

Sergey Melnik

LNCS 2967

Generic Model Management

Concepts and Algorithms



Springer

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2967

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Sergey Melnik

Generic Model Management

Concepts and Algorithms



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Author

Sergey Melnik
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052-6399, USA
E-mail: melnik@microsoft.com

Library of Congress Control Number: 2004104636

CR Subject Classification (1998): H.3, H.2, D.2, D.3, F.3, I.2

ISSN 0302-9743

ISBN 3-540-21980-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable to prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 11007593 06/3142 5 4 3 2 1 0

Preface

Many challenging problems facing information systems engineering involve the manipulation of complex metadata artifacts, or *models*, such as database schemas, interface specifications, or object diagrams, and *mappings* between models. The applications that solve metadata manipulation problems are complex and hard to build. The goal of generic model management is to reduce the amount of programming needed to develop such applications by providing a database infrastructure in which a set of high-level algebraic operators, such as Match, Merge, and Compose, are applied to models and mappings as a whole rather than to their individual building blocks.

This dissertation presents an initial study of the concepts and algorithms for generic model management. We describe the first prototype of a generic model management system, introduce the algebraic operators that are used to manipulate models and mappings, clarify the semantics of the operators, and develop novel algorithms for implementing them. In particular, we present an innovative algorithm based on fixpoint computation that is used for implementing the generic operator Match, which finds correspondences between two models. Using the prototype and the operators presented in the dissertation, we develop solutions for several practically relevant problems, such as change propagation and reintegration.

April 2004

Sergey Melnik

Acknowledgements

I would like to express my deep gratitude to everyone who helped me shape the ideas explored in this dissertation, either by giving technical advice or encouraging and supporting my work in many other ways.

I enjoyed a rare privilege of collaborating closely with several distinguished database researchers. This dissertation would not have come into existence without their hands-on advice and motivation.

Prof. Erhard Rahm supervised and guided my work from the very first day. He gave me the opportunity to conduct this doctoral research and helped me make the right strategic decisions at many forks along the way. He kept me on track while allowing me to broaden my research horizon in tangential areas. His insightful comments, which densely filled the margins of each draft that I gave to him, gave rise to many creative ideas.

Prof. Hector Garcia-Molina invited me to Stanford University and taught me the art of turning hard research challenges into fun and expressing my thoughts clearly using examples. From him I learned that solid research requires patience: for example, he suggested that a draft of our joint paper [Melnik, Garcia-Molina, Rahm 2002] needed more polishing and so we missed a conference deadline. Later that paper, which underpins Part III of the dissertation, received the Best Student Paper Award at the Intl. Conf. on Data Engineering.

Prof. Emeritus Gio Wiederhold showed me what it takes to step back and see a big picture, and yet keep the details in focus. He gave me the opportunity to collaborate in the DARPA DAML project at Stanford and to get a foretaste of metadata management problems in the context of interoperation on the Semantic Web.

Dr. Philip A. Bernstein has been the driving force behind the emerging research area of generic model management, the subject of the dissertation. His vision papers and talks inspired much of the work done in this thesis. His insightful suggestions on our joint papers and his guidance in designing the first prototype for model management, which is presented in Part I of the dissertation, were invaluable.

Prof. Alon Halevy helped me keep my spirits high while I worked on Part II, a more theoretical part of the dissertation. His encouragement and advice made this work a real pleasure.

I am grateful to Profs. Serge Abiteboul, Paolo Atzeni, Stefano Ceri, Martin Kersten, Renée Miller, and Gerhard Weikum for helpful discussions.

I would like to thank my colleagues and friends in the Database Groups in Leipzig and Stanford, the members of the Graduate Programme on Knowledge Representation in Leipzig, and the colleagues in the RDF Core Working Group at the World-Wide Web Consortium for fruitful exchanges of ideas. The members of the Stanford Database Group helped me conduct the user study presented in Part III.

I am indebted to Drs. Stefan Decker, Andreas Paepcke, Bertram Ludäscher, Felix Naumann, and Arturo Crespo for their support and many informal discussions, which helped me put my academic research into perspective.

I owe very special thanks to my wife Teresa. Her love and energy constantly recharged my forces. She has been my perpetual source of creativity and inspiration, in so many respects.

This dissertation is dedicated to my parents, Tanja and Juri, who are truly the origin of all great things that ever happened to me.

The contributions of the above people made the work on this dissertation a rewarding and memorable experience. I thank you all.

April 2004

Sergey Melnik

Table of Contents

Part I. A Programming Platform for Model Management

1. Introduction	3
1.1 Metadata Management	3
1.2 The Problem	5
1.3 A Vision for Management of Complex Models	6
1.4 Outline and Contributions of the Dissertation	9
2. Conceptual Structures and Operators	13
2.1 Motivating Scenario	13
2.2 Conceptual Structures	18
2.2.1 Models	19
2.2.2 Morphisms	20
2.2.3 Selectors	22
2.3 Operators	22
2.3.1 Primitive Operators	23
2.3.2 Derived Operators	25
2.3.3 Extract and Delete	26
2.3.4 Match	27
2.3.5 Merge	28
3. Implementation and Applications	29
3.1 Conceptual Structures	29
3.2 Operators	30
3.2.1 Extract and Delete	30
3.2.2 Dependencies	32
3.2.3 ExtractMin	33
3.2.4 DeleteHard and DeleteSoft	35
3.2.5 Diff	36
3.2.6 Match	37
3.2.7 Merge	38
3.3 Prototype “Rondo”	42
3.4 View-Reuse Scenario	45
3.5 Reintegration Scenario	47
3.6 Conclusions	50

Part II. A Semantics for Model Management Operators

4. State-Based Semantics	55
4.1 Basic Concepts	56
4.1.1 Models	56
4.1.2 Mappings	58
4.1.3 Formal Notation	60
4.1.4 Semantics of Scripts	61
4.1.5 Preliminaries	62
4.2 Operators	64
4.2.1 Compose Operator	65
4.2.2 Invert Operator	67
4.2.3 Extract Operator	68
4.2.4 Merge Operator	73
4.2.5 Diff Operator	77
4.2.6 Confluence Operator	84
4.2.7 Match Operator	85
4.3 Materialization	86
5. Change Propagation Scenario	91
5.1 Propagating Additions	92
5.2 Propagating Deletions	93
5.3 A General Solution	95
5.4 Schema Evolution Scenario	96
5.5 Variants of Change Propagation	98
6. State-Based Semantics in Rondo	101
6.1 Semantics of Morphisms	101
6.2 Semantics of Selectors	105
6.3 Structural vs. State-Based Operators	106
6.4 Revisiting Change Propagation	109
6.5 Conclusions	112

Part III. Schema Matching

7. Similarity Flooding Algorithm	117
7.1 Overview of the Approach	119
7.2 Similarity Flooding Algorithm	122
7.2.1 Similarity Propagation Graph	122
7.2.2 Fixpoint Computation	123
7.3 Generalized Version of the Algorithm	124
7.4 Convergence and Complexity of the Algorithm	126
7.5 Features of the Algorithm by Example	127

7.5.1	Semistructured Data	128
7.5.2	XML Schemas	129
7.5.3	Matching XML Schemas Using Instance Data	131
7.5.4	Finding Related Data	134
8.	Filters	137
8.1	Constraints	138
8.2	Selection Metrics	139
8.3	FilterBest Algorithm	142
8.4	Expressing FilterBest in SQL	144
9.	Evaluation and Tuning	147
9.1	Matching Accuracy	148
9.2	Intended Match Result	150
9.3	User Study	151
9.4	Evaluation of Algorithm and Filters	153
9.5	Propagation Coefficients	155
9.6	Conclusions and Open Issues	156
<hr/>		
Part IV. Model Management in Perspective		
<hr/>		
10.	Related Work	163
10.1	Data Integration and Merge	164
10.1.1	Schema Integration	165
10.1.2	Answering Queries Using Views	170
10.2	Schema Matching and Match	173
10.3	Mapping Composition and Compose	178
10.4	View Selection and Extract	181
10.5	View Complement and Diff	182
10.6	Approaches to Specifying Semantics	184
10.6.1	Semantics of Models and Mappings	184
10.6.2	Information Capacity	186
10.6.3	Category Theory	187
10.7	Metadata Repositories	189
10.8	Metadata-Intensive Applications	190
10.8.1	Declarative Mediation	190
10.8.2	Change Propagation	193
10.9	Other Related Work	195
11.	Conclusions and Outlook	199
11.1	Summary of Contributions	199
11.2	Concluding Discussion	200
11.3	Open Technical Challenges	205
11.3.1	Decidability and Complexity	205

11.3.2	Equivalence and Entailment of Scripts	205
11.3.3	Completeness and Redundancy	206
11.3.4	<i>N</i> -ary Mappings	209
11.3.5	Formalization of Model-Management Problems	210
A.	User Study	213
A.1	BizTalk Schemas (XML)	214
A.2	Property Listing Schemas (XML)	215
A.3	Library Schemas (XML)	215
A.4	Product Schemas with Data Instances (XML)	215
A.5	University Schemas with Data Instances (XML)	216
A.6	Catalogs with Data Instances (XML)	217
A.7	Personnel Schemas (Relational)	219
A.8	University Schemas (Relational)	219
A.9	Personnel/University Schemas (Relational)	220
B.	Proofs of Simplification Theorems	221
B.1	Extract Operator	221
B.2	Merge Operator	223
B.3	Diff Operator	225
	References	229

List of Figures

1.1	A high-level architecture of model management	8
2.1	Scenario illustrating propagation of changes from a relational schema to an XML schema	14
2.2	Schematic representation of a solution for change propagation scenario of Fig. 2.1	15
2.3	Converted schema c and support element ORDERS in c'	16
2.4	Sample model shown as graph and 4-tuples	19
2.5	A morphism between a relational and an XML schema	21
2.6	Graph representation of XML schema in Fig. 2.5	21
2.7	Example of a selector	22
2.8	Examples of copying the model of Fig. 2.4 using selector $\{a1, a2, a3, a4\}$	26
3.1	Examples of extraction and deletion from a relational schema m ..	31
3.2	Example of existential dependencies in a relational schema	33
3.3	Example of existential dependencies in an XML schema	33
3.4	Merging two sample schemas	39
3.5	Architecture of the prototype	42
3.6	Code size breakdown in prototype (in lines of code)	45
3.7	Morphism between sources S_1 and S_2	45
3.8	Merging two SQL views	46
3.9	Reintegration scenario (3-way merge)	48
3.10	Schematic representation of the reintegration scenario	50
4.1	Some instances of relational schema R(Name: char(3), Sex: bool)	57
4.2	Portion of a mapping	58
4.3	Schematic representation for Example 4.2.6 (Extract)	68
4.4	Illustration of Extract operator	70
4.5	Schematic representation for Example 4.2.12 (Merge)	73
4.6	Illustration of Merge operator	74
4.7	Schematic representation for Example 4.2.17 (Diff)	77
4.8	Illustration of Diff operator	78
4.9	Example of Diff result by Theorem 4.2.5	80

4.10	The output mapping in Diff is not determined up to isomorphism	80
4.11	Illustration of Theorem 4.2.11 (Mirror Merge)	86
4.12	Materialization of models and mappings	88
5.1	Propagating additions	92
5.2	Propagating deletions	94
5.3	Propagating deletions over bijection	94
5.4	Change propagation: a general solution	96
5.5	Schema evolution: a special case of change propagation	97
5.6	Addition only, convert first then Diff	98
5.7	Addition only, Diff first, then convert	99
6.1	Three alternative semantics for a morphism	102
6.2	Relationship between cites and zip codes is not preserved on composition	104
6.3	Structural composition vs. state-based composition (the latter with and without NULLs; predicate \leftrightarrow denotes if-and-only-if)	107
6.4	Structural extraction yields materialization of the state-based operator	108
6.5	Schematic representation for structural change propagation script	111
7.1	Matching two relational schemas: Personnel and Employee-Department	119
7.2	A portion of graph representation G_1 for relational schema S_1	120
7.3	Example illustrating the Similarity Flooding algorithm	122
7.4	Matching of semistructured data	128
7.5	Matching of two XML schemas: AccountOwner (S_1) vs. Customer (S_2)	130
7.6	Two different representations of XML data: OEM/Lore-like vs. XML/DOM-like	131
7.7	Matching of two XML schemas using instance data in DOM graph representation	133
7.8	Excerpt of relationships in the Stanford DB Group	135
8.1	Cumulative similarity vs. “stable marriage”	137
8.2	Relative similarities for the example in Fig. 8.1	138
8.3	Example illustrating execution of FilterBest in SQL	144
9.1	Matching accuracy as a function of t_{rel} -threshold for intended match results Sparse, Expected, and Verbose from Table 9.1	151
9.2	Average matching accuracy for 7 users and 9 matching problems	152
9.3	Matching accuracy for different filters and four versions of the algorithm	153

9.4	Impact of randomizing initial similarities on matching accuracy ..	155
9.5	Impact of different ways of computing propagation coefficients on overall matching accuracy in the user study	156
10.1	Use of composition in (Shanmugasundaram et al. 2001a)	180
11.1	Schematic representation for Conjecture 11.3.1 (Associative Merge)	206
11.2	Illustration of Intersect operator	207

List of Tables

2.1	Summary of key operators in Rondo	23
2.2	Definitions of primitive operators	24
3.1	Comparison of variants of extraction and deletion	36
4.1	Summary of key model-management operators	64
7.1	A portion of <i>initialMap</i> obtained by string matching (10 of total 26 entries are shown)	120
7.2	The mapping after applying <code>SelectThreshold</code> on result of <code>SFJoin</code>	121
7.3	Variations of the fixpoint formula	124
7.4	The mapping after applying <code>SFJoin</code> \circ <code>SelectLeft</code> to semistructured data in Fig. 7.4	129
7.5	Parameters of the fixpoint computation for S_1 and S_2	131
7.6	Match results for XML schemas in Fig. 7.5 using two different graph representations	132
7.7	Match results for XML element tags in Fig. 7.7 using similarity threshold 0.05	134
7.8	Relatedness of faculty members in the DB group based on data in Fig. 7.8	135
9.1	Three plausible intended match results for matching problem in Fig. 7.1	149
9.2	Sizes of graphs in the user study	152
9.3	Illustration of convergence properties of variations of fixpoint formula for tasks T_1, \dots, T_9 in the user study. Shows iterations needed until length of residual vector got below 0.05.	154
9.4	Different approaches to computing the propagation coefficients $\pi_{\{l,r\}}(\langle x, p, A \rangle, \langle y, q, B \rangle)$	157
10.1	Data integration scenarios	165

1. Introduction

“Life is pretty simple: You do some stuff. Most fails. Some works. You do more of what works. If it works big, others quickly copy it. Then you do something else. The trick is the doing something else.”

– Leonardo da Vinci (1452-1519)

This chapter highlights the background of the dissertation and outlines its structure. In Sect. 1.1, we introduce metadata management, the general subject of this work. The deficiencies of today’s metadata management techniques are examined in Sect. 1.2. In Sect. 1.3, we sketch the approach to metadata management explored in the dissertation, called generic model management, and formulate our main objectives. An overview of the structure and contributions of the dissertation is given in Sect. 1.4.

1.1 Metadata Management

Metadata is descriptive information about data and applications. Metadata is used to specify how data is represented, stored, and transformed, or may describe interfaces and behavior of software components. There are two kinds of metadata that are commonly used (Bretherton and Singley 1994). One kind of metadata, called structural or control metadata, is deployed primarily by computer programs. Examples of structural metadata are an interface definition in a programming environment or a database schema in a database system. The other kind of metadata, called guide metadata, is intended solely for use by humans and is expressed in natural language. It contains keyword descriptions or documentation, and is often used to facilitate information retrieval. The focus of this work is on structural metadata, i.e., schemas, interface definitions, and other data-structure-like artifacts that directly affect database or other computer system operations.

The first use of structural metadata for data processing was reported in (McGee 1959). Since then, metadata-related tasks and applications have become truly pervasive. They arise in data management, website and portal

management, network management, and in various fields of computer-aided engineering. In data management, the flagship application areas that rely heavily on metadata include data integration (Batini et al. 1986), data translation (Shu et al. 1977), and database design (Wiederhold 1977). In website and portal management, metadata is used to generate entire websites from databases (Fernandez et al. 1997; Mecca et al. 1998). In network management, explicit models of devices and services are deployed to facilitate control of complex networks (Ahn 1994). In software engineering, metadata is used to describe the interfaces and behavior of software components (OMG 2002b). Feature descriptions of idealized objects such as a point mass or an ideal rope are utilized in physics tools (Kook and Novak 1991). In applications related to computation and mathematics, metadata is used to describe the properties of computer algorithms (Günther et al. 1997) or discrete optimization problems (Blanning 1982; Becker 1996).

In fact, *metadata management* plays a major role in today's information systems. In addition to the aforementioned areas, its importance has been emphasized in the context of scientific (Shoshani et al. 1984), statistical (McCarthy 1982), geographic (Blott and Vekovski 1995), and biological (Davidson et al. 1995b) information systems. The aim of metadata management is to support the design, manipulation, and maintenance of complex metadata artifacts such as database schemas, interface definitions, or website layouts.

To illustrate some typical metadata management tasks consider data integration, one of the major research topics in database systems (the tasks mentioned below are highlighted in italics). A key objective of data integration is to provide a uniform view covering a number of heterogeneous data sources. Using such a view, the data that resides at the sources can be accessed in a uniform fashion. This data is usually described using database schemas, such as relational, object-oriented, or XML schemas. To construct a uniform view, source schemas are *matched* to identify their similarities and discrepancies. The relevant portions of schemas are *extracted* and *integrated* into a uniform schema. The *translation* of data from the representation used at the sources into the representation conforming to the uniform schema is specified using database transformations, which may be expressed in SQL, XQuery, XSLT or other data manipulation languages. The queries that are stated against the uniform view are transparently *rewritten* into queries on sources. Should the source schemas change, the database transformations and the uniform schema may have to be *updated* accordingly.

Examining metadata-related tasks of data integration leads to two observations. First, these tasks are not specific to database schemas and transformations. Beside database schemas, approaches in the literature addressed integration of ontologies (Mitra et al. 2000; Noy and Musen 2000), knowledge bases (Baral et al. 1991; Subrahmanian 1994), or specifications of software components (Davies and Woodcock 1996). Second, the tasks that we listed

are not unique to data integration scenarios. Some of them have been studied in the context of different or specialized applications, such as website management or data warehousing, and became distinctive names in the literature, such as schema matching, data translation, view selection, or change management.

Although the nature of the metadata artifacts manipulated by metadata-intensive applications often differs, the addressed tasks are strikingly similar. For example, Roddick et al. (2000) notice that many approaches to change management have remarkable similarity while the subject of the change may be quite different. They suggest that development of conceptual modeling tools is needed to support change management. Other authors argue that the data translation task (Atzeni and Torlone 1996) or the schema integration task (Barsalou and Gangopadhyay 1992) can be approached in a uniform fashion for a variety of schema languages.

1.2 The Problem

Despite the commonalities in the design of metadata-intensive tools and applications, little progress has been made in metadata management in the past decades. Applications that address metadata manipulation tasks remain complex and hard to build. Several major reasons contribute to their complexity:

- Metadata applications are developed using low-level programming interfaces. Such interfaces typically provide access to the individual elements of metadata artifacts, such as individual attribute definitions of database schemas. The programming of metadata applications against such interfaces requires extensive amount of navigational code and incurs high development and maintenance cost.
- Most approaches are application-specific. That is, adopting the code and infrastructure developed say for change management to data integration requires a major customization effort.
- The solutions are language-specific, i.e., are developed for SQL, UML, XML, or RDF and are not easily portable to other domains. For example, solutions developed for change management of database schemas are hard to adopt to managing changes of websites.
- No general-purpose platform is available to simplify the development of metadata-intensive tools and applications. The existing general-purpose solutions typically focus on persistent storage or graphical design environments for metadata artifacts and do not go far enough to support the developers of metadata applications. In fact, many of today's metadata-related tasks are still solved manually, because an automated approach requires too much implementation effort due to the lack of a common programming platform.

Akin Problems Call for Akin Solutions. To understand better the nature of the problems that we address and to set the stage for the approach exploited in the thesis, it is instructive to take a brief look at the state of the art in data management that prevailed three decades ago (Wiederhold 1977; Date 1995).

In fact, there appears to be a striking similarity between today's problems in *metadata* management and the challenges in *data* management before the adoption of the relational model in 1970's. At that time, data management applications were developed using extensive amount of navigational code, which was hard to write, maintain, and optimize. The same techniques were reapplied to one new problem after another without getting much leverage from each succeeding step. The data management code was embedded into individual applications which used incompatible storage and access structures and were not portable between different domains. The existing database management systems focused on persistent storage of data but offered little help in programming of database applications.

The groundbreaking idea, which eventually revolutionized the database research field, was to raise the level of abstraction in developing data-intensive applications. In the late 1950's, McGee observed that "there are certain broad data processing operations which are common to all or most data processing applications" and suggested that the key to effective data processing was in identifying such generic operations and making them available to application developers (McGee 1959, page 6).

This idea culminated in the pioneering work by Codd (1970). Instead of then-common navigational access to individual records and data values, Codd suggested a set of algebraic operations on entire relations, such as selection, projection, or join. This approach allowed factoring out many similar aspects of data management and free application code from ordering, indexing, and access path dependencies. The relational algebra helped to drastically simplify the programming of data-intensive applications and laid out the foundation of query optimization. In fact, the relational model and algebra are considered to be "the single most important development in the entire history of the database field" (Date 1995, page 22).

1.3 A Vision for Management of Complex Models

The idea of factoring out common aspects of applications by raising the level of abstraction worked exceptionally well for data management and is, by itself, not new. However, applying a similar approach to metadata management has been suggested only relatively recently. Initial thoughts on a high-level algebraic approach and three operators for manipulation of knowledge bases, Intersection, Union, and Difference, were presented in (Wiederhold 1994). Further operators such as Extract and Match were proposed in (Jannink et al. 1999) for manipulation of ontologies, dictionaries, and schemas. More

recently, Bernstein et al. (2000b) outlined a vision to provide a truly generic and powerful environment to enable rapid development of metadata-intensive applications in different domains. They called this capability *generic model management*.

A central concept in generic model management is that of a *model*. A model is a formal description of a metadata artifact. Examples of models include database schemas, ontologies, interface definitions, object diagrams, control flow diagrams, and form definitions. The manipulation of models usually involves designing transformations between models. Formal descriptions of such transformations are called *mappings*. Examples of mappings are SQL views, XSL transformations, ontology articulations, mappings between class definitions and relational schemas, mappings between two versions of a model, mappings between device specifications and device functions, etc.

The key idea behind generic model management is to develop a set of algebraic operators that generalize the transformation operations utilized across various metadata applications. These operators are applied to models and mappings as a whole rather than to their individual elements, and simplify the programming of metadata applications. The operators are *generic*, i.e., they can be utilized for various problems and different kinds of metadata artifacts. Some of the major model management operators are:

- Match: automatically create a mapping between two models.
- Merge: merge two models into a third model using a mapping between the two models.
- Extract: return a portion of a model that participates in a mapping.
- Compose: return the composition of two mappings.

