
    title       String,
    number      integer,
    validUntil  String,
    author      String,
    confidential integer,
    respEmp     Employee,
    xrefs       Set(XRef)
            inverse XRef.refBy,
    refBy       Set(DocRef)
            inverse DocRef.ref,
    keyword     Set(Keyword)
            inverse Keyword.docs
)
class OnlineDocument inherit Document
type Tuple (
    contents    octet,
    format      integer
)

class OfflineDocument inherit Document
type Tuple (
    archive     String
)
class CommodityGroup
type Tuple (
    name        String,
    id          integer
    prodGrps    Set(ProductGroup)
                inverse
ProductGroup.comGrp
)
class ProductGroup
type Tuple (
    name        String,
    id          integer,
    comGrp      CommodityGroup,
    manager     Employee,
)
class Product
type Tuple (
    name        String,
    number      integer,
    prodGrp     ProductGroup
)
class Keyword
type Tuple (
    keyw        String,
    docs        Set(Document)
            inverse Document.keywords
)
```

Figure 5.48. Object schema description for *ObjectDRIVER*

classes and subclasses

Classes without a generalization are mapped to so-called *base tables*. In *ObjectDRIVER*, this mapping is defined by the class name followed by the keyword *on* and the name of the base table (cf. class *XRef* in Figure 5.49). Subclasses are automatically mapped to the same base table like their generalizations. Textual constraints are used to specify which database entries qualify as valid members of a certain subclass. Figure 5.49 illustrates this concept for the four subclasses of class *XRef*. These subclasses logically correspond to the four different variants of entries in table *PRODREF* (cf. page 20). For example the constraint for class *PrdGrpRef* in Figure 5.49 specifies that only those tuples with a null-value in column *prod* but with valid values in columns *cg* and *pg* represent product group references.

```
class XRef on PRODREF
type Tuple
  (
  no        integer,
  refBy     Document
)
class DocRef inherit XRef
type Tuple (
  ref       Document,
  constrainedBy((PRODREF.pg == NULL)
            && (PRODREF.prod == NULL)
            && (PRODREF.cg == NULL))
)
class ComGrpRef inherit XRef
type Tuple (
  ref       CommodityGroup,
  constrainedBy((PRODREF.cg != NULL)
            && (PRODREF.pg == NULL)
            && (PRODREF.prod == NULL)
)

class ProdGrpRef inherit XRef
type Tuple (
  ref       ProductGroup,
  constrainedBy((PRODREF.cg != NULL)
            && (PRODREF.pg != NULL)
            && (PRODREF.prod == NULL)
)
class ProdRef inherit XRef
type Tuple (
  ref       Product,
  constrainedBy((PRODREF.cg != NULL)
            && (PRODREF.pg != NULL)
            && (PRODREF.prod != NULL)
)
```

Figure 5.49. Mapping description for classes and subclasses

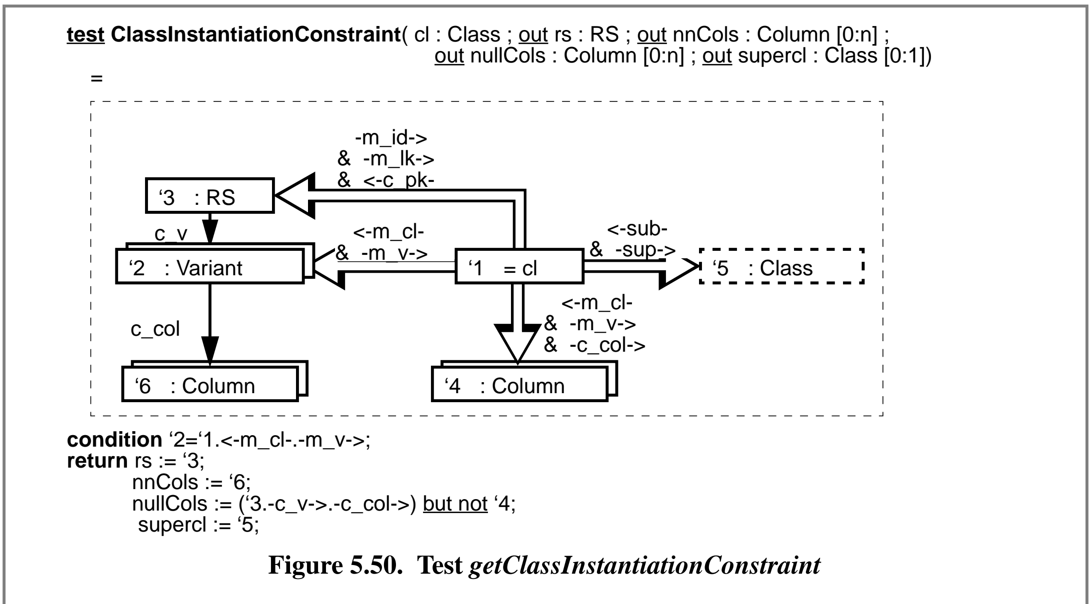


Figure 5.50. Test *getClassInstantiationConstraint*

Figure 5.50 shows the graph test that can iteratively be called for each class *cl* to extract the necessary information from the migration graph. If *cl* has a generalization it is matched to the optional node *'5* and returned in parameter *supercl*. The base table is identified as node '3 by traversing the *m_id* edge and the *m_lk* to the primary key of the corresponding RS in the logical schema (cf. Figure 5.2 on page 117). All variants that have been mapped to *cl* are collected in node set '2. Node set '6 represents all columns that are common to the variants in node set '2. These columns have to carry valid values (not null) in order to qualify for instances of class *cl*. On the other hand, the set of columns that *have* to carry null values for instances of *cl* is returned in parameter *nullCols*. This set is defined by all columns of the base table minus all columns that are includes by any variant mapped to *cl* (node set '4).

base table attributes

Mappings of attributes which correspond to columns in the base table are described by simply adding the key word *on* followed by the qualified name of corresponding columns (cf. Figure 5.51). The graph test to check the validity of this mapping for each attribute *attr* is presented in Figure 5.52. Node ‘5 represents a negative application condition which ensures that *attr* is not mapped over a foreign key to a column in a different table. If this condition is fulfilled the corresponding column in the base table is returned in parameter *col*.

```
class Document on DOCUMENT
type Tuple (
  title        String
    on DOCUMENT.dname,
  number  integer
    on DOCUMENT.docno,
  validUntil String
    on DOCUMENT.valid,
  author    String
    on DOCUMENT.author,
  confidential integer
    on DOCUMENT.rd,
...
)
```

Figure 5.51. Mapping description for base table attributes

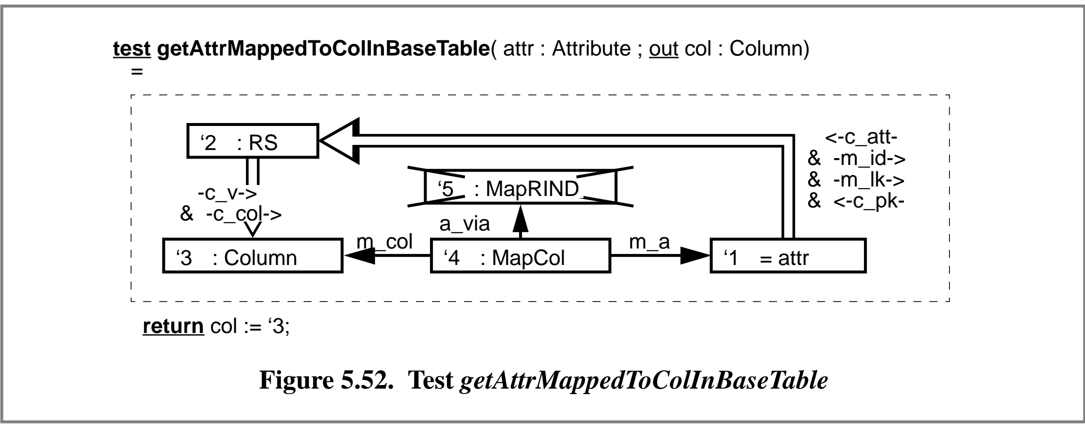


Figure 5.52. Test *getAttrMappedToColInBaseTable*

remote attributes

If a class contains attributes which belong to (*remote*) tables different from the corresponding base table these tables have to be joined. So far, our sample scenario does not include such a situation. However, let us assume a situation where MIS users who are managers have been mapped to a specialization of class *Employee* named *Manager*. Furthermore, let us assume that managers have an additional attribute *secretariate* which has been relocated via association manager to class *ProductGroup* using the conceptual redesign transformations *MoveAttribute* and *RenameAttribute* (cf. Section 5.3.2, on page 138). This scenario is illustrated on the left-hand side of Figure 5.53. Its right-hand side contains the corresponding mapping description for *ObjectDRIVER*. It shows that the relocated and renamed attribute *contactInfo* of class *ProductGroup* is mapped to column *secretariate* of table *USER*. Both tables are joined over the foreign key that is the logical representation of association *manager*.

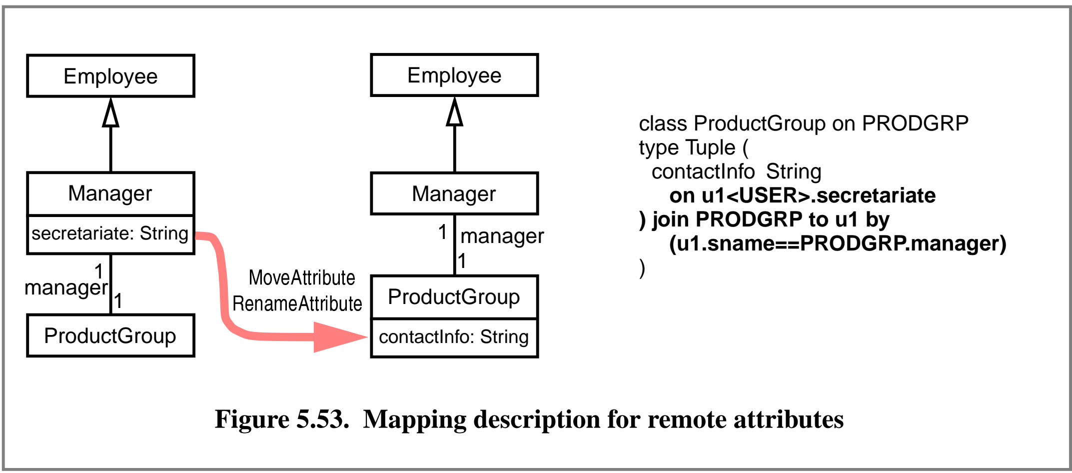


Figure 5.53. Mapping description for remote attributes

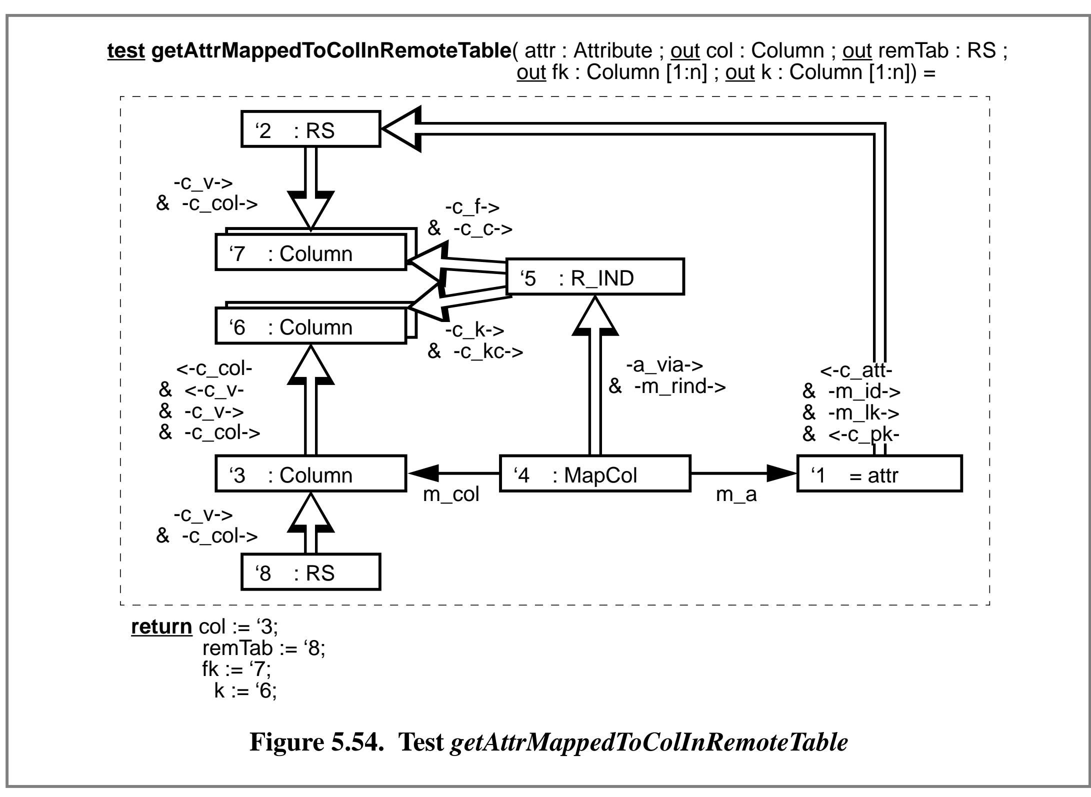


Figure 5.54. Test *getAttrMappedToColInRemoteTable*

The graph test that specifies the extraction of the information needed to generate the mapping description for each remote attribute *attr* is presented in Figure 5.54. The foreign key that is used for the join is matched to node ‘5 by traversing edges *a_via* and *m_rind* from the column mapping node ‘4. The remote table itself is represented by node ‘8 and returned in output parameter *remTab*, while the join columns in both tables are returned in parameters *k* and *fk*, respectively. Note, that generally it is also possible to relocate an attribute over more than one association. In this case, more than one *R-IND* node can be matched to node ‘5 and several joins have to be generated for the *ObjectDRIVER* mapping description. This situation cannot be specified with one single graph test but additional control structures are necessary to extract this information.

base table relationships

Relationships that have been created by splitting classes in the conceptual schema are represented in the *ObjectDRIVER* mapping description as a cyclic join over the key column(s) of the corresponding base table. In our case study, we have split class *Employee* in two classes *Employee* and *Department* with an association *worksFor* (cf. Figure 5.42 on page 161). The generated mapping information for reference *worksFor* in class *Employee* is given in Figure 5.55.

```
class Employee inherit User
type Tuple (
  shortName String
    constrainedBy(USER.sname != NULL),
  trusted   boolean
    on USER.trusted,
  worksFor Department
    on u1<USER>.usrid
) join USER to u1 by (u1.usrid == USER.usrid)
```

Figure 5.55. Mapping description for base table relationships

The graph test in Figure 5.56 specifies that the given relationship *rel* may not be mapped to foreign keys and that its source and target class have to be mapped to the same base table (node ‘4). If the test succeeds it returns the set of all primary key columns in output parameter *k*.

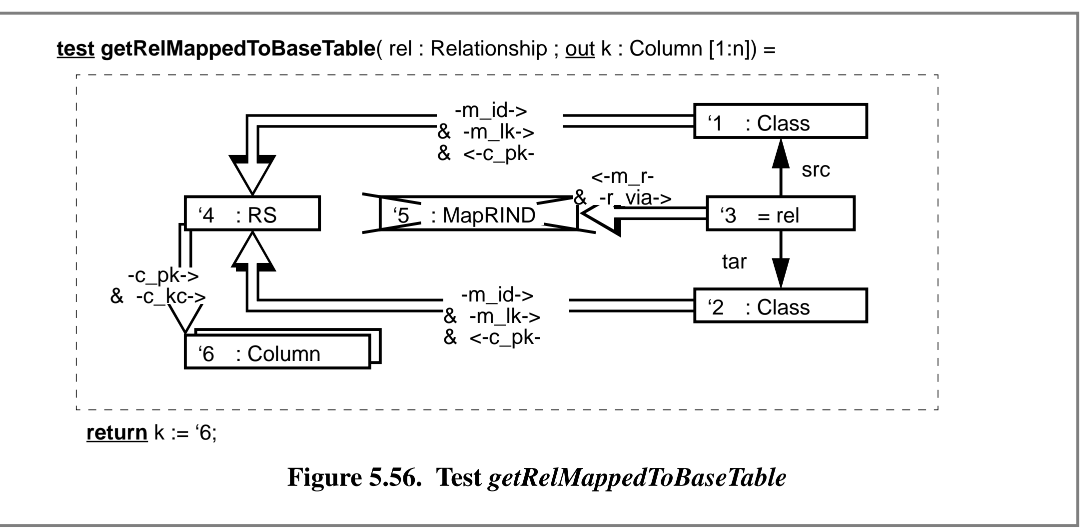


Figure 5.56. Test *getRelMappedToBaseTable*

remote relationships

Similar to the mapping description for remote attributes, we have to add joins to the mapping description of relationships if they have been mapped to foreign keys in the migration graph. Depending on the cardinalities of such *remote* relationships the corresponding references in the participating source and target classes are either declared to be set-valued or single valued. Further cardinality constraints like specific limits and totality constraints have to be checked in the application code which can also be generated. As an example, Figure 5.57 shows the mapping description for the *one-to-many* association *referencedBy* among classes *Document* and *XRef* (cf. Figure 2.19 on page 27). This association is represented as a set-valued reference *xrefs* in class *Document* which is inverse to a single valued reference *refBy* in class *XRef*.

```
class Document on DOCUMENT
type Tuple (
  xrefs     Set on x1<XRef> (
            aRef XRef
                 on PRODREF.id
            join DOCUMENT to x1
            by(DOCUMENT.docno == x1.doc)
    ) inverse XRef.refBy
...)
```

```
class XRef on PRODREF
type Tuple
  (
  ...
  refBy     Document
     on d1<DOCUMENT>.docno
) join PRODREF to d1
                 by(PRODREF.doc == d1.docno)
```

Figure 5.57. Mapping description for remote relationships

Obviously, different graph tests are needed to check for the various possible cardinalities of relationships. With respect to our example, Figure 5.58 shows the graph test that validates a *one-to-many* relationship *rel* and retrieves the necessary information about the foreign key which is mapped to *rel*. The defined folding clause is necessary to allow for cyclic joins, i.e., that *rel* has the same class as its source and target.

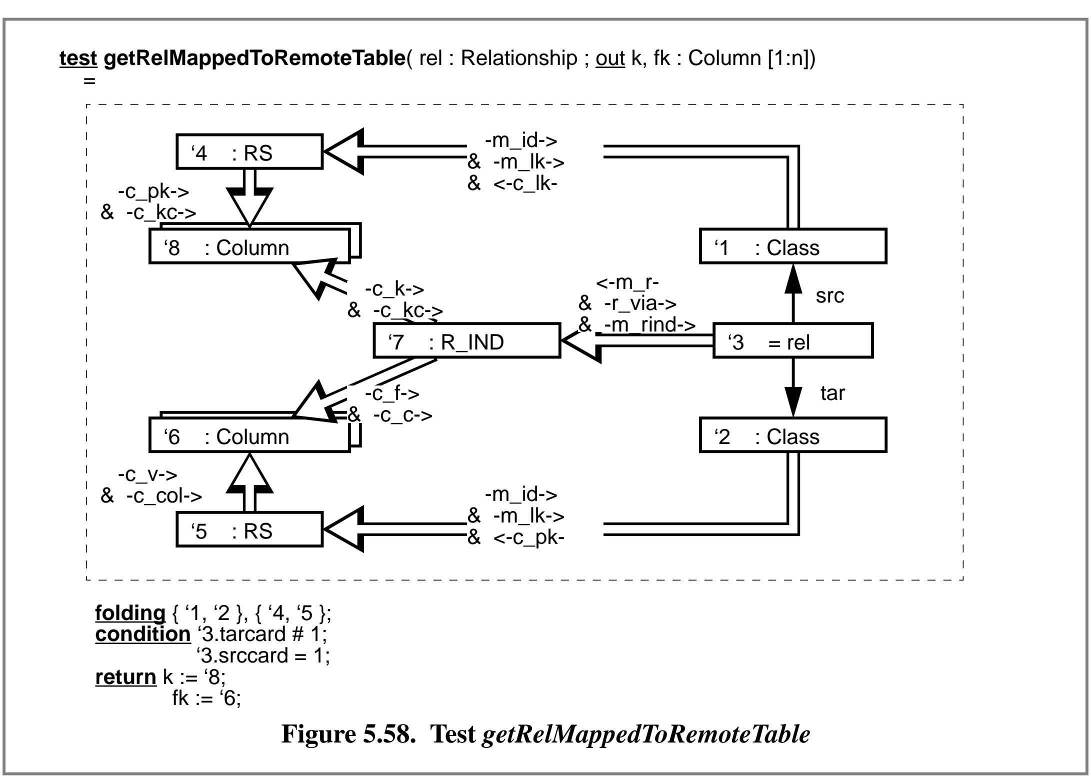


Figure 5.58. Test *getRelMappedToRemoteTable*

IND-based inheritance

Finally, we have to specify how inheritance relationships that have been mapped to inclusion dependencies (I-INDs) are represented in *ObjectDRIVER* mapping descriptions. Our sample case study includes such a situation for the specialization *DocRef* of class *XRef* (cf. Figure 5.42 on page 161). Figure 5.59 shows that this constellation is represented by adding a join over the foreign key columns between both participating tables in the relational schema. The corresponding graph test in Figure 5.60 is very similar to the previous test in Figure 5.58. The complete *ObjectDRIVER* mapping description for the seventeen classes in our case study is summarized in Figure 5.61.

```
class DocRef inherit XRef
type Tuple (
  ref        Document
    on d1<DOCUMENT>.docno
) join DOCREF to PRODREF
    by((DOCREF.id == PRODREF.id)
    && (DOCREF.sdoc == PRODREF.doc)),
  join d1 to DOCREF
    by(d1.docno=DOCREF.tdoc)
```

Figure 5.59. Mapping description for IND-based inheritance relationships

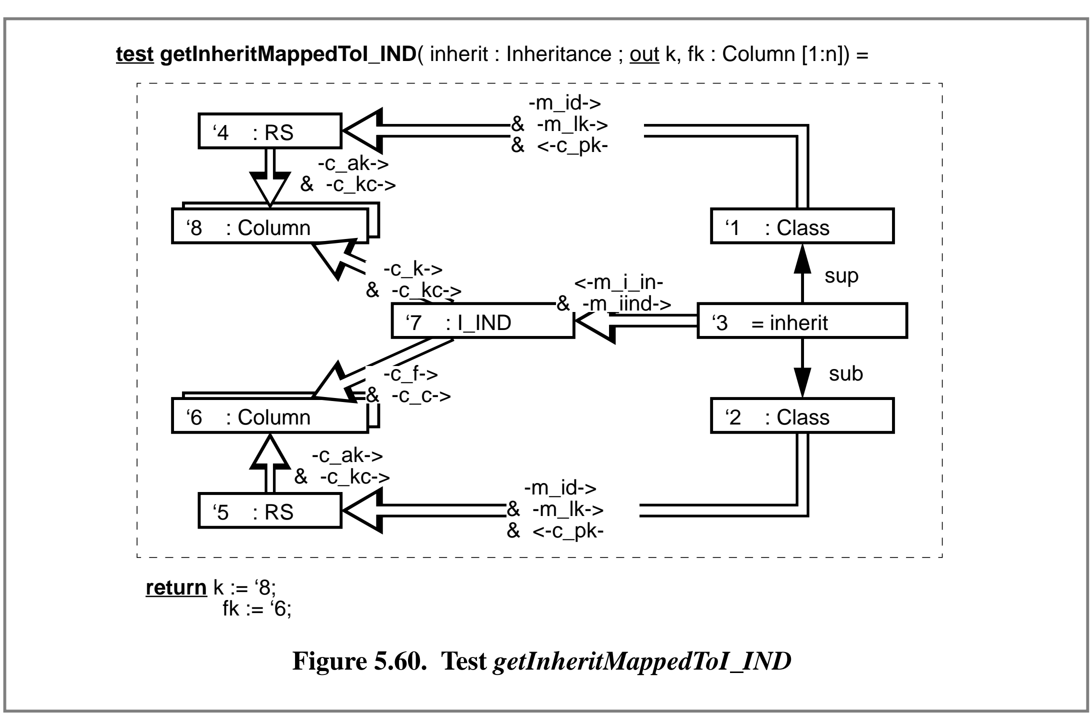


Figure 5.60. Test *getInheritMappedToI_IND*

object-oriented application code

The current version of *ObjectDRIVER* (1.1) does not yet provide further support for the generation of object-oriented application code in *Java* or *C++*. This means that the programmer is responsible to define all application classes with their methods. The *ObjectDRIVER* data integration mechanism requires that all class properties (attributes and references) are accessed exclusively by method calls (*set* and *get* accessor methods). Besides the usual reasons for encapsulation, this is especially important because *ObjectDRIVER* creates run-time objects from persistent (relational) data on demand only, i.e., when the corresponding resource is accessed. This is illustrated by the sample application code for class *Document* in Figure 5.62. The first statement in each accessor method is a call to the predefined method *getObject()* which initiates that the object is filled with the actual data maintained in the relational LDB. Before the first call to *getObject()* the object is represented by a proxy. This lazy data migration strategy is needed to avoid efficiency problems that could otherwise be caused by the eager generation of huge object structures due to a large amount of data. It is possible and desirable to generate application classes with such canonical accessor methods automatically. We have implemented such a generator for a different ODMG middleware [Sch98].