Model-management operators can be used for solving schema evolution, data integration, and other scenarios using short programs, or scripts. For example, consider the simple script shown below:

```

$$m_1\_m_2 = \text{Match}(m_1, m_2);$$


$$\langle m, m\_m_1, m\_m_2 \rangle = \text{Merge}(m_1, m_2, m_1\_m_2);$$

```

In the first line of the script, the models m_1 and m_2 are “matched”. The result of matching is the mapping $m_1_m_2$ that describes the correspondences between m_1 and m_2 . Then, the models are “merged”. The merging is driven by the mapping produced in the previous step and yields the model m and the mappings m_m_1 , m_m_2 that describe how m relates to m_1 and m_2 .

The scripts such as the one above are executed by a model management system. Although each of the operators can be invoked by the applications individually, the maximal benefit is achieved when an entire sequence of operations is passed to the model management system as a script for execution and optimization. A high-level architecture of model management is depicted in Fig. 1.1. The tools that deploy a model management system may maintain models and mappings in their own repositories. In this case, models and

mappings that are utilized in a script need to be imported into the model management system before the script runs. Alternatively, the tools may exploit the persistence capabilities of the model management system and use it as a shared repository (Do and Rahm 2000). The tools remain responsible for the management of model *instances*, such as data that resides in operational databases, XML documents, web pages, or device specifications, and may be capable of executing the mappings, i.e., transforming instances of one model into instances of another model.

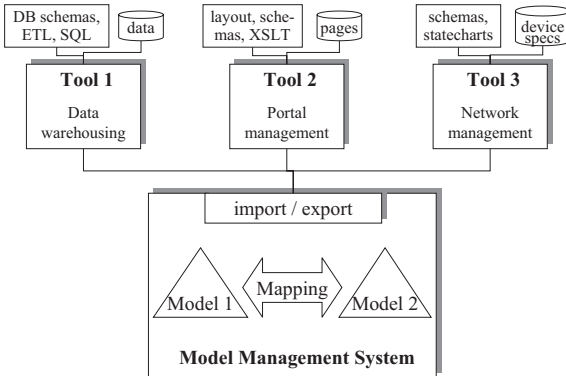


Fig. 1.1. A high-level architecture of model management

If successful, generic model management may improve programmer productivity for metadata-intensive applications by an order of magnitude. However, the vision for management of complex models raises many hard questions. In fact, at a panel that took place at the VLDB 2000 conference in Cairo it was debated whether the approach is feasible at all (Bernstein et al. 2000a). Some of the questions discussed by the panelists were the following:

- Is it feasible to develop a generic infrastructure for managing models and mappings? If so, what would it need to do, beyond what is offered in today’s database management systems and repositories?
- Can we devise a useful generic notion of model that treats all popular information structures as specializations (SQL schemas, ER diagrams, XML DTDs, object-oriented (OO) schemas, website maps, make scripts, etc.)?
- Can we produce a generic model manipulation algebra that generalizes transformation operations developed for data integration, data translation, and data warehousing?
- Does a generic approach offer any advantages for metadata management areas of current interest, such as data integration and XML?

The questions raised in the panel set the stage for the subject of this dissertation. One of the conclusions of the panel discussion was that realizing the vision of generic model management would take years of research and

that substantial implementation effort and theoretical work was required to answer the above questions to the full extent.

The objective of this first dissertation on generic model management is to demonstrate that model management operators are implementable and useful. The problem that we address is very challenging. It took dozens of Ph.D. theses, hundreds of thousands lines of code, and many years of work to demonstrate that the relational model and algebra were implementable and useful (Stonebraker 2003). We do not expect the investigation of practicality of generic model management to be any easier. In fact, an additional complicating factor is that the formal foundations of generic model management are much less clear than those underlying the relational algebra.

1.4 Outline and Contributions of the Dissertation

The dissertation presents an initial study of the concepts and algorithms for generic model management. It consists of four parts.

Part I. We present the first implemented prototype of a programming platform for model management, called Rondo¹. The prototype supports the execution of model management scripts that are written using high-level operators, which manipulate models and mappings as first-class objects. The usefulness of operators is studied in several model-management scenarios, such as change propagation and reintegration, which involve different kinds of models and mappings. In prior work, e.g., in (Bernstein et al. 2000b; Bernstein and Rahm 2000), detailed walkthroughs of various model-management problems have been examined to address the question of whether metadata management can be done in a generic fashion. Our contribution is that we succeeded in making such abstract programs executable.

Primarily, our prototype supports the developers of model-management solutions, by providing a high-level programming environment. However, it also addresses the needs of the engineers who deploy these solutions by offering a graphical user interface (GUI) to receive their feedback in semiautomatic operations. In designing and implementing our prototype, we consciously focus on simplicity. We investigate how far we can go with a comparatively weak representation of models and mappings that can be used to solve an interesting class of problems. We also determine how much code is needed for a basic, but still useful, model management system.

The conceptual structures and operators used in the prototype are presented in Chap. 2. The implementation of the prototype, its architecture, and the algorithms that we developed are addressed in Chap. 3. The results

¹ Rondo is a musical work in which the main theme returns a number of times. We called our prototype Rondo to reflect the fact that different variations of similar metadata problems keep arising in numerous applications.

presented in Part I have been published in (Melnik et al. 2003a; Melnik et al. 2003b).²

Part II. The operator definitions presented in Part I are largely syntactic: the models, such as relational and XML schemas, are represented as graphs, and the semantics of the operators is defined in terms of graph transformations. We call this semantics *structural*, since it is driven by the structural properties of models, i.e., by the relationships between the individual models elements.

And yet, the effect of applying “syntactic” operators to models ultimately needs to be expressed in terms of what the operators do to the instances of these models, such as entire database states. We call this other kind of semantics *state-based* semantics. Focusing on state-based semantics makes it possible to define the properties of operators without relying on a particular representation of models.

In Chap. 4, we define the state-based semantics for models, mappings, operators, and scripts. We present detailed examples that illustrate the state-based definitions using relational schemas and SQL views. We derive alternative formulations of operator definitions that are substantially easier to work with. In Chap. 5, we revisit the change propagation scenario presented in Part I and argue the correctness of our solution using state-based semantics. In Chap. 6, we discuss the state-based semantics of the conceptual structures and operators used in our prototype.

Part III. Although many model-management tasks can be automated, there remain critical places where human decision-making is needed, e.g., to address the semantic heterogeneity. Thus, some of the operations are inherently semiautomatic and require feedback of a human engineer before, during, or after the operator execution. The operator Match, which establishes correspondences between models, is among the most difficult to automate.

In Chap. 7, we present an algorithm called Similarity Flooding (SF) that can be used for matching of diverse data structures and is utilized for implementing the operator Match in the prototype. The input models are represented as directed labeled graphs and are used in an iterative fixpoint computation whose results tell us what nodes in one graph are similar to nodes in the second graph. For computing the similarities, we rely on the intuition that elements of two distinct models are similar when their adjacent elements are similar. Over a number of iterations, the initial similarity of any two nodes propagates through the graphs. We demonstrate the applicability of the algorithm for diverse matching tasks and examine its computational properties.

Usually, for every element in the matched models, the SF algorithm delivers a large set of match candidates. Hence, the immediate result of the fixpoint computation may still be too voluminous for many matching tasks. In Chap. 8, we examine several filters that can be used for choosing the

² Reprinted from (Melnik et al. 2003a) with permission from Elsevier.

best match candidates from the list of ranked matches returned by the SF algorithm.

The evaluation and tuning of the SF algorithm is addressed in Chap. 9. We suggest a novel accuracy metric for evaluating automatic schema matching algorithms and evaluate the effectiveness of our algorithm on the basis of a user study that we conducted. A summary of the results presented in Part III has been published in (Melnik et al. 2002).

Part IV. The individual aspects of metadata management have been studied extensively in the literature. The operator definitions that we give in Part I and the formal properties of the operators that we examine in Part II are inspired by the established model-management problems and scenarios, such as data integration, schema matching, view selection, or view complement.

In Chap. 10, we review in detail the major related work and show how our operator definitions reflect the properties of the approaches suggested in the literature. We also sum up our prior work on declarative mediation that served as part of the motivation to address metadata management in a generic fashion.

Generic model management is an extremely rich emerging area of research. This dissertation presents a first treatment of some fundamental challenging issues in this area. In our work, we uncovered a wide spectrum of exciting open problems, which are summarized in Chap. 11.

2. Conceptual Structures and Operators

“I can’t work without a model.”

– Vincent Van Gogh (1853-1890)

In this chapter, we describe the conceptual structures and operators that are used in the prototype of a programming platform for model management that we developed. The chapter is organized as follows.

- In Sect. 2.1, we walk through a model-management scenario to motivate the conceptual structures and operator definitions that we present.
- In Sect. 2.2, we introduce conceptual structures used for representing models and mappings. We explore a simple class of mappings between models that we call morphisms and suggest a new structure called selector.
- In Sect. 2.3, we define the structural semantics of the key model-management operators on the conceptual structures that we introduce, and suggest several new generic operators.

2.1 Motivating Scenario

To motivate the operator definitions that we give in this chapter, we use a scenario that is illustrated in Fig. 2.1 and exemplifies one of the patterns that can be found in many metadata-intensive applications.

Example 2.1.1. Consider an e-commerce company that needs to supply its purchase order data to a business partner that does the accounting, invoicing or data warehousing. The data is stored in a relational database according to a relational schema s_1 . For the purpose of data exchange, both companies agree to use a common XML schema d_1 . (The correspondences between the elements of schema s_1 and d_1 are depicted as light gray lines). Schema d_1 differs from s_1 in terms of structure and naming.

The relational schema used by the company undergoes periodic changes due to the dynamic nature of its business. Assume that s_2 is a new version

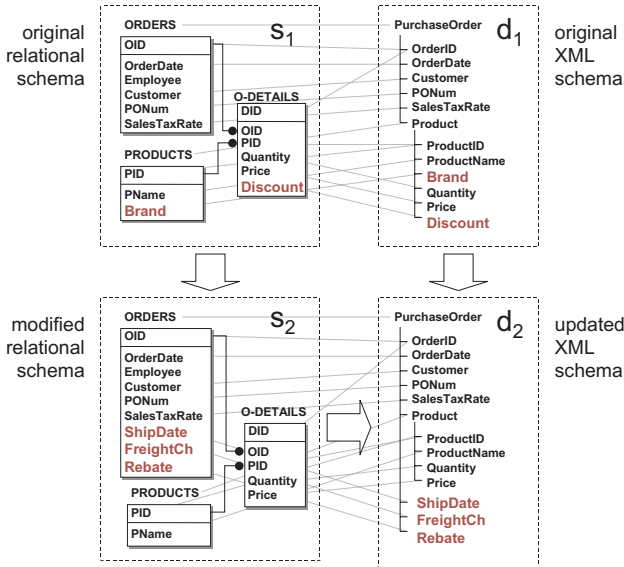


Fig. 2.1. Scenario illustrating propagation of changes from a relational schema to an XML schema

of the relational schema s_1 , in which columns “Brand” and “Discount” have been deleted, and columns “ShipDate”, “FreightCh” (freight charge), and “Rebate” have been added. These changes (highlighted in bold in Fig. 2.1) need to be propagated to the XML schema, so that d_1 becomes d_2 .

The change propagation described above can be done as follows. First, the changes introduced by s_2 need to be detected, i.e., s_1 and s_2 need to be matched. Then, the d_1 images of the elements deleted in s_1 need to be removed from d_1 . Finally, the XML schema counterparts of the added and renamed columns in s_1 need to be merged into d_1 to obtain d_2 . During these steps, intervention of a human engineer may be required, for example, to decide whether the new column “Rebate” should indeed be added to the exchange schema or is not part of the exchanged data and should be omitted. Still, a major portion of the work is mechanical and can be automated.

Notice that the procedure sketched above could be applied in the reverse case, when the XML schema d_1 is the one that has been modified and the changes are to be propagated back to the relational schema s_1 . Another instance of the same pattern is round-tripping the modifications from a relational schema like s_1 to an existing conceptual schema of the data, which may be expressed as an Entity-Relationship (ER) diagram. A key idea of generic model management is to solve such tasks at a high level of abstraction using a concise generic script.

Below we present an actual model-management script that implements the above solution for our change propagation scenario, and is directly exe-

cutable by our prototype. We will use the script to introduce the major model-management operators, which we define in the subsequent sections. To explain the individual steps of the script, we use a schematic representation of the solution shown in Fig. 2.2. The rectangles labeled s_1 , s_2 , d_1 , and d_2 represent the four schemas of Fig. 2.1. The arcs between the rectangles denote the mappings between the schemas. For example, the correspondences between schemas s_1 and d_1 in Fig. 2.1 are shown as a single arc from rectangle s_1 to d_1 in Fig. 2.2.

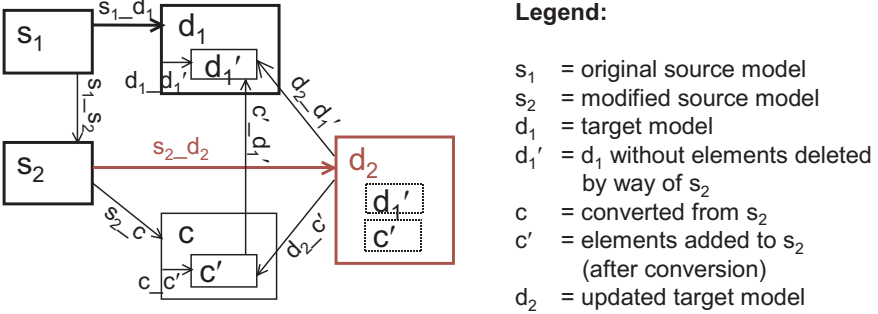


Fig. 2.2. Schematic representation of a solution for change propagation scenario of Fig. 2.1

At the bottom of Fig. 2.2, there is a schema c , which does not appear in Fig. 2.1. To see why it is needed, recall that s_1 and d_1 are expressed using two different schema languages. The new schema elements added to s_1 by way of s_2 have no counterparts in schema d_1 . That is, the new elements need to be converted from the source schema language to the target language. For example, the attribute “ShipDate” added to relation “ORDERS” needs to be converted to a subelement of the complex type “PurchaseOrder” in the XML schema. This step is often referred to as schema translation in the literature. In our solution, we assume that such a translation tool is available as an operator, say SQL2XSD, which takes as input a relational schema and produces as output an XML schema and a mapping between the original and converted schema elements. Thus, the schema c and the mapping s_2_c between s_2 and c shown in Fig. 2.2 are obtained as $\langle c, s_2_c \rangle = \text{SQL2XSD}(s_2)$. Schema c is illustrated in Fig. 2.3. Note that c is not yet the desired result d_2 ; for example, c contains an unneeded complex type O-DETAILS, and differs from d_2 structurally.

Now, our solution for the change propagation scenario can be expressed as the following script:

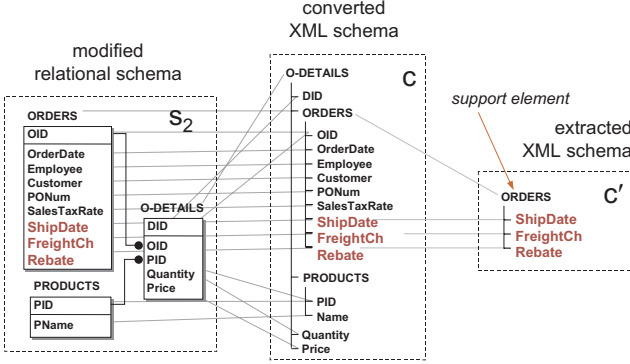


Fig. 2.3. Converted schema c and support element ORDERS in c'

operator $\text{PropagateChanges}(s_1, d_1, s_1_d_1, s_2, c, s_2_c)$

1. $s_1_s_2 = \text{Match}(s_1, s_2)$;
2. $\langle d'_1, d_1_d'_1 \rangle = \text{Delete}(d_1, \text{Traverse}(\text{All}(s_1) - \text{Domain}(s_1_s_2), s_1_d_1))$;
3. $\langle c', c_c' \rangle = \text{Extract}(c, \text{Traverse}(\text{All}(s_2) - \text{Range}(s_1_s_2), s_2_c))$;
4. $c'_d'_1 = \text{Invert}(c_c') * \text{Invert}(s_2_c) * \text{Invert}(s_1_s_2) * s_1_d_1 * d_1_d'_1$;
5. $\langle d_2, d_2_c', d_2_d'_1 \rangle = \text{Merge}(c', d'_1, c'_d'_1)$;
6. $s_2_d_2 = s_2_c * c_c' * \text{Invert}(d_2_c') + \text{Invert}(s_1_s_2) * s_1_d_1 * d_1_d'_1 * \text{Invert}(d_2_d'_1)$;
7. return $\langle d_2, s_2_d_2 \rangle$;

The script defines a generic operator PropagateChanges , which takes six parameters as input (including the converted schema c), and produces two return values $\langle d_2, s_2_d_2 \rangle$ as output. Below, we explain the script line by line.

1. In line 1, schemas s_1 and s_2 are “matched” to detect the changes. The result is a mapping $s_1_s_2$ shown schematically in Fig. 2.2. Speaking informally, the mapping connects the equivalent elements of s_1 and s_2 . The new elements of s_2 (e.g., “ShipDate”) and deleted elements of s_1 (e.g., “Brand”) have no matching counterparts, so they remain unconnected.
2. Line 2 illustrates how operators can be combined. First, the deleted elements of s_1 are identified using the expression $\text{All}(s_1) - \text{Domain}(s_1_s_2)$, i.e., all elements of s_1 without the matched (and thus not deleted) elements. Then, these elements are used to “traverse” the mapping $s_1_d_1$. For example, the deleted relational attribute “Brand” traverses $s_1_d_1$ and yields the XML schema element “Brand” of d_1 . Finally, these d_1 images of the deleted elements are removed from d_1 using the operator Delete . The result is a new schema d'_1 (a “subschema” of d_1), and a mapping $d_1_d'_1$, which describes how d_1 relates to d'_1 .
3. Line 3 is quite similar to line 2. The new elements of s_2 , i.e., those missing from the range of $s_1_s_2$, traverse s_2_c into the converted model c . For example, the image of relational attribute “ShipDate” is an XML schema

element “ShipDate” obtained by conversion. A “subschema” c' containing the images of the new elements is then extracted from c using the operator *Extract*, which also returns the mapping c_c' . In addition to the elements obtained by traversal like “ShipDate”, c' contains an extra element of c , the complex type “ORDERS” that encloses “ShipDate”. Such extra elements are called *support* elements (Bernstein 2003). Support elements may have to be extracted to make c' a well-formed XML schema.

4. At this point, d'_1 is a subschema of d_1 without the deleted elements, and c' contains the added elements and their support elements. Schemas d'_1 and c' need to be merged to obtain the final result d_2 (line 5). As we explain in Sect. 2.3.5, the merging of two schemas is driven by a mapping that tells how elements of the two schemas, specifically the support elements of c' , correspond to each other. The mapping between d'_1 and c' is shown in Fig. 2.2 as an arc connecting the two enclosed rectangles. This mapping can be obtained by “composing” the existing mappings between c' , c , s_1 , s_2 , d_1 , and d'_1 as $\text{Invert}(c_c') * \text{Invert}(s_2_c) * \text{Invert}(s_1_s_2) * s_1_d_1 * d_1_d'_1$. To get the composition right, mappings c_c' , s_2_c , and $s_1_s_2$ need to be ‘inverted’, i.e., the domains and ranges of the mappings need to be swapped. Thus, we determine by composition that the support element “ORDERS” in c' corresponds to the element “PurchaseOrder” in d'_1 .
5. The final result of change propagation, schema d_2 , is computed by the *Merge* operator. Among other things, the operator *Merge* creates a single complex type definition from complex type “ORDERS” from c' and “PurchaseOrder” from d'_1 . Additionally, the operator returns two mappings, d_2_c' and $d_2_d'_1$, which describe how d_2 relates to the inputs to *Merge*, c' and d'_1 .
6. As a last step, we compute $s_2_d_2$, a new version of the mapping $s_1_d_1$ given as part of the input. We need $s_2_d_2$ to ensure that our change propagation script can be re-applied if the source schema evolves again. Since d_2 is obtained by merging d'_1 and c' , the mapping $s_2_d_2$ is essentially a union of two mappings, the one between s_2 and the d'_1 -portion of d_2 , and the one between the s_2 and c' -portion of d_2 . These two mappings can be obtained by composition as $s_2_c * c_c' * \text{Invert}(d_2_c')$ and $\text{Invert}(s_1_s_2) * s_1_d_1 * d_1_d'_1 * \text{Invert}(d_2_d'_1)$, respectively. Their union is denoted using the plus sign (+). To illustrate, the first mapping establishes the correspondences between the added elements “ShipDate”, “FreightCh”, “Rebate” in s_2 and their d_2 counterparts. The second mapping in the union tells us that “OID” in s_2 corresponds to “OrderID” in d_2 , etc.

Notice that the above script is not limited to propagating changes from relational schemas to XML schemas. In fact, the reverse propagation problem can be solved using the same script by assigning the original and modified XML schemas to s_1 and s_2 , and the relational schema to d_1 . Of course, the

input parameters c and s_2_c need to be obtained using a different converter, e.g., as $\langle c, s_2_c \rangle = \text{XSD2SQL}(s_2)$.

In our implementation, every intermediate result of a script such as the one above can be examined and adjusted by a human engineer using a graphical tool. Specifically, the result of `Match` in line 1 can be post-processed to remove the incorrectly suggested matches and add the missing ones. Similarly, the merging step of line 5 is in general a semiautomatic process, which may require human feedback. Finally, by adjusting the intermediate results of operator compositions in lines 2 and 3 the engineer can decide which additions and deletions should not be propagated.

In the above discussion, we introduced several operators informally. To make these operators effective and usable by developers, their semantics needs to be specified precisely. Our goal is to make the semantics as “generic” as possible, so the operators can serve a broad range of model-management tasks. In the next two sections we describe this semantics, first by defining the structures on which they operate, and then by describing the operators themselves.

2.2 Conceptual Structures

Model-management applications deal with a wide range of metadata artifacts, which include not only schemas, such as the relational and XML schemas in our motivating scenario, but also view definitions, interface specifications, etc. We represent the formal descriptions, or models, of these artifacts as directed labeled graphs. This graph representation is quite flexible and can accommodate virtually any type of model.

We also introduce two additional structures, called *morphisms* and *selectors*. Morphisms are binary relationships that establish $n : m$ correspondences between the elements of two models (i.e., nodes of two graphs). For example, in our motivating scenario morphisms are used for keeping track of the XML counterparts of the relational schema elements. Two morphisms, one between s_1 and d_1 and another between s_2 and d_2 , are shown in Fig. 2.1 using light gray lines. The third conceptual structure, selector, is a set of elements used in models. A major benefit of using selectors is that various operations, in particular the set operations, which would typically produce non-well-formed models if used on models, can be applied to selectors safely.

In the following subsections, we define models, morphisms, and selectors as abstract graph and set structures. We also describe them in an equivalent representation as relations. The latter will make it easier to define the semantics of the operators, which follow later.

We briefly review the conventional metadata terminology that we use below. A *meta-model* can be thought of as a model that describes the structure of another model. Typically, it contains the type definitions for the objects used in models. For example, the Open Information Model (OIM) (Bernstein

et al. 1999) defines meta-models for several database schema and transformation languages. A *meta-meta model* is a representation language in which models and meta-models are represented. For example, the Unified Modeling Language (UML) specification uses an object-oriented meta-meta model called MOF (Meta-Object Facility (OMG 2002a)). The meta-meta model of our prototype, which we discuss below, is based on directed labeled graphs. All models and meta-models can be viewed as instances of the meta-meta model.

2.2.1 Models

We represent models as directed labeled graphs. The nodes of such graphs denote *model elements*, such as relations and attributes in relational schemas, type definitions in XML schemas, clauses of SQL statements, etc. We assume that each element is uniquely identified by an object identifier (OID). A directed labeled graph is a set of edges $\langle s, p, o \rangle$ where s is the source node, p is the edge label, and o is the target node¹. The order of the nodes in a graph can be captured by an ordinal property on edges. Thus, conceptually a graph can be viewed as a relation M with four attributes, $M(S: \text{OID}, P: \text{OID}, O: \text{OID} \cup \text{Literal}, N: \text{integer})$, where N is an optional attribute used for ordering and S, P, O form a unique key. The node identifiers and edge labels are drawn from the set of OIDs, which can be implemented as integers, pointers, URIs, etc. The literals include strings, integers, floats, and other data types. The type of attribute O is defined as a union type of OIDs and literals.

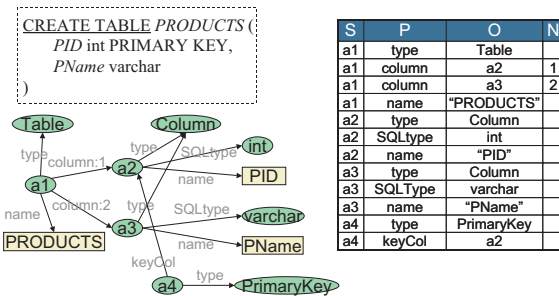


Fig. 2.4. Sample model shown as graph and 4-tuples

Consider the example in Fig. 2.4. It illustrates how a relational table PRODUCTS defined in SQL DDL (top left) is represented as a graph (bottom left) and as a corresponding set of 4-tuples (on the right). The ovals in the graph denote OIDs, and rectangles denote literals. Nodes a1, a2, a3 represent the table PRODUCTS and its columns PID and PName, respectively. Node a4 represents the primary key constraint on PID. For readability, the identifiers such as Table or Column are spelled out as names rather than opaque IDs.

¹ The notation $\langle s, p, o \rangle$ stands for (subject, predicate, object).

The order of the columns identified by the nodes `a2` and `a3` is determined by the values 1 and 2 of attribute `N` (fourth attribute of the table with 4-tuples). In general, the node ordering for a given `{src node}` and `{edge label}` is determined by the SQL query: `SELECT M.O FROM M WHERE M.S={src node} AND M.P={edge label} ORDER BY M.N`. In the example, we have `M.S=a1 AND M.P=column`.

A formal specification of the rules for encoding a model as a graph is called a *meta-model*. A model is *well-formed* if it conforms to its meta-model. For example, Fig. 2.4 illustrates a graph encoding of relational schemas that uses specific edge labels, such as `SQLtype` or `name`, and auxiliary nodes, such as `Table`, `varchar`, or `PrimaryKey`. If we know the relational meta-model, we can tell whether or not a given graph represents a well-formed relational schema. For example, if we know that each column must have an SQL type, then removing the edge `(a2, SQLtype, int)` from the graph in Fig. 2.4 yields a model that is not well-formed. For the purposes of this chapter, it is unimportant how a meta-model is represented and how one checks that a model conforms to its meta-model. The details of the graph representation of models remain opaque to the developer of model management applications. Of course, the representation is visible to developers of model management operators. So, a developer must be aware of the representation to implement a custom, non-generic operator, e.g., an operator to normalize relational schemas.

2.2.2 Morphisms

Many metadata-intensive applications, such as data integration and warehousing tools, use a graphical metaphor like the one shown in Fig. 2.1 for representing schema mappings. These mappings are shown to the engineer as sets of lines connecting the elements of two schemas. We call such mappings (schema) morphisms. Thus, a morphism is a binary relation over two (possibly overlapping) sets of OIDs, i.e., a set of pairs $\langle l, r \rangle$ drawn from $\text{OID} \times \text{OID}$.

Clearly, a morphism is a weaker representation of a transformation between two models than an SQL view or the mapping languages and expressions suggested in (Bergamaschi et al. 1999; Bernstein et al. 2000b; Davidson et al. 1995a; Miller et al. 1994; Mitra et al. 2000). In particular, a morphism carries no semantics about the transformation of instances that conform to the models (e.g., no SQL `WHERE`-clause). Still, we have found that many mappings can be expressed in this way such as in our change propagation scenario of Sect. 2.1. The morphisms have several other advantages. Given our graph representation of models, a morphism can represent a mapping between different kinds of models, e.g., between a relational and XML schema. A morphism can always be inverted and composed. (In contrast, an SQL view cannot be composed with an XSL transformation in an obvious way). And since morphisms can be expressed as binary relations, they can be implemented and manipulated easily.

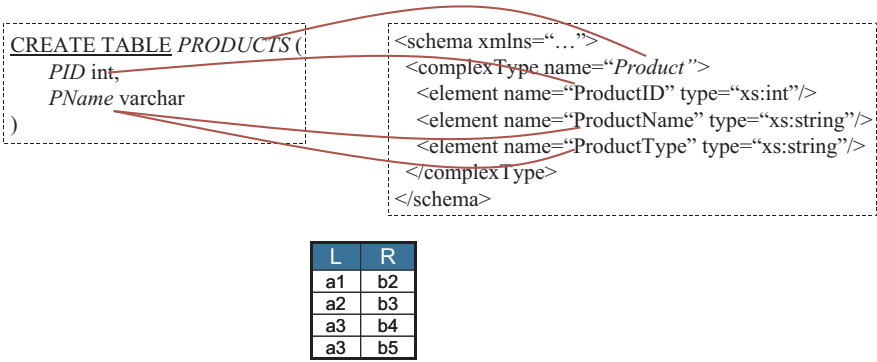


Fig. 2.5. A morphism between a relational and an XML schema

Consider the example in Fig. 2.5. The top part of the figure shows the relational schema of Fig. 2.4 and an XML schema. A morphism between the two schemas is depicted graphically as four arcs that connect the elements of the schemas. The bottom part of the figure shows the same morphism represented as a relation. The node identifiers a1, a2, a3 correspond to those of Fig. 2.4. The nodes b2, b3, b4, b5 denote respectively the complex type “Product” and the elements “ProductID”, “ProductName”, and “ProductType” defined in the XML schema (its graph representation is illustrated in Fig. 2.6). Notice that a node can be connected to multiple nodes; e.g., a3 (“PName”) is connected to b4 (“ProductName”) and b5 (“ProductType”). Moreover, various kinds of model elements, such as relations or attributes, can participate in a morphism.

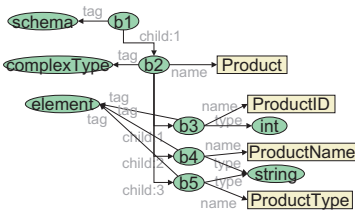


Fig. 2.6. Graph representation of XML schema in Fig. 2.5

In an implementation, it may be convenient to annotate the pairs $\langle l, r \rangle$ with additional properties. For example, most implementations of the Match operator compute similarity values between the elements of two models. These values can be returned conveniently using a morphism in which each pair has an additional similarity property. Hence, although we define a morphism conceptually as a binary relation $H(L: \text{OID}, R: \text{OID})$, it may contain additional attributes, as required by the individual operators. Typically, the L elements originate from one model, and the R elements from another.

2.2.3 Selectors

A selector is a set of node identifiers, which may originate from a single or multiple models. It can be represented as a relation with a single attribute, $S(V: \text{OID})$, where V is a unique key. Fig. 2.7 shows an example of a selector that contains all OIDs used in the model depicted in Fig. 2.4.

V
a1
a2
a3
a4
Table
Column
PrimaryKey
int
varchar

Fig. 2.7. Example of a selector

2.3 Operators

In our motivating scenario, we introduced several high-level operators whose inputs and outputs are models, morphisms and selectors, such as *Match*, *Delete*, *Traverse*, *Extract*, and *Invert*. Such operators raise the level of abstraction of manipulating metadata structures by considering whole models and morphisms at a time, as opposed to using node-at-a-time primitives. For easy reference, the signatures and informal descriptions of the operators that are used in scripts most frequently are summarized in Table 2.1. In this section, we define the precise semantics of these operators on the structures defined in Sect. 2.2. We call this semantics *structural*. The implementation of the operators is covered in Chap. 3.

We start our presentation of operator semantics in Sect. 2.3.1 with what we call *primitive* operators. These are generic operators whose semantics can be defined formally using the relational algebraic manipulation of the relational representations of Sect. 2.2. For notational convenience, we express this manipulation in SQL. After that, we introduce the other more powerful operators: such as *Extract*, *Delete*, *Match*, and *Merge*, whose semantics is more subtle and still a subject of ongoing research².

As we will see, some operators, such as *Subgraph* or *Copy*, are agnostic about the kind of models passed as input, whereas the semantics of others depends on the underlying meta-model. The GUI operators *EditMap* and *EditSelector* allow arbitrary transformations of morphisms and selectors by an engineer. Thus, their semantics cannot be constrained any further.

² In (Melnik et al. 2003a; Melnik et al. 2003b), we used slightly different operator signatures for *Extract*, *Merge*, and *Diff*. In this dissertation, we changed the directionality of the output mappings of these operators to facilitate a more natural notation when the mappings are functional. This difference is, however, purely syntactic.

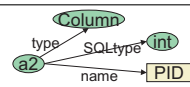
Table 2.1. Summary of key operators in Rondo

Signature	Description
Primitive operators	
$s = \text{Domain}(map)$	Returns selector s that holds the elements in the domain of morphism map
$map_2 = \text{Invert}(map_1)$	Swaps the “left” and “right” side of the input morphism map_1
$map_2 = \text{RestrictDomain}(map_1, s)$	Restricts the domain of morphism map_1 to the elements in selector s
$s = \text{All}(m)$	Returns a selector s containing all elements of model m
$map = \text{Id}(s)$	Returns the identity morphism map for selector s
$m_1_m_3 = \text{Compose}(m_1_m_2, m_2_m_3)$ $= m_1_m_2 * m_2_m_3$	Composes morphisms $m_1_m_2$ and $m_2_m_3$
Derived operators	
$s = \text{Range}(map)$	Returns selector s that holds the elements in the range of morphism map
$map_2 = \text{RestrictRange}(map_1, s)$	Restricts the range of morphism map_1 to the elements in selector s
$s_2 = \text{Traverse}(s_1, map)$	Returns selector s_2 holding the elements in the range of morphism map that are reachable by traversing map from s_1
$\langle m_d, m_m_d \rangle = \text{Delete}(m, s)$	Returns a “submodel” m_d of m that does <i>not</i> contain the elements in selector s
More complex operators	
$\langle m_x, m_m_x \rangle = \text{Extract}(m, s)$	Extracts a “submodel” m_x of m that contains the elements in selector s
$m_1_m_2 = \text{Match}(m_1, m_2 [, seed])$	Computes a morphism $m_1_m_2$ between m_1 and m_2 using an optional initial morphism $seed$
$\langle m, m_m_1, m_m_2 \rangle =$ $\text{Merge}(m_1, m_2, m_1_m_2)$	Merges models m_1 and m_2 using morphism $m_1_m_2$

2.3.1 Primitive Operators

Table 2.2 lists the definitions of seven primitive operators. The left column contains the operator definitions expressed in SQL. Variables m , s , and map hold a model, a selector, and a morphism, respectively. The right column illustrates the application of the operators using simple examples. All primitive operators defined in the table are standard set-theoretic operators. Notice that their definitions are expressed declaratively, i.e., the implementation of these operators, or functional combinations thereof, can be optimized using standard query optimization techniques.

Table 2.2. Definitions of primitive operators

Definition	Example
$\text{Domain}(map) :=$ SELECT DISTINCT <i>map</i> .L AS V FROM <i>map</i>	$\text{Domain}\left(\begin{array}{ c c } \hline a1 & b1 \\ \hline a2 & b2 \\ \hline \end{array}\right) = \begin{array}{ c } \hline a1 \\ \hline \end{array}$
$\text{RestrictDomain}(map, s) :=$ SELECT * FROM <i>map</i> WHERE <i>map</i> .L IN <i>s</i>	$\text{RestrictDomain}\left(\begin{array}{ c c } \hline a1 & b1 \\ \hline a2 & b2 \\ \hline \end{array}, \begin{array}{ c } \hline a1 \\ \hline \end{array}\right) = \begin{array}{ c c } \hline a1 & b1 \\ \hline \end{array}$
$\text{Invert}(map) :=$ SELECT <i>map</i> .R AS L, <i>map</i> .L AS R FROM <i>map</i>	$\text{Invert}\left(\begin{array}{ c c } \hline a1 & b1 \\ \hline a2 & b2 \\ \hline \end{array}\right) = \begin{array}{ c c } \hline b1 & a1 \\ \hline b2 & a2 \\ \hline \end{array}$
$\text{Compose}(map_1, map_2) :=$ SELECT DISTINCT <i>map</i> ₁ .L, <i>map</i> ₂ .R FROM <i>map</i> ₁ , <i>map</i> ₂ WHERE <i>map</i> ₁ .R = <i>map</i> ₂ .L	$\text{Compose}\left(\begin{array}{ c c } \hline a1 & b1 \\ \hline a2 & b2 \\ \hline \end{array}, \begin{array}{ c c } \hline b1 & c1 \\ \hline \end{array}\right) = \begin{array}{ c c } \hline a1 & c1 \\ \hline \end{array}$
$\text{TransitiveClosure}(map) :=$ WITH RECURSIVE TC(L, R) AS (<i>map</i> UNION SELECT DISTINCT TC.L, <i>map</i> .R FROM TC, <i>map</i> WHERE TC.R = <i>map</i> .L) SELECT * FROM TC	$\text{TransitiveClosure}\left(\begin{array}{ c c } \hline a & b \\ \hline b & c \\ \hline \end{array}\right) = \begin{array}{ c c } \hline a & b \\ \hline b & c \\ \hline a & c \\ \hline \end{array}$
$\text{Id}(s) :=$ SELECT <i>s</i> .V AS L, <i>s</i> .V AS R FROM <i>s</i>	$\text{Id}\left(\begin{array}{ c } \hline a1 \\ \hline a2 \\ \hline \end{array}\right) = \begin{array}{ c c } \hline a1 & a1 \\ \hline a2 & a2 \\ \hline \end{array}$
$\text{Subgraph}(m, s) :=$ SELECT * FROM <i>m</i> WHERE <i>m</i> .S IN <i>s</i> AND (<i>m</i> .O IN <i>s</i> OR isLiteral(<i>m</i> .O))	$\text{Subgraph}(M, \begin{array}{ c } \hline a2 \\ \hline \text{Column} \\ \hline \text{int} \\ \hline \end{array}) =$  <p>M = model of Fig. 2.4</p>

The operator Domain extracts the “left” elements from a morphism and returns a selector that holds the result. The operator RestrictDomain restricts a morphism to a smaller element domain, which is specified by the selector passed as a second parameter of the operator. The Invert operator swaps the left and right elements of a morphism. The Compose (*) operator is defined as the natural join of two morphisms, yielding another morphism. The TransitiveClosure operator on morphisms is specified using a recursive SQL definition. The Id operator creates an identity morphism over a given selector.

The operator Subgraph(*m*, *s*) extracts from model *m* a subgraph induced by the nodes referenced in *s*. The literals attached to the nodes in *s* are also extracted from *m*. In the example of Table 2.2, the literal “PID” is not contained in the input selector *s*, but the edge ⟨a2, name, “PID”⟩ is nevertheless returned as part of the result. The extracted subgraph may not

be a well-formed model. That is, it may not be fully connected and may not conform to its meta-model.

The set operators Union (+), Difference (-), and Intersection (\cap) are another three important primitive operators. We define these on models, morphisms, and selectors by the corresponding set operations on their representation as relations. For example,

$$\text{Union}(x, y) := \text{SELECT } * \text{ FROM } x \text{ UNION SELECT } * \text{ FROM } y$$

Note that applying the set operations to well-formed models may produce a model that is not well-formed.

The last two primitive operators are All and Copy. The operator All(m) returns a selector that contains only those nodes of m that denote the model elements of the model's meta-model, such as tables or columns in the relational meta-model. For example, for the model of Fig. 2.4 the operator All yields the selector {a1, a2, a3, a4} and filters out all auxiliary nodes, such as Table or PrimaryKey, that are used in the graph encoding.

Frequently, it is important to ensure that a given node identifier is used in exactly one model. Furthermore, unique node IDs make it possible to refer to model elements across model boundaries. For these reasons, we use the operator Copy to create a copy of a model m in which the selected node IDs are replaced by new, uniquely created IDs. In the following definition of Copy, the function uniqueOID() generates a unique OID on each call, and the function ifNULL(x, y, z) returns y whenever x is a NULL value, z otherwise. If $s = \text{All}(m)$, the output morphism m_m' is a bijection between All(m) and All(m').

```
Copy( $m, s$ ) :=
   $m\_m' = \text{SELECT } s.V \text{ AS } L, \text{ uniqueOID}() \text{ AS } R \text{ FROM } s;$ 
   $m' = \text{SELECT ifNULL}(T1.R, m.S, T1.R), m.P,$ 
           ifNULL(T2.R, m.O, T2.R)
           FROM  $m, m\_m'$  AS T1,  $m\_m'$  AS T2
           LEFT OUTER JOIN ON  $m.S=T1.L, m.O=T2.L;$ 
  return  $\langle m', m\_m' \rangle;$ 
```

Fig. 2.8 illustrates the operator Copy. The operator takes as input the model m of Fig. 2.4 and selector {a1, a2, a3, a4} = All(m). As a result of copying, a new model has been created (on the right), in which the nodes IDs a1, a2, a3, a4 have been replaced by the generated unique IDs a5, a6, a7, a8, respectively.

2.3.2 Derived Operators

The derived operators are functional combinations of other operators. For example, consider the definitions shown below.

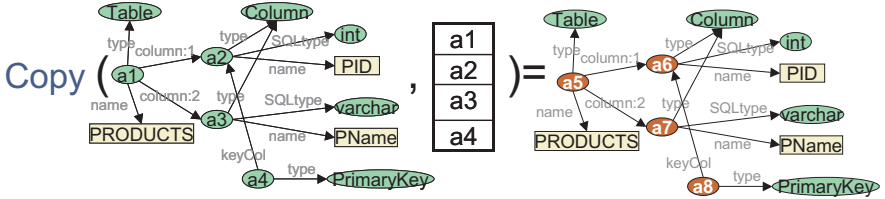


Fig. 2.8. Examples of copying the model of Fig. 2.4 using selector $\{a1, a2, a3, a4\}$

```

operator Range(map)
    return Domain(Invert(map));

operator RestrictRange(map, s)
    return Invert(RestrictDomain(Invert(map), s));

operator Traverse(s, map)
    return Range(RestrictDomain(map, s));

operator Restrict(map, m1, m2)
    return RestrictRange(RestrictDomain(map, All(m1)), All(m2));
    
```

The Range of a morphism is obtained as the domain of an inverted morphism, by combining the primitive operators Domain and Invert of Table 2.2. Similarly, RestrictRange is specified in terms of the operator RestrictDomain by first inverting the input morphism, then applying RestrictDomain, and finally inverting the resulting morphism once again.

The third operator, Traverse, was used in our motivating scenario for locating the d_1 images of the elements deleted from the relational schema s_1 . To traverse the nodes in the selector over a morphism, the morphism is first domain-restricted by the selector, and the range of the restricted morphism is returned as output.

The last operator, Restrict, confines the domain and range of a morphism to the elements of two models m_1 and m_2 . Notice that the definitions of the derived operators above are expressed declaratively, allowing the implementations to be optimized.

2.3.3 Extract and Delete

Extracting and deleting portions of models are operations that are heavily deployed in metadata applications. To perform these operations, we propose the generic operators Extract and Delete. The operator Extract is applied as follows: $\langle m', m_m' \rangle = \text{Extract}(m, s)$. The inputs are a well-formed model m and a selector s that identifies the set of nodes to be extracted. The output model m' satisfies the following properties:

- i. m' contains all selected nodes,
- ii. m' is a well-formed model,

- iii. m' is an equally or less expressive model than m , i.e., m can represent all information of m' , and
- iv. m' is a ‘minimal’ model that satisfies (i)-(iii).

Condition (iii) can be characterized formally in terms of dominance and information capacity as suggested in (Hull 1986; Miller et al. 1994). The morphism $m _ m'$ is an injective function from $\text{All}(m)$ to $\text{All}(m')$, i.e., each model element of m has at most one counterpart in m' .

In general, a model may contain implicit information, such as transitive relationships between model elements. In such cases, the result of `Extract` may need to make such information explicit. For example, consider a class diagram with three classes A , B , C , and two explicit subclass definitions: A is a subclass of B , and B is a subclass of C . Due to condition (iii), `Extract(m , $\{A, C\}$)` should return a class diagram in which A is defined as a subclass of C . This example illustrates that extraction is a rich operation, whose semantics and implementation may be non-trivial.

Conceptually, the semantics of the operator `Extract(m , s)` can be realized using the following algorithm:

1. Create a “closure” of m , i.e., a model m' in which all implicit information of m is represented explicitly.
2. Assign $s' = s$, where s' is a temporary selector.
3. For each x in s' , extend s' with elements needed to satisfy conditions (ii) and (iii).
4. Apply 3 until a fixpoint is reached, i.e., s' will not change.
5. Extract subgraph t' induced by s' as $t' = \text{Subgraph}(m', s')$.
6. Obtain a “cover” of t' , i.e., a minimal model t that is semantically equivalent to t' .
7. Return `Copy(t , $\text{All}(t)$)` as result of extraction. Notice that the operator `Copy` (Sect. 2.3.1) returns a model and a mapping.

Deleting a selected portion of a model can be defined as extraction of the unselected portion. Thus, we define

```
operator Delete( $m$ ,  $s$ )
  return Extract( $m$ ,  $\text{All}(m) - s$ );
```

Note that the nodes of s that do not represent the model elements of m , i.e., are not members of $\text{All}(m)$, have no impact on the result of deletion due to applying $\text{All}(m) - s$.