```
class User on USER
type Tuple (
  name        String
     on USER.name,
  login       String
     on USER.login,
  addr        String
     on USER.addr,
  telephone Telepone
     on USER.usrid
)
class Employee inherit User
type Tuple (
  shortName String
     constrainedBy(USER.sname != NULL),
  trusted     boolean
     on USER.trusted,
  worksFor  Department
     on d1<USER>.usrid
) join USER to d1 by (d1.usrid==USER.usrid)

class Customer inherit User
type Tuple (
  company   String
     constrainedBy(USER.company != NULL)
)
class Telephone on USER
type Tuple
  (
  office      String
     on USER.telo,
  private     String
     on USER.telp
)
class XRef on PRODREF
type Tuple
  (
  no          integer
     on PRODREF.id,
  refBy       Document
     on d1<DOCUMENT>.docno
) join PRODREF to d1
              by(PRODREF.doc == d1.docno)

class DocRef inherit XRef
type Tuple (
  ref         Document
     on d1<DOCUMENT>.docno
  constrainedBy((PRODREF.pg == NULL)
              && (PRODREF.prod == NULL)
              && (PRODREF.cg == NULL))
) join DOCREF to PRODREF
  by((DOCREF.id == PRODREF.id)
     && (DOCREF.sdoc == PRODREF.doc)),
  join d1 to DOCREF
     by(d1.docno=DOCREF.tdoc)

class ComGrpRef inherit XRef
type Tuple (
  ref         CommodityGroup
  on c1<COMGRP>.cg
  constrainedBy((PRODREF.cg != NULL)
              && (PRODREF.pg == NULL)
              && (PRODREF.prod == NULL)
) join c1 to PRODREF
     by(c1.cg=PRODREF.cg)

class ProdGrpRef inherit XRef
type Tuple (
  ref         ProductGroup
     on p1<PRODGRP>.pg
  constrainedBy((PRODREF.cg != NULL)
              && (PRODREF.pg != NULL)
              && (PRODREF.prod == NULL)
  )  join p1 to PRODREF
              by((p1.cg == PRODREF.cg)
              && (p1.pg == PRODREF.pg))

class ProdRef inherit XRef
type Tuple (
  ref         Product
     on p1<PRODUCT>.no
  constrainedBy((PRODREF.cg != NULL)
              && (PRODREF.pg != NULL)
              && (PRODREF.prod != NULL)
  )  join p1 to PRODREF
              by((p1.cg == PRODREF.cg)
              && (p1.pg == PRODREF.pg)
              && (p1.no == PRODREF.prod))

class Document on DOCUMENT
type Tuple (
  title       String
     on DOCUMENT.dname,
  number      integer
     on DOCUMENT.docno,
  validUntil  String
     on DOCUMENT.valid,
  author      String
     on DOCUMENT.author,
  confidential integer
     on DOCUMENT.rd,
  respEmp   Employee
     on e1<USER>.usrid,
  xrefs       Set on x1<XRef> (
     aRef     XRef
     on PRODREF.id
     join DOCUMENT to x1
              by(DOCUMENT.docno == x1.doc)
     ) inverse XRef.refBy,
  refBy       Set on d1<DocRef> (
     aRef     DocRef
     on PRODREF.id
     join DOCUMENT to d1
              by(DOCUMENT.docno == d1.tdoc)
     ) inverse DocRef.ref
  keyword   Set on k1<Keyword> (
     akeyw    Keyword
     on KEYW.keyw
     join DOCUMENT to k1
              by(DOCUMENT.docno == k1.doc)
     ) inverse Keyword.docs
) join e1 to DOCUMENT
     by(e1.usrid==DOCUMENT.usr)

class Department on USER
type Tuple (
  deptName String
     on USER.dpt
)

class OnlineDocument inherit Document
type Tuple (
  contents    octet
     constrainedBy(DOCUMENT.contents != NULL),
  format      integer
     on DOCUMENT.format
)
class OfflineDocument inherit Document
type Tuple (
  archive     String
     constrainedBy(DOCUMENT.archive != NULL)
)
class CommodityGroup on COMGRP
type Tuple (
  name        String
     on COMGRP.name,
  id          integer
     on COMGRP.cgid,
  prodGrps  Set on p1<ProductGroup> (
     aPG      ProductGroup
     on PRODGRP.pg
     join COMGRP to p1
              by(COMGRP.cgid == p1.cg)
     ) inverse ProductGroup.comGrp
)
class ProductGroup on PRODGRP
type Tuple (
  name        String
     on PRODGRP.grpname,
  id          integer
     on PRODGRP.pg,
  comGrp    CommodityGroup
     on c1<COMGRP>.cgid
  manager   Employee
     on u1<USER>.usrid
) join PRODGRP to u1
              by (u1.sname==PRODGRP.manager),
  join PRODGRP to c1
              by (c1.cgid==PRODGRP.cg)

class Product on PRODUCT
type Tuple (
  name        String
     on PRODUCT.name,
  number      integer
     on PRODUCT.no,
  prodGrp     ProductGroup
     on p1<PRODGRP>.pg
) join PRODUCT to p1
              by ((p1.cg==PRODUCT.cg)
              && (p1.pg==PRODUCT.pg))

class Keyword on KEYW
type Tuple (
  keyw        String
     on KEYW.keyw,
  docs        Set on d1<Document> (
     aDoc     Document
     on DOCUMENT.docno
     join KEYW to d1
              by(KEYW.doc == d1.docno)
     ) inverse Document.keywords
)
```

Figure 5.61. Mapping Description for ObjectDRIVER

```
public class Document extends ObjectDRIVERObject {
    private    String   title;
    private    int      number;
    private    String   validUntil;
    private    String   author;
    private    boolean confidential;
    private    Employee respEmp;
    private    Set      xrefs;
    private    Set      refBy;
    private    Set      keywords;
    private    transient int status;

    public     Document() {
    }
    public getTitle() {
        getObject();
        return title;
    }
    public setTitle(String aName) {
        getObject();
        title=aname;
    }
    public Employee getRespEmp() {
        getObject();
        return respEmp;
    }
    public Set getReferencedProducts() {
        getObject();
        SetOfObjects prods = new SetOfObjects();
        Enumeration e = xref.elements();
        ProdRef pr;
        While (e.hasMoreElements()) {
         try {
            pr = (ProdRef) e.get();
            prods.add(pr.getRef());
         } catch (Exception) {};
         e.next();
        }
        return prods;
    }
...
}
```

Figure 5.62. MIS application code (example)

5.7 Evaluation

During the last three years we iteratively implemented, evaluated, and refined our approach to conceptual schema migration and data integration. In 1996, we implemented the idea of using triple graph grammars to describe the translation between logical and conceptual database schemas in a first prototype of the *Varlet Migrator* [JSZ96]. The major motivation for this approach was that experiments with existing tools for schema migration and data integration showed that they provided too little flexibility for *alternative* schema mappings [ONT96, Sie98]. We had the hypothesis that by using triple graph grammars to define and generate schema translators, we would obtain a database migration environment that is easily extensible w.r.t. alternative schema mapping rules. Moreover, the triple graph grammar approach to incremental document integration introduced by Lefering and Schürr [LS96] seemed suitable to overcome the inability of current tools to cope with iterations among schema analysis and migration activities.

experiences with triple graph grammars

We evaluated our first prototype with small application examples and discussed the concepts with other researchers and practitioners in this domain [JSZ97a, JSZ97b]. The ability of the prototype to propagate incremental changes in the logical schema to the conceptual schema and vice-versa received broad attention. In order to increase the flexibility of our schema translation tool, we defined many alternative mapping rules. A drawback of this approach was that it became increasingly difficult for the user of our tool to comprehend all possible alternative translations [Wad98]. Inspired by the research of Hainaut et al. on transformation-based database reverse engineering [HTJC94], we discovered that it is significantly easier for the user to invoke redesign operations on a given conceptual schema than to select from many

alternative translations from a logical to a conceptual schema. Hence, we decided to combine an automatic initial schema translation step (defined by a limited set of triple graph grammar rules) with an interactive conceptual redesign phase. This approach greatly improved the usability of the *Varlet Migrator* but it also required the development of additional mechanisms for change propagation to retain the tool's ability to cope with process iterations. We developed the concept of the history graph to meet this requirement (Section 5.4).

case study

This extended version of the *Varlet Migrator* has been tested and refined in the context of an industrial project in collaboration with two German companies. The analyzed logical schema included 85 tables, 347 attributes, and 138 INDs. The automatic initial translation to the conceptual data model took 2.5 minutes on a SUN Ultra-Sparc II with 300Mhz processor. In experiments with several (internal and external) users, we have validated the advantages of the proposed automatic change propagation mechanism to support process iterations. The most frequent changes of the logical schema have been due to additional INDs or changed semantic classifications of INDs. Depending on how many applied redesign transformations have been affected by a given change, the propagation time ranged from about 30 seconds up to minutes. The users considered this performance as satisfactory compared to the alternative of validating and re-establishing the consistency, manually. Furthermore, all of them appreciated the reliability of using a persistent graph repository and accepted to trade some of the run-time performance for having the advantage of a recovery mechanism after a crash of the *Varlet Migrator*. One common point of criticism was that the current version of our tool does not preserve layout information (for different views) for those increments which have been affected by the change. Still, this weakness is not an inherent characteristic of our approach but we have chosen default layout information to simplify our implementation. Currently, we are working on a new version of the *Varlet Migrator* that overcomes this problem.

middleware generation

The possibility of using the dependency information maintained in the schema mapping graph to generate middleware components for data integration is self-evident. Still, we had to find a data structure that provides suitable flexibility for alternative schema mappings but is simple enough to facilitate its maintenance and interpretation. For our first experiments, we developed an own middleware generator as a test bed to conduct experiments with different data structures [Sch98]. Subsequently, we investigated the possibility of integrating existing commercial middleware generators as a back-end to our DBRE environment. We selected *ObjectDRIVER* [CER99] because it has been freely available for research purposes and it provides suitable flexibility to deal with legacy schemas. Extracting the schema mapping description for *ObjectDRIVER* was possible with little effort and without any modifications to our migration graph structure. Hence, we are confident that other middleware products can be integrated, likewise.

5.8 Related work

According to the two main aspects covered in this chapter (*schema migration* and *data integration*), we split the discussion of related work in two subsections: the following section compares related approaches w.r.t. to their support for schema migration and consistency management, whereas Section 5.6 covers the aspect of data integration.

5.8.1 Conceptual schema migration and consistency management

For more than one decade, many approaches to conceptual schema migration have been developed based on algorithms that perform canonical translations of logical to conceptual schemas [NA87, BDH+87, JK90, MM90, SK90, And94, PKBT94, MCAH95, RH97, Fon97]. Recently, several critics have stated that these approaches provide little flexibility for different possible schema mappings. Because of this problem, Vossen and Fahrner suggest a further manual *redesign phase* after the canonical translation [FV95]. Behm et al. propose an interactive schema migration environment that provides a set of alternative schema mapping rules [BGD97]. In an iterative process, the reengineer chooses an adequate mapping rule for each schema artifact that has to be mapped. This approach is similar to our migration environment in its early stages [JSZ96]. However, we discarded this approach for several reasons: we made the experience that in order to achieve a reasonable flexibility for alternative schema mappings, the set of mapping rules became very large. User experiments showed that with a growing number of alternatives it became increasingly difficult for the reengineer to grasp the semantics of the different mapping rules and choose the best alternative. It turned out that it is much easier for the reengineer to redesign an initial conceptual translation of the logical schema than having to think of alternative mappings between the logical schema and the conceptual schema, explicitly.

Vossen and Fahrner

Behm et al.

Jeusfeld and Johnen

Jeusfeld and Johnen propose an approach to schema migration that employs a generic *meta model* as mediator [MAJ94]. This meta model includes general modeling concepts like objects, types, and links with different cardinality. The schema migration process is performed as follows. In a first step, the concepts of the concrete data model of the LDB are classified in terms of concepts of the meta model. The same is done for the target data model. The classification of the source data model is the basis to map all LDB schema artifacts to equivalent artifacts in the meta model. Analogously, the classification of the target data model is used to map this meta schema back to an equivalent schema in the target data model. These mapping steps are performed in an interactive process and the tool prompts the reengineer in case of ambiguities. Even though the idea of a common meta model as a mediator among different concrete data models is appealing, the advantages of the described approach over a direct translation are questionable. This is because Jeusfeld and Johnen evaluated their approach only for the translation of relational schemas to ER schemas.

Hainaut et al.

Hainaut et al. propose to skip the initial translation step completely and use a common generic data model that subsumes conceptual constructs as well as logical (and physical) constructs [HHHR96, Hai89]. Based on this common data model Hainaut et al. have defined a catalog of schema transformations which are used to gradually replace low-level implementation constructs by more abstract concepts [HTJC94]. An implicit assumption behind this approach is that all relevant information about the legacy schema is available at the beginning of the

migration process. In this dissertation, we have argued that this assumption is unrealistic because, in practice, iterations among analysis and migration activities might occur for different reasons. However, the execution of *in-place* transformations (as suggested by Hainaut) impede such iterative DBRE processes because the original LDB schema is lost during the migration process. A possibility to overcome this limitation is to make an initial copy of the LDB schema and perform all transformations on this copy. This initial copy operation can be implemented as (very simple) initial schema mapping transformations and, thus, the consistency management mechanism defined in this thesis can be used to enable iterations.

problem of consistency

The problem of consistency management in case of DBRE process iterations is not adequately solved in any of the above approaches. As exemplified in the previous paragraph, the mechanism for incremental change propagation, which has been developed in this dissertation, can be used with little modifications to complement these approaches and overcome this limitation. None of the above DBRE tools supports automatic propagation of extensions made to the conceptual schema back into the original implementation.

problem of idiosyncrasies

Most approaches referenced above presume a logical schema in third normal form [EN94]. Some authors argue that this requirement can always be satisfied by inserting a preprocessing (normalization) step before migrating the schema [FV95]. However, this solution is not feasible for unforeseen idiosyncratic optimization patterns [BP95]. Hence, it is important that a DBRE tool can easily be adapted to deal with such patterns. Most existing tools do not provide the necessary adaptability, because their schema migration process and mapping rules are hard-coded in general programming languages. A notable exception that employs a dedicated language to describe transformation systems (TXL) has been developed by Cordy et al. [MCAH95]. Still, such textual transformation patterns are significantly harder to formulate and comprehend than graphical transformation rules. Because of this reason many authors have used diagrams to communicate their transformation rules to their readers, e.g., [BP96, HTJC94, BCN92, Tre95]. By choosing graph grammars, the approach presented in this thesis combines the expressiveness of diagrams with the executability of formal replacement systems. The *Progres* graph grammar engineering environment allows the reengineer to specify additional mapping rules and redesign transformations. This facilitates to add further mapping rules to deal also with denormalized RS, e.g., the rules described by Ramanathan and Hodges [RH97].

problem of variant structures

The formal definition and automatic translation of variant structures in LDB schemas to inheritance structures in the conceptual model is new in our approach. Other existing DBRE tools do not consider variant structures even though they are broadly used in forward engineering relational database schemas [HHEH96, BCN92].

5.8.2 Data integration

Behm et al.

Fong

Only a few of the approaches to schema migration also tackle the problem of data integration. Behm et al. [BGD97] and Fong [Fon97] aim on a complete replacement of relational by object-oriented databases. Based on the schema correspondences created in the schema migration step, they present algorithms to migrate the data in a batch-oriented process. Due to our experience, a complete replacement of relational by object-oriented database platforms is often not desired, not viable, or implies a significant risk. Hence, there has been an increasing

industrial demand for approaches to wrap and integrate LDB systems with modern technologies.

Hainaut et al.

In the *InterDB* project [THB+98], Hainaut et al. use their transformation-based approach to schema migration to generate the data integration wrappers for LDBs [TCHH99]. Their approach is based on the definition of data conversion operations (instance mappings) for all schema transformations. A logging mechanism records all schema transformations which have been applied during the interactive schema migration and redesign phase. This history log is the basis to generate a data conversion program which consists of a concatenation of the instance mappings of all applied transformations. The main difference to the approach described in this dissertation is that we maintain *explicit* schema dependencies in a schema mapping graph (SMG). This explicit information allows us to generate *declarative* schema mapping descriptions as an input for various commercial off-the-shelf (COTS) middleware products. This is not possible or at least problematic for Hainaut's approach because schema correspondences are *implicitly* defined in *operational* data conversion programs.

COTS middleware

Examples for commercial middleware products which require declarative textual schema mapping descriptions are Ardent Software's *Java-Relational-Binding* [Gre98], *ObjectDRIVER* [ObjDrv99], *OpenDM* [Sie98], and *CocoBase* [Tho99]. Other products also provide graphical user interfaces to build schema mappings, e.g., *Object Integration Server* [ONT96], *ObjectMatter* [Obj99b], and *TOPLink* [Obj99a]. The common aim of these products is to wrap LDB applications with a modern API that facilitates integration with object-oriented, distributed, and platform independent technology, e.g., *CORBA*, *COM*, and *Java* [Uma97].

Web-gateways

Still, in projects that focus on integrating legacy data with Web-based services it might also be sufficient to use a more light-weight approach in terms of so-called *Web-gateways*. Currently, almost every database vendor offers such a gateway solution. Typically, Web-gateways provide the possibility to embed database queries into HTML pages. Kappel et al. present a taxonomy for the different technical solutions in this domain [EKR97].

5.9 Summary

In this chapter, we elaborated an *incremental* approach to conceptual schema migration which is based on a tight integration of tools for legacy schema analysis and conceptual translation and redesign. A major benefit of this approach is that it provides support for iterations between analysis and conceptual migration activities rather than imposing a strictly phase-oriented DBRE process. We showed that a common graph repository is a suitable platform for this tight integration. Furthermore, it allows to employ *graph grammars* as an abstract formalism to facilitate specification of schema translation and redesign transformations. We argued that this high level of abstraction is particularly important because it facilitates extension and adaption of schema transformations due to unforeseen design patterns in LDB schemas. Based on the concept of input/output dependencies of schema transformations, we described an incremental change propagation mechanism that allows the reengineer to reestablish schema consistency after iterations in the DBRE process, automatically. We used the *Progres* graph grammar engineering environment to implement our approach in a customizable DBRE tool called the *Varlet Migrator*. We argued that another benefit of this tight integration is the possibility to generate schema mapping descriptions for existing middleware components. We selected the object-relational middleware product *ObjectDRIVER* to validate this hypothesis.

CHAPTER 6 CONCLUSIONS AND FUTURE PERSPECTIVES

6.1 Major contributions

Database reengineering (DBRE) activities inherently deal with uncertain information about the internal structure of legacy systems. This uncertainty and the fact that legacy systems evolve during ongoing migration activities often cause iterations in DBRE processes. The direct result of such process iterations are inconsistencies between the implementation of the legacy system and its conceptual (re)design. In this dissertation, we have explored concepts and techniques to manage aspects of uncertainty and inconsistency in computer-aided DBRE processes. The major contributions of our research are summarized in the following paragraphs.

selection of a theory to manage uncertainty

Based on our experiences with practical DBRE case studies, we elaborated a catalog of central requirements on a theory as a basis to represent and reason about imperfect DBRE knowledge. With this catalog we studied and evaluated major theories in the domain of approximate reasoning. As a result of this evaluation, we have identified possibilistic logic as the theory which is most suitable to provide the framework for our research.

GFRN as a basis for LDB analysis

In this framework, we have developed Generic Fuzzy Reasoning Nets (GFRNs) as a dedicated formalism to specify and adapt DBRE heuristics and processes. GFRN specifications provide the basis to integrate and combine many existing schema analysis operations and methods. By distinguishing between data-driven and goal-driven analysis operations, GFRNs allow for the specification of *active* analysis tools. Such tools are capable of executing analysis operations depending on the state of information about the legacy system, automatically. This is in contrast to traditional (*passive*) tools where all analysis operations have to be invoked explicitly by the user. The GFRN language has a sound declarative semantics based on a formal translation to necessity-valued possibilistic logic. In order to execute GFRN specifications in human-centered DBRE tools, we have developed a non-monotonic inference algorithm based on fuzzy Petri nets.

implementation and evaluation

Incorporating imperfect knowledge in DBRE tools has a significant impact on their user interfaces. New concepts and interaction mechanisms are required to communicate uncertain and contradicting information to the reengineer and guide him/her to a consistent analysis result. In our prototype CARE tool *The Varlet Analyst*, we have developed filter mechanisms and an advanced agenda concept to meet these requirements. *The Varlet Analyst* has been used as a test bed to evaluate our approach to legacy schema analysis with practical case studies. These experiments showed that the concepts and techniques developed in this thesis represent a valuable improvement over currently existing tool support for legacy schema analysis.

flexible schema translation

We have developed a hybrid approach to conceptual schema migration which consists of an automatic initial translation step followed by an interactive redesign and extension phase. The entire migration process has been specified on a high-level of abstraction using graph transformation systems. A generation mechanism which is mainly based on the *Progres* graph grammar engineering environment enables the produce executable transformation tools based

on this abstract specification. This generative approach provides a high amount of flexibility and extensibility which is important to consider unforeseen idiosyncrasies in legacy database (LDB) schemas. Moreover, the proposed schema mapping mechanism is *bidirectional*, i.e., it allows the reengineer to map modifications in the conceptual model back to the implemented logical schema.

incremental consistency preservation

Using graph grammars to specify schema translation and redesign operations enabled us to derive a formal notion of their input/output dependencies according to left-hand side and the right-hand side of each graph production rule. Based on these dependencies, we have defined a data structure (history graph) that logs information about all steps performed during the schema migration process. In case of iterations in the DBRE process, the history graph is interpreted by an algorithm that performs incremental change propagation and reestablishes document consistency, automatically. This technique enables to intertwine analysis and migration activities in evolutionary DBRE processes. Consequently, our approach provides a suitable basis to construct CARE environments which provide more adequate support for DBRE projects than existing, strictly phase-oriented tools. We have implemented this consistency management mechanism in the prototype CARE tool *The Varlet Migrator* which has been evaluated with industrial collaboration.

heterogeneous data integration

We have demonstrated the suitability of the information maintained in our schema mapping graph model to generate declarative schema mapping descriptions. This facilitates the integration of our DBRE tool with various available middleware products for heterogeneous data integration to obtain a flexible and comprehensive environment for LDB analysis, migration, and encapsulation. We have evaluated this approach with a commercial object-relational middleware product.

6.2 Transferability of results

schema analysis

Even though the focus of this dissertation is on reengineering legacy relational databases, most of our results are not limited to this specific application domain. The requirements that were used to select a suitable theory to manage uncertain DBRE knowledge remain valid in many other scenarios that aim on software comprehension and design recovery. For example, we have noticed similar problems and challenges in the domain of architectural design recovery for object-oriented software. A mechanism to detect and classify design patterns [GHJV95] would be very supportive for software comprehension. Recently, researchers have started to investigate in techniques that can be used to detect such patterns [KSRP99, Bro96, KDBM94, TFAM96]. As a common problem, they encountered that different software systems contain various derivations of the same design pattern. Typically, their detection is ambiguous and inherently deals with heuristics, e.g., naming conventions, structural characteristics, and caller/callee relationships. Current tools for design pattern detection lack explicit concepts to deal with imperfect knowledge. Their heuristics are often hard-coded and cannot be adapted easily. The concepts and mechanisms in Chapter 4 are suitable to complement these approaches and overcome their current limitation. First attempts to employ GFRNs for the detection of *C++* and *Java* design patterns have shown that this approach is promising and feasible [Jahn97a].

conceptual migration

Many tools for conceptual abstraction and interactive redesign of software are based on some formal notion of a transformation system [HTJC94, MCAH95, YB94, War96, PMdP98]. The mechanism to incremental consistency management developed in this dissertation can

complement these approaches and enable them to deal with iterations in this migration process. Furthermore, we have demonstrated that the application of graph grammar engineering techniques in combination with automatic code generators can contribute significantly to decrease the complexity of constructing and customizing tools for software abstraction and migration.

6.3 Open problems

selection of CVs

While applying our approach to practical case studies we encountered a number of open problems which need further investigation. One of these open problems considers the selection of confidence values (CVs) for GFRN implications. In this dissertation, we argued that the credibility of DBRE heuristics depend highly on various technical and non-technical characteristics of the LDB under investigation, e.g., different naming conventions, design paradigms, and DBMS functionality. In principle, the GFRN approach facilitates customizing the credibility of the different heuristics used in the semi-automatic analysis process by adjusting the CVs of implications. In Section 4.1, we proposed that this adjustment should be done according to the results of an initial *domain analysis* activity. However, we have experienced that selecting "good" CVs *a-priori* (before the actual analysis starts) is far from being trivial. This is because many characteristics, especially non-technical characteristics, remain undetected in the initial domain analysis step. Consequently, it is likely that the reengineer starts the analysis process with suboptimal CVs whenever the application context of the CARE tool has changed to an LDB from another company, developer team, or on a different platform. Of course, (s)he can adjust the CVs *on-the-fly* during the analysis process when (s)he learns more about the LDB implementation. Still, this entailed that every user of our schema analysis tool also has to learn about the GFRN formalism. A much more preferable solution was if the tool would adjust the CVs automatically during the interactive analysis process. An automatic adaption mechanism could exploit interactive decisions of the reengineer to decrease CVs of heuristics which have lead to false hypotheses and increase CVs which (could) have lead to a correct indication.

top-down migration

A different open problem considers the fact that our approach to conceptual schema migration is limited to *bottom-up* migration only. This means that our technique supports incremental creation of an abstract conceptual design from an implemented logical schema. However, there are many practical DBRE scenarios where an abstract design is (partly) existent at the beginning of the migration process. It often occurs that companies have (obsolete) design documents for specific subsystems of their LDBs. Even more important are scenarios that aim on federating several (heterogeneous) LDBs into an enterprise-wide business object model. In the latter case, specific parts of the conceptual design are predefined and the reengineer has to map this design to the existing LDB schema. So far, these *top-down* migration scenarios are not considered by our approach.

loss of layout information during change propagation

Even though the developed consistency management mechanism has been well accepted by the users of our DBRE tool, most of them criticized that after propagating a schema update, the layout information has been lost for certain schema increments (classes and relationships). More precisely, the layout information has been lost for all those schema increments which represent the output of transformations that have been re-evaluated during the change propagation step. The reason for this irritating and annoying effect is that whenever a

transformation application is going to be re-evaluated, its former output is discarded and reproduced. The layout information which is associated to the former output is discarded as well. In the case that all transformation applications in a dependency chain remain valid, this problem can be solved by copying the layout information from the former output of the last transformation applications to their new output. The situation becomes more difficult for transformation applications which are no longer valid. In these cases, layout information for their former input increments are no longer available because these increments are only represented by place holder nodes in the history graph. One possible solution is to annotate these place holders with their layout history.

6.4 Future perspectives

generalizing GFRNs

One focus of our future research is on generalizing the GFRN approach for other applications in the RE domain. For this purpose, we have designed and implemented the GFRN editor and the inference engine in a modular and portable way that facilitates integration with other CARE tools. We plan to make this component freely available for academic purposes. In a project called *FUJABA* (*From UML to Java And Back Again*) [KNNZ99], we have started to experiment with GFRN specifications to analyze *Java* software and detect design patterns. Preliminary experiences show that this is a suitable application although the problems involved seem to be harder than in the application described in this dissertation: we noticed that the structure of typical object-oriented design patterns is much more complex than the structure of most relational schema constraints. Defining complex patterns in terms of predicates and implications results in rather large GFRN specifications which are difficult to read. Therefore, we plan to develop a more adequate notation for such search patterns with a semantics based on GFRN specifications. We have begun to investigate the suitability of annotated UML object-diagrams for this purpose.

self-adaptation

In a Master Thesis, we developed a first prototype for a learning mechanism that adjusts the CVs of GFRN implications automatically during the semi-automatic analysis process [Str99]. The motivation for this research is the aforementioned problem for the user to estimate the right CVs when the application context of the analysis tool has changed. The goal is to minimize the classification error, i.e., to decrease the CVs of those implications which lead to a large number of false hypotheses and increase the CVs of implications which (could) lead to true hypotheses. The idea of our approach is to exploit the interactive feedback of the reengineer during the analysis process to adapt the CVs in the GFRN (cf. Figure 6.1). For this purpose, we employ techniques known from the area of neural network learning [Gal93]. Based on the hypotheses indicated by the GFRN inference engine and the (refutation and confirmation) decisions of the reengineer, our tool creates a so-called *learning task* (LT). Then, the LT is fed back into a feed-forward neural network (NN) which has been generated from the current GFRN specification. We use the standard *backpropagation* algorithm [Gal93] to train the weights in the NN that correspond to the CVs in the GFRN. Finally, the CVs in the GFRN are adjusted according to the new weights in the NN.

The technique outlined above could be a possible basis to develop *adaptive* CARE tools. First experiences with the described mechanism show that this approach is feasible [JS99]. Still, several questions remain in this context which need further investigation. For example, a central question is on how to select the parameters of the backpropagation algorithm (learning

rate, momentum factor, etc.) to achieve a fast yet stable adaption process. These parameters define the influence of the current application context of the analysis tool w.r.t. previous experiences. The general idea is to increase the learning rate temporarily when the tool is applied in a new RE project, in a different company, or for new subcomponent that has been developed by another developer team. Practical case studies will play an important role within our efforts to evaluate and refine this technique.

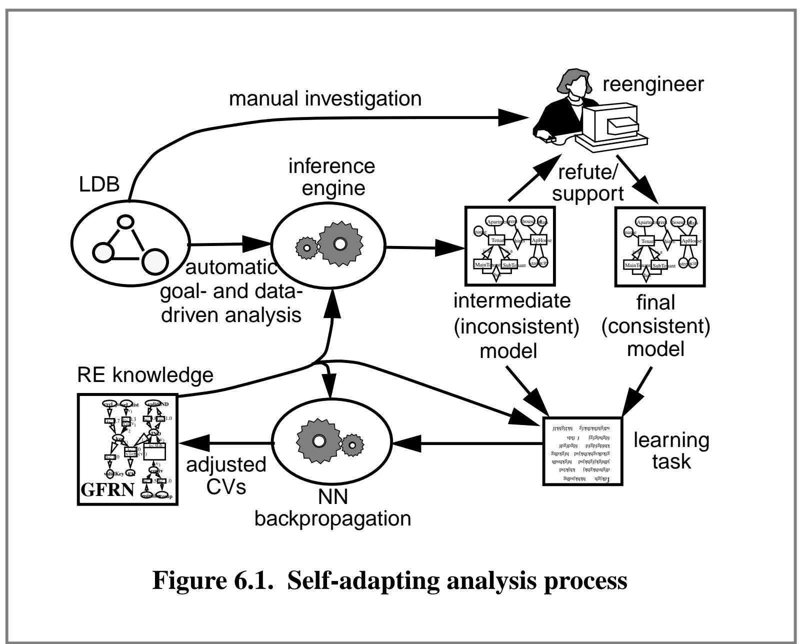


Figure 6.1. Self-adapting analysis process

LDB federation and evolution

In this dissertation, we developed methods and techniques to support reengineering and integration of *single* LDB systems with object-oriented technology. However, an increasing number of companies strive to federate *several* heterogeneous information systems (IS) to achieve integrated, enterprise-wide information infrastructures [Rad95]. An important condition for the efficiency of such net-centric IS is their ability to evolve in step with changing market conditions and changes in the organizational structure of the company. Tools which allow to modify and evolve net-centric IS on a high-level of abstraction have a great potential to contribute to the desired flexibility. In the future, we plan to generalize our graph grammar approach to schema integration and redesign for its application in IS federation and evolution scenarios. As mentioned in Section 6.3, a first step of this generalization will be the extension of our approach by a technique for *top-down* schema migration.

abstract losslessness criterion

In Section 5.3, we followed a broadly used approach to categorize schema transformations according to their impact on the information capacity of the target schema w.r.t. the source schema [BCN92, HTJC94, Tre95, Sch93]. We elaborated semi-formal proofs for these classifications in [Rum98]. Like other researchers in the domain of schema redesign, we have noticed that constructing such proofs requires experiences and skills which cannot be expected from a typical reengineer who wants to extend the catalog of schema transformations available in our DBRE environment. Therefore, it was beneficial to have an *abstract losslessness criterion* which can easily be applied to proof properties of newly specified schema transformations. In [JZ99, JZ98], we have begun to develop such a formal criterion based on

the rich theory of parallel graph rewriting rules [Tae96]. For the future, we plan to refine this approach such that it can be integrated with our tool customization process to facilitate reasoning about properties of newly added transformations.

user experiments

Incorporating uncertain and contradicting knowledge in tool-based RE processes requires new human-computer interaction schemes to eliminate this imperfect knowledge efficiently and arrive at a consistent result. Such efficient and user-friendly interaction schemes are crucial to exploit the benefits of this new technology and achieve broad commercial acceptance in industry. In Section 4.4.2.2, we proposed a first user interface solution based on an advanced agenda concept with query and filter mechanisms. This user interface has to be evaluated and refined in practical user experiments. We will conduct these experiments in tight collaboration with industry and the Software Engineering Group at the University of Victoria, B.C., Canada. Their scientific background in tool evaluation [MWS97, Sto98] and the new *Experimental Software Engineering Lab* at the University of Victoria represent an ideal environment to conduct these experiments.

APPENDIX A ADDITIONAL DEFINITIONS AND SPECIFICATIONS

A.1 Interpretation of a logical schema

The interpretation of a relational database schema is well-defined in the literature. Still, in Definition 4.1, we substituted the problematic notion of NULL-values by a new concept of relational *variants*.[a] Consequently, we have to define the interpretation of this new concept. The following Definition A.1 formalizes the interpretation of a logical schema with variants. Note, that this formalization does not include the intentional semantics of the annotation function $\mathcal{A}$. Operation Π denotes the usual relational projection of the relational algebra [EN94].

Definition A.1 Interpretation of a logical schema

The ***interpretation of a logical schema*** *$(T, R, \Delta, \mathcal{A})$ is a tuple:=$(\Im_T, \Im_R, \Im_\Delta)$, where*

- *$\Im_T: T \rightarrow \overline{SET}$ is a function that maps column type names to finite sets, i.e., their domains.*
- *$\Im_R: R \rightarrow \overline{REL} \times \overline{FUN} \times \mathcal{L}\{L^1\}$, $\Im_R(r:(n,X,\Sigma,V)) = (\Im_X, \Im_V, \Im_\Sigma)$, $r \in R$, is a function that maps each RS to a tuple of a relation, a function, and a constraint represented by a logical implication;*
    - *relation $\Im_X$ is a subset of the cartesian product of the domains of all columns (including the special value NULL), i.e., $\Im_X \subseteq \Im_T(t_1) \cup \{NULL\} \times .. \times \Im_T(t_m) \cup \{NULL\}$, for $X = \{(n,c_1,t_1),...,(n,c_m,t_m)\}$, $m \in \mathbb{N}$;*
    - *function $\Im_V: V \rightarrow \overline{REL}$ maps variants to relations; for each variant $v \in V$, $\Im_V$ is a subset of the cartesian product of the domains of all columns in v, i.e., $v:\{(n,c_1,t_1),...,(n,c_m,t_m)\} \in V$, $m \in \mathbb{N}$, $\Im_V(v) \subseteq \Im_T(t_1) \times .. \times \Im_T(t_m)$;*
    - *$\Im_\Sigma$ is an implication that specifies that all tuples in $\Im_X$ can uniquely be identified by the values in their key columns, i.e., $\Im_\Sigma$= '$\forall s_1, s_2 \in \Im_X : (\Pi_\Sigma(s_1) = \Pi_\Sigma(s_2) \rightarrow s_1 = s_2)$';*
- *$\Im_\Delta: \Delta \rightarrow \mathcal{L}\{L^1\}$ is a function which maps each IND to a logical implication: $\Im_\Delta(d:(l, r, I))$= '$\forall s_1 \in \Im_V(l)\ \exists s_2 \in \overline{\Im}_{\overline{X}}(r) : (\forall i \in I : \Pi_i(s_1) = \Pi_i(s_2))$', with $\Im_R(RS(l)) = (\Im_X, \Im_V, \Im_\Sigma)$, $\Im_R(r) = (\overline{\Im}_{\overline{X}}, \overline{\Im}_{\overline{V}}, \overline{\Im}_{\overline{\Sigma}})$.*

D

---

a NULL-values often cause problems during the migration of relational to object-oriented platforms because object-oriented data models typically lack the concept of NULL-valued attributes.

A.2 Specification of the migration graph model

In this section, we employ the formal specification language *Progres* [SWZ95] to define the migration graph model discussed in Section 5.1.

logical schema ASG