2.3.4 Match

The purpose of `Match` is to uncover how two models “correspond” to each other. It takes two models as input and returns a morphism between them. `Match` is inherently heuristic. So following the previous literature on `Match`

(Rahm and Bernstein 2001), we do not offer a formal definition of what constitutes a correct output morphism. In general, matching two schemas requires information that is not present in the schemas and cannot be fully automated. Hence, a human engineer needs to review and adjust the suggestions produced by an automatic procedure, either in a post-processing step or iteratively.

2.3.5 Merge

To combine two models into one, we utilize the operator *Merge*, applied as $\langle m, m_{_}m_1, m_{_}m_2 \rangle = \text{Merge}(m_1, m_2, \text{map})$. If the input models m_1 and m_2 are well-formed, *Merge* should produce a well-formed model m that

- i. is at least as expressive as each of the input models, i.e., capable of representing the information contained in both models, and
- ii. is “minimal”, i.e., the elements shared between the input models are not replicated unnecessarily.

The third parameter to *Merge* is a morphism *map* that describes model elements of m_1 and m_2 that are equivalent and should be “merged” into a single model element in m . The output morphisms $m_{_}m_1$ and $m_{_}m_2$ identify the counterparts of the elements of m_1 and m_2 in the merged model m .

The conceptual definition of *Merge* given above does not say anything about the naming and ordering of model elements. For example, it does not prescribe that the attribute names of m_1 take precedence over those of m_2 , or the other way around. These details are not considered to be part of the semantics of *Merge* because they inherently involve end-user decision making. They are discussed in Sect. 3.2.7.

In the next chapter, we discuss the implementation of the conceptual structures and operators presented above.

3. Implementation and Applications

“My way is to seize an image that moment it has formed in my mind, to trap it as a bird and to pin it at once to canvas. Afterward I start to tame it, to master it. I bring it under control and I develop it.”

– Joan Miró (1893-1983)

This chapter is devoted to the implementation and deployment aspects of the first prototype for model management developed as part of the thesis. The chapter is structured as follows:

- In Sections 3.1 and 3.2, we describe the implementation of the conceptual structures and operators, respectively. In particular, we present new algorithms developed for the operators `Extract` and `Merge`.
- In Sect. 3.3, we present our prototype in more detail and demonstrate how it can be extended to embrace new kinds of models.
- In Sections 3.4 and 3.5, we examine the solutions for two further important model-management tasks, view reuse and reintegration, that involve manipulations of relational schemas, XML schemas, and SQL views.

We conclude the chapter and Part I in Sect. 3.6.

3.1 Conceptual Structures

In this section we discuss our implementation of the conceptual structures. We have found that the relations that were used in Sect. 2.2 as standard mathematical representation of graphs actually are a convenient implementation structure too. Our graph representation is based on the classical relational data model, in which node identifiers are constants that can be shared across models. We chose a relational approach instead of an object-oriented one (e.g., the one in (Bernstein et al. 2000b)) to simplify the implementation and

specification of the operators, which can often be done using SQL. Our relational graph model is based on the W3C's Resource Description Framework (RDF) (Lassila and Swick 1998; Powers 2003).

For encoding relational schemas, XML schemas, and SQL views as graphs we use the following approach. Our meta-model for relational schemas is based on OIM (Bernstein and Bergstraesser 1999). For example, the model elements of a relational schema comprise tables, columns, and constraints; a table contains an ordered list of columns, each of which has a type; tables and columns carry names; the constraints are specialized into primary key, unique key, non-null, or referential constraints; a referential constraint refers to two columns, one of which is a foreign key and the other is a primary key; etc. Our graph representation of XML schemas builds on XML DOM (DOM 1998). The graph representation of SQL views that we deploy is comparable to a parse tree produced by an SQL processor (see Fig. 3.8 in Sect. 3.4). All clauses, statements, alias definitions, functional terms, etc. are represented as separate nodes. A view graph does not replicate the names of attributes and relations used in schemas, but refers directly to the respective nodes in the schema graphs.

3.2 Operators

Now we turn to the implementation of the operators of Sect. 2.3. The output of the primitive operators is defined uniquely in Sect. 2.3.1, except for the operator All, which is implemented differently for each meta-model. For example, for relational schemas the implementation of All is specified as follows:

$$\text{All}(m, s) := \text{SELECT } m.S \text{ FROM } m \text{ WHERE } m.P=\text{type AND } m.O \text{ IN} \\ \{\text{Table, Column, PrimaryKey, UniqueKey,} \\ \text{NonNull, ReferentialConstraint}\}$$

3.2.1 Extract and Delete

To describe our implementation of the Extract and Delete operators we focus on the relational schemas. Consider the schema m shown on the left of Fig. 3.1. The primary key constraints on PID and DID are depicted as horizontal bars underlining the respective attributes. The referential constraint is shown as a line connecting PRODUCTS.PID and O-DETAILS.PID. Assume that in the graph representation of m the three constraints are denoted by the nodes $c1$, $c2$, and $c3$, respectively. For brevity, we henceforth refer to the graph nodes representing the attributes of m simply by using their names.

Fig. 3.1 illustrates six examples of extraction and deletion. The output morphisms $m_{-m_1}, \dots, m_{-m_6}$ are omitted in the figure for compactness; they simply connect the respective elements of the input and output schema that

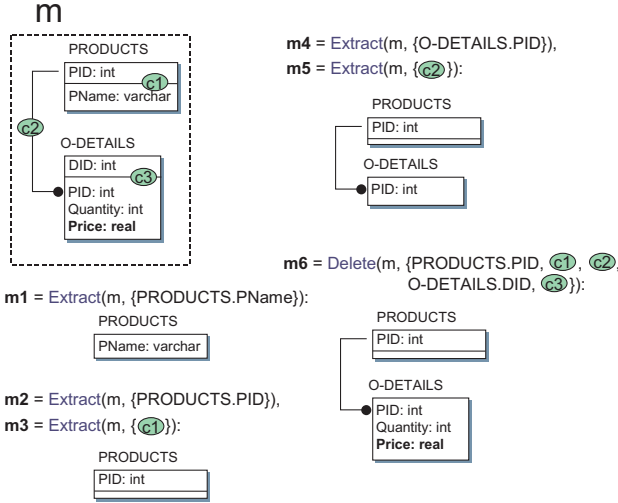


Fig. 3.1. Examples of extraction and deletion from a relational schema m

carry the same name. The first example demonstrates extraction of the attribute PName, which produces schema m_1 . Condition (ii) of Sect. 2.3.3 requires that m_1 be a well-formed relational schema, i.e., attribute PName belongs to a relation and has a type specification. Applied to relational schemas, condition (iii), which requires the output model to no more expressive as the input model, makes the extracted schema contain all constraints present in the original schema that affect the selected model elements. For example, extracting the attribute PRODUCTS.PID from m causes the primary key constraint c_1 to be extracted as well, yielding the schema m_2 . Dropping c_1 would violate (iii), since it would allow the attribute PID to contain duplicates and thus the original schema m could not represent all information of m_2 . Analogously, extracting O-DETAILS.PID from m (as schema m_4) needs to preserve the referential constraint c_2 , which in turn requires the presence of PRODUCTS.PID and its primary key constraint c_3 . Condition (iv) prevents any other attributes from appearing in m_4 .

In our prototype, the implementation of operator $\text{Extract}(m, s)$ for relational schemas is based on the conceptual algorithm of Sect. 2.3.3. Steps 1 (“closure”) and 6 (“cover”) are equality assignments. Step 3 of the algorithm is implemented as follows:

- If s' contains constraint x , add to s' all attributes that participate in the constraint definition.
- If s' contains attribute x , s' is extended to include
 - a. the enclosing relation of x ,
 - b. the type definition of x ,
 - c. the referential constraint or non-null constraint for x ,

- d. the primary key or unique key definition for x , but only when all attributes participating in the key definition are contained in x .

In Fig. 3.1, schemas m_3 and m_5 illustrate the extraction of nodes that denote constraints. To illustrate case (d), consider a relation $P(\text{Name}, \text{DOB}, \text{Addr})$ with a unique key constraint on $(\text{Name}, \text{DOB})$. According to the algorithm, $\text{Extract}(m, P.\text{Name})$ yields $P(\text{Name})$. The unique key constraint is not included since $P.\text{DOB}$ is not selected.

Notice that condition (iii) of Extract makes it impossible to delete a constraint on a relational attribute without deleting the attribute definition, or to delete the primary key attribute participating in a referential constraint without deleting its foreign key attribute. For example, consider schema m_6 in Fig. 3.1. Selecting PRODUCTS.PID and the constraints c_1 and c_2 is not sufficient for deleting this attribute: the attribute O-DETAILS.PID , which is a foreign key on PRODUCTS.PID , is not selected; therefore, dropping PRODUCTS.PID would extend the set of possible values that O-DETAILS.PID may take beyond those contained in PRODUCTS.PID and hence violate condition (iii). In Sections 3.2.3 and 3.2.4, we present more flexible operators ExtractMin , DeleteHard , and DeleteSoft , which allow such deletions by providing fewer consistency guarantees than Extract and Delete .

Extraction from XML schemas is implemented analogously to the above algorithm. Type references in XML schemas are treated similarly to the referential constraints in relational schemas. Currently, derived types are not supported.

3.2.2 Dependencies

As we observed above, the operators Extract and Delete disallow semantically questionable transformations on schemas, such as dropping arbitrary constraints, and are defined for schemas only. In general, deletion on models, which may or may not be schemas, needs to be done in a careful way to ensure that the consistency of the resulting model with respect to its meta-model is not violated. For example, consider the relation ORDERS shown at the bottom of Fig. 3.2. If we were to delete just the definition of the table ORDERS , we risk getting an inconsistent model, in which fields like OID do not belong to any table. Or, if we delete the field ORDERS.OID , we might get a malformed referential constraint for O-DETAILS.OID , whose target key definition is now missing. To deal with such consistency issues in a more general way, we exploit the concept of existential dependencies between model elements.

Figures 3.2 and 3.3 show examples of dependencies that hold between the elements of a relational schema, and between the elements of an XML schema. Each of the arcs specifies that the source element of the arc is existentially dependent on the target element. For example, in the relational schema of Fig. 3.2, the attribute “UnitPrice” cannot exist without its type

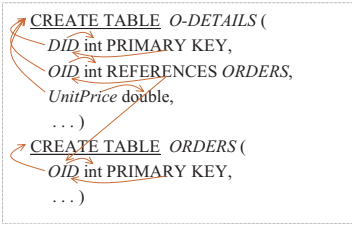


Fig. 3.2. Example of existential dependencies in a relational schema

definition (arc from “UnitPrice” to “double”). Similarly, the primary key constraint in table O-DETAILS is malformed if the constrained field “DID” is missing. The referential constraint between the fields O-DETAILS.OID and ORDERS.OID spans two tables, and requires both a foreign key and a primary key. Analogously, in the XML schema of Fig. 3.3, the definition of the element “shipTo” depends on the existence of the complex type “Address” as well as on the enclosing sequence element, etc.

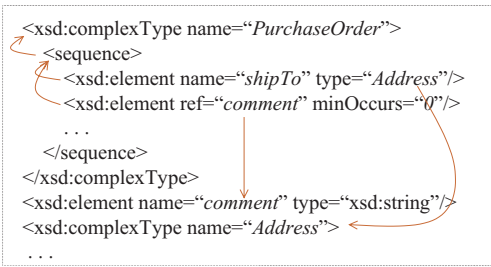


Fig. 3.3. Example of existential dependencies in an XML schema

As illustrated in Figures 3.2 and 3.3, dependencies are binary relations over the elements of a single model. Thus, we represent dependencies as intra-model morphisms, whose left elements are dependent on the right ones. To obtain the dependencies for a given model, we use the operator `Dependencies`, which invokes a non-generic implementation to compute the dependency morphism for the given model. For each supported model type, one such non-generic implementation is provided (one for relational schemas, another one for XML schemas, etc.). In our implementation, the operator `Dependencies` uses the arc types defined in the meta-model to determine what arcs are dependency arcs. For example, the arcs `column` and `SQLType` of Fig. 2.4 are marked as dependency arcs in our representation of the meta-model for relational schemas; the target of an arc of type `SQLType` depends on the source, and the source of arc of type `column` depends on its target.

3.2.3 ExtractMin

A general intuition behind extraction is that we want to obtain a minimal model that contains the nodes in the selector and all those nodes and edges

that are necessary to make the resulting subgraph a “complete”, well-formed model. Obviously, such model has to contain at least those nodes that are existentially required for the nodes in the selector. This minimalist subgraph can be obtained using the operator `ExtractMin` defined below, which uses an auxiliary derived operator `Reachable`.

```
operator ExtractMin(M, selector, dependencies)
  T = Subgraph(M, selector + Reachable(selector, dependencies));
  return Copy(T, All(T));

operator Reachable(selector, map)
  return Range(RestrictDomain(TransitiveClosure(map), selector));
```

The operator `ExtractMin` takes three parameters as input, a source model M , a selector that identifies the elements to be selected, and the dependency morphism for M . The operator returns the subgraph of M induced by the union of the nodes in the selector and all nodes that are required to satisfy the existential dependencies of the selected nodes. These required nodes are obtained using the operator `Reachable`.

To illustrate how `Reachable` works, imagine that it is called with parameters $\{a,d\}$ as selector and $\{(a,b),(b,c)\}$ as the dependency morphism of model M . We get: $\text{Reachable}(\{a,d\}, \{(a,b), (b,c)\}) = \text{Range}(\text{RestrictDomain}(\{(a,b), (b,c), (a,c)\}, \{a,d\})) = \text{Range}(\{(a,b), (a,c)\}) = \{b,c\}$. Thus, selecting $\{a,d\}$ from model M yields $\text{Subgraph}(M, \{a,d\} + \{b,c\}) = \text{Subgraph}(M, \{a,b,c,d\})$. The resulting subgraph contains by definition all edges between $\{a,b,c,d\}$ and their incident literals. Notice that the operator `Reachable` can be executed by the optimizer efficiently, without materializing the transitive closure. This observation is important, since the dependency closures of even moderately-sized models may contain hundreds of thousands of entries.

As another example, consider selecting a single node denoting the attribute “UnitPrice” from the model of the relational schema of Fig. 3.2 using `ExtractMin`. As shown in the figure, the type definition of “UnitPrice” and the relation “O-DETAILS” are required for the attribute definition, so that the operator `Extract` returns a subgraph of the model that represents the relational schema

```
CREATE TABLE O-DETAILS (UnitPrice double)
```

Similarly, if a single node denoting the primary key of table `ORDERS` is selected, we get

```
CREATE TABLE ORDERS (OID int PRIMARY KEY)
```

In this case, the node identifying the table `ORDERS` is pulled out due to the transitive dependency of the primary key on the table definition via the attribute definition.

3.2.4 DeleteHard and DeleteSoft

As noted in Sect. 2.3.3, extracting a selected portion of a model can be viewed as deletion of the unselected portion. To support a broader range of model management scenarios, we define additional two variants of deletion, `DeleteHard` and `DeleteSoft`. Both operators remove a portion of a model referenced by a selector. The intuition behind `DeleteHard` is that we want to obtain a maximal consistent submodel without the selected nodes. It is defined as follows.

```
operator DeleteHard(M, selector, dep)
  toDelete = selector + Reachable(selector, Invert(dep));
  toKeep = All(M) – toDelete;
  return ExtractMin(M, toKeep, dep);
```

Essentially, the operator `DeleteHard` takes $All(M)$ elements of M , subtracts from this set the elements to be deleted, and applies `ExtractMin` to extract the unselected portion of the model. To take the existential dependencies into account, `DeleteHard` extends the selector passed as input to include all elements of M that would become “dangling”, i.e., elements that are existentially dependent on the elements to be deleted. Such would-be dangling elements are obtained by passing the selector and the inverted dependency morphism to the operator `Reachable`. That is, the dependencies are traversed in the reverse direction.

Consider again the example in Fig. 3.2. Imagine that we `DeleteHard` the nodes representing the attribute `O-DETAILS.UnitPrice` and the table `ORDERS`. The set of elements `Reachable` from these selected elements over the inverted dependency morphism are the foreign key constraint on `O-DETAILS.UnitPrice` and all attributes of `ORDERS` (to see that, the arcs in the figure need to be traversed in the reverse direction). That is, the constraint and the table `ORDERS` with all its attributes will be removed, and we get the schema

```
CREATE TABLE O-DETAILS (DID int PRIMARY KEY, OID int)
```

In contrast to `DeleteHard`, the operator `DeleteSoft` removes each selected element only if it has no unselected dependent elements. That is, in the above example, the table `ORDERS` would not be deleted since it is referenced by the unselected foreign key on `O-DETAILS.OID`. The result of applying `DeleteSoft` for the same input parameters is shown below. Only `O-DETAILS.UnitPrice` has been removed.

```
CREATE TABLE O-DETAILS (
  DID int PRIMARY KEY,
  OID int REFERENCES ORDERS)
CREATE TABLE ORDERS (OID int PRIMARY KEY, ...)
```


The operator `DeleteSoft` is defined below. Instead of extending the selector to cover the would-be dangling elements, it is restricted to make sure that no unselected elements are removed. The selector that keeps the elements that cannot be deleted (`cannotBeDeleted`) is first obtained by collecting all elements which the unselected elements depend on. Now, the input selector is adjusted to eliminate all these undeletable elements. Finally, the operator `ExtractMin` is applied, just as in the operator `DeleteHard`.

```
operator DeleteSoft( $M$ , selector, dep)
  cannotBeDeleted = Reachable(All( $M$ ) – selector, dep);
  toDelete = selector – cannotBeDeleted;
  toKeep = All( $M$ ) – toDelete;
  return ExtractMin( $M$ , toKeep, dep);
```

Table 3.1 summarizes the differences between the operators discussed above and illustrates them using a single characteristic example for relational schemas. The “hard” version of deletion in schemas is similar to the cascading delete of existentially dependent data tuples, which is supported by many relational database systems.

Table 3.1. Comparison of variants of extraction and deletion

Operator	Example
Extract	Cannot extract a field without the constraints defined for the field.
ExtractMin	Can extract a field without the constraints defined for the field.
Delete	Cannot delete a constraint defined on a field without deleting the field.
DeleteSoft	Can delete a constraint defined on a field without deleting the field. Cannot delete fields referenced by unselected fields.
DeleteHard	Can delete fields even if they are referenced by unselected fields. In this case, dangling references would be deleted, too.

3.2.5 Diff

The `Diff` operator computes the difference between a model m and another model m' that is connected to m using a mapping m_m' . Intuitively, the difference between two models is a sub-model of m that does not participate in the mapping m_m' . In other words, to obtain the difference we eliminate from m all elements that do have matching counterparts in the other model. Thus, we define the operator `Diff` as shown below:

```
operator Diff( $m$ ,  $m\_m'$ )
  return Delete( $m$ , Domain( $m\_m'$ ));
```

Similarly to the operators `DeleteSoft` and `DeleteHard`, we provide additional two versions of the `Diff` operator: `DiffSoft` and `DiffHard`.

```
operator DiffSoft(m, m_m')
  return DeleteSoft(m, Domain(m_m'));
```

```
operator DiffHard(m, m_m')
  return DeleteHard(m, Domain(m_m'));
```

Notice that given the the differencing operators, we could define deletion as derived operations. For example, the operator `Delete` could be defined based on `Diff` as

```
operator Delete(m, s)
  return Diff(m, Id(s));
```

3.2.6 Match

In our prototype, the `Match` operator takes as input two models of the same kind, e.g., two relational schemas, and returns as output a morphism. We implemented `Match` using the Similarity Flooding (SF) algorithm, a graph-matching algorithm presented in Chap. 7. The SF algorithm exploits the structure of the graphs to be matched and performs especially well for detecting the differences between two versions of a schema, which is the case in our motivating scenario and many other typical metadata applications.

The SF algorithm takes as input two graphs m_1 and m_2 , and a set of initial similarity values between the nodes of the graphs, expressed as a weighted binary relation seed. Each pair $\langle l, r \rangle$ of seed carries a similarity value between zero and one. In a fixpoint computation, the algorithm iteratively propagates the initial similarity of nodes to the surrounding nodes, using the intuition that neighbors of similar nodes are similar. The output of the algorithm is another weighted binary relation.

In Sect. 2.2.2 we defined a morphism as a binary relation. To include weights in a morphism, we add to it a third attribute `Sim` that holds a similarity value for each pair of nodes. The primitive operators in Sect. 2.3.1 ignore this extra information. We implement the operator `Match` as

```
operator Match(m1, m2, seed)
  multimap = SFJoin(m1, m2, seed);
  multimap = Restrict(multimap, m1, m2);
  map = FilterBest(multimap);
  return  $\langle$ map, multimap $\rangle$ ;
```

The operator `SFJoin` encapsulates the SF algorithm. As explained in Chap. 7, the `multimap` returned by the algorithm may contain a large fraction of the cross product between nodes in m_1 and m_2 , and needs to be

filtered. The operator `FilterBest` implements the filter suggested in Chap. 8, which exploits the stable-marriage property. In addition to filtering, we restrict the result of the `SFJoin` operator to the nodes that represent the model elements of m_1 and m_2 using the operator `Restrict` (Sect. 2.3.2). The input morphism *seed* is typically obtained using another auxiliary operator `NGramMatch(m_1, m_2)`, which computes the similarities of literals in m_1 and m_2 based on the number of n -grams that they have in common. Alternatively, *seed* can be obtained by composition of morphisms. If *seed* is omitted, `NGramMatch` is invoked in `SFJoin` by default.

The above `Match` implementation returns both the filtered morphism *map*, and the unfiltered *multimap*. The morphism *map* can be adjusted by the engineer using a graphical tool by invoking the operator `EditMap` on the outputs of `Match`, e.g., as $map = \text{EditMap}(map, multimap)$. The graphical tool allows the engineer to inspect all candidate matches suggested in *multimap*.

The script used above for implementing the `Match` operator can be easily adapted to call other external schema matchers, which may deploy thesauri, analyze schema annotations, mine samples of instance data, reuse previous match results, etc., to reduce the manual post-processing effort.

3.2.7 Merge

We discuss our implementation of the `Merge` operator using the example in Fig. 3.4. On the top, two sample models m_1 and m_2 get merged into m (the output morphisms are omitted). The morphism *map* is depicted using directed arcs. The direction of each arc establishes a preference between two model elements; when collapsing the two elements, the target element is kept in the output m , whereas the source element is discarded. For example, the attribute `PO.OrderDate` is kept and `ORDER.ODate` is discarded, as illustrated in the figure. Such preferences are not part of the semantics of the `Merge` operator (Sect. 2.3.5), but are essential for practical deployment.

The input morphism *map* contains an extra attribute `Dir` to hold the direction of the arcs (\rightarrow or \leftarrow). Before `Merge` is executed, a human engineer has a chance to specify the arc direction in a graphical tool by invoking the operator `EditMap`. The output morphisms provide the engineer an auditable trail of how the elements of the input models have been transformed into the elements of the output model. For example, although `ORDER.ODate` is discarded in m , the morphism $m _ m_1$ would tell the engineer that `ORDER.ODate` from m_1 has become `ORDER.OrderDate` in m .

The bottom of Fig. 3.4 depicts m_1 and m_2 as graphs. For brevity, the arc labels, type edges, and literals are omitted (compare to Fig. 2.4). Node x corresponds to relation `ORDER`, $x1$ denotes `ORDER.ODate`, etc. The morphism *map* is $\langle x, y, \leftarrow \rangle, \langle x1, y2, \rightarrow \rangle, \langle x2, z1, \rightarrow \rangle$.

To implement the `Merge` operator, we developed an algorithm called `GraphMerge`, which we describe below. Similar to (Buneman et al. 1992; Pottin-

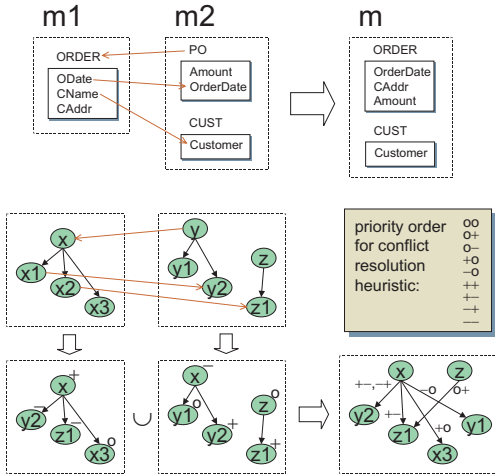


Fig. 3.4. Merging two sample schemas

ger and Bernstein 2003), the algorithm consists of three conceptual steps: node renaming, graph union, and conflict resolution.

1. In the first step, the graph nodes at the blunt ends of *map* are renamed to their targets at sharp ends, in both graphs m_1 and m_2 . The result of renaming is shown on the bottom left of Fig. 3.4. Nodes y , x_1 , and x_2 of both graphs have been renamed respectively to x , y_2 , and z_1 .
2. In the second step, we do a graph union, i.e., a set union of two sets of edges, and obtain the graph depicted on the bottom right of the figure. This graph is not a well-formed model, because the node z_1 , which used to represent the attribute $CUST.Customer$ in m_2 , has now become an attribute of two different relations, x ($ORDER$) and z ($CUST$).
3. Such conflicts are resolved in the third and final step of the GraphMerge algorithm. The above conflict is eliminated by deleting either the edge between x and z_1 , or the edge between z and z_1 , effectively making $Customer$ either an attribute of relation $CUST$ or an attribute of relation $ORDER$ in the merged schema. The choice between the two options is made by a human engineer.

Step 3 is the costliest step of the algorithm, since it requires human feedback. To partially automate conflict resolution, we developed the following heuristic. Observe that in Fig. 3.4 it seems more “natural” to keep the attribute $Customer$ in relation $CUST$ than to move it to $ORDER$. To generalize this observation, we track the origin of each edge in the merged graph, and assign to each edge a tag, such as $+-$ or $o+$, which indicates whether each of the nodes incident at the edge was a source node of *map* ($-$), a target node ($+$) of *map*, or none of the two (o) (these are the only three possible cases assuming that source and target nodes of *map* are disjoint). For example, the edge $\langle x, z_1 \rangle$ obtained by renaming from $\langle x, x_2 \rangle$ is tagged with $+-$, since x is

a target node and x_2 is a source node of map . Analogously, the edge $\langle z, z_1 \rangle$ is tagged with $o+$, since z does not appear in map at all.

If we knew that $o+$ edges are always preferred over $+ -$ edges, then, in a conflict $\langle x, z_1 \rangle$ could be eliminated without asking the engineer. We examined a variety of merge problems in the context of relational schemas, XML schemas, and SQL views, and established empirically a total order among all tag variations, which helps resolve many conflicts automatically in a way that matches human intuition. This order is shown in the middle right of Fig. 3.4. Intuitively, edges between unchanged nodes (oo) are least likely to be rejected in a conflict, and thus have the highest priority. Similarly, edges incident at $+$ seem more likely to be preferred than those incident at $-$. Thus, Steps 2 and 3 are realized as follows. First, all edges in the merged graph are sorted by decreasing priority. Then, iteratively, each edge is taken off the top of the sorted list and is appended to an (initially empty) graph G . If appending the edge violates model consistency, it is rejected. Once all edges have been appended, the engineer examines the result and the choices made heuristically, and makes any necessary adjustments. The execution trace of the algorithm is stored in a log file, which lists the rejected alternatives.

In the above description of the algorithm, we factored out an important aspect, the ordering of nodes within parent. To illustrate how we reestablish a correct order in the merged schema, consider Fig. 3.4. Node y denoting the relation PO is renamed to x . Thus, when merging this node with the original x in m_1 , we move attributes y_1 (Amount) and y_2 (OrderDate) to the last position in the merged schema m . However, OrderDate “overrides” ODate, the first attribute in relation ORDER, and should remain at the first position. Hence, in schema m , the resulting order of attributes is OrderDate, CAddr, Amount.

The GraphMerge algorithm is summarized below:

Algorithm GraphMerge(m_1, m_2, map)

$M := m_1 \cup m_2$; $L :=$ empty list; $G :=$ empty graph

for each edge e in M **do**

 rename nodes of e using map ; assign tag to e ; append e to L ;

end for

sort edges in L by decreasing tag priority;

$maxN :=$ SELECT max($M.N$) FROM M ;

while L **not** empty **do**

 take edge $e = \langle s, p, o, n \rangle$ off top of L ;

if tag(e) one of {“ $-o$ ”, “ $-+$ ”, “ $--$ ”} **then**

$n := n + maxN$;

if o is literal **then continue loop end if**

end if

if exists $e' = \langle s, p, o, n' \rangle$ in G **then**

 replace e' in G by $\langle s, p, o, \min\{n, n'\} \rangle$;

else if not conflictsWith($\langle s, p, o, n \rangle, G$) **then**

```

    append  $\langle s, p, o, n \rangle$  to  $G$ ; end if
  end if
end while
return  $G$ 

```

The number $maxN$ is obtained as the highest existing value of the ordinal property N in m_1 and m_2 (compare Sect. 2.2.1). It is used to move the nodes hanging off renamed nodes to the last positions. To test for renamed nodes, we check whether the corresponding edge tag starts with $-$, i.e., is one of $-o$, $-+$, or $--$. The literals belonging to such renamed nodes are removed, to ensure that, e.g., the relation corresponding to node x in the merged graph of Fig. 3.4 will be named “ORDER” and not “PO”. The function `conflictsWith()` checks whether appending a new edge to G causes a conflict.

The GraphMerge algorithm can be used for various kinds of models by implementing the function `conflictsWith()` appropriately. In our prototype, we deploy the algorithm for merging relational schemas, XML schemas, and SQL views. For example, conflict detection for relational schemas checks that relations cannot contain relations instead of attributes, or that attributes cannot be shared among relations, etc.

The Merge operator is implemented as follows:

```

operator Merge( $m_1, m_2, map$ )
   $G = \text{GraphMerge}(m_1, m_2, map)$ ;
   $s = \text{SELECT L FROM } map \text{ WHERE Dir} = \text{“}\rightarrow\text{” UNION}$ 
     $\text{SELECT R FROM } map \text{ WHERE Dir} = \text{“}\leftarrow\text{”};$ 
   $m_1\_G = \text{RestrictDomain}(map, \text{All}(m_1) \cap s) + \text{Id}(\text{All}(m_1) - s)$ ;
   $m_2\_G = \text{RestrictDomain}(map, \text{All}(m_2) \cap s) + \text{Id}(\text{All}(m_2) - s)$ ;
   $\langle m, G\_m \rangle = \text{Copy}(G, \text{All}(G))$ ;
  return  $\langle m, \text{Invert}(m_1\_G * G\_m), \text{Invert}(m_2\_G * G\_m) \rangle$ ;

```

Recall that Merge must also return morphisms from each of its input models to its output model. Thus, after applying `GraphMerge` to obtain the merged model G , we compute the morphisms m_1_G and m_2_G . The selector s contains all source nodes of map . For the example of Fig. 3.4, we obtain m_1_G as union of domain-restricted map, $\{\langle x1, y2 \rangle, \langle x2, z1 \rangle\}$, which maps each renamed m_1 node to its new name, and the identity morphism on not renamed nodes, $\{\langle x, x \rangle, \langle x3, x3 \rangle\}$. Finally, G is copied to make the node IDs of the output model m unique, and the morphisms m_1_G and m_2_G are composed with G_m , so they range over m instead of G .

The GraphMerge algorithm does not “invent” new model elements or establish new relationships between the existing elements. Therefore, the operator Merge as implemented above cannot reorganize schemas to resolve structural conflicts. For example, consider two XML schemas, S_1 with element `FullName` and S_2 with elements `FirstName` and `LastName`. Merging S_1 and S_2 should ideally create a new complex type `Name` with subordinate elements `FirstName` and `LastName`. Such structural conflicts can be addressed by using

n -way merges, in which intermediate schemas S_j are used for describing the desired structural transformations.

In Sect. 2.3.5 we postulated two “semantic” conditions that Merge should satisfy. Our implementation does not automatically ensure that condition (i) holds. For example, the engineer might decide to “override” a non-null constraint on an attribute in one schema S_1 by a primary key constraint of the other schema S_2 , in which case the output model would be less expressive (i.e., more constrained) than S_1 . This flexibility is sometimes desirable in practice.

3.3 Prototype “Rondo”

In this section, we describe the architecture and main features of the prototype in more detail¹. Its architecture is shown in Fig. 3.5. A central component of the architecture is an interpreter that executes scripts. Its main task is to orchestrate the data flow between the operators. The interpreter can be run either from the command line, or invoked programmatically by external applications and tools. The operators can be defined either by providing a native implementation, or by means of scripts. For example, a native operator like ReadSQLDDL reads a text document containing the definition of a relational database and creates its graph representation, whereas WriteSQLDDL exports the graph back as text.

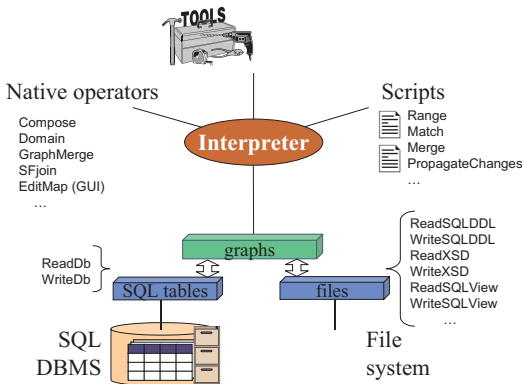


Fig. 3.5. Architecture of the prototype

Schemas and instances in the native format such as XML or SQL DDL files are stored in the local file system. The models can additionally be loaded and stored in a (remote) metadata repository. The repository is an SQL-compatible database. The interpreter communicates with the repository via

¹ A demo of the prototype is available for download at <http://www-db.stanford.edu/~melnik/mm/rondo/>

JDBC. Two native operators, `ReadDb` and `WriteDb`, load and store arbitrary graphs in the database.

Native operators are defined in scripts using statements like

```
alias ReadSQLDDL [Java class name];
```

Every native operator corresponds to a Java class compliant to a certain interface. Other operators that have been implemented natively include

- all primitive operators of Sect. 3.2,
- operators that launch GUIs for editing morphisms and selectors, such as `EditMap` or `EditSelector`,
- schema translation and conversion operators such as `SQL2XSD`, and
- the operators that implement the individual algorithms such as `Similarity Flooding (SFJoin)`, `GraphMerge`, or the string matching operators described below.

All other operators, such as `Range`, `Match`, or `Merge`, are implemented by scripts presented in the previous sections. The specification of the commonly used native or derived operators can be grouped in a single script and utilized in other scripts using `include` statements.

`StringMatch` provides a hint how literal nodes in one graph match those in another. This string matcher splits a text string into a set of words and compares the word in two sets pairwise. In word comparison, we examine only common prefix and suffix. Optionally, term frequencies are used to reduce the impact of common terms in large schemas. In Chap. 7, we give an example of string matching in Table 7.1. `NGramMatch` is another, more efficient, string matching operator. It builds an in-memory inverted n -gram index over the labels used in the models. Then, both indexes are merged producing a list of pairs of labels. The complexity of `NGramMatch` is $O(n \log n)$ instead of $O(n^2)$ of `StringMatch`; it is determined by the sorting phase of the index construction.

The scripting language that we use is quite simple. Every operator takes a list of models as input and produces a list of models as output. `Load/store` and `import/export` operators are an exception, since they accept additional parameters that are not models. Recall that mappings are models and therefore can be used whenever model is expected as a parameter. For compactness of scripts, operators can be nested.

The interpreter provides a debugging facility that allows examining the execution traces of complex scripts, and supports flexible handling of the input and output parameters of operators. For example, if an operator returns more than one argument (as does our implementation of the operator `Match`), some of which are not used subsequently (as in script `PropagateChanges` in Sect. 2.1), they can be tacitly ignored.

For minimizing the amount of GUI programming needed for visualizing various kinds of models, we used the following technique. We require an operator like `WriteSQLDDL` to output not only the textual representation of the

model, but also a data structure that describes how the terms in the text relate to the model elements, or graph nodes. In this way the schema elements shown in Fig. 3.7 enclosed in boxes are associated with the graph nodes representing those elements, and the GUI operators `EditMap` and `EditSelector` can be used in exactly the same way for relational schemas (Fig. 3.7) or SQL views (Fig. 3.8).

At the current stage, our prototype supports the basic features of SQL DDL, XML Schema, RDF Schema, and SQL views, and, in preliminary form, UML. To introduce a new modeling language in the prototype, two steps are required. First, the import/export operators need to be provided, which ensure lossless round-tripping from the native format to graphs and back. Second, several callbacks need to be implemented for supporting the operators `All`, `Extract`, and `GraphMerge`.

The code breakdown of the prototype is shown in Fig. 3.6. A large share of the implementation effort was due to the graph APIs responsible for in-memory representation and manipulation of graphs and morphisms, and the database support. The key generic model-management functionality comprises less than 7K lines of code. It includes the interpreter (2050), primitive operators (660), `SFJoin` (1760) and `GraphMerge` (700) implementations, as well as the generic GUI operators (1400). The non-generic part is essentially divided among the code needed to support SQL DDL, XML schemas, and SQL views. The smallest portion of code is due to converters: `XSD2SQL` (260), `SQL2XSD` (250), `View2Morphism` (90), and `Morphism2View` (200). The compactness of the converters is mostly due to the fact that they operate on the internal graph representation using expressive queries. The total amount of code in the prototype is below 24K lines. The total scripting code developed so far is measured in hundreds of lines.

The implemented scenarios run in a few seconds on a 600 MHz laptop with 256 MB of memory for moderately-sized schemas, which contain up to a few hundred model elements². However, we found that our graphical user interface is inadequate for visualizing medium-size and large schemas. For example, schemas that contain around 40 table definitions stretch over dozens of computer screens, are hard to navigate, and the mappings between them clutter the screen. For schemas containing over a thousand table definitions, the running time performance of our GUI and the matching algorithm is inadequate. Intelligent GUI design and efficiency require future work.

Further scenarios that we implemented include a reintegration scenario from the context of version management, iterative merge, a warehousing scenario, in which we extract a subset of the schema that is sufficient to answer a given set queries, and a view reuse scenario. The view reuse scenario is in Sect. 3.4. Among other aspects, it illustrates how views can be merged, presents the GUIs used in our prototype, and demonstrates the use of the

² The test schemas that we used are available at ifr.sap.com, www.xcbl.org, and www.microsoft.com/biztalk.

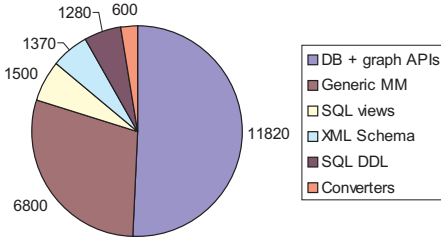


Fig. 3.6. Code size breakdown in prototype (in lines of code)

operators `Morphism2View` and `View2Morphism`. The reintegration scenario is covered in Sect. 3.5.

3.4 View-Reuse Scenario

In this section, we examine another scenario, which illustrates the use of the operators presented in this chapter for addressing a typical data warehousing task. Consider adding a new source S_2 to a data warehouse D . Assume that S_2 is similar to an existing source S_1 . The morphism $S_1 \rightarrow S_2$ between the two source schemas is shown in Fig. 3.7. Let an existing SQL view $v_{S_1 \rightarrow D}$ describe how the instances of S_1 populate D . The view $v_{S_1 \rightarrow D}$ is depicted in the middle of Fig. 3.8 (the relevant portion of the warehouse schema can be seen in the `CREATE VIEW` clause). Our goal is to reuse the view $v_{S_1 \rightarrow D}$ for importing S_2 data into D , i.e., creating the view $v_{S_2 \rightarrow D}$. Conventionally, this problem is solved manually involving a tiresome and error-prone renaming of the attribute and relation names used in $v_{S_1 \rightarrow D}$ based on the similarities between S_1 and S_2 . In our prototype, we obtain $v_{S_2 \rightarrow D}$ using the following script:

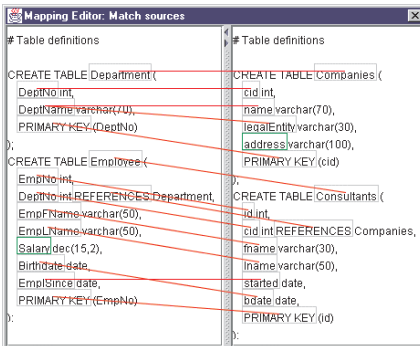


Fig. 3.7. Morphism between sources S_1 and S_2

1. $S_1_S_2 = \text{Match}(S_1, S_2)$;
2. $S_1_D = \text{View2Morphism}(vS_1_D)$;
3. $S_2_D = \text{Invert}(S_1_S_2) * S_1_D$;
4. $vS_2_D' = \text{Morphism2View}(S_2_D)$;
5. $map = \text{Match}(vS_2_D', vS_1_D, \text{Invert}(S_1_S_2))$;
6. $vS_2_D = \text{Merge}(vS_2_D', vS_1_D, map + S_1_S_2)$;

First, we match S_1 and S_2 to determine the correspondences between the schemas. As can be seen in Fig. 3.7, some of the elements of S_1 and S_2 remain unmatched, whereas others, such as `Department.DeptName` are matched to two elements, `Companies.name` and `Companies.legalEntity`. In Step 2, we extract the morphism S_1_D from the view definition vS_1_D using a non-generic operator `View2Morphism`. For example, the morphism S_1_D , which is omitted in the figures for brevity, associates the attribute `Personnel.Pname` with two attributes, `Employee.EmpFName` and `Employee.EmpLName`, etc. Next, we compute the morphism S_2_D by composition. In Step 4, a ‘template’ view definition vS_2_D' is generated from S_2_D using another non-generic operator `Morphism2View`. It is shown on the left of Fig. 3.8. Morphism S_2_D contains no information as to how the values of the attribute `Personnel.Affiliation` are obtained from `Companies.name` and `Companies.legalEntity`. Therefore, a functional term `fct1` is generated in vS_2_D' as a placeholder.

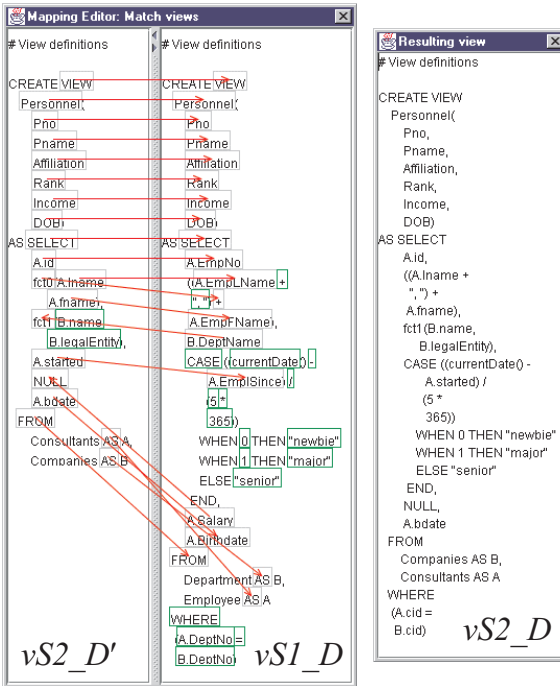


Fig. 3.8. Merging two SQL views

In Step 5, the template vS_2_D' and the existing view vS_1_D are matched, using as a seed the morphism between S_1 and S_2 . The resulting morphism, after some minor manual corrections, is depicted in Fig. 3.8. Finally, in Step 6 both view definitions are merged to obtain vS_2_D , shown on the right. Notice that the function symbol `fct0` has been correctly replaced by the nested concatenation, whereas `fct1` was left as is. The unmatched WHERE clause was borrowed from vS_1_D ; the attribute references have however been correctly replaced by `Companies.cid` and `Consultants.cid`. To achieve that, the morphism map passed to Merge is extended to include $S_1_S_2$. The heuristic deployed in the GraphMerge algorithm produces vS_2_D fully automatically, due to relative simplicity of the input views.

3.5 Reintegration Scenario

In this section, we illustrate another scenario called reintegration, or 3-way merge. The reintegration problem arises when a model is modified independently by several engineers or tools. We focus on the case when there are two such independent modifications. Assume that model m was changed independently into m_1 by Ann and into m_2 by Bob. Our goal is to obtain the reconciled model m_3 that incorporates the changes done by Ann and Bob, and the mappings $m_3_m_1$, $m_3_m_2$ and m_3_m that describe how the models m_1 , m_2 , and m relate to the reconciled version m_3 .

Consider the example in Fig. 3.9. The original (relational) schema m is depicted on the top of the figure. In table ORDERS in schema m , employees are represented by an opaque identifier. To store employees' names, Ann creates the table EMPLOYEES and makes ORDERS.EID a foreign key into the new table. Also, she deletes ORDERS.PONum and ORDERS.UnitPrice and adds PRODUCTS.PDesc. Meanwhile, Bob creates the table BRANDS and replaces the attribute PRODUCTS.Brand by a foreign key pointing to the new table. In addition, he adds a new attribute PRODUCTS.ISIC that holds the classification description of products. He deletes DETAILS.UnitPrice, just as Ann, and in addition he also deletes DETAILS.Discount.