```
spec MigrationGraphModel

  node class Increment end;

  section LogicalSchemaASG

    node type LSchema : Increment end;

    node type RS : Increment
      intrinsic
        rsname : string;
    end;

    node type LType : Increment
      intrinsic
        ltname : string;
    end;

    node type Variant : Increment end;

    node type LKey : Increment end;

    node type I_IND : IND end;

    node type ForKey : Increment end;

    node type C_IND : IND end;

    node type R_IND : IND
      intrinsic
        invkb : boolean;
    end;

    node type Column : Increment
      intrinsic
        colname : string;
    end;

    edge type c_RS : LSchema [1:1] -> RS [0:n];

    edge type c_lt : LSchema [1:1] -> LType [1:n];

    edge type c_v : RS [1:1] -> Variant [1:n];

    edge type c_ak : RS [1:1] -> LKey [1:1];

    edge type c_col : Variant [0:n] -> Column [1:n];

    edge type c_fk : Variant [1:1] -> ForKey [0:n];

    edge type c_c : ForKey [0:n] -> Column [1:n];

    edge type c_kc : LKey [0:n] -> Column [1:n];
```

```
  edge type lt : Column [0:n] -> LType [1:1];

  node class IND end;

  edge type c_k : IND -> LKey;

  edge type c_f : IND -> ForKey;

end;

section ConceptualSchemaASG

  node type CSchema : Increment end;

  node type CType : Increment
    intrinsic
      ctname : string;
  end;

  node type Class : Increment
    intrinsic
      clname : string;
      abstract : boolean;
  end;

  node type Inheritance : Increment end;

  node type CKey : Increment end;

  node type Attribute : Increment
    intrinsic
      aname : string;
      default : string;
  end;

  node type Association : Relationship
    intrinsic
      srctotal : boolean;
      srccard : integer;
  end;

  node type Aggregation : Relationship end;

  edge type c_ct : CSchema [1:1] -> CType [1:n];

  edge type c_cl : CSchema [1:1] -> Class [0:n];

  edge type sup : Inheritance [0:n] -> Class [1:1];

  edge type sub : Inheritance [0:1] -> Class [1:1];

  edge type c_ck : Class [1:1] -> CKey [0:1];

  edge type c_ka : CKey [0:n] -> Attribute [1:n];
```

conceptual schema ASG

```
  node class Relationship is a Increment
    intrinsic
      srcname : string;
      tarname : string;
      tartotal : boolean;
      tarcard : integer;
  end;

  edge type src : Relationship [0:n] -> Class [1:1];

  edge type tar : Relationship [0:n] -> Class [1:1];

  edge type c_att : Class -> Attribute [0:n];

  edge type ct : Attribute [0:n] -> CType;

end;
```

SMG model

```
section SchemaMappingGraphModel

  node type MapSch : Increment end;

  edge type m_v : MapV [0:n] -> Variant [1:n];

  node type MapType : Increment end;

  node type MapV : Increment end;

  node type MapInc : Increment end;

  node type MapIIND : Increment end;

  node type MapKey : Increment end;

  node type MapCol : Increment end;

  node type MapRIND : Increment end;

  edge type m_ls : MapSch [0:1] -> LSchema [1:1];

  edge type m_cs : MapSch [1:1] -> CSchema [1:1];

  edge type m_lt : MapType [0:1] -> LType [1:1];

  edge type m_ct : MapType [0:1] -> CType [1:1];

  edge type m_cl : MapV [0:1] -> Class [1:1];

  edge type m_v_in : MapInc [0:1] -> Inheritance [1:1];

  edge type m_iind : MapInc [0:n] -> Variant [1:n];

  edge type m_i_in : MapIIND [0:n] -> Inheritance [1:1];

  edge type m_lk : MapKey [0:n] -> LKey [1:1];

  edge type m_ck : MapKey [0:1] -> CKey [1:1];

  edge type m_col : MapCol [0:1] -> Column [1:1];
```

```
    edge type m_a : MapCol [0:1] -> Attribute [1:1];

    edge type m_rind : MapRIND [0:1] -> R_IND [1:1];

    edge type m_vs : MapInc [0:n] -> Variant [1:n];

    edge type m_id : Class [0:n] -> MapKey [1:1];

    node type MapRel : Increment end;

    edge type m_r : MapRel [0:1] -> Relationship [1:1];

    edge type r_via : MapRel [0:n] -> MapRIND [0:n];

    edge type a_via : MapCol [0:n] -> MapRIND [0:n];

  end;

end.
```

history graph model

```
section HistoryGraphModel

  node type Transformation : Increment end;

  node type Parameter : Increment
    intrinsic
      nr : integer;
  end;

  edge type In : Transformation [0:1] -> Parameter [1:n];

  edge type Out : Transformation [0:1] -> Parameter [0:n];

  edge type con1 : Transformation [0:1] -> Increment [0:n];

  edge type actual : Parameter [0:n] -> Increment [0:n];

end;
```

graph tests to check for constraint violations

```
section Constraints

    test DoubleAggregation =

                  (     (    <-sub-
                           &  -sup-> )
                    or  (    <-sup-
                           &  -sub-> ) ) *
     '1 : Class  ==============>  '2 : Class

       tar                          tar

     '3 : Aggregation             '4 : Aggregation

      folding { '1, '2 };

    end;
```

```
test DuplicateAttrName =
```

'1 : Class
c_att
c_att
'2 : Attribute
'3 : Attribute

```
  condition '2.aname = '3.aname;

end;

test DuplicateClassName =
```

'1 : CSchema
c_cl
c_cl
'2 : Class
'3 : Class

```
  condition '2.clname = '3.clname;

end;

test DuplicateRelName1 =
```

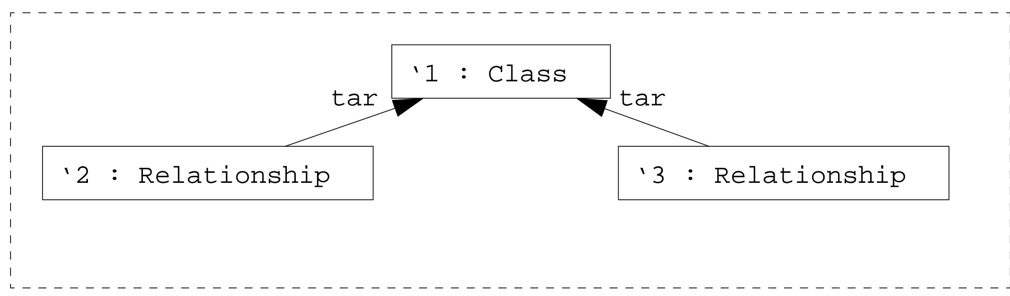
'1 : Class
src
src
'2 : Relationship
'3 : Relationship

```
  condition '2.srcname = '3.srcname;

end;

test DuplicateRelName2 =
```

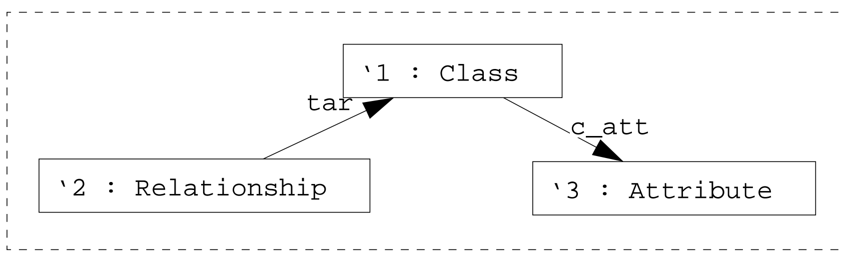
'1 : Class
tar
src
'2 : Relationship
'3 : Relationship

```
  condition '2.tarname = '3.srcname;

end;
```

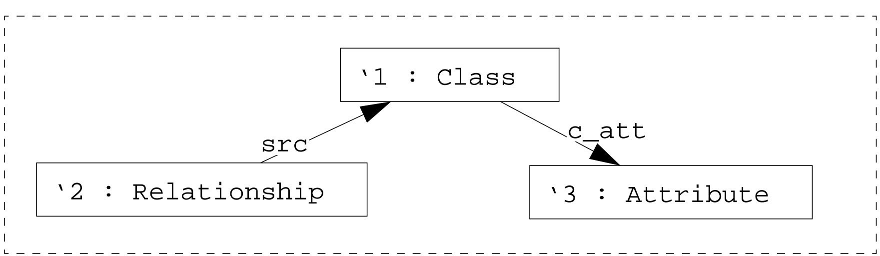
```
test DuplicateRelName3 =
```