One way of obtaining m_3 is to simply merge m_1 and m_2 . That is, in the script shown below, we first match m_1 and m_2 (line 1) and apply the Merge operator (line 2). To compute the mapping m_3_m , we need to know how m corresponds to each of m_1 and m_2 . So, we match them in lines 3-4. Now, each of the compositions $m_3_m_1 * \text{Invert}(m_m_1)$ and $m_3_m_2 * \text{Invert}(m_m_2)$ describes a part of the mapping from m_3 to m . To obtain m_3_m , we combine both compositions in line 5.

operator `ReintegrateFirstCut(m, m_1, m_2)`

1. $m_1_m_2 = \text{Match}(m_1, m_2)$;
2. $\langle m_3, m_3_m_1, m_3_m_2 \rangle = \text{Merge}(m_1, m_2, m_1_m_2)$;

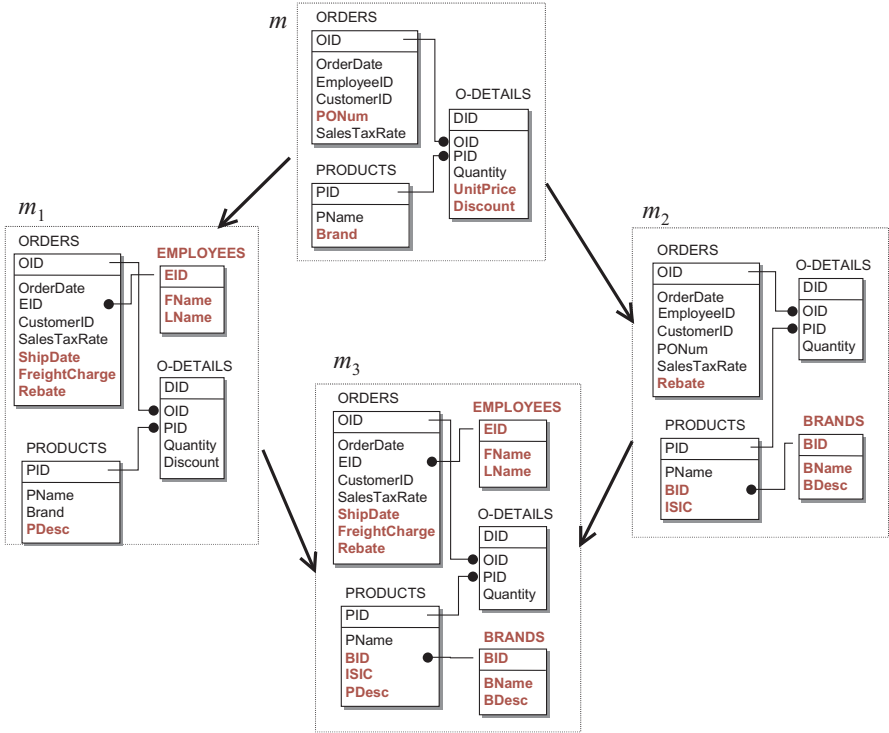


Fig. 3.9. Reintegration scenario (3-way merge)

3. $m_m_1 = \text{Match}(m, m_1)$; // or given
4. $m_m_2 = \text{Match}(m, m_2)$; // or given
5. $m_3_m = m_3_m_1 * \text{Invert}(m_m_1) + m_3_m_2 * \text{Invert}(m_m_2)$;
6. return $\langle m_3, m_3_m, m_3_m_1, m_3_m_2 \rangle$;

The above approach has two major weaknesses. First, we have to apply the Match operator three times, each potentially requiring expensive human intervention. In practice, m_m_1 and m_m_2 could be tracked automatically by the schema editing tool used by Ann and Bob. Still, matching m_1 and m_2 from scratch can be costly. Second, the above script discards all deletions done exclusively by either Ann or Bob. That is, ORDERS.PONum and O-DETAILS.Discount would appear in m_3 albeit both have been deleted. O-DETAILS.UnitPrice would, however, be correctly removed.

To address the first problem, we could modify the above script by moving lines 3-4 to the top and obtaining $m_1_m_2$ as the composition $m_1_m_2 = \text{Invert}(m_m_1) * m_m_2$. By doing so, however, we duplicate the equivalent additions done by both Ann and Bob, since the added equivalent elements have no counterparts in m and hence their correspondences get lost upon composition. That is, after executing such modified script,

ORDERS.Rebate would appear in m_3 twice. And yet, we could use $m_{1_}m_2$ computed by composition to drive the match between m_1 and m_2 , as in $m_{1_}m_2 = \text{Match}(m_1, m_2, \text{Invert}(m_{_}m_1) * m_{_}m_2 + \text{NGramMatch}(m_1, m_2))$. Moreover, when m_1 and m_2 are large, it may be more effective to extract only the new portions of m_1 and m_2 and match those.

To address the second problem, which is due to losing deletions done exclusively by Ann or Bob, we could apply to m_1 all deletions done in m_2 , and likewise apply to m_2 all deletions of m_1 . We incorporate both ideas in the script below:

operator Reintegrate(m, m_1, m_2)

1. $m_{_}m_1 = \text{Match}(m, m_1);$ // or given
2. $m_{_}m_2 = \text{Match}(m, m_2);$ // or given
3. $\langle m'_1, m_{1_}m'_1 \rangle = \text{Delete}(m_1, \text{Traverse}(\text{All}(m) - \text{Domain}(m_{_}m_2), m_{_}m_1));$
4. $\langle m'_2, m_{2_}m'_2 \rangle = \text{Delete}(m_2, \text{Traverse}(\text{All}(m) - \text{Domain}(m_{_}m_1), m_{_}m_2));$
5. $\langle m_{1x}, m'_{1_}m_{1x} \rangle = \text{Extract}(m'_1, \text{Traverse}(\text{All}(m_1) - \text{Range}(m_{_}m_1),$
 $m_{1_}m'_1));$
6. $\langle m_{2x}, m'_{2_}m_{2x} \rangle = \text{Extract}(m'_2, \text{Traverse}(\text{All}(m_2) - \text{Range}(m_{_}m_2),$
 $m_{2_}m'_2));$
7. $m_{1x_}m_{2x_}core = \text{Invert}(m'_{1_}m_{1x}) * \text{Invert}(m_{1_}m'_1) * \text{Invert}(m_{_}m_1) *$
 $m_{_}m_2 * m_{2_}m'_2 * m'_{2_}m_{2x};$
8. $m_{1x_}m_{2x} = \text{Match}(m_{1x}, m_{2x}, m_{1x_}m_{2x_}core + \text{NGramMatch}(m_{1x}, m_{2x}));$
9. $m'_{1_}m'_2 = m'_{1_}m_{1x} * m_{1x_}m_{2x} * \text{Invert}(m'_{2_}m_{2x}) +$
 $\text{Invert}(m_{1_}m'_1) * \text{Invert}(m_{_}m_1) * m_{_}m_2 * m_{2_}m'_2;$
10. $\langle m_3, m_{3_}m'_1, m_{3_}m'_2 \rangle = \text{Merge}(m'_1, m'_2, m'_{1_}m'_2);$
11. $m_{3_}m_1 = m_{3_}m'_1 * \text{Invert}(m_{1_}m'_1);$
12. $m_{3_}m_2 = m_{3_}m'_2 * \text{Invert}(m_{2_}m'_2);$
13. $m_{3_}m = m_{3_}m_1 * \text{Invert}(m_{_}m_1) + m_{3_}m_2 * \text{Invert}(m_{_}m_2);$
14. return $\langle m_3, m_{3_}m, m_{3_}m_1, m_{3_}m_2 \rangle;$

To illustrate the script, consider the schematic representation in Fig. 3.10. In line 3, we obtain the model m'_1 that contains all of m_1 , i.e., the model produced by Ann, without the elements deleted by Bob by way of m_2 (DETAILS.Discount). The expression $\text{All}(m) - \text{Domain}(m_{_}m_2)$ produces a selector that holds the elements of m that do not appear in m_2 . The images of these elements obtained by traversing $m_{_}m_1$ into m_1 are then deleted. Analogously, m'_2 contains all of m_2 without the elements deleted by way of m_1 , such as ORDERS.PONum.

In line 5, we extract a portion m_{1x} of m'_1 that comprises only the elements added by Ann (e.g., PRODUCTS.PDesc) and their support elements (e.g., PRODUCTS). We achieve this by traversing the added elements $\text{All}(m_1) - \text{Range}(m_{_}m_1)$ from m_1 to m'_1 . Line 6 does a similar job for m_{2x} . Notice that line 5 could be realized alternatively as $\langle m_{1x}, m'_{1_}m_{1x} \rangle = \text{Diff}(m'_1, \text{Invert}(m_{1_}m'_1) * \text{Invert}(m_{_}m_1));$

In line 7, we compute the mapping $m_{1x_}m_{2x_}core$ between m_{1x} and m_{2x} to establish the correspondences between the support elements of m_{1x} and

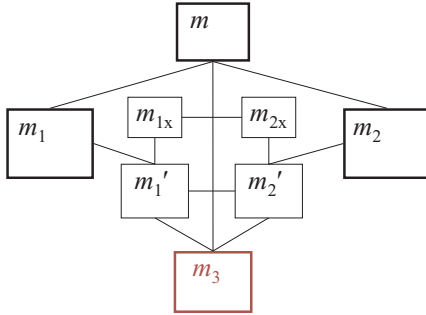


Fig. 3.10. Schematic representation of the reintegration scenario

m_{2x} . This mapping is then used to drive the Match between the added portions in line 8. Here, the engineer executing the script has a chance to decide whether or not `ORDERS.Rebate` added by Ann is equivalent to `ORDERS.Rebate` added by Bob. Notice that this Match is relatively inexpensive to perform, since we only have to reconcile the additions introduced by Ann and Bob.

In line 9, we compute the mapping between m'_1 and m'_2 to drive the Merge in line 10. To compose m'_1 - m'_2 , we need to consider both “paths” between the two models. One of them includes the matches between the added elements, m_{1x} - m_{2x} , and the other goes over the original model m . Similarly, the mapping m_3 - m is obtained in line 13 by joining two paths, one going through m_1 and the other through m_2 , portions of which are computed in lines 11-12. In line 14, the results of the script execution are returned.

3.6 Conclusions

In Chapters 2 and 3 we presented a programming platform for model management that implements all generic operators suggested so far in the literature. We explored the use of morphisms and selectors and introduced several novel generic operators. We discussed the structural operator semantics and the algorithms that we developed for implementing them. We showed that introducing a new model type like SQL DDL schemas in our prototype requires a moderate programming effort, but brings a large new class of model-management tasks within reach.

The main conclusions that we draw at this point are the following:

1. One can solve practical problems using the model management operators.
2. The solutions require a relatively small amount of code.
3. One can get quite far using a relatively weak representation for models and mappings.

The operator definitions presented in Chap. 2 above are mostly syntactic, just like the conceptual structures, and are expressed as graph transformati-

ons. Focusing on syntax allows the operators like Match or Merge to be implemented in a generic fashion for different kinds of models, as we demonstrated in Chap. 3. However, a deeper understanding of the semantics of these operators is crucial for assessing the correctness of model-management scripts. That is, the effect of applying “syntactic” operators to schemas ultimately needs to be expressed in terms of what these operators do to the instances of these schemas. Conditions (i)-(iv) for the Extract operator (Sect. 2.3.3), or (i)-(ii) for Merge (Sect. 2.3.5) reflect the semantics of these operators to a limited degree.

In Part II, we discuss the instance-level semantics of the model-management operators. It allows us to characterize the properties of the operators without assuming a particular meta-meta model representation of models and mappings.

4. State-Based Semantics

“He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.”

– Leonardo da Vinci (1452-1519)

In Part I, we described the first prototype for model management, called Rondo, which offers a set of high-level operators for solving metadata-related problems. Using Rondo, we developed scripts for several practically relevant scenarios, such as change propagation, view reuse, and reintegration. We found that the scripts produce intuitively correct results and that the structural operator definitions that we give are useful for solving practical problems.

In Chap. 2, we defined the semantics of the model-management operators for morphisms, a very simple mapping language. A morphism is represented as a set of arcs connecting the elements of two schemas. Although the desired result of the operators seems intuitively clear when morphisms are utilized, the treatment in Chap. 2 provides little guidance with respect to what results the operators should return if SQL views, XQuery, Datalog, or other more expressive languages are deployed in scripts instead of morphisms.

In this chapter, we present a way of defining the semantics of the operators in a truly generic fashion, without assuming any specific model and mapping languages. The main idea of our approach is to express the effect of applying the operators to models in terms of what the operators do to the instances of these models. For example, the effect of applying the operators to a database schema is expressed in terms of the valid database states described by the schema. In this way, we can characterize the semantics of operators without relying on any particular meta-model or meta-meta model. We call this kind of semantics *state-based*, or *instance-based*, semantics. In contrast, the semantics defined in Chap. 2 is driven by the structural properties of models, i.e., by the relationships between the individual models elements. To distinguish the state-based definitions from the structural definitions, in this chapter we use a distinct font face for the operators. Thus, we write *Extract* instead of `Extract`, and denote the composition using \circ instead of `*`.

This chapter is structured as follows:

- In Sect. 4.1, we introduce the state-based approach and formal notation used in this chapter. We define the state-based semantics of model-management scripts and explain what it means to execute a script.
- In Sect. 4.2, we specify the state-based semantics of the operators. We derive alternative formulations of operator definitions that are substantially easier to verify for concrete schema and mapping languages. We present detailed examples of how the operators can be applied to relational schemas and SQL views.
- In Sect. 4.3, we consider in more detail the problem of computing the results of a script, which we refer to as *materialization*.

In Chap. 5, we revisit the change propagation scenario from the state-based perspective, and address the relationship between the structural and state-based operator definitions in Chap. 6.

Specifying the state-based semantics of the operators allows us to lay out a clear extensibility path for supporting more complex mapping languages in our prototype. Furthermore, it helps us study formally the properties of the operators and the behavior of model-management systems: the existing ones, such as Rondo, as well as systems that will be built in the future.

4.1 Basic Concepts

In this section we present the concepts of a model and a mapping and explain the notation used in the rest of the chapter. For clarity, in the examples that we give we put schema and mapping definitions in French quotation marks $\langle \dots \rangle$. For example, $\langle R(A,B), S(C) \rangle$ denotes a relational schema with two tables, R and S. Furthermore, we distinguish between the set semantics for relational tables, when the table is not allowed to contain duplicate tuples, and the multiset semantics used in SQL. For the former we write $\langle R(A,B) \rangle$, for the latter we use square brackets: $\langle R[A,B] \rangle$. We abbreviate SELECT DISTINCT clauses in view definitions as SELECTD.

4.1.1 Models

A model is a formal description of an application artifact, such as a relational schema, a workflow definition, or an interface specification. Typically, a model serves as a template for instances. For example, an instance of a relational schema is a valid database state; an instance of a workflow is a valid transition graph; an instance of a programming interface is an implementation that conforms to the interface. Let $\text{Inst}(m)$ denote the set of all possible instances of m . If m is a database schema, every instance $db \in \text{Inst}(m)$ must satisfy the constraints present in m .

Definition 4.1.1 (State-based semantics of models). *The state-based semantics of model m is defined as the set of all possible instances of m , denoted as $\text{Inst}(m)$.* ■

We do not specify the nature of instances of models any further.

Example 4.1.1. Let m be the relational schema $\langle R(\text{Name: char}(3), \text{Sex: bool}) \rangle$. Several instances of this schema are shown in Fig. 4.1. Attributes Name and Sex can take $|\text{char}(3)|$ and $|\text{bool}|$ different values, respectively. Using these we can construct $|\text{char}(3)| \cdot |\text{bool}|$ different tuples to populate the table R. Any subset of these tuples describes a valid database state of m . Thus, schema m has $2^{|\text{char}(3)| \cdot |\text{bool}|}$ valid instances. Notice that an instance of m is not an individual string or Boolean value, but an entire populated database. ■

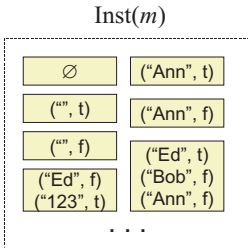


Fig. 4.1. Some instances of relational schema $R(\text{Name: char}(3), \text{Sex: bool})$

Example 4.1.2. Consider the schema $\langle R[A: \text{bool}] \rangle$ (multiset semantics). Each ordered list of Boolean values is a valid instance of the schema. Thus, the schema has infinitely many instances. ■

Definition 4.1.1 intensionally leaves the concept of model unspecified. Other semantics, e.g., the structural semantics defined in Chap. 2, which establishes the connection to the representation of m in a concrete meta-meta model, can be introduced using different function symbols, such as $\text{Struct}(m)$. Model m itself is not identical to $\text{Inst}(m)$. Instead, $\text{Inst}(m)$ provides a mechanism for describing a part of the semantics of m , its state-based semantics.

A model m can itself be an instance of another model. The latter is called the meta-model of m . For example, a relational schema can be viewed as an instance of a meta-model that describes the concepts Table, Column, Datatype, ReferentialConstraint, etc. Such a meta-model for relational schemas can be defined, e.g., using a UML diagram (Bernstein et al. 1999).

4.1.2 Mappings

A mapping establishes a semantic correspondence between models. A mapping between m_1 and m_2 identifies all mutually consistent states of m_1 and m_2 , i.e., those states that can exist at the same time in an application that deploys the mapping. For example, consider an application that uses a program $m_1 _ m_2$ to generate XML reports complying with DTD m_2 from a database with schema m_1 . The state-based semantics of $m_1 _ m_2$ tells us whether a given database state $x \in m_1$ and a given document $y \in m_2$ are mutually consistent, i.e., whether y could possibly have been generated from x . Thus, the program defines a binary relation on the instances of m_1 and m_2 .

Definition 4.1.2 (State-based semantics of mappings). *The state-based semantics of a mapping $m_1 _ m_2$ between models m_1 and m_2 is defined as the binary relation $\text{Inst}(m_1 _ m_2) \subseteq \text{Inst}(m_1) \times \text{Inst}(m_2)$. ■*

Example 4.1.3. Let

$$\begin{aligned} m_1 &= \langle\langle R(\text{ID}, \text{Age}) \rangle\rangle, \\ m_2 &= \langle\langle S(\text{SSN}) \rangle\rangle \end{aligned}$$

be relational schemas. Let the mapping map be defined using a relational algebra expression as

$$map = \langle\langle \pi_{\text{ID}}(\sigma_{\text{Age}=20}(\text{R})) = \text{S} \rangle\rangle$$

Formally, $(db_1, db_2) \in \text{Inst}(map)$ if and only if $\pi_{\text{ID}}(\sigma_{\text{Age}=20}(db_1.\text{R})) = db_2.\text{S}$. A portion of map is shown in Fig. 4.2. Let mapping map' be specified as an SQL view definition,

$$\begin{aligned} map' &= \langle\langle \text{CREATE VIEW S(SSN) AS} \\ &\quad \text{SELECT ID AS SSN FROM R WHERE Age=20} \rangle\rangle \end{aligned}$$

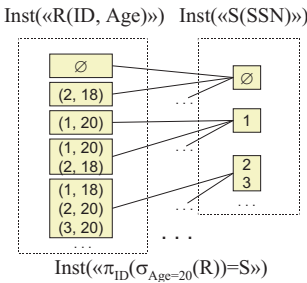


Fig. 4.2. Portion of a mapping

Mappings map and map' are equivalent, i.e., $\text{Inst}(map) = \text{Inst}(map')$. Although map and map' are expressed in different languages they both describe the same correspondence between m_1 and m_2 . Now, let mapping map'' be specified by the view definition

$$\begin{aligned} \text{map}'' &= \langle\langle \text{CREATE VIEW R(ID, Age) AS} \\ &\quad \text{SELECT D.SSN AS ID, 20 AS Age FROM S} \rangle\rangle \end{aligned}$$

Mappings map' and map'' are not equivalent. To see that, notice that by applying the view map' to the database state $(1, 1) \in m_1$ we get $\emptyset \in m_2$, but we cannot obtain $(1, 1)$ from \emptyset using the view map'' . That is, $((1, 1), \emptyset) \in \text{Inst}(\text{map}')$, but $((1, 1), \emptyset) \notin \text{Inst}(\text{map}'')$. In fact, $\text{Inst}(\text{map}'') \subset \text{Inst}(\text{map}')$. ■

Example 4.1.4. Let

$$\begin{aligned} m_1 &= \langle\langle \text{R(ID, Age)} \rangle\rangle, \\ m_2 &= \langle\langle \text{S(SSN, Tel)} \rangle\rangle, \\ \text{map} &= \langle\langle \pi_{\text{ID}}(\text{R}) = \pi_{\text{SSN}}(\text{S}) \rangle\rangle \end{aligned}$$

That is, $(db_1, db_2) \in \text{Inst}(\text{map})$ iff $\pi_{\text{ID}}(db_1.R) = \pi_{\text{SSN}}(db_2.S)$. The mapping establishes a correspondence between two databases $db_1 \in \text{Inst}(m_1)$ and $db_2 \in \text{Inst}(m_2)$ whenever they agree on the values of ID and SSN. This mapping cannot be expressed using a SQL view definition because it is non-functional and not injective; an instance of m_1 does not determine uniquely an instance of m_2 , nor the other way around. ■

A mapping can be thought of as a set of constraints that hold between two models. If $\text{Inst}(m_1 _ m_2)$ yields the whole cross-product $\text{Inst}(m_1) \times \text{Inst}(m_2)$, the relationship between the models is unconstrained, i.e., each pair of states $x \in m_1$, $y \in m_2$ are mutually consistent. For example, think of a mapping between a university payroll database schema and an airline reservation database schema. Such mapping is likely to be unconstrained. If $\text{Inst}(m_1 _ m_2) = \emptyset$, the mapping can be viewed as a contradictory set of constraints. In general, $\text{Inst}(m_1 _ m_2)$ is an arbitrary relationship between instances. It may have several properties whose definitions are summarized below:

Definition 4.1.3. *Let X and Y be two sets. A relation $r \subseteq X \times Y$ is functional, if $(x, y_1), (x, y_2) \in r$ implies $y_1 = y_2$; injective, if $(x_1, y), (x_2, y) \in r$ implies $x_1 = x_2$; total, if $\{x \mid \exists y : (x, y) \in r\} = X$; surjective (onto), if $\{y \mid \exists x : (x, y) \in r\} = Y$. A total, functional, injective, surjective relation is called a bijection.* ■

If $\text{Inst}(m_1 _ m_2)$ is bijective (surjective, functional, etc.), we call the respective mapping a bijection (surjection, function, etc.).

In the examples that we give in the subsequent sections, we refer to certain kinds of mappings as database transformations, views, and queries. A database transformation specifies how an instance of one schema is transformed into an instance of another schema. That is, a database transformation is a functional mapping or simply a function. This function does not need to be total: there may be certain database states on which the transformation is undefined due to a violation of integrity constraints assumed by the transformation but not enforced by the schema. A database query is a database

transformation defined for each initial database state, i.e., a total functional mapping from some model m . We use the term query and view synonymously. More precisely, a database view is a named query, whose result schema, called view schema, can be specified explicitly. This distinction is unimportant for the purposes of state-based semantics.

In this dissertation, we focus on binary mappings, i.e., those that hold between exactly two models. We consider n -ary mappings in Chap. 11 when we discuss future work.