```
  condition `2.tarname = `3.tarname;
end;
test RelnameEqualAttrname =
```




```
  condition `2.tarname = `3.aname;
end;
test RelnameEqualAttrname1 =
```




```
  condition `2.srcname = `3.aname;
end;
end;
```

APPENDIX B A CATALOG OF REDESIGN TRANSFORMATIONS

This appendix presents the specification for the primitive schema redesign transformations implemented in this dissertation. The following table gives an overview of their purpose and their location in this appendix.

Transformation	**Short description**	**Type**	**Page**
Aggregate	Transforms an association into an aggregation	IP	196
AssociationToClass	Transforms an association between two classes to an intermediate class with two associations	IP	197
ChangeAssoc-Cardinality	Modifies the cardinality of a given association	IC	198
ChangeAttributeType	Changes the type of an attribute	IC	198
ClassToAssociation	Transforms a class that participates in two *one-to-many* associations to a *many-to-many* association	IP	199
CreateAssociation	Creates an association between two given classes	IA	200
CreateAttribute	Creates an attribute in a given class	IA	200
CreateClass	Creates a new class	IA	201
CreateInheritance	Creates an inheritance relationship between two given classes	IA	201
CreateKey	Creates a key for a given class	IR	202
ConvertAbstract	Converts a concrete class into an abstract class	IR	202
ConvertConcrete	Converts an abstract class into a concrete class	IA	203
DisAggregate	Transforms an aggregation into an association	IP	204
Generalize	Creates a generalization for a given class	IA	205
MergeClasses	Merges two classes which are associated by a *one-to-one* relationship into a single class	IP	206
MoveAttribute	Moves an attribute from one class to an associated class via a given *one-to-one* relationship	IP	207
PushDownAttribute	Moves an attribute of a given class to its specialization	IR	208
PushDown-Association	Moves a relationship of a given class to its specialization	IR	209
PushUpAttribute	Moves an attribute of a given class to its generalization	IA	210
PushUpAssociation	Moves a relationship of a given class to its generalization	IA	211
Remove	Removes an increment from the conceptual schema	IC	212
RenameAttribute	Changes the name of an attribute	IP	212
RenameClass	Changes the name of a class	IP	212
RenameRelationship	Changes the role names of a relationship	IP	213
Specialize	Creates a specialization for a given class	IA	214
SplitClass	Splits a class in two classes connected by a *one-to-one* relationship	IP	213
SwapAssocDirection	Swaps source and target of a given association	IP	215

Aggregate

Transformation *Aggregate* converts an association (*rel*) into an aggregation. Its application condition specifies that the source of association *rel* has to be a total, single reference.

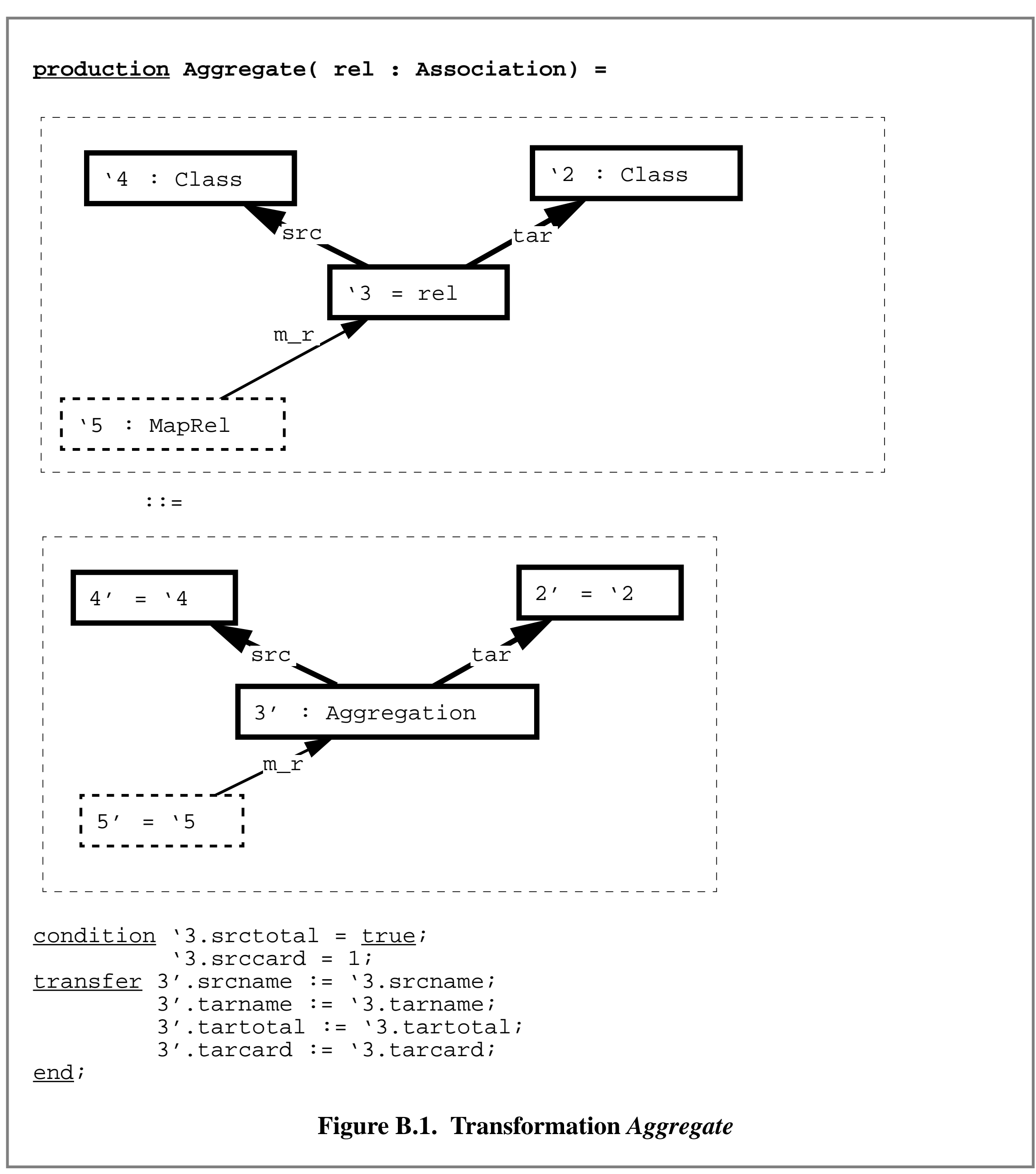


Figure B.1. Transformation *Aggregate*

Production *AssociationToClass* specifies the reverse transformation for transformation *ClassToAssociation*, i.e., it transforms an association to a class with two associations.

AssociationToClass

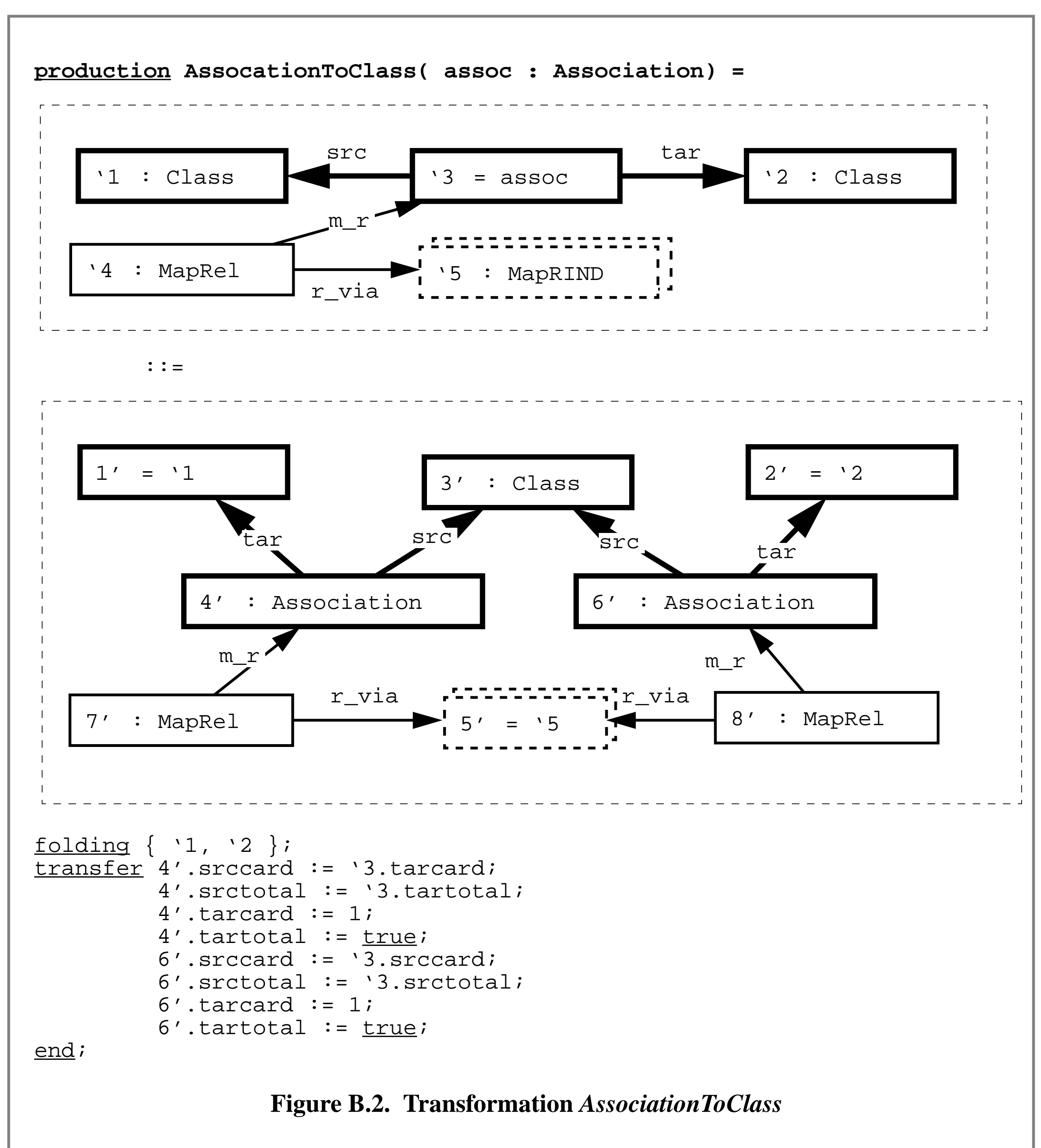


Figure B.2. Transformation *AssociationToClass*

ChangeAssoc-Cardinality

Transformation *ChangeAssocCardinality* modifies the cardinality of a given association. The choose statement determines whether the transformation application is information-reducing (IR). If this is the case, the cardinality of the given association *assoc* is adjusted according to the actual parameters of the transformation. Otherwise, the existing association is replaced by a new association with the desired cardinality constraints. Note, that this implies the loss of all correspondence information with the logical schema which might have existed for the original association *assoc*.

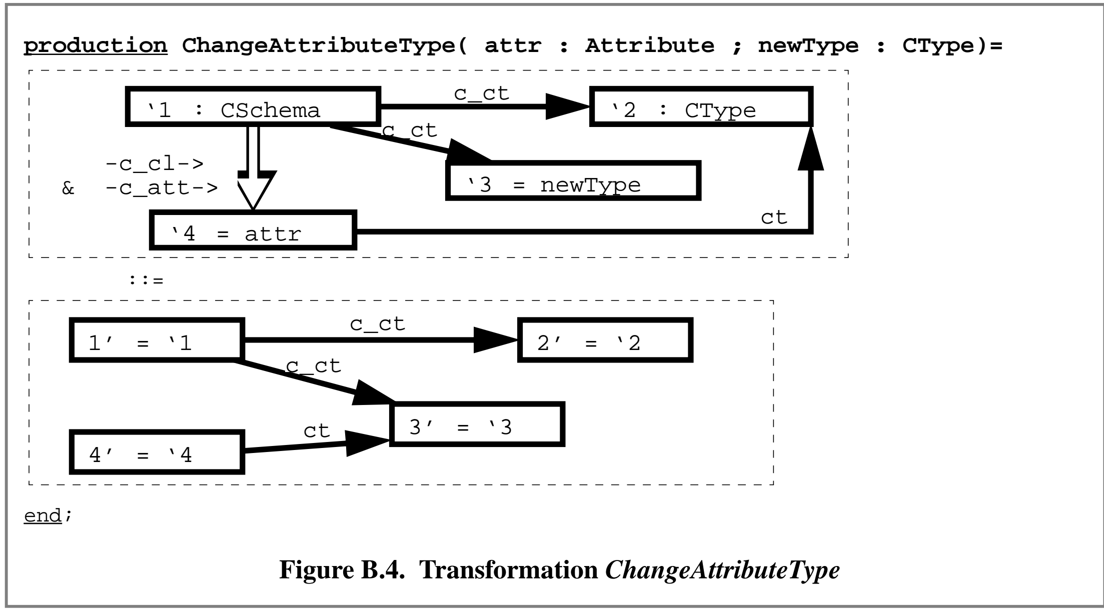
```
transaction ChangeAssocCardinality( assoc : Association; srcCard : integer;
                                    srcTotal : boolean; tarCard : integer ;
                                    tarTotal : boolean)=
choose
 when (* IR transformation? *)
     ((assoc.srccard > srcCard) and (assoc.tarcard > tarCard))
 then
    assoc.srccard := srcCard
  & assoc.tarcard := tarCard
  & assoc.srctotal := srcTotal
  & assoc.tartotal := tarTotal
 else
    use newAssoc : Association
    do
       CreateAssociation ( assoc.-src->, assoc.-tar->, assoc.srcname,
                           assoc.tarname,out newAssoc )
     & Remove ( assoc )
     & newAssoc.srccard := srcCard
     & newAssoc.tarcard := tarCard
     & newAssoc.srctotal := srcTotal
     & newAssoc.tartotal := tarTotal
    end
 end
end;
```

Figure B.3. Transformation *ChangeAssocCardinality*

ChangeAttribute-Type

Transformation *ChangeAttributeType* changes the type of a given attribute *attr* to *newType*.

```
production ChangeAttributeType( attr : Attribute ; newType : CType)=
```




```
end;
```

Figure B.4. Transformation *ChangeAttributeType*

ClassToAssociation

Transformation *ClassToAssociation* transforms a class with two associations into an association. Negative application conditions (nodes ‘6, ‘7, and ‘12) ensure that the class has no properties other than the required two associations (‘4 and ‘5) and does not participate in an inheritance hierarchy. The application conditions of *ClassToAssociation* restrict the two associations of the given class *cl* to be single valued w.r.t. the participating classes (‘2,‘3). Note, that the requirement that *cl* is the source of both associations can be satisfied by executing primitive transformation *SwapAssocDirection* first.

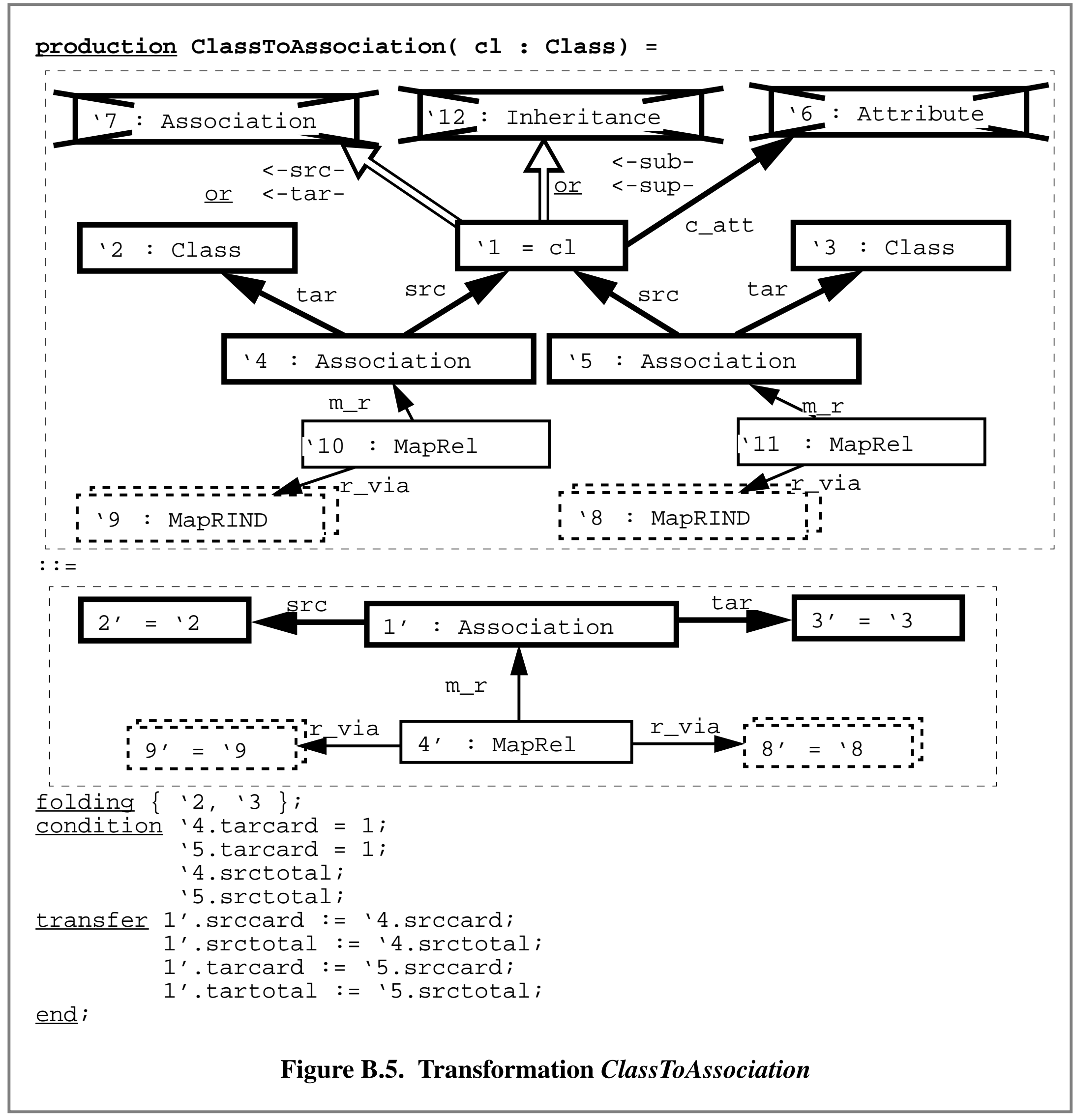


Figure B.5. Transformation *ClassToAssociation*

CreateAssociation

The following transformations *CreateAssociation, CreateAttribute*, *CreateClass*, *CreateInheritance*, and *CreateKey* extend the conceptual schema by a new association, attribute, class, inheritance relationship, and key, respectively.

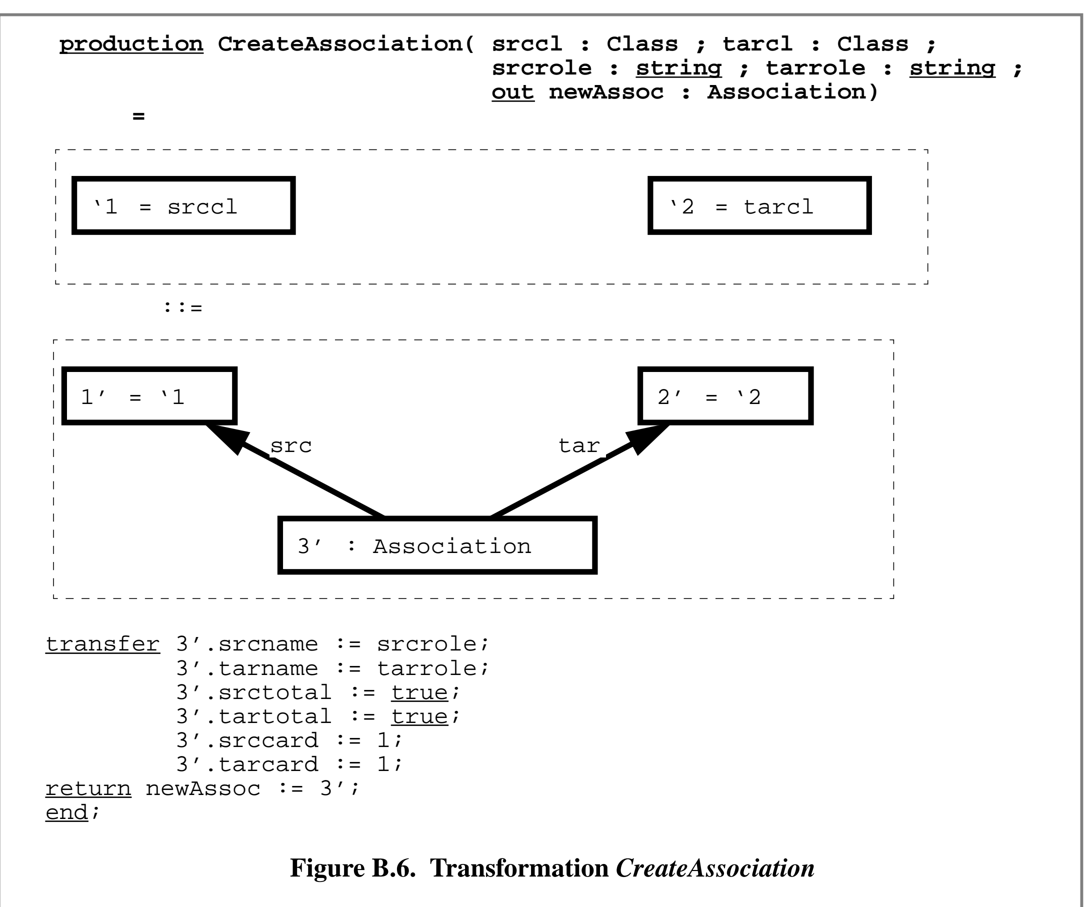


Figure B.6. Transformation *CreateAssociation*

CreateAttribute

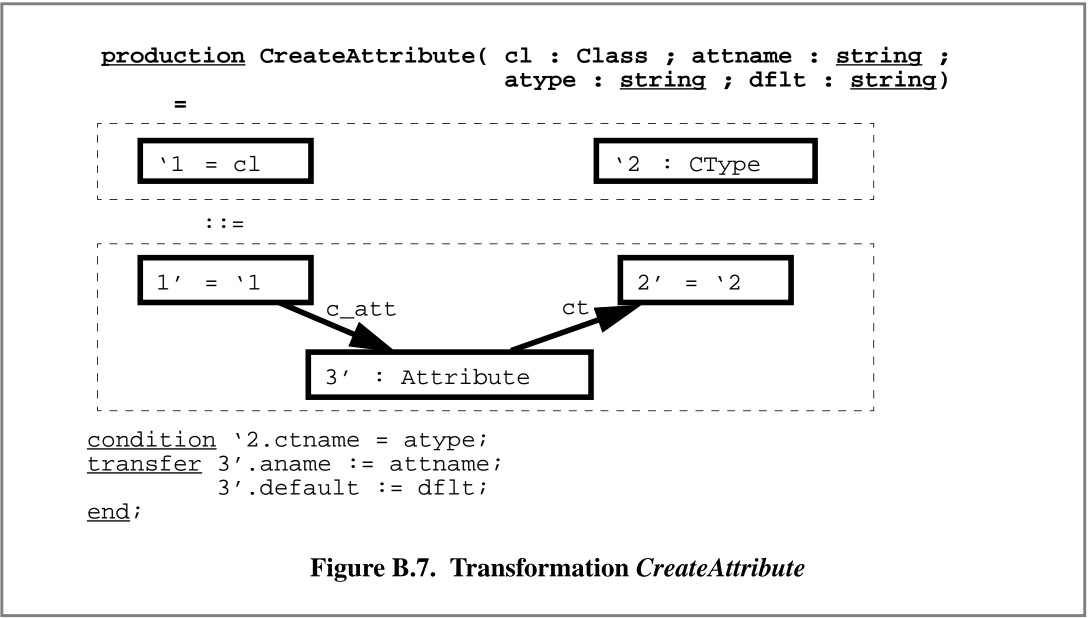


Figure B.7. Transformation *CreateAttribute*

The following transformations *CreateClass*, *CreateAttribute*, *CreateAssociation*, *CreateKey*, and *CreateInheritance* extend the conceptual schema by a new class, attribute, association, key, and inheritance relationship, respectively.

CreateClass

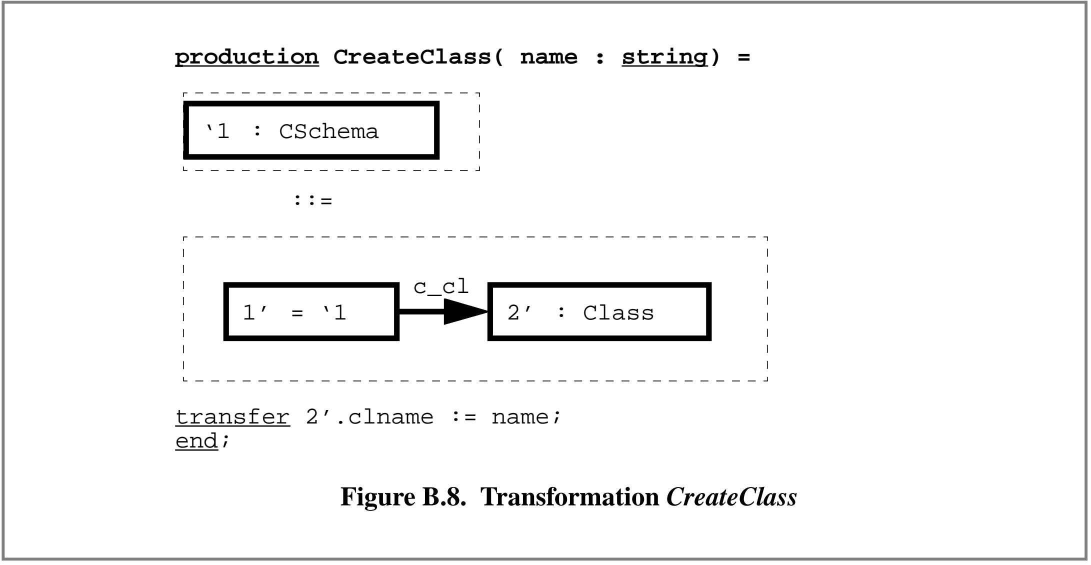


Figure B.8. Transformation *CreateClass*

CreateInheritance

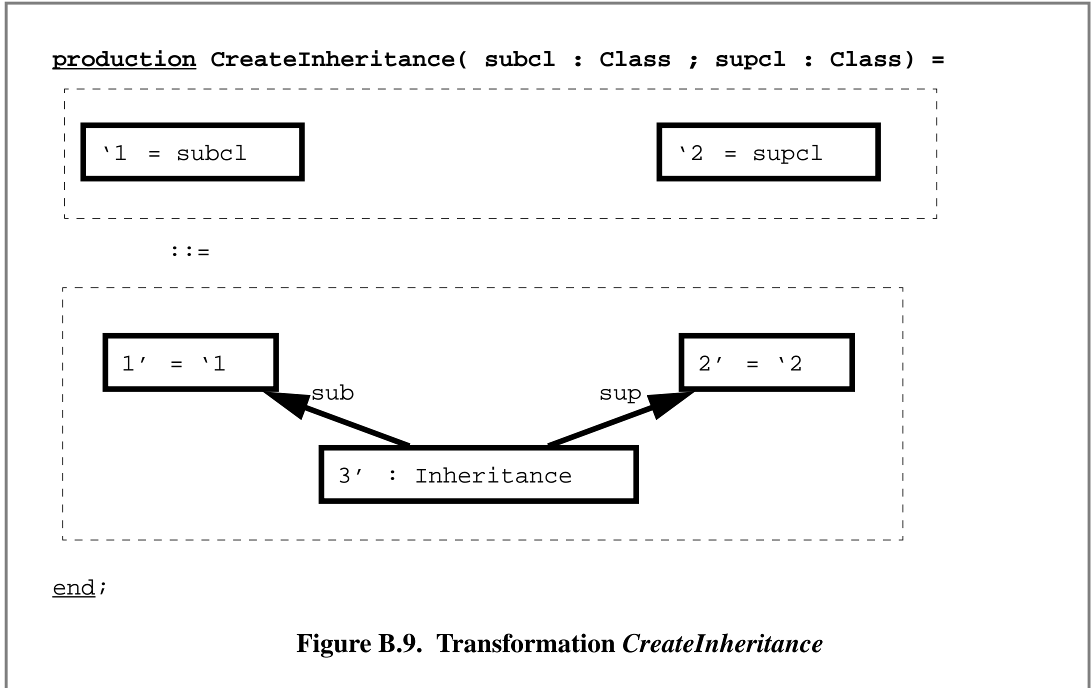


Figure B.9. Transformation *CreateInheritance*

CreateKey

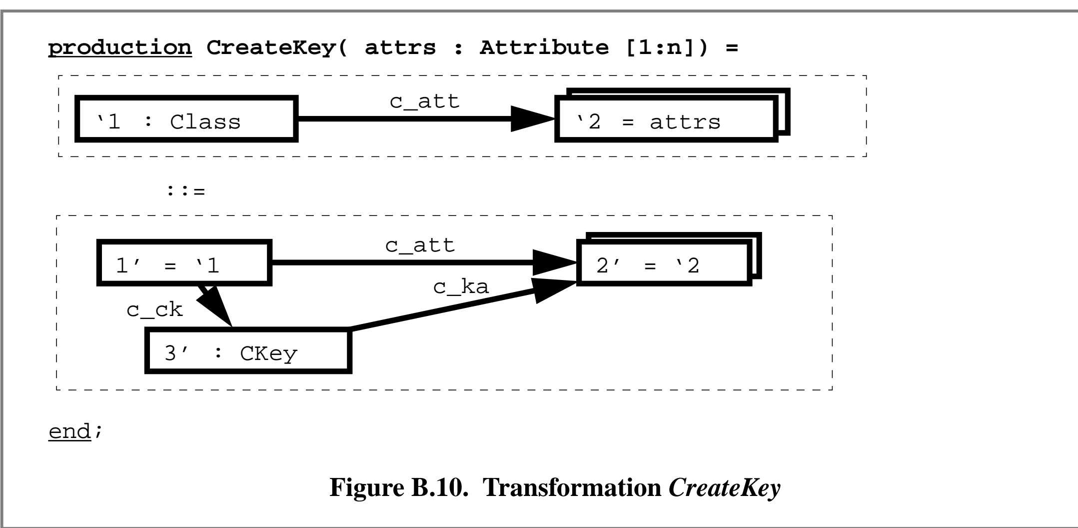


Figure B.10. Transformation *CreateKey*

ConvertAbstract

Transformation *ConvertAbstract* transforms a given concrete class *cl* to an abstract class. Figure B.11 shows that the variant that has been mapped to *cl* (node '2) is removed from the logical schema and the (new) abstract class is mapped to all variants (node set '4) which have commonly been mapped to all subclasses of *cl*.

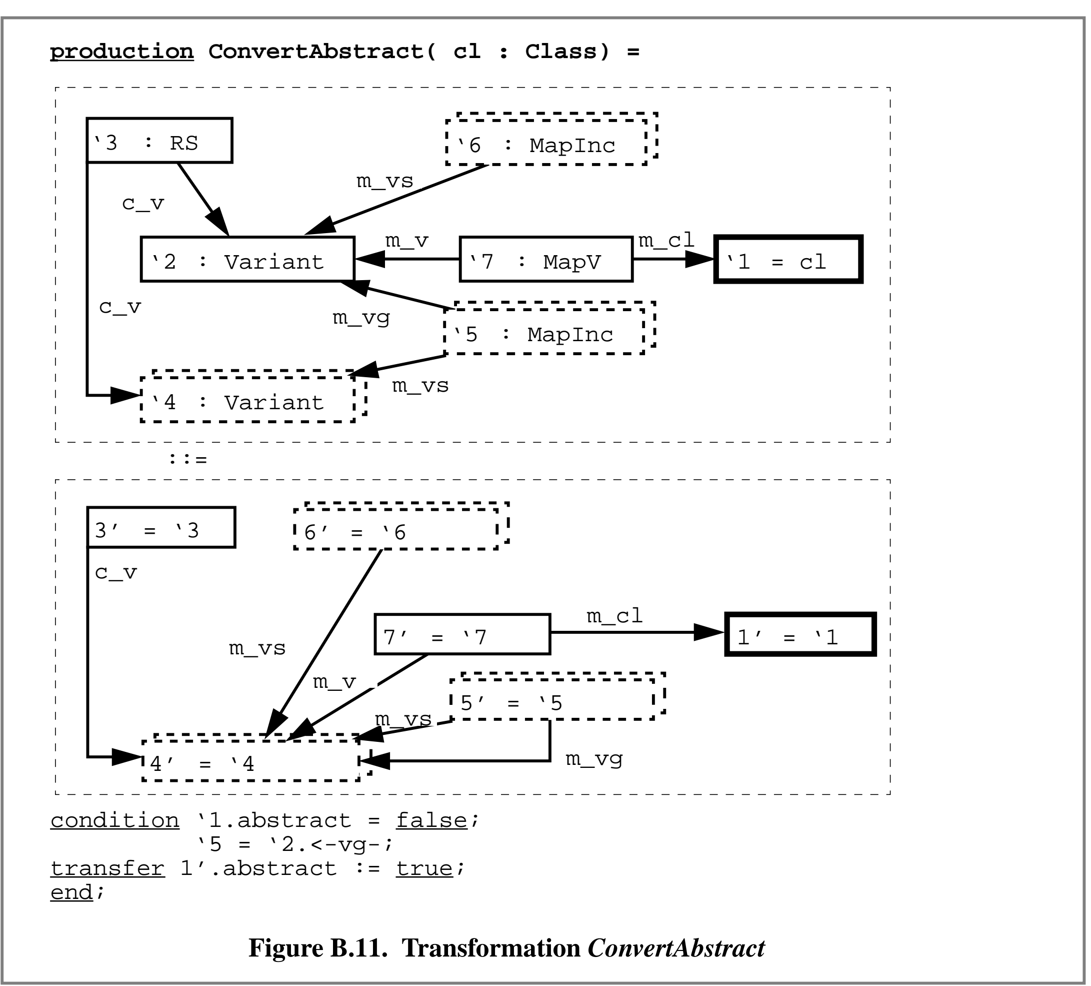


Figure B.11. Transformation *ConvertAbstract*

ConvertConcrete

Production *ConvertConcrete* specifies the reverse transformation for the previous transformation *ConvertAbstract*, i.e., it converts an abstract class to a concrete class. Figure B.12 shows that a new variant (9') is added to represent instances of the (new) concrete class. This new variant includes all foreign keys (4') and columns (8') which are common to all variants that were mapped to the former abstract class.

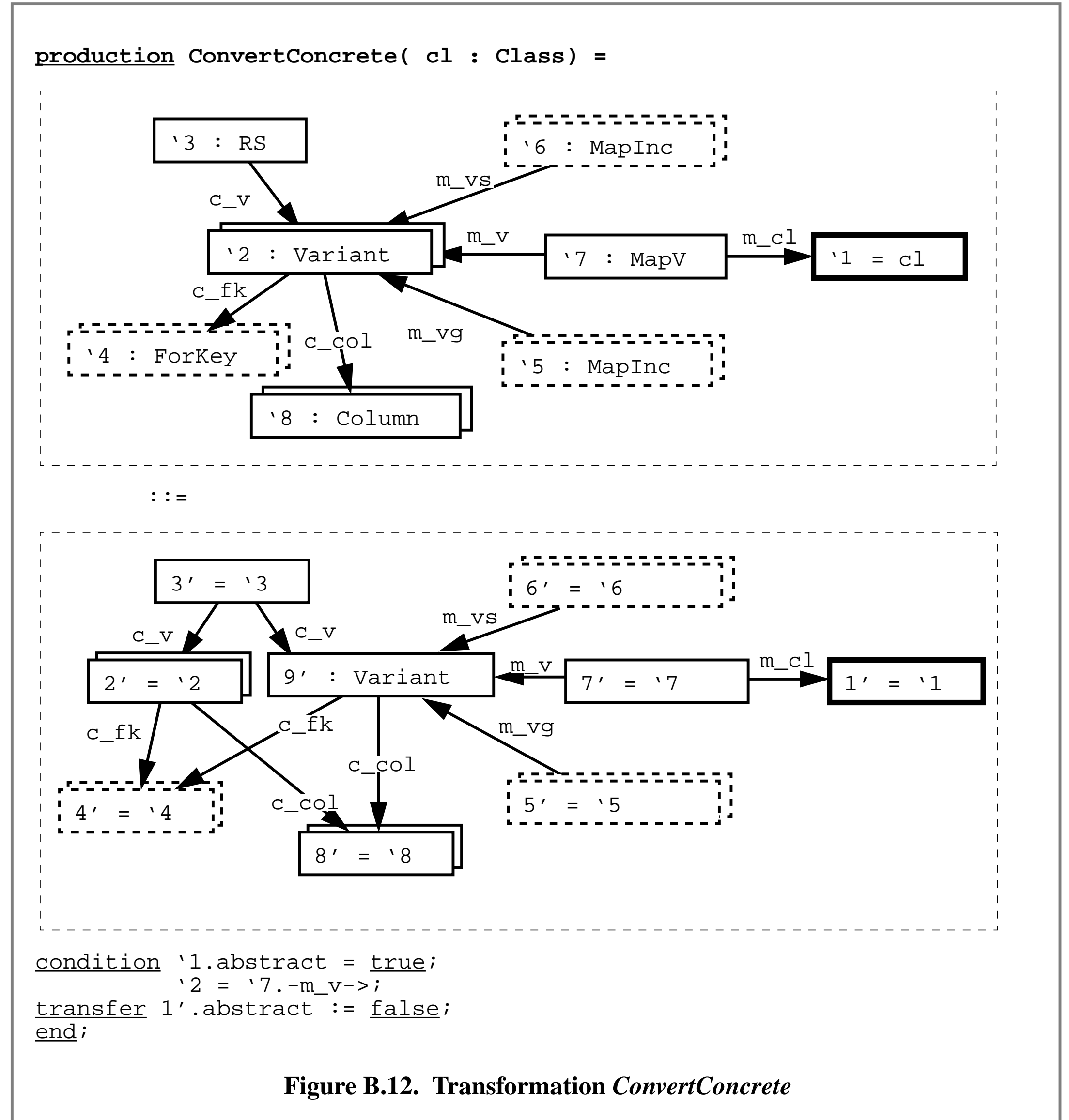


Figure B.12. Transformation *ConvertConcrete*

DisAggregate

Production *Disaggregate* specifies the inverse transformation for transformation *Aggregate*, i.e., it transforms an aggregation relationship to an association.

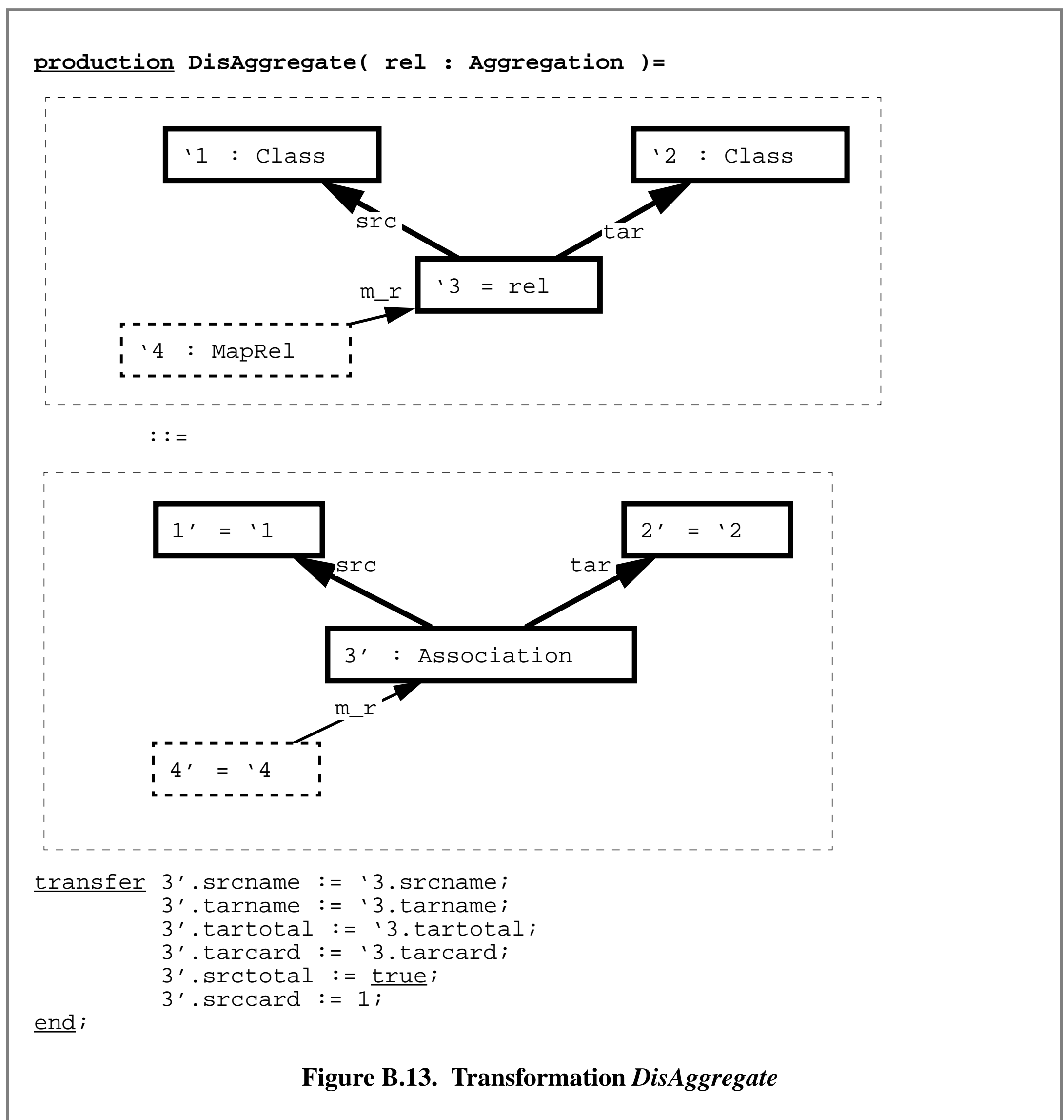


Figure B.13. Transformation *DisAggregate*

Transformation *Generalize* creates a generalization for a given root class, i.e., a class that does not have a superclass (cf. page 142).

Generalize

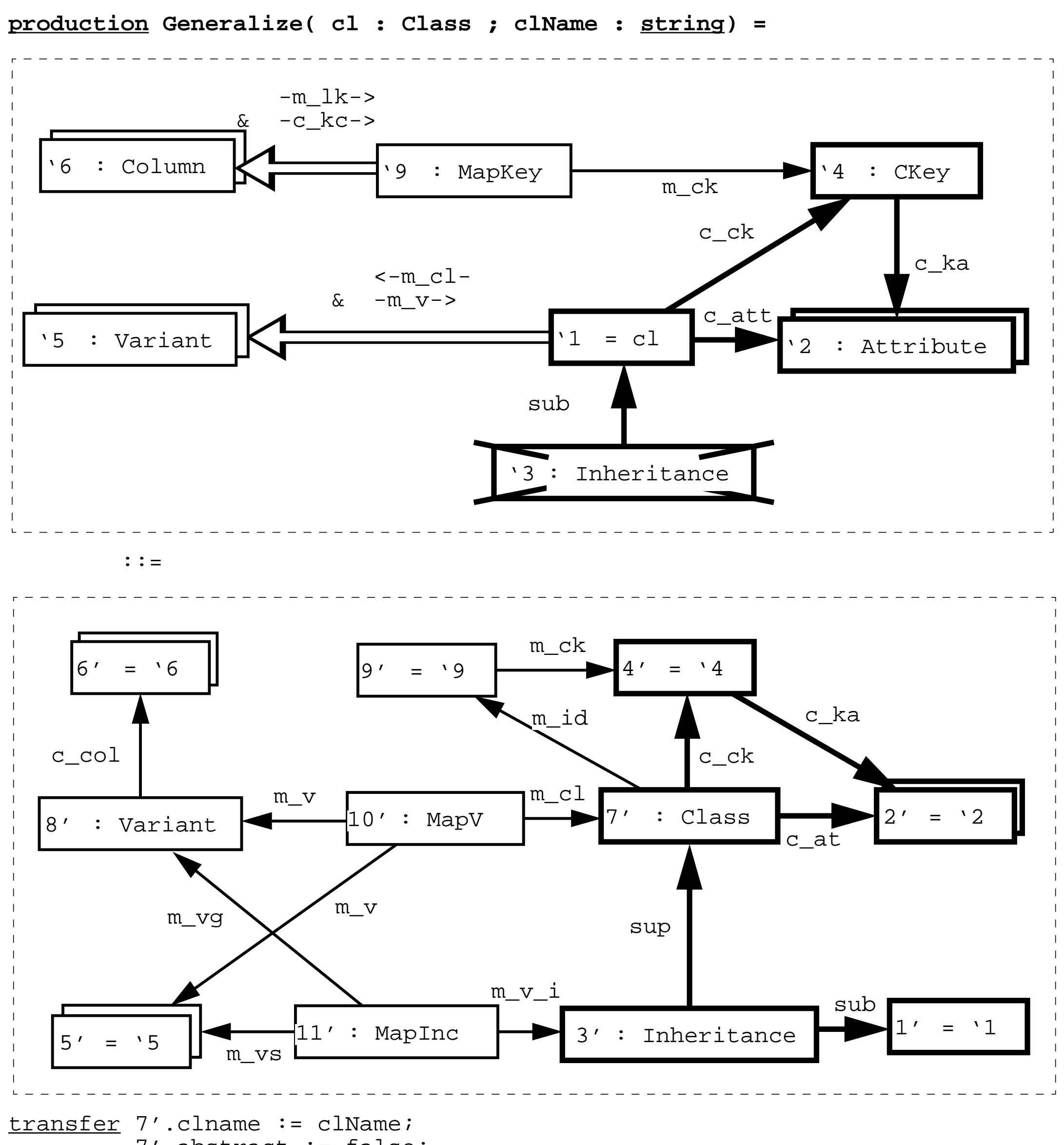


Figure B.14. Transformation *Generalize*

MergeClass

Production *MergeClass* specifies the reverse transformation for transformation *SplitClass*. Note, that one of the two classes to be merged (node ‘3) has to have no other property than the association (*assoc*) that is used for the merge operation. If such properties exist they can be relocated to class ‘2 by using primitive transformations *MoveAttribute* and *MoveAssociation* first.

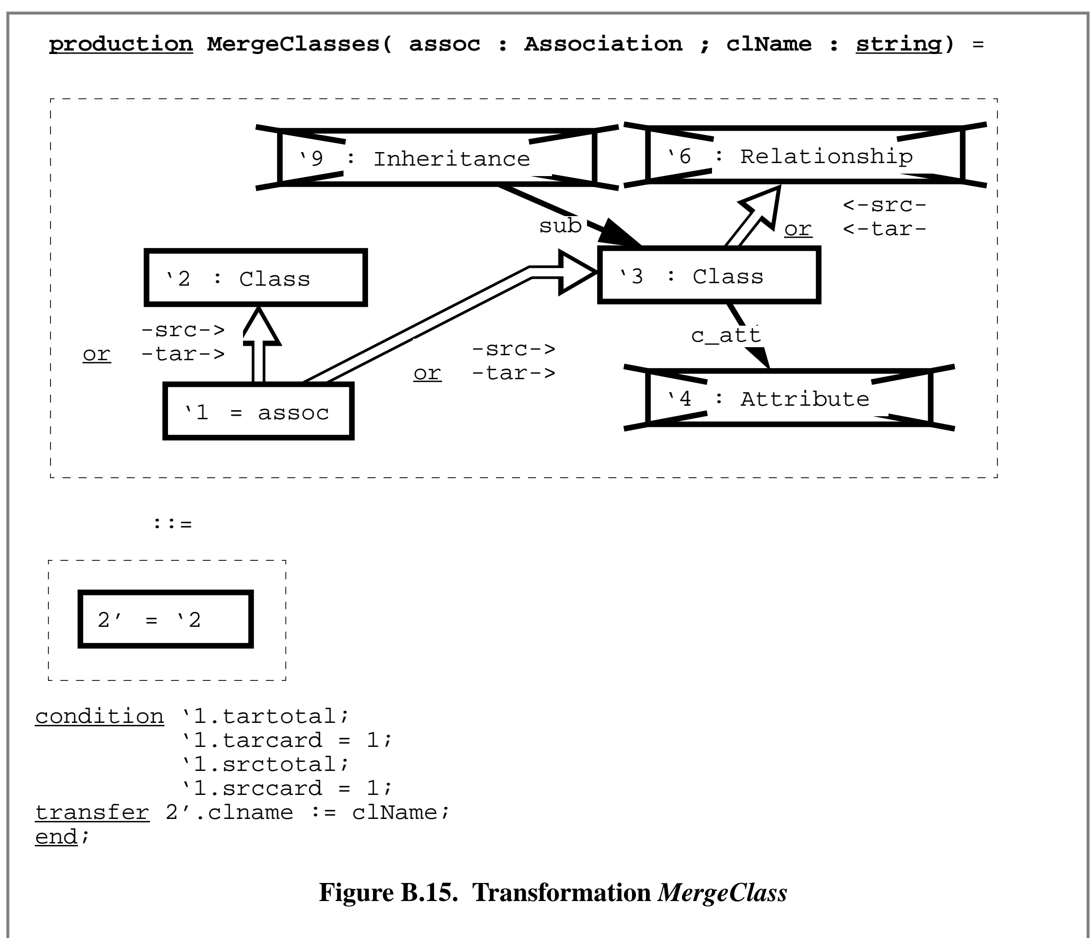


Figure B.15. Transformation *MergeClass*

Transformation *MoveAttribute* relocates an attribute from one class to another class via a given association. This transformation is described in detail on page 140.

MoveAttribute

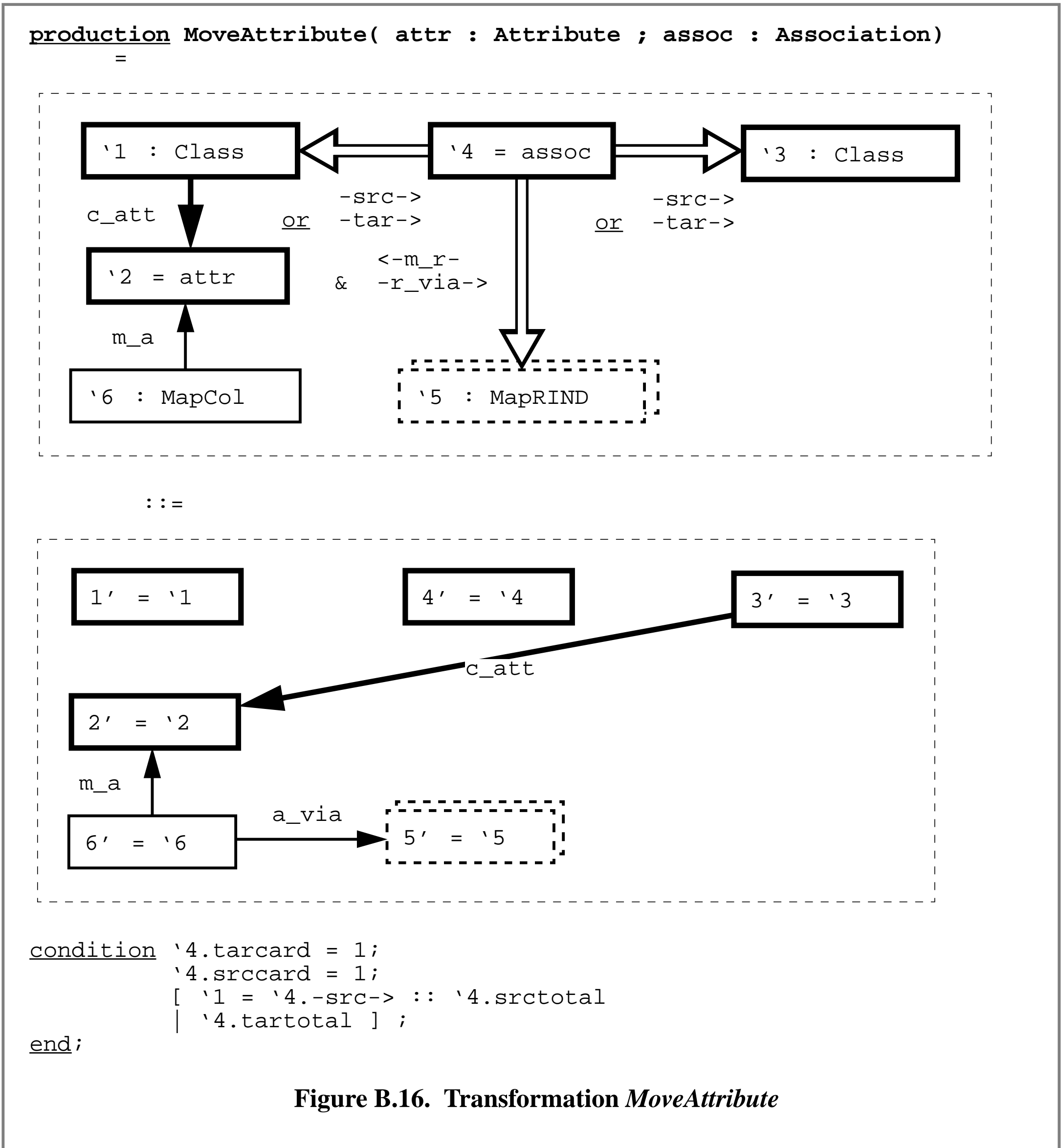


Figure B.16. Transformation *MoveAttribute*

PushDown-Attribute

Transformation *PushDownAttribute* specializes an attribute of a given class to its subclass. In order to avoid the necessity to reorganize data, this transformation is restricted to inheritance relationships that have been mapped to variants of the same RS (cf. page 143). Note, that the negative application condition (node ‘8) prohibits that attributes are specialized which belong to the key of the class.

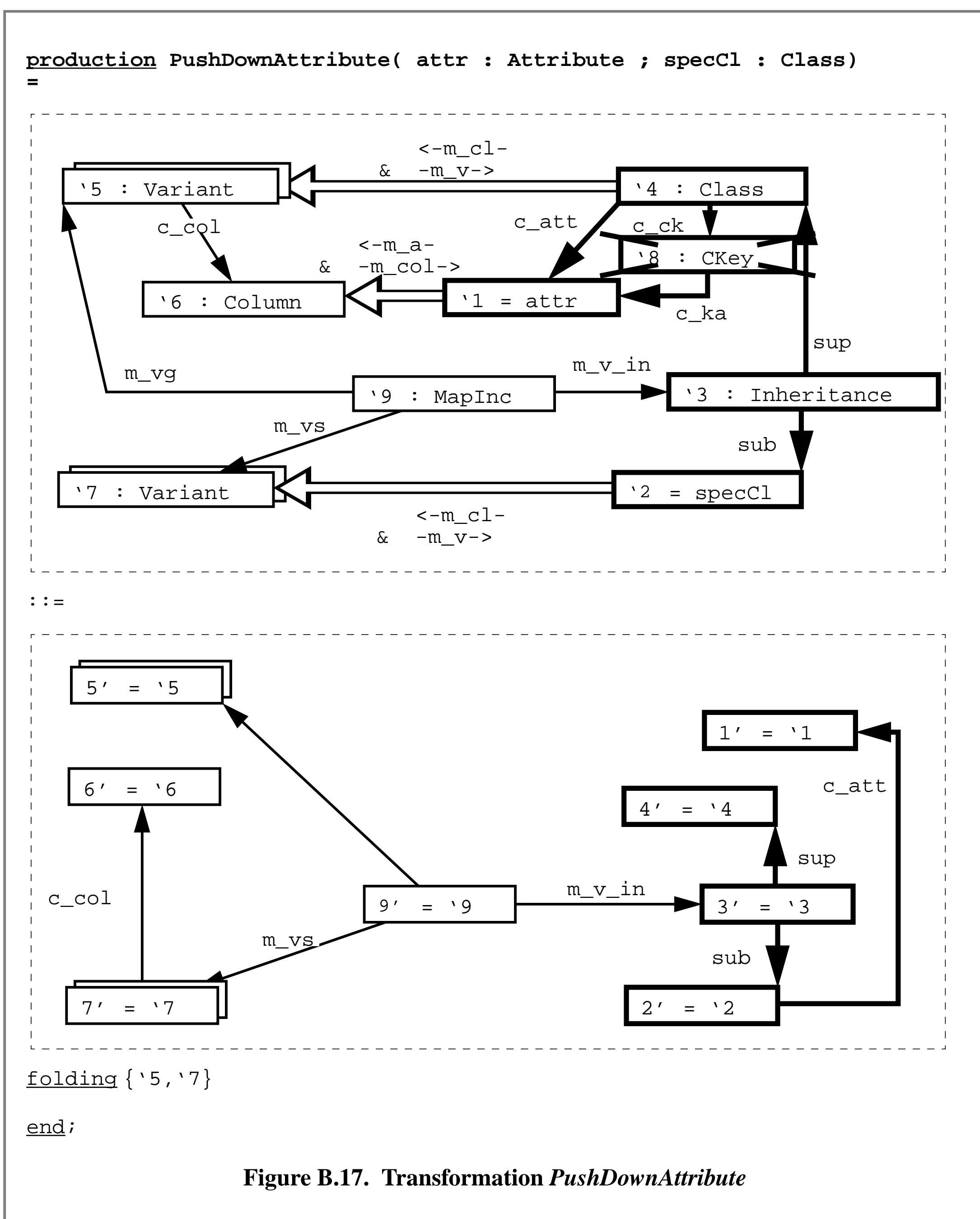


Figure B.17. Transformation *PushDownAttribute*

In analogy to the transformation *PushDownAttribute*, the following transformation *PushDownAssocoation* specializes the source role of an association a given subclass in the inheritance hierarchy.

PushDown-Association

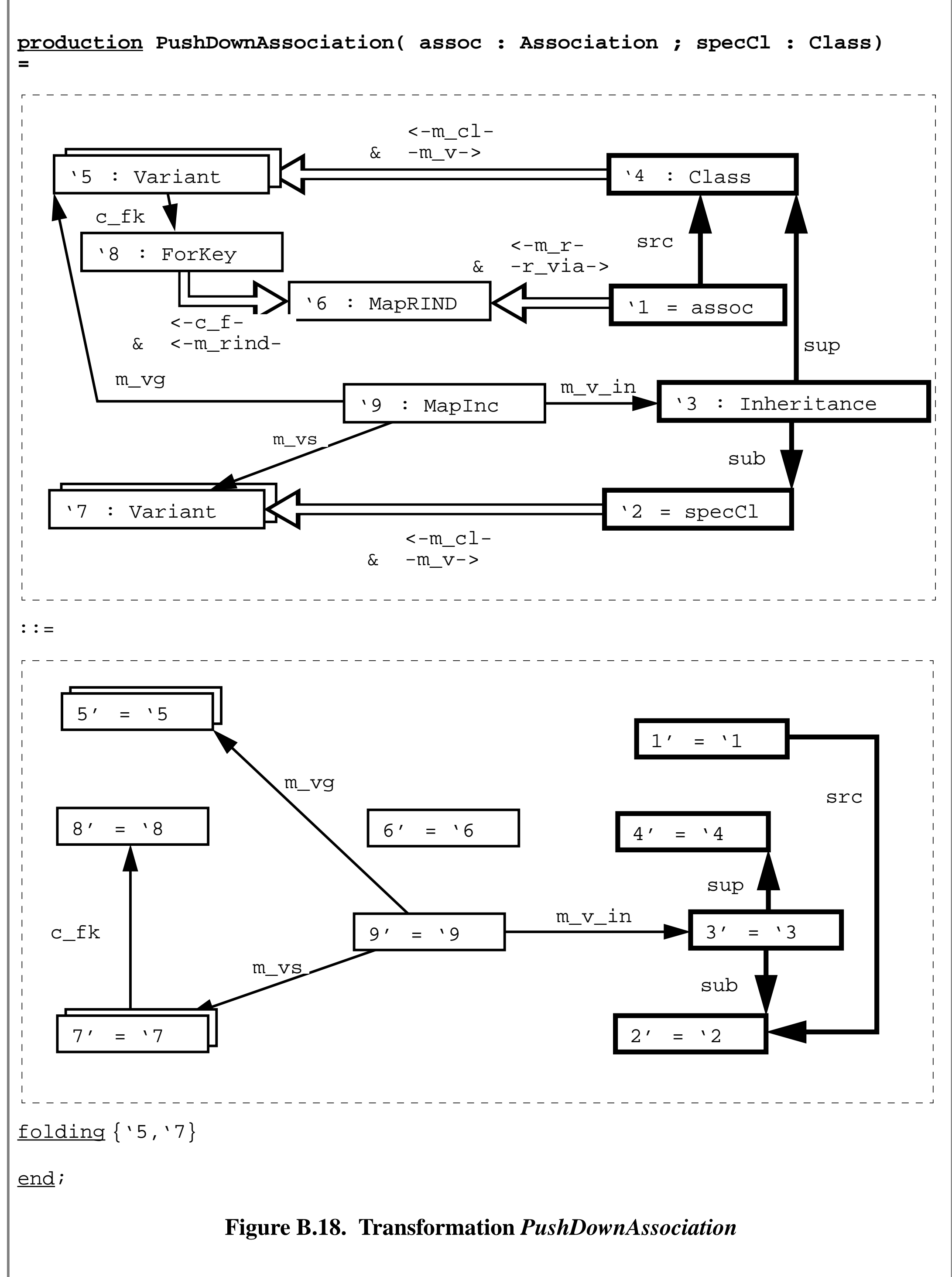


Figure B.18. Transformation *PushDownAssociation*

PushUpAttribute

Transformation *PushUpAttribute* generalizes an attribute of a given class to its superclass (cf. page 143).

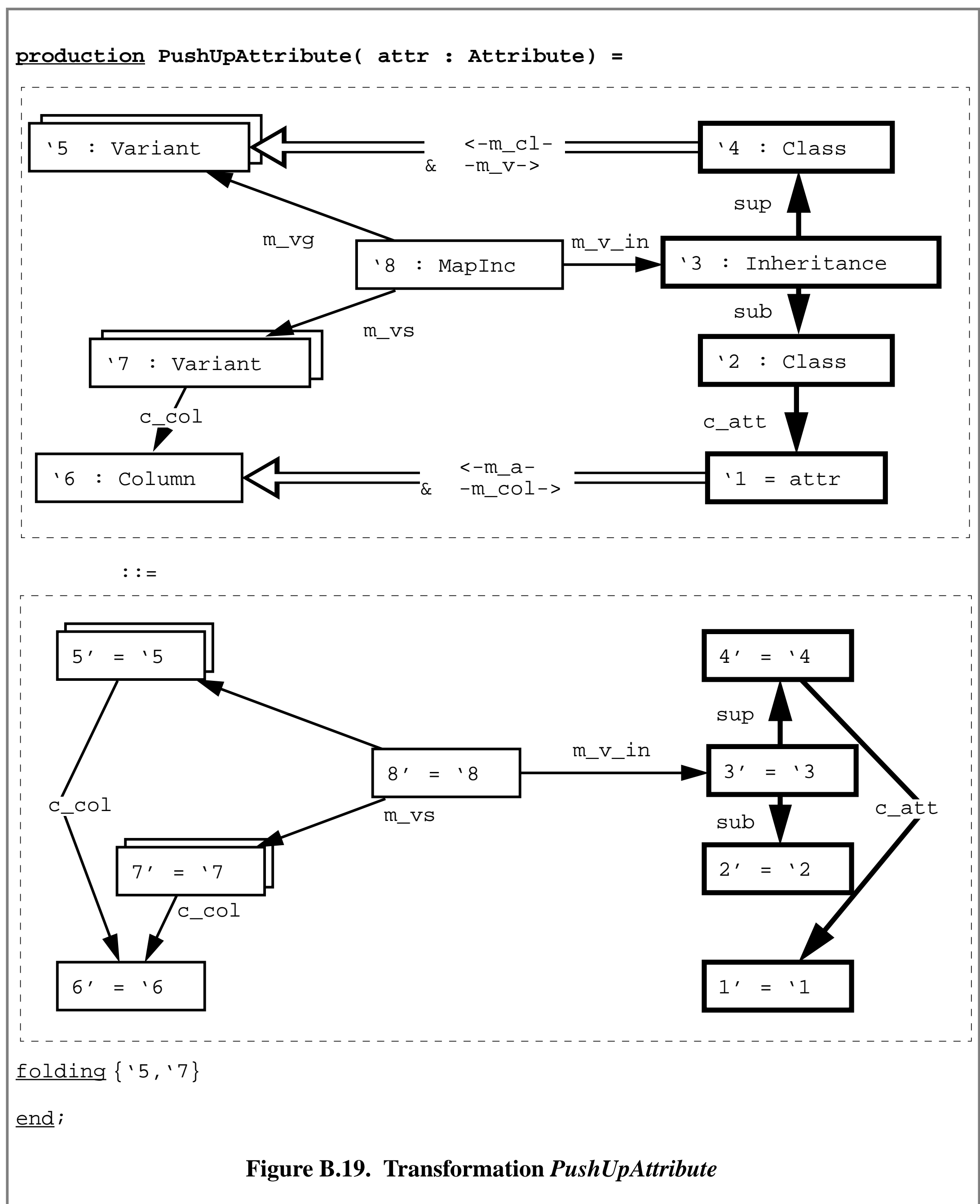


Figure B.19. Transformation *PushUpAttribute*

PushUp-Association

Transformation *PushUpAssocoation* generalizes the source role of an association to the superclass in the inheritance hierarchy.

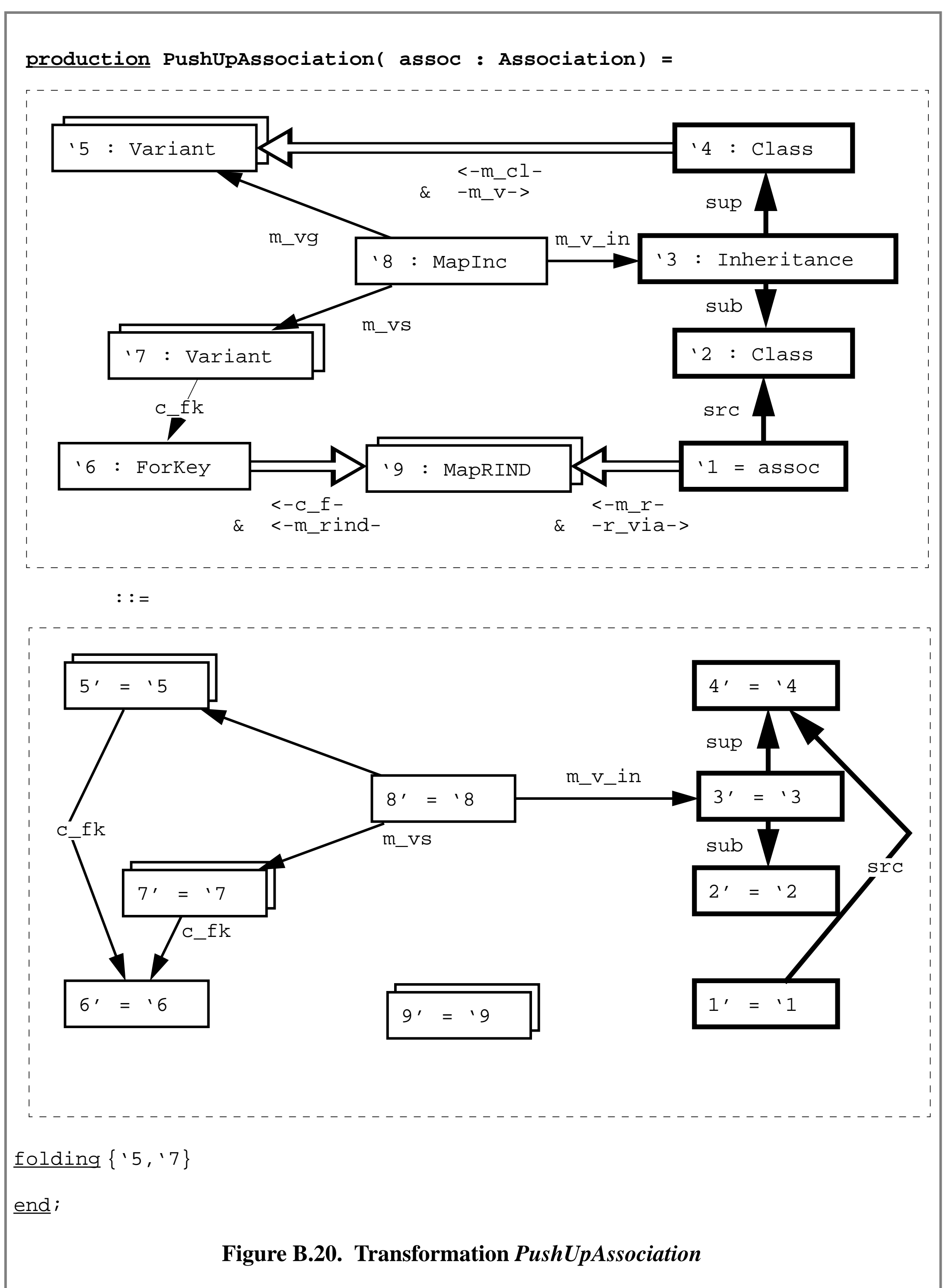


Figure B.20. Transformation *PushUpAssociation*

Remove

Transformation *Remove* deletes an increment from the conceptual schema.

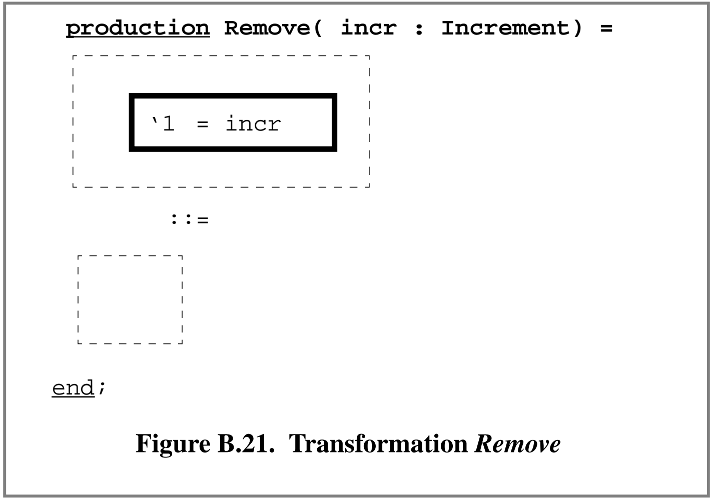


Figure B.21. Transformation *Remove*

RenameClass

The following transformations *RenameClass*, *RenameAttribute*, and *RenameRelationship* change the names of classes, attributes, and relationships, respectively.