4.1.3 Formal Notation

In this chapter, we consider only the state-based semantics of models and mappings, as described by the instantiation function Inst . To simplify the notation, we henceforth interpret the variables used for models, such as m_1 or m_2 , as set variables, or unary predicate variables. That is, instead of writing $db \in \text{Inst}(m)$ and $\text{Inst}(m_1) = \text{Inst}(m_2)$, we simply write $db \in m$ and $m_1 = m_2$. Similarly, we consider mapping variables, such as $m_1_m_2$ or map , as binary predicate variables and write $(db_1, db_2) \in m_1_m_2$ instead of the more verbose $(db_1, db_2) \in \text{Inst}(m_1_m_2)$. We borrow this notation from the second-order logic, which has variables and quantifiers not only for individuals but also for subsets of the universe and for n -ary relations. Despite using this simplified notation, we stress that a model is more than a set of instances – the latter only characterizes its state-based semantics.

The concrete schema and mapping definitions such as $\langle\langle R(\text{ID}, \text{Age}) \rangle\rangle$, $\langle\langle \text{SELECT A FROM R} \rangle\rangle$, or $\langle\langle \pi_A(R) = S \rangle\rangle$ are interpreted as constant symbols. Notice that a mapping definition is often closely coupled with the models it relates. For example, a relational algebra expression or a SQL view makes sense only when we have the definitions of the schemas it applies to. That is, a more correct notation for the mapping of Example 4.1.3 would be $map = \langle\langle \pi_{\text{ID}}(\sigma_{\text{Age}=20}(R)) = S :: R(\text{ID}, \text{Age}) :: S(\text{SSN}) \rangle\rangle$, which identifies the “left” and “right” models precisely. We continue to use the shorthand notation when the participating models are clear from the context, or abbreviate as $\langle\langle \pi_{\text{ID}}(\sigma_{\text{Age}=20}(R)) = S :: R :: S \rangle\rangle$ where appropriate.

For mapping variables we often use the underscore notation, such as $m_1_m_2$. By convention, in a constant assignment such as $m_1_m_2 = \langle\langle \text{CREATE VIEW R(A) AS SELECT A FROM S :: R(A) :: S(A,B) \rangle\rangle$, the left instances of the mapping are assumed to originate from $\langle\langle R(A) \rangle\rangle$ while the right instances originate from $\langle\langle S(A,B) \rangle\rangle$. The create view statement suggests that the instances of $\langle\langle R(A) \rangle\rangle$ are obtained as a function of the instances of $\langle\langle S(A,B) \rangle\rangle$. That is, $m_1_m_2$ above is non-functional, whereas $\text{Invert}(m_1_m_2)$ is a function. The underscore notation has no special semantics and in particular does not ensure that $\text{Domain}(m_1_m_2) \subseteq m_1$. We state the necessary conditions explicitly where needed.

We describe the application of model-management operators using operational notation. For example, we write $\langle m_x, m_m_x \rangle = \text{Extract}(m, map)$.

Here, the operator `Extract` takes two variables as input and produces two variables as output. We use this notation instead of the predicate-centric one, `Extract($m_x, m_{_}m_x, m, map$)`, because the former is more intuitive. In fact, in all operator definitions that we give, the variables can be clearly divided into input and output variables; the operator definition places constraints on the output variables based on the values of the input variables. Nevertheless, formally the operator `Extract` is a quadrarity predicate, `Invert` is a binary predicate, `Merge` is a sextary predicate etc.

4.1.4 Semantics of Scripts

In this section we explain what it means to compute the results of a model-management script. A model-management script is a conjunction of formulas built of the model-management operators and free variables and constants for models and mappings. That is, a script is a logical formula. The scripts have declarative semantics, which is defined as the standard model-theoretic semantics for logical formulas. Hence, the order of operator “invocations” in a script is irrelevant. We call two scripts t_1 and t_2 *equivalent*, denoted as $t_1 \equiv t_2$, when they are logically equivalent formulas. The fact that it may be possible to rewrite a script into another equivalent script provides the foundation for optimizing the script execution.

Since a script is a formula, executing the script amounts to finding a variable substitution that satisfies it, or makes the script true. Recall that the variables in a script range over relational schemas, SQL views, and other kinds of models and mappings. To compute the results of the script effectively, we construct concrete schema and mapping definitions that make the script true.

In many cases it is impossible to represent the results of scripts exactly using existing schema or mapping languages due to the limited expressiveness of the languages. For example, if we compose a SQL view with an XQuery or with a set of Datalog rules, it may be impossible to describe the result of the composition using a closed expression in an existing database transformation language, so that a special language may need to be invented to hold the result (Shanmugasundaram et al. 2001a). Even if we compose two transformations in the same language, the result may not be expressible in that language, or may produce an infinite set of formulas (Madhavan and Halevy 2003). To use the result of composition in practical applications, we may have to construct a transformation that is equivalent to the one we are looking for except that it covers some irrelevant database states.

The problem of limited expressiveness arises for database schemas, too. For example, in (Buneman et al. 1992) the schema language had to be extended to make sure that all schemas obtained by merging two input schemas can be represented explicitly. In practical applications, it may often be acceptable to construct a more expressive schema if some of the schema constraints are not representable in the target language. Intuitively, we call a schema

more expressive if it allows more valid database states. For example, suppose that schema $\langle\langle R(A,B), S(B,C); \pi_B(R) = \pi_B(S) \rangle\rangle$ is an exact result of a script. The schema defines two tables with a schema constraint. Assume that we need to deploy this schema with a SQL DBMS. The set semantics can be enforced by defining two unique keys over the attributes of each table. The constraint $\pi_B(R) = \pi_B(S)$ is however not expressible in the standard SQL DDL. (If B is a unique key of S , then a foreign key constraint can express that $\pi_B(R) \subseteq \pi_B(S)$. If B is not a key of either table, then neither $\pi_B(R) \subseteq \pi_B(S)$ nor $\pi_B(S) \subseteq \pi_B(R)$ is expressible.) Still, it may be acceptable to delegate the enforcement of the constraint to the application and use a more expressive schema $\langle\langle R[A,B], S[B,C] \rangle\rangle$ that can be defined in SQL DDL.

In many other cases, multiple schema or mapping definitions may satisfy the script. For example, if the variable assignment $map := \langle\langle \pi_{ID}(\sigma_{Age=20}(R)) = S \rangle\rangle$ makes a script true, so do $map := \langle\langle S = \pi_{ID}(\sigma_{Age=20}(R)) \rangle\rangle$ and $map := \langle\langle S = \pi_{ID}(\sigma_{Age=20}(\pi_{ID, Age}(R))) \rangle\rangle$. These expressions are equivalent under state-based semantics, but differ with respect to their syntactic representation. Human input or tuning parameters are required to specify the desired result in such cases, much like a format specification is needed to specify whether the floating-point number 1.3 is to be printed out as “1.3” or “0.13E1”. The script execution environment, such as Rondo, may provide such format specifications implicitly. We do not consider them in this chapter.

The problem of computing the results of scripts effectively appears as one of the most challenging and exciting open issues in model management. As mentioned above, this problem is very hard even if we consider relatively simple languages and just a single operator, such as **Merge** (Buneman et al. 1992; Pottinger and Bernstein 2003) or **Compose** (Madhavan and Halevy 2003). We address it in more detail in Sect. 4.3 and in Chap. 10.

4.1.5 Preliminaries

From now on, we use the notation introduced in Sect. 4.1.3, without the instantiation function `Inst`.

Definition 4.1.4 (Submodel). *A model m' is called a submodel of m if all instances of m' are also instances of m , i.e., $m' \subseteq m$. ■*

Definition 4.1.5 (Subordinate model). *A model m' is called a subordinate model of m , denoted as $m' \leq m$, if m' has at most as many instances as m , i.e., there exists a surjective function from m onto m' . ■*

If $m' \leq m$, we say that m' is equally or less expressive than m , or is *dominated* by m (Hull 1986).

Definition 4.1.6 (Minimal model). *Model m_{min} is a minimal model of the class $C = \{m_1, \dots, m_k\}$ if $m_{min} \in C$ and $m_{min} \leq m_i$ for each $m_i \in C$.*

Example 4.1.5. Schema

$$m_1 = \langle\langle S(\underline{\text{Name}}: \text{char}(3), \text{Sex}: \text{bool}) \rangle\rangle$$

in which Name is a primary key, is a submodel of

$$m_2 = \langle\langle R(\text{Name}: \text{char}(3), \text{Sex}: \text{bool}) \rangle\rangle.$$

In general, if m is a database schema, adding a constraint to m yields a submodel of m . Both m_1 and m_2 are subordinate models of

$$m_3 = \langle\langle T(\text{Name}: \text{char}(4)) \rangle\rangle.$$

All of m_1 , m_2 , and m_3 are subordinate models of

$$m_4 = \langle\langle U(\text{FN}: \text{char}(2), \text{LN}: \text{char}(2)) \rangle\rangle.$$

m_4 is not a subordinate model of m_3 , i.e., it describes more database states than m_3 . Indeed, observe that two strings of size ≤ 2 cannot always be encoded losslessly in a string of size ≤ 4 . For example, concatenations “a” + “bc” and “ab” + “c” both yield “abc”.

Models m_1 and m_2 are minimal models of the class $\{m_1, m_2, m_3, m_4\}$. ■

Definition 4.1.7 (Equivalence). *Models m and m' are equivalent if they have identical instance sets, denoted as $m = m'$.* ■

Definition 4.1.8 (Equipotence). *Models m and m' are equipotent, or equally expressive, denoted as $m \cong m'$, if m has exactly as many instances as m' , i.e., there exists a bijection between m and m' .* ■

Example 4.1.6. Schemas

$$m_1 = \langle\langle S(\text{A}: \text{char}(3), \text{B}: \text{bool}) \rangle\rangle,$$

$$m_2 = \langle\langle R(\text{Name}: \text{char}(3), \text{Sex}: \text{bool}) \rangle\rangle$$

are equivalent. They are not identical: in the abbreviated notation introduced in Sect. 4.1.3, $m = m'$ is a shortcut for $\text{Inst}(m) = \text{Inst}(m')$. However, if the full notation is used, $\text{Inst}(m) = \text{Inst}(m')$ is not equivalent to and does not imply $m = m'$. Schema

$$m_3 = \langle\langle T(\text{Val}: \text{int}(1..33686018)) \rangle\rangle$$

is equipotent with m_1 and m_2 assuming that the characters are drawn from an alphabet of size 256; in this case, $|\text{char}(3)| \cdot |\text{bool}| = 33686018$ (compare Example 4.1.1). ■

We borrow our definitions of equivalence and equipotence from the standard set theory. Notice that schema equivalence is defined differently in (Hull 1986; Miller et al. 1994). Their definition corresponds to that of equipotence (Definition 4.1.8).

4.2 Operators

In this section we define the state-based semantics of the key model-management operators. The signatures and informal descriptions of the operators are summarized in Table 4.1. Wherever possible, we illustrate the results of operators using examples of concrete schema and mapping languages. We also point out when the results cannot be represented exactly in the languages that we consider. Some of the examples we give are non-trivial. In these cases, we provide the proofs for the propositions stated in the examples.

Table 4.1. Summary of key model-management operators

Signature	Description
$m_1_m_3 = \text{Compose}(m_1_m_2, m_2_m_3)$ $= m_1_m_2 \circ m_2_m_3$ $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$	Composes mappings $m_1_m_2$ and $m_2_m_3$ Extracts a subordinate model m_x of m that participates in mapping m_m'
$\langle m, m_m_1, m_m_2 \rangle =$ $\text{Merge}(m_1, m_2, m_1_m_2)$ $\langle m_d, m_m_d \rangle = \text{Diff}(m, m_m')$	Merges models m_1 and m_2 using mapping $m_1_m_2$ Returns a subordinate model m_d of m that does <i>not</i> participate in mapping m_m'
$map_3 = \text{Confluence}(map_1, map_2)$ $= map_1 \oplus map_2$ $m_1_m_2 = \text{Match}(m_1, m_2)$	Combines mappings map_1 and map_2 into mapping map_3 Returns a mapping $m_1_m_2$ between m_1 and m_2
Auxiliary operators	
$m_1_m_2 = m_1 \times m_2$ $map = \text{Id}(m)$ $m = \text{Domain}(map)$ $m = \text{Range}(map)$ $map_2 = \text{Invert}(map_1)$	Returns the “unrestricted” cross-product mapping between models m_1 and m_2 Returns the identity mapping map for model m Returns model m that holds the instances in the domain of mapping map Returns model m that holds the instances in the range of mapping map Swaps the “left” and “right” side of the input mapping map_1

We use the following auxiliary operators:

1. $m_1 \times m_2 =_{\text{df}} \{(x, y) \mid x \in m_1 \text{ and } y \in m_2\}$ defines the cross-product of two models.
2. $\text{Id}(m) =_{\text{df}} \{(x, x) \mid x \in m\}$ is the identity mapping on m .
3. $\text{Domain}(map) =_{\text{df}} \{x \mid (x, y) \in map\}$.
4. $\text{Range}(map) =_{\text{df}} \{y \mid (x, y) \in map\}$.
5. $\text{Invert}(m_1_m_2) =_{\text{df}} \{(y, x) \mid (x, y) \in m_1_m_2\}$. The operator **Invert** is discussed in more detail in Sect. 4.2.2.

4.2.1 Compose Operator

To motivate the definition of the Compose operator, consider the following example.

Example 4.2.1. Let $m_1_m_2$ be a mapping between a relational schema m_1 and an XML schema m_2 used for data exchange, $m_1_m_2 \subseteq m_1 \times m_2$. For a given database $x \in m_1$ the mapping generates an XML document $y \in m_2$. Assume that the XML schema m_2 has been modified into schema m_3 . Let $m_2_m_3 \subseteq m_2 \times m_3$ be the mapping between the original and the new XML schema. To derive the updated export mapping, we compute the composition of $m_1_m_2$ and $m_2_m_3$, denoted as $\text{Compose}(m_1_m_2, m_2_m_3)$ or simply as $m_1_m_2 \circ m_2_m_3$. ■

Definition 4.2.1 (Compose, \circ).

$m_1_m_2 \circ m_2_m_3 =_{\text{df}} \{(x, z) \mid (x, y) \in m_1_m_2 \text{ and } (y, z) \in m_2_m_3\}$ ■

Obviously, $m_1_m_2 \circ m_2_m_3 \subseteq m_1 \times m_3$. Next, we consider three examples of composition of SQL views. In each of the examples, the views have distinct directionality: $m_1 \rightarrow m_2 \rightarrow m_3$, $m_1 \rightarrow m_2 \leftarrow m_3$, or $m_1 \leftarrow m_2 \rightarrow m_3$.

Example 4.2.2 ($m_1 \rightarrow m_2 \rightarrow m_3$). Let

$$\begin{aligned} m_1 &= \langle\langle R(A,B), S(B,C) \rangle\rangle, \\ m_2 &= \langle\langle T(A,C) \rangle\rangle, \\ m_3 &= \langle\langle U(A) \rangle\rangle, \\ m_1_m_2 &= \langle\langle \text{CREATE VIEW T(A, C) AS} \\ &\quad \text{SELECTD R.A, S.C FROM R, S WHERE R.B=S.B} \rangle\rangle, \\ m_2_m_3 &= \langle\langle \text{CREATE VIEW U(A) AS} \\ &\quad \text{SELECTD T.A FROM T WHERE T.C=5} \rangle\rangle \end{aligned}$$

Then, the composition $m_1_m_3 = m_1_m_2 \circ m_2_m_3$ can be specified as

$$\begin{aligned} &\langle\langle \text{CREATE VIEW U(A) AS} \\ &\quad \text{SELECTD R.A FROM R, S WHERE R.B=S.B AND S.C=5} \rangle\rangle \end{aligned}$$

Proof: Observe that $m_1_m_2 = \langle\langle \pi_{A,C}(R \bowtie S) = T \rangle\rangle$ and $m_2_m_3 = \langle\langle \pi_A(\sigma_{C=5}(T)) = U \rangle\rangle$. That is, $(x, y) \in m_1_m_2$ iff $\pi_{A,C}(x.R \bowtie x.S) = y.T$. Similarly, $(y, z) \in m_2_m_3$ iff $\pi_A(\sigma_{C=5}(y.T)) = z.U$. By Definition 4.2.1, $m_1_m_3 = \{(x, z) \mid (x, y) \in m_1_m_2 \text{ and } (y, z) \in m_2_m_3\} = \{(x, z) \mid \pi_{A,C}(x.R \bowtie x.S) = y.T \text{ and } \pi_A(\sigma_{C=5}(y.T)) = z.U\} = \{(x, z) \mid \pi_A(\sigma_{C=5}(\pi_{A,C}(x.R \bowtie x.S))) = z.U\} = \{(x, z) \mid \pi_A(\sigma_{C=5}(x.R \bowtie x.S)) = z.U\}$. That is, $m_1_m_3 = \langle\langle \pi_A(\sigma_{C=5}(R \bowtie S)) = U \rangle\rangle$, which is equivalent to the above view definition. ■

Example 4.2.3 ($m_1 \rightarrow m_2 \leftarrow m_3$). Let

$$\begin{aligned} m_1 &= \langle\langle R(A,B), S(B,C) \rangle\rangle, \\ m_2 &= \langle\langle U(A) \rangle\rangle, \end{aligned}$$

$$\begin{aligned}
m_3 &= \langle\langle T(A,C) \rangle\rangle, \\
m_1_m_2 &= \langle\langle \text{CREATE VIEW U(A) AS} \\
&\quad \text{SELECTD R.A FROM R, S} \\
&\quad \text{WHERE R.B IN (SELECTD S.B FROM S)} \rangle\rangle, \\
m_2_m_3 &= \langle\langle \text{CREATE VIEW U(A) AS} \\
&\quad \text{SELECTD T.A FROM T WHERE T.C=5} \rangle\rangle
\end{aligned}$$

(The mapping $m_2_m_3$ is described using a view definition that maps m_3 to m_2 .) Then,

$$m_1_m_3 = m_1_m_2 \circ m_2_m_3 = \langle\langle \pi_A(R \bowtie S) = \pi_A(\sigma_{C=5}(T)) \rangle\rangle.$$

There is no equivalent view definition for this relational algebra expression because the mapping $m_1_m_3$ is non-functional and not injective.

Proof: Observe that $m_1_m_2 = \langle\langle \pi_A(R \bowtie S) = U \rangle\rangle$ and $m_2_m_3 = \langle\langle \pi_A(\sigma_{C=5}(T)) = U \rangle\rangle$. We obtain the proposition using the same arguments as in Example 4.2.2. \blacksquare

Example 4.2.4 ($m_1 \leftarrow m_2 \rightarrow m_3$). Let

$$\begin{aligned}
m_1 &= \langle\langle T(A,C) \rangle\rangle, \\
m_2 &= \langle\langle R(A,B), S(B,C) \rangle\rangle, \\
m_3 &= \langle\langle U(A) \rangle\rangle.
\end{aligned}$$

Let $|A|, |B|, |C|$ be domain sizes of attributes A, B, and C. Assume that $|B| > |A| \cdot |C|$. Further, let

$$\begin{aligned}
m_1_m_2 &= \langle\langle \text{CREATE VIEW T(A, C) AS} \\
&\quad \text{SELECTD R.A, S.C FROM R, S WHERE R.B=S.B} \rangle\rangle, \\
m_2_m_3 &= \langle\langle \text{CREATE VIEW U(A) AS} \\
&\quad \text{SELECTD R.A FROM R WHERE R.B=3} \rangle\rangle.
\end{aligned}$$

Then, the composition $m_1_m_3 = m_1_m_2 \circ m_2_m_3$ is unrestricted, i.e., $m_1_m_3 = m_1 \times m_3$. Obviously, $m_1_m_3$ cannot be represented using a SQL view.

Proof: Observe that $m_1_m_2 = \langle\langle \pi_{A,C}(R \bowtie S) = T \rangle\rangle$ and $m_2_m_3 = \langle\langle \pi_A(\sigma_{B=3}(R)) = U \rangle\rangle$. Thus, $m_1_m_3 = \{(x, z) \mid \pi_{A,C}(y.R \bowtie y.S) = x.T \text{ and } \pi_A(\sigma_{B=3}(y.R)) = z.U\}$. Now we show that for each $x \in m_1$ and for each $z \in m_3$ we can find $y \in m_2$ so that the above condition holds. Assume that database instances x and z are given. We construct the database y using the following view definitions:

$$\begin{aligned}
&\text{CREATE VIEW R(A, B) AS} \\
&\quad (\text{SELECTD U.A, 3 AS B FROM U}) \text{ UNION} \\
&\quad (\text{SELECTD T.A, Sk(T.A, T.C) AS B FROM T}) \\
&\text{CREATE VIEW S(B, C) AS} \\
&\quad \text{SELECTD Sk(T.A, T.C) AS B, T.C FROM T}
\end{aligned}$$

where $\text{Sk}(\dots)$ is a Skolem function that generates a distinct value $b = \text{Sk}(a, c) \neq 3$ from the domain of B for each pair of A and C values. Such a Skolem function exists, since $|B| > |A| \cdot |C|$. When this property is not guaranteed, we get $m_1 _ m_3 = \{(x, z) \mid \pi_{A,C}(y.R \bowtie y.S) = x.T\}$, i.e., the “right side” of the mapping is unconstrained. ■

Proposition 4.2.1. *Operator Compose is associative, i.e., $map_1 \circ (map_2 \circ map_3) = (map_1 \circ map_2) \circ map_3$.*

Proof: follows from Definition 4.2.1. ■

4.2.2 Invert Operator

The operator **Invert** swaps the “left” and “right” side of a mapping. For convenience, its definition is repeated below:

Definition 4.2.2 (Invert). $\text{Invert}(m_1 _ m_2) =_{\text{df}} \{(y, x) \mid (x, y) \in m_1 _ m_2\}$. ■

By reversing the sides of a mapping we can ensure that its directionality fits other operations, such as composition or merging. An inverted mapping still describes the same correspondence between two models. When we use mapping constants, we have to specify the “left” and “right” schemas explicitly, as we explained in Sect. 4.1.3, to distinguish the mapping from its inverted mapping.

Example 4.2.5. Consider the mapping

$$map = \ll \pi_{\text{ID}}(\sigma_{\text{Age}=20}(\text{R})) = \text{S} :: \text{R}(\text{ID}, \text{Age}) :: \text{S}(\text{SSN}) \gg$$

from Example 4.1.3. The inverted mapping can be represented as

$$\text{Invert}(map) = \ll \pi_{\text{ID}}(\sigma_{\text{Age}=20}(\text{R})) = \text{S} :: \text{S}(\text{SSN}) :: \text{R}(\text{ID}, \text{Age}) \gg. \quad \blacksquare$$

As mentioned earlier, the state-based semantics does not prescribe the exact syntactic representation for models and mappings. For example, we cannot state that $\text{Invert}(map)$ in the example above should be computed as a view definition, and not as a relational algebra expression, and that in this view definition the relation R should be defined as a view on S and not vice-versa. This constraint is part of the structural semantics. Still, using the state-based semantics we could tell that such a view on S cannot possibly exist, since it would require $\text{Invert}(map)$ to be a function, but the mapping map is not injective.