```
production RenameClass( cl : Class ; newName : string) =
    `1 = cl
  ::=
    1' = `1
transfer 1'.clname := newName;
end;
```

Figure B.22. Transformation *RenameClass*

RenameAttribute

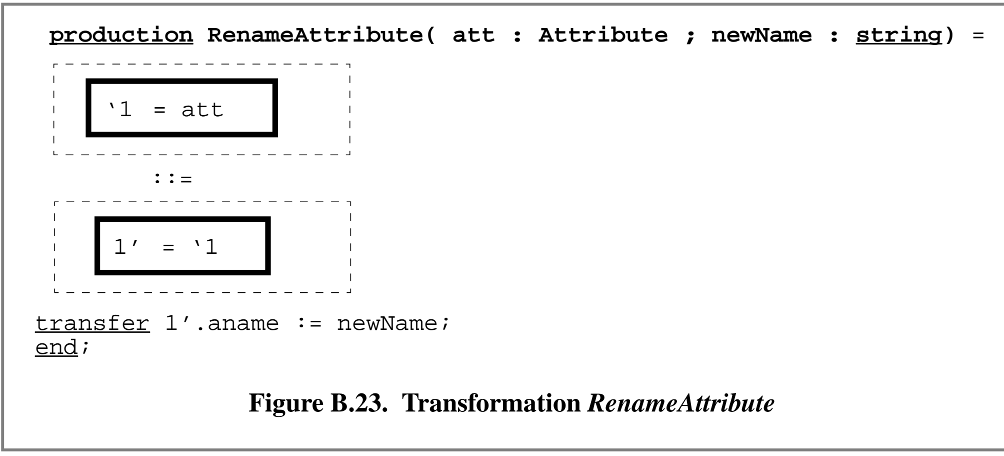


Figure B.23. Transformation *RenameAttribute*

Rename-Relationship

```
production RenameRelationship( rel : Relationship ; newSrcname,
                               newTarname : string) =

  `1 = rel

     ::=

  1' = `1

transfer 1'.srcname := newSrcname;
         1'.tarname := newTarname;
end;
```

Figure B.24. Transformation *RenameRelationship*

SplitClass

Transformation *SplitClass* splits a given class in two classes which are connected by a *one-to-one* associations. This transformation has been described in detail in Section 5.3.

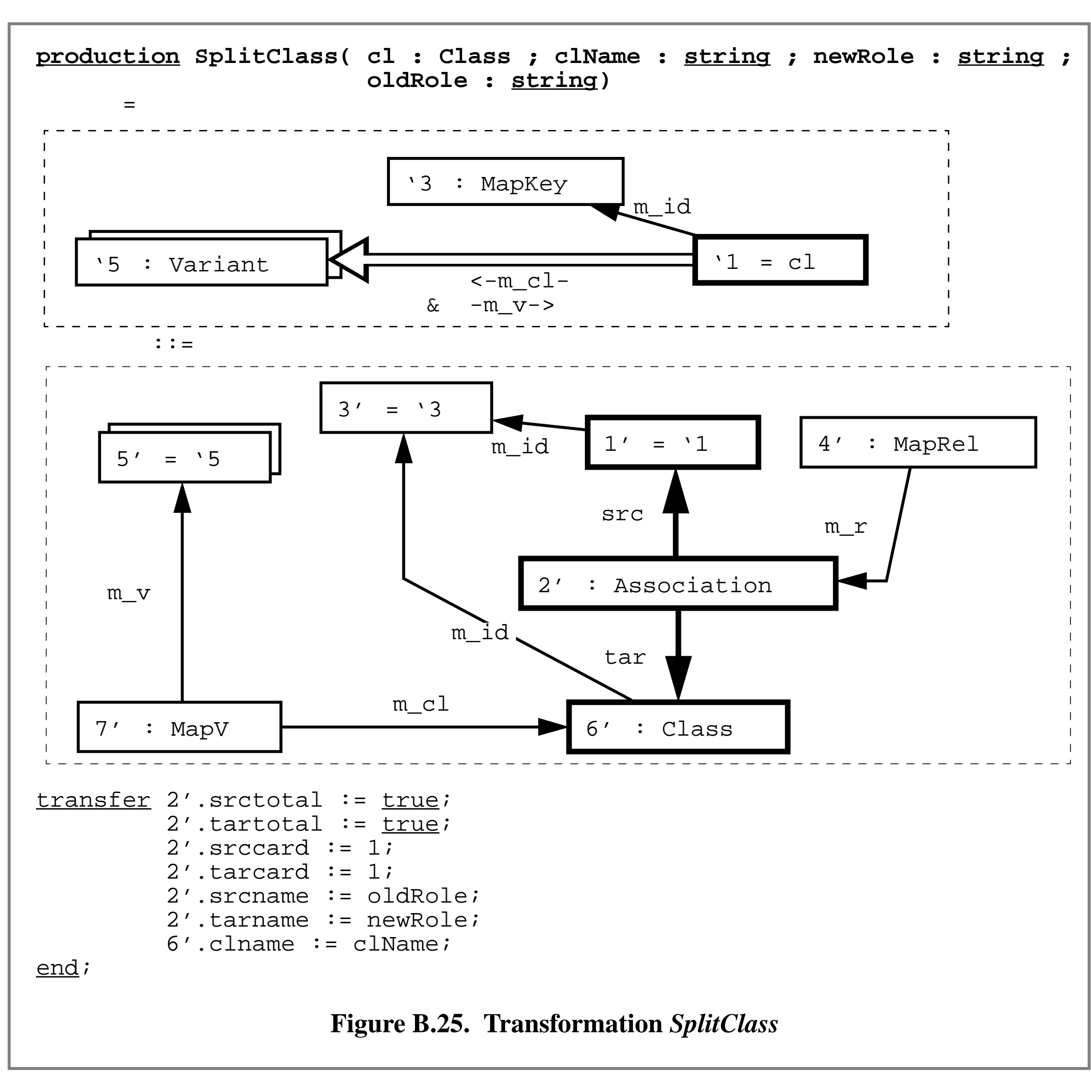


Figure B.25. Transformation *SplitClass*

Specialize

Transformation *Specialize* creates a specialization for a given class.

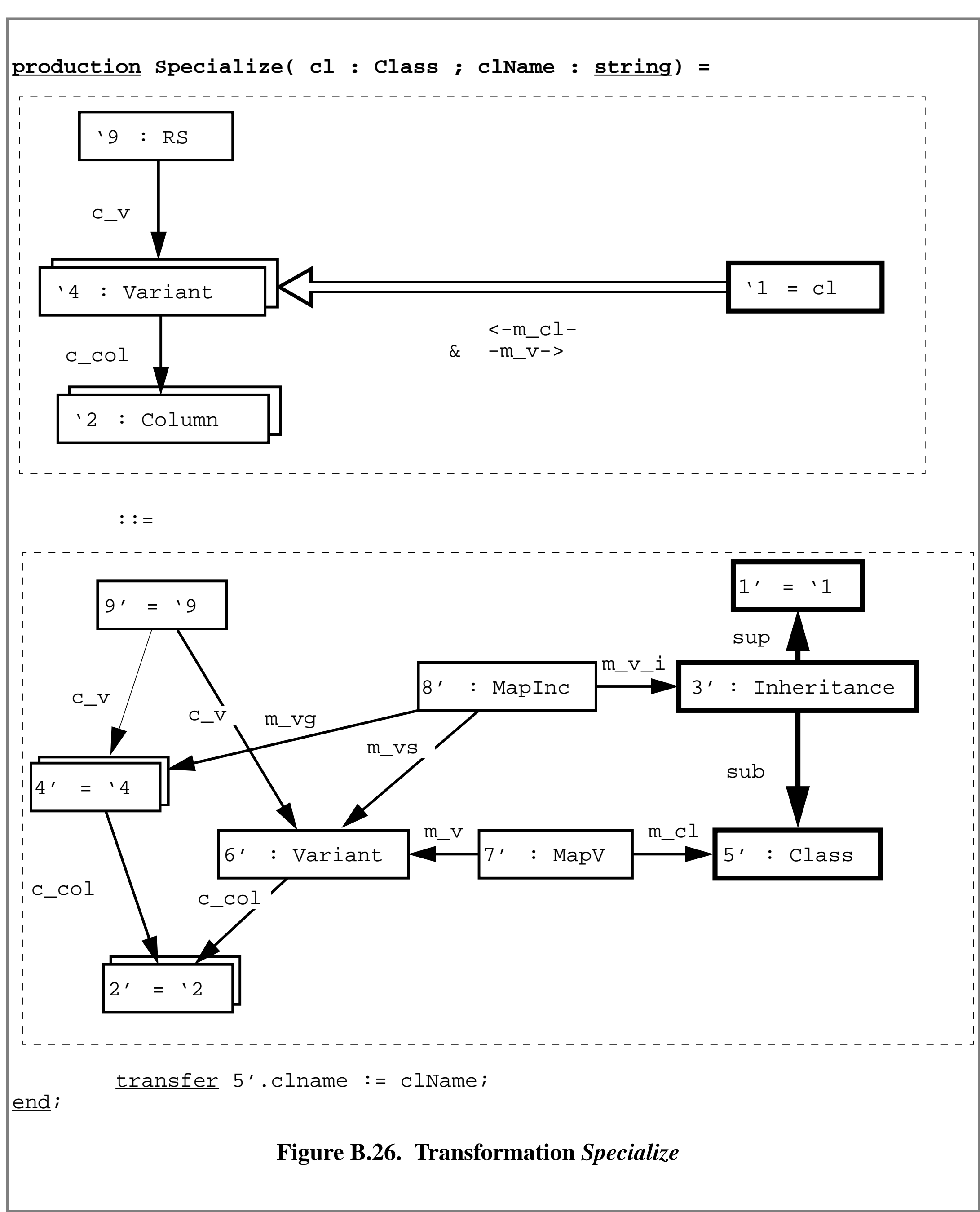


Figure B.26. Transformation *Specialize*

SwapAssoc-Direction

Transformation *SwapAssocDirection* swaps source and target of a given associations.

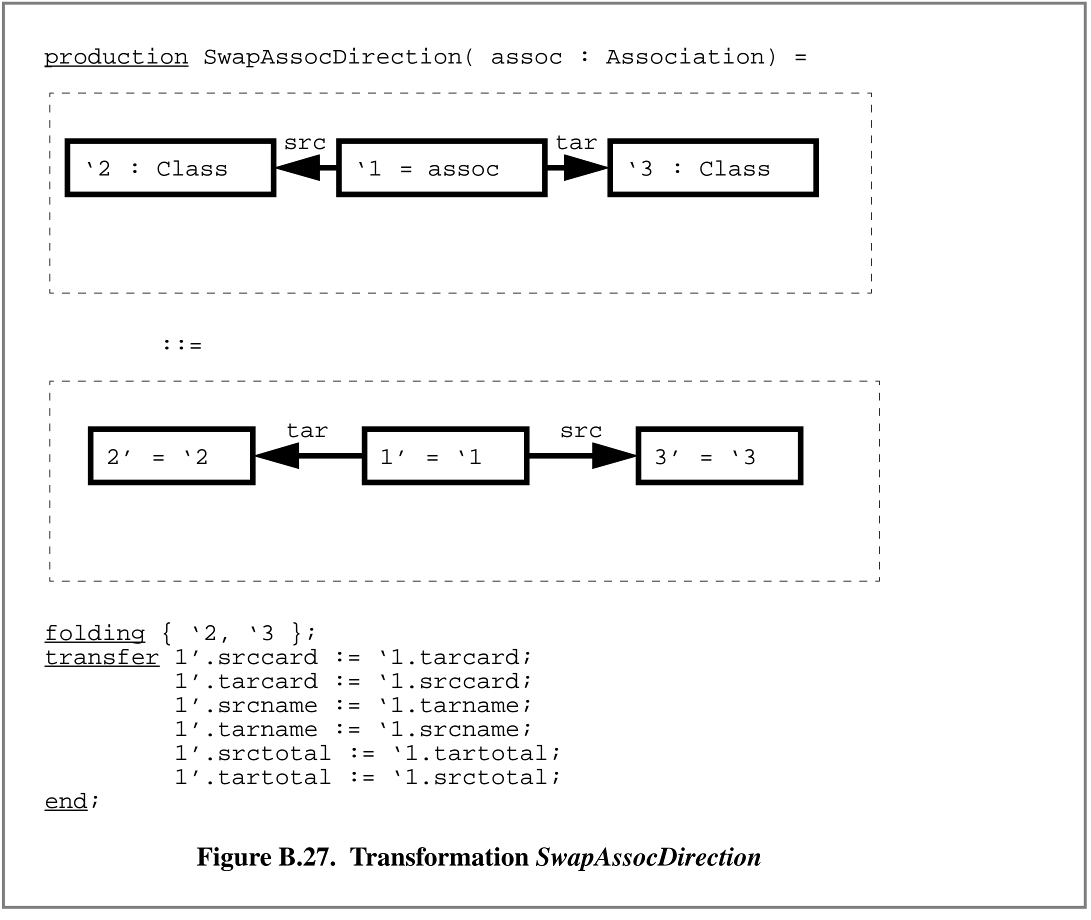


Figure B.27. Transformation *SwapAssocDirection*

REFERENCES

[Ada76] J.B. Adams. A probability model of medical reasoning and the mycin model. *Mathematical Biosience*, 32:177–186, 1976.

[AdBHL86] E. H. L. Aarts, F. M. J. de Bont, J. H. A. Habers, and P. J. M. Laarhoven. A parallel statistical cooling algorithm. In *3rd Annual Symposium on Theoretical Aspects of Computer Science, Orsay, France*, *Lecture Notes in Computer Science*. Springer Verlag, 1986.

[AEP96] J. M. Antis, S. G. Eick, and J. D. Pyrce. Visualizing the Structure of Large Relational Databases. *IEEE Software*, pages 72–79, 1996.

[AG96] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. of the 18th Int. Conf. on Software Engineering, Berlin, Germany*, pages 16–27. IEEE Computer Society Press, 1996.

[AGM85] C. A. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: partial meet contraction and revision functions. *The Journal of Symbolic Logic*, 50:510–530, 1985.

[Aik95] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[AL94] D. Aebi and R. Largo. Methods and tools for data value re-engineering. In *Applications of Databases (ADB-94)*, volume 819 of *Lecture Notes in Computer Science*, pages 400–411. Springer Verlag, 1994.

[ALV93] F. Abbattista, F. Lanubile, and G. Visaggio. Recovering conceptual data models is human-intensive. In *Proc. of 5th Intl. Conf. on Software Engineering and Knowledge Engineering, San Francisco, California, USA*, pages 534–543, 1993.

[AMR94] P. Aiken, A. Muntz, and R. Richards. DoD legacy systems: reverse engineering data requirements. *Communications of the ACM*, 37(5):26–41, 1994.

[And94] M. Andersson. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In *Proc. of the 13th Int. Conference of the Entity Relationship Approach, Manchester*, volume 881 of *Lecture Notes of Computer Science*, pages 403–419. Springer Verlag, 1994.

[AT98] M. N. Armstrong and C. Trudeau. Evaluating architectural extraction tools. In *Proc. of the 5th Working Conference on Reverse Engineering, Hawaii, USA*, pages 30–39. IEEE Computer Society Press, 1998.

[Bau94] M. Bauer. Integrating probabilistic reasoning into plan recognition. In *Proc. of the 11th European Conference on Artificial Intelligence (ECAI '94)*, pages 620–624. John Wiley & Sons, 1994.

[Bau95] M. Bauer. A Dempster-Shafer approach to modeling agent preferences for plan recognition. *User Modeling and User-Adapted Interaction*, 5:317–348. Wolters Kluwer Publishers, 1995.

[BB92] L. Bolc and P. Borowik. *Many-valued Logics: Theoretical Foundations*. Springer Verlag, Berlin, 1992.

[BB94] A. J. Bugarin and S. Barro. Fuzzy reasoning supported by petri nets. *IEEE Transactions on Fuzzy Systems*, 2(2):135–150, 1994.

[BC90] T. J. M. Bench-Capon. *Knowledge Representation - An Approach to Artificial Intelligence*. Academic Press, London, 1990.

[BCN92] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database design*. Benjamin/Cummings, 1992.

[BDH+87] H. Briand, C. Ducateau, Y. Hebrail, D. Herin-Aime, and J. Kouloumdjian. From Minimal Cover to Entity-Relationship Diagram. In *Proc. of the 6th Intl. Conference of the Entity Relationship Approach, New York*, pages 287–304. North-Holland, 1987.

[BED94] J. S. Bowman, Sandra L. Emerson, and M. Darnovsky. *The Practical SQL Handbook - Using Structured Query Language*. Addison-Wesley Developers Press, Reading, MA, USA, 1994.

[Ber80] J.O. Berger. *Statistical Decision Theory*. Springer Verlag, New York, 1980.

[Bew98] B. Bewermeyer. Cliche-Erkennung in relationalen Datenbankanwendungen. Master's Thesis, University of Paderborn, Dept. of Mathematics and Computer Science, 33095 Paderorn, Germany, 1998.

[BGD97] A. Behm, A, Geppert, and K. R. Dittrich. On the migration of relational schemas and data to object-oriented database systems. In *Proc. 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria*, pages 13-33. Österreichische Computer Gesellschaft, 1997.

[Big90] T. J. Biggerstaff. Human-oriented conceptual abstractions in the reengineering of software. In *Proc. of the 12th International Conference on Software Engineering*, page 120-122. IEEE Computer Society Press, 1990.

[BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record*, 16(3):311–322, 1987.

[BL97] H. Kleine Büning and T. Lettmann. Skriptum zur Vorlesung wissensbasierte Systeme. Scriptum for the class on Knowledge-Based Systems at the University of Paderborn, Dept. of Mathematics and Computer Science, 33095 Paderborn, Germany, 1997.

[Bla98] M. Blaha. On reverse engineering of vendor databases. In *Proc. of the 5th Working Conference on Reverse Engineering*, pages 183–190, Hawai, USA. IEEE Computer Society Press, 1998.

[BM98] E. Baniassad and G. Murphy. Conceptual module querying for software reengineering. In *Proc. of the 20th International Conference on Software Engineering*, pages 64–73. IEEE Computer Society Press, 1998.

[BP95] M. Blaha and W. Premerlani. Observed idiosyncracies of relational database designs. In *Second Working Conference on Reverse Engineering, Toronto, Ontario, Canada.* IEEE Computer Society Press, 1995.

[BP96] M. Blaha and W. Premerlani. A catalog of object model transformations. In *Proc. of 3rd Working Conference on Reverse Engineering, Monterey, California. USA*. IEEE Computer Society Press, 1996.

[BP98] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[BR97] H. Blockeel and L. D. Raedt. Relational knowledge discovery in databases. In *Proc. of the 6th Intl. Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 199–211, Berlin, August 1997. Springer Verlag.

[BRH95] S. Bridges, S. Ramanathan, and J. Hodges. A prototype object-oriented geophysical database system developed by re-engineering a relational database system. Technical Report MSU-950612, Department of Computer Science, Mississippi State University, USA, June 1995.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, USA, 1st edition, 1999.

[Bro96] K. Brown. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical Report TR-96-07, Department of Computer Science, North Carolina State University, 1996.

[BS84] B. G. Buchanan and E. H. Shortliffe, editors. *Rule-Based Expert Systems*. Addison-Wesley, Reading, MA, USA, 1984.

[BS95] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems*. Morgan Kaufmann Publishers, San Francisco, USA, 1995.

[CBB+97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Los Altos, CA, USA, 1997.

[CER99] CERMICS Database Team, *ObjectDRIVER V1.1 User Manual*, 2004 route des lucioles, 06902 Sophia Antipolis Cedex, France, 1999.

[Che76] P. Chen. The Entity-Relationship Model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Chr75] N Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, New York, 1975.

[CI90] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17. IEEE Computer Society Press, 1990.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae,* 26(3):241–265. IOS Press, Amsterdam, 1996.

[Coo90] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[CVD96] J. Cardoso, E. Valette, and D. Dubois. Fuzzy Petri nets - an overview. In *Proc. of the 13th World Congress of the Intl. Federation of Automatic Control, San Francisco*, pages 443–448, 1996.

[CW85] R.T. Clemen and R.L. Winkler. Limits for the precision and value of information from dependent sources. *Operations Research*, 33:427–442, 1985.

[Dat84] C. J. Date. *A Guide to DB2*. Addison Wesley, Reading, MA, USA, 1984.

[Dat89] C. J. Date. *A Guide to the SQL standard*. Addison Wesley, Reading, MA, USA, 1989.

[DD92] D. Driankov and P. Doherty. A nonmonotonic fuzzy logic. In L. A. Zadeh and J. Kacprzyk, editors, *Fuzzy Logic for the Management of Uncertainty*, pages 171–190. John Wiley & Sons, 1992.

[DLP92] D. Dubois, J. Lang, and H. Prade. Dealing with multi-source information in possibilistic logic. In *Proc. of the 10th European Conference on Artificial Intelligence*, pages 38–42, Vienna, Austria. John Wiley & Sons, 1992.

[DLP94] D. Dubois, J. Lang, and H. Prade. Possibilistic Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 439–503, Clarendon Press, Oxford, 1994.

[DP83] D. Dubois and H. Prade. Unfair coins and necessity analysis: Towards a possibilistic interpretation of histograms. *Fuzzy Sets and Systems*, 10(1):15–20, 1983.

[DP88] D. Prade and H. Prade. An introduction to possibilistic and fuzzy logics. In P. Smets, E. H. Mamdani, D. Dubois, and H. Prade, editors, *Non-Standard Logics for Automated Reasoning*, pages 287–326. Academic Press, London, 1988.

[DP97] D. Dubois and H. Prade. Synthetic view of belief revision with uncertain inputs in the framework of possibility theory. *International Journal Of Approximate Reasoning*, 17(2-3), pages 295–324, 1997.

[EKR97] G. Ehmayer, G. Kappel, and S. Reich. Connecting databases to the web - a taxonomy of gateways. In *Proc. of the 8th International Conference on Database and Expert Systems Applications, Toulouse, France*, volume 1308 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 1997.

[EM77] H. Ebrahim and D. Mamdani. Application of fuzzy logic to approximate reasoning. *IEEE Transactions on computers,* volume 26, 1977.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994.

[Eng86] G. Engels. *Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung*. Ph.D. Thesis, Universität Osnabrück. VDI-Verlag, 1986.

[Eng98] V. Englebert. *Voyager 2 (version 4.0) - Reference manual*. Institut d'Informatique, University of Namur, Belgium, rue grandgaggnage B-5000 Namur, Belgium, 1998.

[Fen67] J. E. Fenstad. Representations of probabilities defined on first-order languages. In J. N. Crossley, editor, *Sets, models and recursion theory*. North-Holland, 1967.

[FG90] C. Froidevaux and C. Grossete. Graded default theory for uncertainty. In *Proc. of the 9th European Conference on Artificial Intelligence, Stockholm, Sweden*, pages 283–288. Pitman, London, 1990.

[FH97] J. S. P. Fong and S.-M. Huang. *Information Systems Reengineering*. Springer Verlag, Singapore, 1997.

[FHK+97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564-593, 1997.

[Fla97] D. Flanagan. *Java in a Nutshell: a desktop quick reference*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2nd edition, 1997.

[Fon97] J. Fong. Converting relational to object-oriented databases. *ACM SIGMOD Record*, 26(1), 1997.

[Fou92] Open Software Foundation. *Introduction to OSF/DCE*. Prentice Hall, New Jersey, 1992.

[Fry95] B. Fryer. Prudential gets healthy. *Information Week*, pages 60–64, 1995.

[FS97] A. Fay and E. Schnieder. Fuzzy petri nets for knowledge representation and reasoning in rule-based systems. In *Proc. of the 2nd Intl. ICSC Symposium on Fuzzy Logic and Applications, Zurich*, pages 146–150, 1997.

[FS98] A. Fay and E. Schnieder. On the combination of expert systems and petri nets. In *Proc. of the 7th Intl. Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems. Paris, La Sorbonne*, pages 1626–1632, 1998.

[Fus97] M. L. Fussell. Foundations of object relational mapping. 1220 N. Fair Oaks Ave, #1314, Sunnyvale, CA 94089, 1997.

[FUZ98] *Proc. of 7th IEEE Intl. Conf. of Fuzzy Systems. Anchorage, USA*. IEEE, 1998.

[FV95] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. In *Proc. of the 4th Intl. Conference on Deductive and Object-Oriented Databases*, 1995.

[Gal93] S. I. Gallant. *Neural Network Learning and Expert Systems*. The MIT Press, Cambridge, MA, USA, 1993.

[Gär75] P. Gärdenfors. Qualitative probability as an intensional logic. *Journal of Philosophical Logic*, 4(2):171–185, 1975.

[Gei95] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, Reading, MA, USA, 1997.

[GK93] H. Gall and R. Klösch. Capsule oriented reverse engineering for software reuse. In *Proc. of the European Conference on Software Engineering*, volume 717 of *Lecture Notes in Computer Science*, pages 418–433. Springer Verlag, 1993.

[Got88] S. Gottwald. *Mehrwertige Logik*. Akademie-Verlag, Berlin, Germany, 1988.

[Gra95] A. Grauel. *Fuzzy-Logik*. BI, Braunschweig, Germany, 1995.

[Gre98] R. Grehan. Object marries relational — Ardent's Java Relational Binding turns a relational database into a Java object-oriented database management system. *Byte Magazine*, 23(3):101–102, 1998.

[Gro98] K. Grotenhuis. Crossing the Euro rubicon. *IEEE Spectrum*, 35(10):30–33, 1998.

[Hai89] J.-L. Hainaut. A generic entity-relationship model. In *Information System Concepts: An In-depth Analysis*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1989.

[Hai91] J-L. Hainaut. Entity-generating schema transformations for entity-relationship models. In *Proc. of the 10th Conference on the Entity-Relationship Approach, San Mateo*. Springer Verlag, 1991.

[Haj94] P. Hajek. On logics of approximate reasoning. In *Knowledge Representation and Reasoning Under Uncertainty*, volume 808 of Lecture Notes in Artificial Intelligence, pages 17–29. Springer Verlag, 1994.

[Hal90] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990.

[HB86] M. Haber and M. B. Brown. Maximum likelihood methods for log-linear models when expected frequencies are subject to linear constraints. *Journal of the American Statistical Association*, 81(394):477–482, 1986.

[HCTJ93] J-L. Hainaut, M. Chandelon, C. Tonneau, and M. Joris. Contribution to a theory of database reverse engineering. In *First Working Conference on Reverse Engineering, Baltimore, USA*, pages 161-170. IEEE Computer Society Press, 1993.

[HEH+96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering*, 3(1-2), 1996.

[HEH+98] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, and J.-L. Hainaut. Program understanding in database reverse engineering. In *Proc. of 9th International Conference on Database and Expert Systems Applications, Vienna, Austria*, volume 1460 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[Hei98] M. Heitbreder. Eine Ausführungsmaschine für Generic Fuzzy Reasoning Nets auf Basis unscharfer Petrinetze. Master's Thesis, University of Paderborn, Dept. of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1998.

[Her94] D. Hernandez. Qualitative representation of spatial knowledge. Volume 804 in *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[HHEH96] J.-L. Hainaut, J.-M. Hick, V. Englebert, and J. Henrard. Understanding the implementation of IS-A relations. Volume 1157 of *Lecture Notes in Computer Science*, pages 42-57. Springer Verlag, 1996.

[HHHR96] J.-L. Hainaut, J. Henrard, J.-M. Hick, and D. Roland. Database design recovery. *Volume* 1080 of *Lecture Notes in Computer Science*, pages 272-300. Springer Verlag, 1996.

[Him97] Himel Inc, *DBInformer User's Manual*, 17153 President Drive, Castro valley, CA 94546, USA, 1997.

[HK94] G. T. Heineman and G. E. Kaiser. Incremental process support for code reengineering. In *Proc. of the Intl. Conference on Software Maintenance*, pages 282–290. IEEE Computer Society Press, 1994.

[HMW95] D. Heckerman, A. Mamdani, and M. P. Wellman. Real-world applications of Bayesian networks: Introduction. *Communications of the ACM*, 38(3):24–26, 1995.

[Hol97] J. Holle. Ein Generator für integrierte Werkzeuge am Beispiel der objekt-relationalen Datenbankschemamigration. Master's Thesis, University of Paderborn, Dept. of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1997.

[Hol98] R.C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Working Conference on Reverse Engineering*, pages 210–219, Hawaii, USA. IEEE Computer Society Press, 1998.

[HR87] J. Y. Halpern and M. O. Rabin. A logic to reason about likelihood. *Artificial Intelligence*, 32(3):379–405, 1987.

[HTJC94] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Transformation-based database reverse engineering. Volume 823 of *Lecture Notes in Computer Science*, pages 362-373. Springer Verlag, 1994.

[Hül96] E. Hüllermeier. *Reasoning about Systems based on incomplete an uncertain models*. Ph.D. Thesis, University of Paderborn, Dept. of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1996.

[Hüs97] F. Hüsemann. Migration relationaler Datenbanken in objektorientierte Umgebungen. In *Tagungsband des 3. Fachkongresses Smalltalk und Java in Industrie und Ausbildung, Erfurt, Germany*, pages 5-10, 1997.

[Hüs98] F. Hüsemann. Eine erweiterte Schemaabbildungskomponente für Datenbank–Gateways. In *10. Workshop "'Grundlagen von Datenbanken"'*, pages 52–56, Konstanz. Konstanzer Schriften in Mathematik und Informatik Nr. 63, Universität Konstanz, 1998.

[JEJ95] I. Jacobson, M. Ericsson, and A. Jacobson. *The Object Advantage*. Addison Wesley, Workingham, UK, 1995.

[JH98a] J. H. Jahnke and M. Heitbreder. Design recovery of legacy database applications based on possibilistic reasoning. In *Proc. of 7th IEEE Intl. Conference on Fuzzy Systems, Anchorage, USA,* pages 1332–1337. IEEE, 1998.

[JH98b] S. Jarzabek and R. Huang. The case for user-centered case tools. *Communications of the ACM*, 41(8):93–99, 1998.

[JK90] P. Johannesson and K. Kalman. A method for translating relational schemas into conceptual schemas. In *Entity-Relationship Approach to Database Design and Querying: Proc. of the 8th Intl. Conference on Entity-Relationship Approach.* North Holland, 1990.

[JNW98] J. H. Jahnke, U. Nickel, and D. Wagenblasst. A case study in supporting evolution of complex engineering information systems. In *Proc. of 22nd Intl. Computer Software and Applications Conference*, pages 513-520. IEEE Computer Society Press, 1998.

[Joh86] R. Johnson. Independence and Bayesian updating methods. In L. N. Kanal and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence*, pages 197–201. Elsevier Science Publishers, Amsterdam, 1986.

[JP92] V. S. Jacob and H. Pirkul. Organizational decision support systems. *Intl. Journal of Man-Machine Studies*, 36(6):817–832, 1992.

[JS99] J. H. Jahnke and C. Strebin. Adaptive tool support for database reverse engineering. In *Proc. of 1999 Conference of the North American Fuzzy Information Processing Society, New York, USA*, pages 278-282. IEEE Press, 1999.

[JSWZ99] J. H. Jahnke, W. Schäfer, J. Wadsack, and A. Zündorf. Managing inconsistency in evolutionary database reengineering processes. *Science of Computer Programming*, 1999. (submitted)

[JSZ96] J. H. Jahnke, W. Schäfer, and A. Zündorf. A design environment for migrating relational to object oriented database systems. In *Proc. of the 1996 Intl. Conference on Software Maintenance*, pages 163-170. IEEE Computer Society Press, 1996.

[JSZ97] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *Proc. of European Software Engineering Conference*, number 1302 in *Lecture Notes in Computer Science*, pages 193-210. Springer Verlag, 1997.

[JSZ97a] J. H. Jahnke, W. Schäfer, and A. Zündorf. A design environment for migrating relational to object-oriented database systems (Abstract). In *Software Engineering and Database Technology*. Dagstuhl-Seminar-Report 173, Dagstuhl, Germany, 1997.

[JZS97b] J. H. Jahnke, W. Schäfer, and A. Zündorf. The NewPORT prototype V0, with demonstration. Joint Seminar of O_2 Technology and INRIA, Versaille, France, February, 26th, 1997.

[JW99a] J. H. Jahnke and J. Wadsack. Human-centered reverse engineering environments should support human reasoning. In *Proc. of the 1st Intl. Workshop on Soft Computing Applied to Software Engineering. Limerick, Ireland*, pages 77-83. Limerick University Press, 1999.

[JW99b] J. H. Jahnke and J. Wadsack. Integration of analysis and redesign activities in information system reengineering. In *Proc. of the 3rd European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands*, pages 160–168. IEEE Computer Society, 1999.

[JW99c] J. H. Jahnke and J. Wadsack. Varlet: Human-centered tool support for database reengineering. In *Proc. of the Workshop Software Reengineering. Bad Honnef, Germany*, 1999. (to appear)

[JZ97] J. H. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. In *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical University of Vienna, Information Systems Institute, Distributed Systems Group, 1997. Technical Report TUV-1841-97-10.

[JZ98] J. H. Jahnke and A. Zündorf. Using graph grammars for building the varlet database reverse engineering environment. In *Proc. of Theory and Application of Graph Transformations, Paderborn, Germany*. Technical Report tr-ri-98-201, University of Paderborn, D-33095 Paderborn, Germany, 1998.

[JZ99] J. H. Jahnke and A. Zündorf. *Handbook of Graph Grammars and Computing by Graph Transformation - Application*, volume 2, chapter Applying Graph Transformations To Database Re-Engineering. World Scientific, Singapore, 1999. (to appear)

[Kas80] U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[Kas96] N. K. Kasabov. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*. MIT Press, Cambridge, 1996.

[KDBM94] K. Kontogiannis, R. DeMori, M. Bernstein, and E. Merlo. Localization of design concepts in legacy systems. In *Proc. of the Intl. Conference on Software Maintenance 1994*, pages 414–423. IEEE Computer Society Press, 1994.

[Ker92] E. E. Kerre. A comparative study of the behavior of some popular fuzzy implication operators on the generalized modus ponens. In *Fuzzy logic for the management of uncertainty*. John Wiley & Sons, New York, 1992.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: A database application system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2), page 499, 1998.

[KM96] A. Konar and A. K. Mandal. Uncertainty management in expert systems using fuzzy petri nets. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):96–105, 1996.

[KM98] N. N. Karnik and J. M. Mendel. Introduction to Type-2 Fuzzy Logic Systems. In *Proc. 7th Intl. Conference on Fuzzy Systems FUZZ-IEEE'98, Anchorage, USA*, pages 915–920. IEEE, 1998.

[KNNZ99] T. Klein, U. Nickel, J. Niere, and A. Zündorf. From UML to Java and back again. University of Paderborn, Department of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1999.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KSRP99] R. K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proc. of the 21st International Conference on Software Engineering*, pages 226–235. ACM Press, 1999.

[KSW95] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Sciences*, 20(1):21–51, 1995.

[KWDE98] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proc. of the 5th Working Conference on Reverse Engineering*, pages 135–143, Hawaii, USA. IEEE Computer Society Press, 1998.

[Lan91] J. Lang. *Logique possibiliste: aspects formels, deduction automatique, et applications*. Ph.D. Thesis, IRIT, Univ. P. Sabatier, Toulouse, France, 1991.

[Lef95] M. Lefering. *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Informatik. Verlag Shaker, 1995.

[Lem77] E. J. Lemmon. *An Introduction to Modal Logic*. Basil Blackwell, 1977.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 6:707–710, 1966.

[LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, Sebastopol, 2nd edition, 1992.

[LMS98] A. L. Lederer, D. A. Mirchandani, and K. Sims. Using WISs to enhance competitiveness. *Communications of the ACM*, 41(7):94–95, 1998.

[LO98] T. Lin and L. O'Brian. FEPSS: A flexible and extensible program comprehension support system. In *Proc. of 5th Working Conference on Reverse Engineering*, pages 40–49, Hawaii, USA. IEEE Computer Society Press, 1998.

[Loe78] M. Loeve. *Probability Theory*. Springer Verlag, New York, 4th edition, 1978.

[Log97] Logic Works Inc., University Square at Princeton, 111 Compus Drive, Princeton NJ 08540. *ERwin User's Guide*, 3rd edition, 1997.

[Loo88] C. G. Looney. Fuzzy Petri nets for rule-based decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):178–183, 1988.

[LR89] C. L'Ecluse and P. Richard. The O_2 Database Programming Language. In *Proc. of the 15th Intl. Conference on Very Large Data Bases, Amsterdam, The Netherlands*, pages 411–422. Morgan Kaufmann Publishers, 1989.

[LS96] M. Lefering and A. Schürr. Specification of Integration Tools. In *Building tightly integrated software development environments*, volume 1170 of *Lecture Notes in Computer Science*, pages 324-334. Springer Verlag, 1996.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. of the 19th Intl. Conf. on Software Engineering, Boston, MA, USA*, pages 349-359. *ACM Press*, 1997.

[LS98a] D.-M. Lincke and B. Schmid. Mediating electronic product catalogs. *Communications of the ACM*, 41(7):86–88, 1998.

[LS98b] G. L. Lohse and P. Spiller. Electronic shopping. *Communications of the ACM*, 41(7):81–87, 1998.

[MAJ94] U. A. Johnen M. A. Jeusfeld. An executable meta model for re-engineering of database schemas. Technical Report 94-19, Technical University of Aachen, Germany, 1994.

[Mar97] R. A. Martin. Dealing with dates: Solutions for the Year 2000. *Computer*, 30(3):44–51, 1997.

[MCAH95] P. Martin, J. R. Cordy, and R. Abu-Hamdeh. Information capacity preserving of relational schemas using structural transformation. Technical Report ISSN 0836-0227-95-392, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1995.

[McC75] C. L. McClure. Structured programming in COBOL. *ACM SIGPLAN Notices*, 10(4):25–33, 1975.

[McC98] T. J. McCabe. Does reverse engineering have a future? Keynote of the *5th Working Conference on Reverse Engineering, Honolulu, Hawaii, USA*, 1998.

[MM90] R. W. Mathews and W. C. McGee. Data modeling for software develpment. *IBM Systems Journal*, 29(2):228–235, 1990.

[MN95] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *Proc. of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116-127. ACM Press, 1995.

[MNB+94] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994.

[MNL96] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *Proc. of the 18th Intl. Conference on Software Engineering*, pages 90–98, Berlin, Germany. IEEE, 1996.

[MNS95] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proc. of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[MT93] V. W. Marek and M. Truszczynski. *Nonmonotonic Logic*. Springer Verlag, Berlin, 1993.

[MWS97] K. Wong, and M.-A.D. Storey, H.A. Müller. How do program understanding tools affect how programmers understand programs? In *Proc. of 4th Working Conference on Reverse Engineering, Amsterdam, Holland*, pages 12–21. IEEE Computer Society Press, 1997.

[MWT94] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software systems using reverse engineering technology. In *Proc. of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences*, pages 41–48, Montreal, Canada, 1994.

[MZ82] M. Mizumoto and H. J. Zimmerman. Comparison of Fuzzy Reasoning Methods. *Fuzzy Sets and Systems*, 8:253–283, 1982.

[NA87] S. B. Navathe and A. M. Awong. Abstracting Relational and Hierarchical Data with a Semantic Data Model. In *Proc. of the 6th Intl. Conference of the Entity Relationship Approach, New York*, pages 305–333. North-Holland, 1987.

[NAF99] *Proc. of the 18th Conference of the North American Fuzzy Information Processing Society, New York, USA*. IEEE, 1999.

[Nag96] M. Nagl, editor. *Building tightly integrated software development environments*, volume 1170 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1996.

[Nea92] R. E. Neapolitan. A survey of uncertain and approximate reasoning. In *Fuzzy Logic for the Management of Uncertainty*, pages 55–82. John Wiley & Sons, 1992.

[NH95] L. Ngo and P. Haddawy. Probabilistic logic programming and Bayesian networks. Volume 1023 of *Lecture Notes in Computer Science*, pages 286-300. Springer Verlag, 1995.

[Nil93] N. J. Nilsson. Probabilistic logic revisited. *Artificial Intelligence*, 59(1-2):39–42, 1993.

[Nov92] V. Novak. Fuzzy logic as a basis of approximate reasoning. In *Fuzzy Logic for the Management of Uncertainty*, pages 247–264. John Wiley & Sons, 1992.

[Nov97] Novera Software Inc., 3 Burlington Woods, Burlington, MA 01830, USA. *Novera EPIC Database Builder (TM)*, release 1.3, September 1997.

[O2 93] O2 Technology. *The O_2 Application Designer's Manual – Version 4.3*. 7 rue du Parc de Clagny, 78000 Versailles, France, 1993.

[Obj99a] The Object People Inc., 885 Meadowlands Dr., Suite 509, Ottawa, Ontario. *TOPLink for Java 2.0 User's Manual*, 1999.

[Obj99b] ObjectMatter Inc., 2450 S.W. 137 Ave. Suite 206 Miami, Fl. 33175, USA. *Objectmatter VBSF Object-Relational Framework V2.02 User Manual*, 1999.

[ONT96] ONTOS Inc., 3 Burlington Woods, Burlington, MA, USA. *ONTOS Object Integration Server for Relational Databases 2.0 - Schema Mapper User's Guide*, 2.0 edition, 1996.

[Paa88a] G. Paass. Discussion of Chapter 9: Belief Functions. In *Non-Standard Logics for Automated Reasoning*. pages 279–280. Academic Press, London, 1988.

[Paa88b] G. Paass. Probabilistic logic. In *Non-Standard Logics for Automated Reasoning*, pages 213–251. Academic Press, London, 1988.

[PB94] W. J. Premerlani and M. R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, 1994.

[PD93] H. Prade and D. Dubois. Belief revision and updates in numerical formalisms – an overview, with new results for the possibilistic framework. In *Proc. of the Intl. Joint Conferences on Artificial Intelligence*, Chambery, France. Morgan Kaufman Publishers, 1993.

[Pea86] J. Pearl. Fusion, propagation, and structuring in bayesian networks. *Artificial Intelligence*, 29(3), 1986.

[Pea98] J. Pearl. Bayesian networks. Technical Report 980002, University of California, Los Angeles, Computer Science Department, USA, 1998.

[Pet81] J. L. Peterson. *Petri Net Theory and Modeling of Systems*. Prentice Hall, 1981.

[PKBT94] J-M. Petit, J. Kouloumdjian, J-F. Boulicaut, and F. Toumani. Using queries to improve database reverse engineering. In *Proc. of 13th Int. Conference of ERA, Manchester*, volume 881 of *Lecture Notes in Computer Science*, pages 369–386. Springer Verlag, 1994.

[PM96] P. Patel and K. Moss. *Java Database Programming With JDBC*. Coriolis Group Books, Scottsdale, AZ, USA, 1996.

[PMdP98] R. Penteado, P. C. Masiero, and A. F. do Prado. Reengineering of legacy systems based on transformation using the object-oriented paradigm. In *Proc. of 5th Working Conference on Reverse Engineering*, pages 144–153, Hawaii, USA. IEEE Computer Society Press, 1998.

[Poo88] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[Poo93] D. Poole. Average-case analysis of a search algorithm for estimating prior and posterior probabilities in bayesian networks with extreme probabilities. In *Proc. of the Intl. Joint Conferences on Artificial Intelligence*, Chambery, France. Morgan Kaufman Publishers, 1993.

[Pro89] G. M. Provan. A logic-based analysis of Dempster-Shafer theory. Technical Report TR-89-08, Department of Computer Science, University of British Columbia, Canada, 1989.

[PS92] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proc. of the 14th Intl. Conference on Software Engineering, Melbourne, Australia*, pages 262–279. IEEE Computer Society Press, 1992.

[PTBK96] J-M. Petit, F. Toumani, J. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proc. 12th International Conference on Data Engineering*, pages 218–227, New Orleans. IEEE Computer Society, 1996.

[Rad95] E. Radeke. *Federation and Migration among Database Systems*. Ph.D. Thesis, University of Paderborn - Department of Mathematics and Computer Science, D-33095 Paderborn, Germany 1995.

[Rat98] Rational Software Corp., 18880 Homestead Road, Cupertino, CA 95014, USA. *Rational Rose 98 - Using Rational Rose / Oracle 8*, 1998.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N. J. 07632, 1991.

[Res76] N. Rescher. *Plausible reasoning - An introduction to the theory and practice of plausibilistic inference*. Van Gorcum, Assen/Amsterdam, 1976.

[RH96] S. Ramanathan and J. Hodges. Reverse engineering relational schemas to object-oriented schemas. Technical Report MSU-960701, Department of Computer Science, Mississippi State University, USA, 1996.

[RH97] S. Ramanathan and J. Hodges. Extraction of object-oriented structures from existing relational databases. *ACM SIGMOD Record*, 26(1), 1997.

[RHSR94] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proc. of ACM SIGSOFT, New Orleans LA, USA*, pages 11-20. ACM Press, 1994.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, USA, 1st edition, 1999.

[Rog71] R. Rogers. *Mathematical Logic and Formalized Theories*. North-Holland, Amsterdam, 1971.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[RRF90] K. Rosen, R. Rosinski, and J. Farber. *UNIX System V Release 4: An Introduction for New and Experienced Users*. Mc-Graw-Hill, New York, NY, USA, 1990.

[RS97] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing, London, Academic Press.*, 8(1), 1997.

[Rum98] C. Rummel. Ein Transformationsbasierter Ansatz zur Migration von relationalen zu objektorientierten Datenbanken. Master's Thesis, Univeristät-GH Paderborn, Mathematik-Informatik, D-33095 Paderborn, Germany, 1998.

[SCC+93] Y.-P. Shan, T. Cargill, B. Cox, W. Cook, M. Loomis, and A. Snyder. Is multiple inheritance essential to OOP? In *Proc. of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 363–363, Washington, DC, USA. ACM Press, 1993

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, Wiesbaden, Germany, 1991.

[Sch92] J. C. Schryver. Object-oriented qualitative simulation of human mental models of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(3):526–541, 1992.

[Sch93] B. Schiefer. *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen*. Ph.D. Thesis, Universität Karlsruhe, Fakultät für Informatik, FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10, D-76131 Karlsruhe, Germany, 1993.

[Sch95a] K. Schick. The key to client/server - unlocking the power legacy systems. *Gartner Group Conference*, February 1995.

[Sch95b] A. Schürr. Logic based structure rewriting systems. *Fundamenta Informaticae, Special Issue on Graph Transformation Systems*, pages 363–386, 1995.

[Sch98] H. Schalldach. Integration von Java-Anwendungen mit relationalen Informationssystemen. Master's Thesis, University of Paderborn, Department of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1998.

[SdJPeA99] P. Sousa, L. Pedro de Jesus, G. Pereira, and F. Brito e Abreu. Clustering relations into abstract er schemas for database reverse engineering. In *Proc. of the 3rd European Conference on Software Maintenance and Reengineering. Amsterdam, NL*, pages 169–176. IEEE Computer Society Press, 1999.

[Sha76] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, 1976.

[Sha90] G. Shafer. Belief functions. In *Readings in Uncertain Reasoning*. Morgan Kaufmann, San Mateo, California, USA, 1990.

[Sho74] E. H. Shortliffe. *A rule-based computer program for advising physicians regarding antimicrobial therapy selection*. Ph.D. Thesis, Stanford University, 1974.

[Sie98] Siemens AG - C-LAB, Fürstenallee 11, D-33102 Paderborn, Germany. *OpenDM ODMG User's Guide*, 1998.

[Sim94] D. Simpson. Are mainframes cool again. *Datamation*, pages 46–53, 1994.

[SK90] F. N. Springsteel and C. Kou. Reverse Data Engineering of E-R Designed Relational Schemas. In *Proc. of Databases, Parallel Architectures and their Applications*, pages 438–440. Springer Verlag, 1990.

[SLGC94] O. Signore, M. Loffredo, M. Gregori, and M. Cima. Reconstruction of er schema from database applications: a cognitive approach. In *Proc. of 13th Intl. Conference of ERA, Manchester*, pages 387–402, volume 881 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Slo95] A. M. Sloane. An evaluation of an automatically generated compiler. *ACM Transactions on Programming Languages and Systems*, 17(5):691–703, 1995.

[SM95] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proc. of Intl. Conference in Software Maintenance*, pages 275–285. IEEE Computer Society Press, 1995.

[Sme88] P. Smets. Belief functions. In *Non-Standard Logics for Automated Reasoning*, pages 253–286. Academic Press, London, 1988.

[Sne91] H. M. Sneed. Bank application reengineering and conversion at the union bank of switzerland. In *Proc. of the Intl. Conference on Software Maintenance 1991*, pages 60–72. IEEE Computer Society Press, 1991.

[Sne95] H. M. Sneed. Planning the reengineering of legacy systems. *IEEE Software*, 12(1):24–34, 1995.

[Sou98a] C. Soutou. Relational database reverse engineering: Extraction of cardinality constraints. *Data and Knowledge Engineering, Elsevier, North Holland*, 28(2):161–207, 1998.

[Sou98b] C. Soutou. Inference of aggregate relationships through database reverse engineering. In *Proc. of Intl. Conf. on Conceptual Modeling,* volume 1507 of *Lecture Notes in Computer Science,* pages 135-145, Springer Verlag, 1998.

[SP98] P. Stevens and R. Pooley. Systems reengineering patterns. In *Proc. of ACM Foundations of Software Engineering, Lake Buena Vista, Florida, USA*, pages 17-23. ACM Press, 1998.

[Sto98] M. A. D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. Ph.D. Thesis, Simon Fraser University, Vancouver, B.C., Canada, 1998.

[Str97] B. Stroustrup. *The C++ Programming Language: Third Edition*. Addison Wesley, Reading, MA, USA, 1997.

[Str99] C. Strebin. Adaption unsicheren Reverse-Engineering-Wissens auf Basis konnektionistischer Methoden. Master's Thesis, University of Paderborn, Department of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1999.

[SWZ95] A. Schürr, A. J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. *Proc. of the European Software Engineering Conference*, pages 219-234, volume 989 of *Lecture Notes in Computer Science*, Springer Verlag, 1995.

[Tae96] G, Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. Ph.D. Thesis, Technische Universität Berlin, Fachbereich 13, 1996.

[TCHH99] Ph. Thiran, A. Chougrani, J.-M. Hick, and J.-L. Hainaut. Generation of conceptual wrappers for legacy database. In *Proc. of 10th Intl. Conference and Workshop on Database and Expert Systems Applications, Florence*, Lecture Notes in Computer Science. Springer Verlag, 1999. (to appear)

[Tea99] The Progres Developer Team. *The Progres Language Manual Version 9.2*. Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany, 1999.

[Ten98] J. M. Tenenbaum. WISs and electronic commerce. *Communications of the ACM*, 41(7):89–90, 1998.

[TFAM96] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo. Augmenting pattern-based architectural recovery with flow analysis: Mosaic - A case study. In *Proc. of 3rd Working Conference on Reverse Engineering*. IEEE Computer Society, 1996.

[THB+98] P. Thiran, J.-L. Hainaut, S. Bodart, A. Deflorenne, and J.-M. Hick. Interoperation of independent, heterogeneous and distributed databases. methodology and CASE support: the InterDB approach. In *Proc. of the 3rd Intl. Conf. on Cooperative Information Systems, New York City, USA*, pages 54–63. IEEE Computer Society Press, 1998.

[Tho99] Thought Inc., 657 Mission Street, Suite 202, San Francisco, CA 94105, USA. *CocoBase WhitePaper*, 1999.

[Tre95] M. Tresch. *Evolution in Objekt-Datenbanken*. Teubner Verlag, Stuttgart, 1995.

[TWSM94] S. R. Tilley, K. Wong, M-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *Intl. Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[Uma97] A. Umar. *Application (Re)Engineering - Building Web-Based Applications and Dealing with Legacies*. Prentice-Hall International, London, UK, 1997.

[UML97] *UML Notation Guide vers. 1.1*. Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam, 1997.

[vdBKV97] M. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation: an annotated bibliography. *ACM Software Engineering Notes*, 22(1), 1997.

[vDM98] A. van Deursen and L. Moonen. Type inference in cobol systems. In *Proc. of the 5th Working Conference on Reverse Engineering*, pages 220–230, Hawaii, USA. IEEE Computer Society Press, 1988.

[Vin97] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[vM19] R. von Mises. Grundlagen der Wahscheinlichkeitsrechnung. *Mathematische Zeitung*, 5, 1919.

[Voo89] F. Voorbraak. A computationally efficient approximation of Dempster-Shafer theory. *International Journal of Man-Machine Studies*, 30(5):525–536, 1989.

[Wad98] J. P. Wadsack. Inkrementell Konsistenzerhaltung in der transformationsbasierten Datenbankmigration. Master’s Thesis, University of Paderborn, Department of Mathematics and Computer Science, D-33095 Paderborn, Germany, 1998.

[War96] M. P. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7):598–618, 1996.

[Wel97] B. B. Welch. *Practical Programming in Tcl & Tk*. Prentice Hall Press, Upper Saddle River, 2nd edition, 1997.

[Wil86] W. G. Wilson. Prolog for applications programming. *IBM Systems Journal*, 25(2):190–206, 1986.

[Wil94] L. M. Wills. Using attributed flow graph parsing to recognize programs. In *Intl. Workshop on Graph Grammars and Their Application to Computer Science*, *Williamsburg, Virginia, USA,* pages 170-184, volume 1073 in *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[WM97] A. R. Williamson and C. L. Moran. *Java Database Programming: Servlets & JDBC*. Prentice Hall, 1997.

[WS90] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly Associates, Inc., Sebastopol, CA, 1990.

[WSK97] C. Welsch, A. Schalk, and S. Kramer. Integrating forward and reverse object-oriented software engineering. In *Proc of the 19th Intl. Conf. on Software Engineering, Boston, MA, USA,* pages 560-561. ACM Press, 1997.

[YB94] H. Yang and K. Bennett. Extension of A transformation system for maintenance - dealing with data-intensive programs. In *Proc. of the Intl. Conference on Software Maintenance*, *Victoria, Canada,* pages 344–353. IEEE Computer Society Press, 1994.

[YHC97] A. S. Yeh, D. R. Harris, and M. P. Chase. Manipulating recovered software architecture views. In *Proc of the 19th Intl. Conf. on Software Engineering, Boston, MA, USA,* pages 184-194. ACM Press, 1997.

[YLQ98] A. Yang, J. Linn, and D. Quadrato. Developing integrated Web and database applications using JAVA applets and JDBC drivers. In *Proc. of the 29th SIGCSE Technical Symposium on Computer Science Education*, volume 30,1 of *SIGCSE Bulletin*, pages 302–306, New York. ACM Press, 1998.

[Zad65] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad75] L. A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. *Information Sciences*, 8:199–249, 1975.

[Zad78] L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1978.

[Zha98] W.R. Zhang. Bipolar Fuzzy Sets. In *Proc. 7th Intl. Conf. on Fuzzy Systems, Anchorage, USA*, pages 835–840. IEEE, 1998.

[Zün95] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. Deutscher Universitätsverlag, Wiesbaden, 1995.

[Zün99] A. Zündorf. Skript zur Vorlesung Graphentechnik, Sommersemester 1999. University of Paderborn, Department of Mathematics and Computer Science, D-33095 Paderborn, Germany, April 1999.

INDEX

Numerics

1-context 147

A

abstract
  class 119, 129
  syntax graph 114
access path 120
α-cut 45, 50, 60, 105
aggregation 118
Analysis Front-End 100
analyzed logical schema 58
application condition 121, 124
architecture 99, 157
artificial key 21
association 133
attribute transfer clause 121, 123
automatic analysis operation 63, 72
axiom 80
  -based marking 80

B

backward propagation 151
backwards reasoning 77
base table 168
  attribute 170
  relationship 172
basic probability
  assignment 42
  number 42
Bayesian inference 41
belief
  function 43
  revision 78
  revision step 78
best model 51
best valuation 51
bipolar fuzzy set 49

C

canonical translation 178
card-operator 129
cardinality constraint 21
certainty factor 39
change propagation 145, 150, 163
classical projection 50
closed world assumption 36
code patterns 18
cold turkey 2
complex transformation 138, 144
complexity 97
compositional inference law 48
computer-aided reengineering 4
conceptual
  abstraction 24
  extension 24
  migration 24
  redesign 24, 161
  schema 25, 118
  schema migration 113
concrete class 119
conditional
  boolean expression 132, 141
  expression 141
  probability 40
confidence factor 48
consistency management 145
context sensitive menu 159
continuous membership function 45
contradiction 7
contraposition transition 83
CORBA 1
COTS middleware 180
credibilistic reasoning 42
Customization Front-End 100
cyclic join pattern 18

D

data
  analysis 18
  integration 113
  reverse engineering 3
database 38
  reengineering 3
  reverse engineering 3
data-decomposable 30
data-driven analysis 82, 92

operation 63, 71
deduction problem 51
degree of consistency 51
Dempster-Shafer model 42
derivability 94
derived attribute 159
discrete membership function 45
domain analysis 56

E

edge type 116
enabled transition 79
encapsulation 28
Entity-Relationship model 116
equilibrium
state 80
time 80
error models 41
Euro-conversion 3
evaluation 84
expansion 83
expansion
of formulae 72
-evaluation cycle 84
expert system 33
extent of a possibilistic predicate 71

F

fact base 34
focal proposition 42
folding clause 131
forward
engineering 2
mapping 123
production 123
propagation 150
forwards reasoning 77
fuzzy
belief marking 78
composition 47
implication 47
inference 48
logic 44
logical operator 48
Petri net 77
predicate 58
reasoning 44
relation 47
rules 46
set 44
truth token 78

G

generic data model 178
Generic Fuzzy Reasoning Net 7, 55, 57, 66
goal-driven analysis 83
operation 63, 71
graded modus ponens 51
graph 115
constraint 118
grammar 121
production 121
test 118, 155
graphical path expression 133
GRAS 157
grounding 84, 87, 92

H

history graph 146, 149
human-awareness 5

I

ignorance 36
implication 58
rule 39
implies 129
inclusion dependency 22, 38, 133
incremental
reasoning 77
schema migration 114
inductive logic programming 111
inference 51
algorithm 90
engine 8, 34, 76, 99
loop 92
process 81
information capacity 137
inheritance 118
inner universal quantifier 61
instance mapping 137
iteration 161

J

Java 28

database connectivity (JDBC) 14
join pattern 18

K

key dependency 38
knowledge
base 33
-based system 33

L

layered graph grammar 99
left-hand side 121
legacy
database 4
software system 1
Levenshtein distance 60
limitcycle 80
logical schema 57

M

main transition 83
mapping rule 122
mass change 3
match 121
maximum-likelihood 41
MAX-MIN composition 48
measure of belief 39
membership
degree 45
function 45
meta model 178
middleware 26, 113, 165
Migration Front-End 159
migration graph 114
model 114
monotonic reasoning 36
multiple inheritance 116
MYCIN 38, 39

N

naming convention 16
necessity 49, 59
-valued formula 49
-valued possibilistic logic 49
negative application condition 130, 148
node set 129
node type 116
non-monotonic reasoning 36
not-null constraint 38
NULL-value 57

O

Object
identifier 120
Management Group 117
Modeling Technique 109
ObjectDRIVER 165
occurrence of literals 72
ODMG standard 28, 116
Open Database Connectivity (ODBC) 100
open world assumption 36
optimization structure 21
optional graph element 129
ordered association 24

P

path expression 124
pattern library 99
periodic oscillation 80
Petri Net 77
place 78
plausibility function 44
possibilistic reasoning 49, 59
possibility 49
distribution 50
posterior probability 41
predecessor 80
primitive transformation 138
probabilistic logic 40
probability measure 40
process iterations 7
Progres 116, 158

Q

qualitativ reasoning 35
quantitative reasoning 35

R

redesign transformation 137
reengineering 2
process 2
reevaluation 150

relation schema 38
relational database 38
remote
    attribute 170
    relationship 172
restriction 130
reverse
    engineering 2
    mapping 123
    production 123
right-hand side 121

S

scalability 97
schema
    analysis 7
    catalog 4
    mapping graph 114
    migration 7
    redesign 137
    transformation 137
select distinct pattern 18
selection problem 35
semantical enrichment 15
stability 80
start graph 121, 125
structural completion 15
structure transformation 137
subjective
    evidence 42
    probability 40

T

t-conorm 46
t-norm 46
threshold value 60
transformation
    system 121
    template 146
transition 78
transitive
    inheritance 130
    path expression 148
translation 150
triple graph grammar 114, 122
TXL 179
type-2 fuzzy logic 48

U

Unified Modeling Language (UML) 13
universe of discourse 38
unparser 159

V

variable aggregation 61
variant 57
    records 20
Varlet
    Analyst 99, 157
    Migrator 157
view threshold 105

Y

Year-2000 problem 3

ABBREVEATIONS

A

AI - Artificial Intelligence 33
API - Application Programming Interface 5
ASG - Abstract Syntax Graph 100, 114

B

BRS - Belief Revision Step 78

C

CARE - Computer-Aided ReEngineering 4
CF - Certainty Factor 39
C-IND Cardinality INclusion Dependency 58
COTS - Commercial Off-The-Shelf 9
CS - Client/Server 1
CT - Contraposition Transition 83
CV - Confidence Value 59

D

DB - DataBase 11
DBMS - Database Management System 3
DBRE - Database ReEngineering 3
DBRvE - DataBase Reverse Engineering 3
DDL - Data Definition Language 100
DRvE - Data Reverse Engineering 3

E

ER - Entity-Relationship 101

F

FBM - Fuzzy Belief Marking 78
FE - Forward Engineering 2
FPN - Fuzzy Petri Net 77
FTT - Fuzzy Truth Token 78
FUJABA - From Uml to Java And Back Again 184

G

GFRN - Generic Fuzzy Reasoning Net 7
GMP - Graded Modus Ponens 51

I

IA - Information Augmenting 137
IC - Information Changing 137
IE - Inference Engine 8, 81
iff - if and only if 50
I-IND - Isa-INclusion Dependency 58
ILP - Inductive Logic Programming 111
IND - INclusion Dependency 22, 38
IP - Information Preserving 137
IQ - Inner universal Quantifier 61
IR - Information Reducing 137
IS - Information System 185
IT - Information Technology 11

J

JDBC - Java DataBase Connectivity 14

K

KBS - Knowledge-Based System 33

L

L^0 - propositional logic 38
L^1 - first-order logic 38
LC - Limit Cycle 80
LDB - Legacy DataBase 4
LOC - Lines Of Code 45
LSS - Legacy Software System 1
LT - Learning Task 184

M

MB - Measure of Belief 39
MD - Measure of Disbelief 39
MIS - Marketing Information System 12
MT - Main Transition 83

N

NN - Neural Network 184
NPL^1 - Necessity-valued Possibilistic Logic 49

O

ODBC - Open DataBase Connectivity 100

OID - Object IDentifier 120
OMG - Object Management Group 117
OMT - Object Modeling Technique 109
OO - Object-Orientation 1

P

PDIS - Product and Document Information System 11
PN - Petri Net 77
PO - Periodic Oscillation 80
Progres - PROgammed Graph REplacement Systems 116

R

RDB - Relational DataBase 38
RE - ReEngineering 2
R-IND - Reference INclusion Dependency 58
RS - Relation Schema 38
RvE - Reverse Engineering 2

S

SMG - Schema Mapping Graph 114
SQL - Structured Query Language 18

T

TV - Threshold Value 60

U

UML - Unified Modeling Language 13

W

w.r.t. - with respect to 65
Web - World Wide Web 1