The following propositions summarize some important well-known algebraic properties that we use in subsequent sections:

Proposition 4.2.2. *$\text{Invert}(\text{Invert}(map)) = map$.* ■

Proposition 4.2.3. $Invert(map_1 \circ map_2) = Invert(map_2) \circ Invert(map_1)$. ■

Proposition 4.2.4. *Mapping map is a surjective function onto m if and only if $Invert(map) \circ map = Id(m)$.* ■

4.2.3 Extract Operator

The operator **Extract** takes a model m and a mapping map between m and some model m' , and returns a subordinate model m_x of m that “participates” in the mapping. Before we give a formal definition of **Extract**, we explain the intuition behind this operator using a motivating example.

Example 4.2.6. Imagine that m is a legacy database schema and q is a query over m . Our goal is to upgrade the legacy database by producing a new schema m_x that captures only the information that can actually be queried using q and no other information (see Fig. 4.3). That is, m_x is a minimal schema that still allows us to obtain all query results that we can obtain by running q against m . In addition to the new schema m_x , we need a database transformation m_m_x that tells us how the data of m can be migrated to m_x . After migrating all instances of m to m_x , we can reformulate our original query q to run against m_x . To do this, we compose the reverse transformation $Invert(m_m_x)$ and q . Notice that q may yield the same query result for multiple different database states of m , because m includes irrelevant legacy information. Such database states are indistinguishable under q and are “collapsed” into a single database state of m_x by way of m_m_x . ■

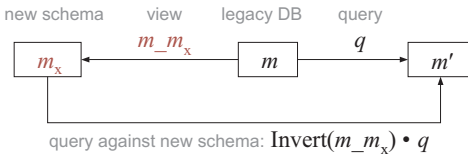


Fig. 4.3. Schematic representation for Example 4.2.6 (**Extract**)

The definition of **Extract** that we present below describes formally the properties of m_x and m_m_x in the above example. The definition covers a general case in which q is an arbitrary, possibly non-functional mapping.

Definition 4.2.3 (Extract). *Let $Domain(q) \subseteq m$. $\langle m_x, m_m_x \rangle = Extract(m, q)$ holds if and only if*

- i. $m_x = Range(m_m_x)$.
- ii. $m_m_x \circ Invert(m_m_x) \circ q = q$.
- iii. m_x is a minimal model satisfying (i) and (ii). ■

To tie the definition to the motivating example, observe that m_m_x is the database transformation from m to the new schema m_x , while $\text{Invert}(m_m_x) \circ q$ is the updated query over m_x . Hence, condition (ii) requires the updated query over m_x to produce the same results as the original query q over m . Notice that in general, $m_m_x \circ \text{Invert}(m_m_x)$ is not one-to-one, i.e., condition (ii) is not a tautology.

Definition 4.2.3 specifies a quadraxy predicate. Whether it holds or not for fixed m_x , m_m_x , m , q is hard to verify formally because condition (ii) is a non-trivial expression and condition (iii) involves a test over all models m_x that satisfy (i) and (ii), as required by Definition 4.1.6. Therefore, before we give detailed examples of **Extract**, we prove the following theorem that allows us to reformulate the Definition 4.2.3 into an equivalent set of conditions that are substantially easier to verify. In the theorem, we utilize an auxiliary predicate *ind* (for “indistinguishable”)

$$\text{ind}(y_1, y_2, m_m') =_{\text{df}} (\{z_1 \mid (y_1, z_1) \in m_m'\} = \{z_2 \mid (y_2, z_2) \in m_m'\})$$

It holds whenever the “projections” of y_1 and y_2 are equal. If $\text{ind}(y_1, y_2, m_m')$, we say that y_1 and y_2 are indistinguishable under m_m' . $\text{ind}(\cdot, \cdot, m_m')$ is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.

Theorem 4.2.1 (Simplification of Extract). *Let $\text{Domain}(m_m') \subseteq m$. $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$ holds if and only if the following conditions are satisfied:*

1. $m_x = \text{Range}(m_m_x)$.
2. $\text{Domain}(m_m_x) = \text{Domain}(m_m')$.
3. For all $(y_1, x_1), (y_2, x_2) \in m_m_x$: $x_1 = x_2$ iff $\text{ind}(y_1, y_2, m_m')$. ■

Condition (2) makes sure that exactly those instances of m participate in m_m_x that are connected in m_m' . Condition (3) requires collapsing any two instances y_1 and y_2 of m into a single instance of m_x if and only if y_1 and y_2 are indistinguishable under m_m' . The proof of the theorem is in Appendix B.

The following examples illustrate the operator **Extract**.

Example 4.2.7. Fig. 4.4 shows a valid result of applying the operator **Extract** to a model m with seven instances. y_2 and y_3 are indistinguishable under m_m' since they are associated with the same z_2 , i.e., $\text{ind}(y_1, y_2, m_m')$ holds. Therefore, they are collapsed into a single instance x_2 of m_x . All other instances of m are pairwise distinguishable. For example, y_5 is associated with $\{z_5, z_6\}$ and y_6 with $\{z_6\}$ so that y_5 and y_6 are connected with two distinct instances of m_x . Instance y_7 is not connected in m_m' and thus has no counterpart in m_x . ■

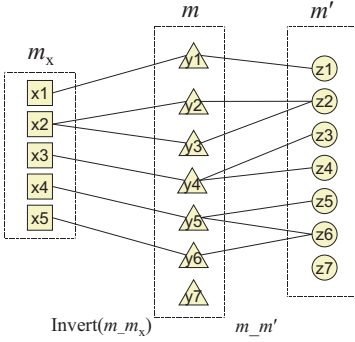


Fig. 4.4. Illustration of Extract operator

Example 4.2.8. Let

$$m = \langle\langle R(A,B), S(C) \rangle\rangle,$$

$$m' = \langle\langle T(A,D) \rangle\rangle,$$

$$m_m' = \langle\langle \text{CREATE VIEW T(A,D) AS} \\ \text{SELECT D A, 5 AS D FROM R} \\ \text{:: R(A,B), S(C) :: T(A,D)} \rangle\rangle.$$

Then,

$$m_x = \langle\langle V(A) \rangle\rangle,$$

$$m_m_x = \langle\langle \text{CREATE VIEW V(A) AS} \\ \text{SELECT D A FROM R} \\ \text{:: R(A,B), S(C) :: V(A)} \rangle\rangle$$

is a valid result of extraction.

Proof: $V(A)$ is a view schema in m_m_x so that m_m_x is a surjective function and condition (1) is satisfied. Condition (2) is trivially true since all instances of m participate in m_m' and m_m_x . To verify condition (3), note that $\{z \mid (y, z) \in m_m'\}$ describes all instances of $\langle\langle T(A,D) \rangle\rangle$ that can be obtained using the view m_m' on the database $y \in m$. The view selects A values from R , therefore $ind(y_1, y_2, m_m')$ holds iff $\pi_A(y_1.R) = \pi_A(y_2.R)$. On the other hand, $m_m_x = \langle\langle \pi_A(R) = V \rangle\rangle$, i.e., $(y, x) \in m_m_x$ iff $\pi_A(y.R) = x.V$. Hence, for all $(y_1, x_1), (y_2, x_2) \in m_m_x$ we obtain: $ind(y_1, y_2, m_m')$ iff $\pi_A(y_1.R) = x_1.V$ and $\pi_A(y_2.R) = x_2.V$ iff $x_1.V = x_2.V$ iff $x_1 = x_2$. Hence, condition (3) holds. ■

Example 4.2.9. Let

$$m = \langle\langle R(\text{Name: char}(10), \text{Salary: real}, \text{Year: int}) \rangle\rangle,$$

$$m_m' = \langle\langle \text{SELECT D Name, SUM(Salary) AS Income} \\ \text{FROM R GROUP BY Name} \rangle\rangle.$$

Then,

$$m_x = \langle\langle S(\text{Name: char}(10), \text{Income: real}) \rangle\rangle$$

is a valid result of extraction. Notice that S.Name is defined as a primary key. Its uniqueness is guaranteed by the GROUP BY clause in m_m' . ■

Example 4.2.10. This example illustrates extraction when m_m' contains a join. Let

$$\begin{aligned} m &= \langle\langle R(A,B), S(B,C) \rangle\rangle, \\ m' &= \langle\langle T(A,B,C) \rangle\rangle, \\ m_m' &= \langle\langle T = R \bowtie S \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m_x &= \langle\langle P(A,B), Q(B,C); \pi_B(P) = \pi_B(Q) \rangle\rangle, \\ m_m_x &= \langle\langle P = \pi_{A,B}(R \bowtie S), Q = \pi_{B,C}(R \bowtie S) \rangle\rangle \end{aligned}$$

is a valid result of **Extract**.

Proof: The proof consists of two parts. First (\rightarrow), we show that each result of the query can be kept by a unique instance of m_x . Then (\leftarrow), we demonstrate that each instance of m_x can be obtained as a result of the query.

(\rightarrow) Notice that the query $\langle\langle T = \sigma_{B \in \mathcal{J}}(R) \bowtie \sigma_{B \in \mathcal{J}}(S), \mathcal{J} = \pi_B(R) \cap \pi_B(S) \rangle\rangle$ produces the identical result as m_m' . That is, if we first select from R and S only the tuples in which B values are shared across R and S, and join them, we obtain the same result as by joining R and S directly. Thus, we can “shred” any given instance of the result $R \bowtie S$ into P and Q with $\pi_B(P) = \pi_B(Q)$ and reconstruct it using a join $P \bowtie Q$ without information loss.

(\leftarrow) Observe that $\pi_{A,B}(P \bowtie Q) = P$ and $\pi_{B,C}(P \bowtie Q) = Q$ for each P and Q such that $\pi_B(P) = \pi_B(Q)$. That is, we can join P and Q into a new table, which represents a valid result of the query m_m' , and reconstruct P and Q again from this new table.

Together, (\rightarrow) and (\leftarrow) yield that m_x is equipotent with the set of possible results of the query (condition (3)). Moreover, the construction used in (\leftarrow) tells us that m_m_x is surjective (condition (1)), and all instances of m participate in m_m' and m_m_x (condition (2)). ■

In the previous examples, m_m' was a total function from m to m' . Next we illustrate the case when m_m' is not a function, but instead $\text{Invert}(m_m')$ is a function from m' into m .

Example 4.2.11. Let

$$\begin{aligned} m' &= \langle\langle R(A,B), S(B,C) \rangle\rangle, \\ m &= \langle\langle T(A,B,C,D) \rangle\rangle, \\ m_m' &= \langle\langle \text{CREATE VIEW } T(A,B,C,D) \text{ AS} \\ &\quad \text{SELECT } D \text{ A, B, C, } 5 \\ &\quad \text{FROM } R, S \\ &\quad \text{WHERE } R.B=S.B \text{ AND } S.C=4 \rangle\rangle \end{aligned}$$

Notice that $\text{Invert}(m_m')$ transforms deterministically each instance of m' to an instance of m (instances of m' in which no tuple of S matches the WHERE clause are mapped to the same instance of m , the empty relation T). Therefore, $\text{Invert}(m_m')$ is a total functional mapping. Because of that, for all $y_1, y_2 \in m$: $\text{ind}(y_1, y_2, m_m')$ iff $y_1 = y_2$. Thus, condition (3) of Theorem 4.2.1 becomes: for all $(y_1, x_1), (y_2, x_2) \in m_m_x$: $x_1 = x_2$ iff $y_1 = y_2$. In other words, m_m_x and $\text{Invert}(m_m_x)$ must be injective functions.

Now, observe that $\text{Invert}(m_m')$ is not onto: there exist instances of m that are not computable from any instance of m' , such as $\{(0, 0, 0, 0)\} \in m$. Condition (3) states that such instances may not participate in m_m_x . The instances computable from instances of m' are precisely those relations $y \in m$ in which every tuple $t \in y$ satisfies the condition ($t.C=4$ and $t.D=5$). Thus, the injective function m_m_x must assign to each such $y \in m$ exactly one instance of m_x . The mapping

$$m_m_x = \langle\langle \text{CREATE VIEW T(A,B,C,D) AS} \\ \text{SELECT D A, B, 4, 5 FROM V} \\ \text{:: T(A,B,C,D) :: V(A,B)} \rangle\rangle$$

defines such a function. m_m_x is a surjective function onto $m_x = \langle\langle V(A,B) \rangle\rangle$, i.e., for each relation $x \in \langle\langle V(A,B) \rangle\rangle$ we can find a source relation $y \in \langle\langle T(A,B,C,D) \rangle\rangle$ such that the view transforms y into x . Hence, the conditions of Theorem 4.2.1 are satisfied. In contrast, specifying m_m_x as

$$\langle\langle \text{CREATE VIEW V(A,B) AS} \\ \text{SELECT D A, B FROM T} \\ \text{:: T(A,B,C,D) :: V(A,B)} \rangle\rangle$$

violates condition (3). ■

Theorem 4.2.2. *If $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$, then m_m_x is a surjective function.*

Proof: Let $(y, x_1), (y, x_2) \in m_m_x$. Assume that $x_1 \neq x_2$. Then, by condition (3) of Theorem 4.2.1, $\text{ind}(y, y, m_m')$ is false. This is a contradiction, since $\text{ind}(\cdot, \cdot, \cdot)$ is reflexive. So, our assumption is false, and $x_1 = x_2$. Hence, $\text{Invert}(m_m_x)$ is a function. m_m_x is surjective by condition (1). ■

Notice that m_m_x is in general not total, i.e., it is a database transformation, but not necessarily a view. For an illustration, see Example 4.2.10.

Proposition 4.2.5. *If $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$ and $\langle m_y, m_m_y \rangle = \text{Extract}(m, m_m')$, then there exists a bijection between m_x and m_y , namely $m_x \cdot m_y = \text{Invert}(m_m_x) \circ m_m_y$. That is, model m_x in Definition 4.2.3 is defined uniquely up to isomorphism.* ■

Proposition 4.2.6. *Let $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$. If $\text{Invert}(m_m')$ is surjective, then m_m_x is a total surjective function. If $\text{Invert}(m_m')$ is a surjective function, then m_m_x is a bijection, i.e., the extracted model is isomorphic to the input model.* ■

Proposition 4.2.7. *Let $\langle m_x, m_{_m_x} \rangle = \text{Extract}(m, m_{_m'})$. If $m_{_m'}$ is a function, then $\text{Invert}(m_{_m'}) \circ m_{_m_x}$ is an injective function. If $m_{_m'}$ is a surjective function, then $\text{Invert}(m_{_m'}) \circ m_{_m_x}$ (and its inverse $\text{Invert}(m_{_m_x}) \circ m_{_m'}$) are bijections. ■*

4.2.4 Merge Operator

We explain the intuition behind the **Merge** operator using the following data integration scenario.

Example 4.2.12. Consider a company with two departments. Each of the departments manages its own database. Let m_1 and m_2 be the respective database schemas (see Fig. 4.5). Schemas m_1 and m_2 are not disjoint; for instance, both databases contain employee data. To simplify the management of data consistency across the departmental databases, the company decides to keep all data in a centralized database, while the departments access the data using view schemas m_1 and m_2 . Thus, the goal is to create a global schema m for the centralized database such that m is minimal, i.e., it captures only the information needed by the departments, and no other information.

Let the mapping $m_1_{_m_2}$ describe how m_1 relates to m_2 , i.e., $m_1_{_m_2}$ identifies all mutually consistent database states $x \in m_1, y \in m_2$. In other words, each pair $(x, y) \in m_1_{_m_2}$ represents a single valid state $z \in m$ of the centralized database. The states x and y need to be “glued” into z in such a way that we can unambiguously reconstruct x and y from z using two functional mappings, $m_{_m_1}$ and $m_{_m_2}$. To prevent information loss, the centralized database must be able to represent each state of m_1 and each state of m_2 even if they are not mutually consistent (for example, think of a temporary inconsistency that may occur when a negative account balance indicating a debt in $x \in m_1$ is disallowed in the billing schema m_2). That is, $m_{_m_1}$ and $m_{_m_2}$ are not total, they are database transformations, but not views. ■

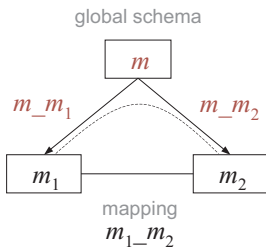


Fig. 4.5. Schematic representation for Example 4.2.12 (Merge)

We define the operator `Merge` as follows:

Definition 4.2.4 (Merge). Let $m_1 _ m_2 \subseteq m_1 \times m_2$. $\langle m, m_m_1, m_m_2 \rangle = \text{Merge}(m_1, m_2, m_1 _ m_2)$ holds if and only if

- i. m_m_1 and m_m_2 are surjective functions onto m_1 and m_2 , respectively.
- ii. $m_1 _ m_2 = \text{Invert}(m_m_1) \circ m_m_2$.
- iii. $m = \text{Domain}(m_m_1) \cup \text{Domain}(m_m_2)$.
- iv. m is minimal model satisfying (i)-(iii). ■

Condition (i) enables us to reconstruct instances x and y from z in a unique fashion and ensures that each instance of m_1 and each instance of m_2 is representable in m . The effect of condition (ii) is that the instances x and y that we obtain using the database transformations m_m_1 and m_m_2 are, if they both exist, mutually consistent. Condition (iii) requires each instance $z \in m$ to represent a valid state of affairs, which can be attributed either to m_1 or m_2 , or both. The minimality condition (iv) prevents `Merge` from “inventing” instances of m that are not absolutely necessary for representing all of m_1 and m_2 .

Example 4.2.13. Fig. 4.6 illustrates the `Merge` operator. The input mapping $m_1 _ m_2$ is shown using light-gray lines. ■

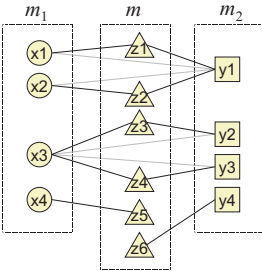


Fig. 4.6. Illustration of Merge operator

Definition 4.2.4 specifies a sextary predicate. Whether it holds or not for fixed $m, m_m_1, m_m_2, m_1, m_2, m_1 _ m_2$ is hard to verify formally even for the simple example of Fig. 4.6, since condition (iv) involves a test over all models m that satisfy (i)-(iii). Therefore, before we give detailed examples of `Merge`, we present the following theorem that allows us to reformulate condition (iv) into another condition that is easier to check. Its proof is in Appendix B.

Theorem 4.2.3 (Simplification of Merge). Let $m_1 _ m_2 \subseteq m_1 \times m_2$. $\langle m, m_m_1, m_m_2 \rangle = \text{Merge}(m_1, m_2, m_1 _ m_2)$ holds if and only if

- the conditions (i)-(iii) of Definition 4.2.4 are satisfied, and

– $|m| = \text{mergeCard}(m_1, m_2, m_1 _ m_2)$, where $\text{mergeCard}(m_1, m_2, m_1 _ m_2) =_{\text{df}} |m_1 _ m_2| + |m_1 - \text{Domain}(m_1 _ m_2)| + |m_2 - \text{Range}(m_1 _ m_2)|$.

If $m_1 _ m_2$ is total and surjective, or if $m _ m_1$ and $m _ m_2$ are total, then $\text{mergeCard}(m_1, m_2, m_1 _ m_2) = |m_1 _ m_2|$.

Example 4.2.14. Let

$$\begin{aligned} m_1 &= \langle\langle R_1(A), S_1(B) \rangle\rangle, \\ m_2 &= \langle\langle R_2(A), T_2(C) \rangle\rangle, \\ m_1 _ m_2 &= \langle\langle R_1 = R_2 \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m &= \langle\langle R(A), S(B), T(C) \rangle\rangle, \\ m _ m_1 &= \langle\langle R_1 = R, S_1 = S \rangle\rangle, \\ m _ m_2 &= \langle\langle R_2 = R, T_2 = T \rangle\rangle \end{aligned}$$

is a valid result of Merge.

Proof: Mappings $m _ m_1$ and $m _ m_2$ are views on m . Hence, conditions (i) and (iii) hold. By definition of $m_1 _ m_2$, $(x, y) \in m_1 _ m_2$ iff $x.R_1 = y.R_2$. By definition of $m _ m_1$ and $m _ m_2$: $(x, y) \in \text{Invert}(m _ m_1) \circ m _ m_2$ iff exists $z \in m$ with $(z, x) \in m _ m_1$ and $(z, y) \in m _ m_2$ iff exists z with $x.R_1 = z.R$, $x.S_1 = z.S$, $y.R_2 = z.R$, $y.T_2 = z.T$ iff exists z with $z.R = x.R_1 = y.R_2$, $z.S = x.S_1$, $z.T = y.T_2$. Such z exists if and only if $x.R_1 = y.R_2$. Thus, condition (ii) is satisfied.

We prove (iv) by Theorem 4.2.3. Notice that $m _ m_1$ and $m _ m_2$ are total, so we have to show that $|m| = |m_1 _ m_2|$. Let $x \in m_1$. Instance $y \in m_2$ participates in $m_1 _ m_2$ iff $x.R_1 = y.R_2$ and $y.T_2$ is arbitrary. That is, for each $x \in m_1$, exactly $|\langle\langle T_2(C) \rangle\rangle|$ instances of m_2 participate in the mapping. Thus, $|m_1 _ m_2| = |m_1| \cdot |\langle\langle T_2(C) \rangle\rangle| = |\langle\langle R_1(A), S_1(B) \rangle\rangle| \cdot |\langle\langle T_2(C) \rangle\rangle| = |\langle\langle R_1(A), S_1(B), T_2(C) \rangle\rangle| = |m|$. Hence, condition (iv) holds. ■

Observe that in the example above, the merged model could also be expressed as

$$m' = \langle\langle R_1(A), R_2(A), S_1(B), T_2(C); R_1 = R_2 \rangle\rangle.$$

This is a union of the schema signatures of m_1 and m_2 , with a constraint contained in $m_1 _ m_2$. Models m' and m are equipotent, $m' \cong m$. We generalize this observation in the following theorem.

Theorem 4.2.4. *Let \mathcal{L} be a schema language, in which a schema consists of two parts: a schema signature Sig and a constraint expression C . A signature Sig is a set of entity definitions $\{e_1, \dots, e_k\}$. C is a formula in the constraint language of \mathcal{L} , such as first-order predicate calculus. Let*

$$\begin{aligned} m_1 &= \langle\langle \text{Sig}_1; C_1 \rangle\rangle, \\ m_2 &= \langle\langle \text{Sig}_2; C_2 \rangle\rangle \end{aligned}$$

be two schemas in \mathcal{L} . Without loss of generality, assume that the entity labels in $Sig_1 = \{e_1^1, \dots, e_p^1\}$ are disjoint from those of $Sig_2 = \{e_1^2, \dots, e_q^2\}$. Let $m_1 _ m_2$ be a total surjective mapping expressed as a constraint C over Sig_1 and Sig_2 in the constraint language of \mathcal{L} . Then, we can construct a valid result for Merge by creating a union of schema signatures of m_1 and m_2 and a conjunction of constraints C_1, C_2, C , as

$$\begin{aligned} m &= \langle\langle Sig_1 \cup Sig_2; C_1 \wedge C_2 \wedge C \rangle\rangle, \\ m _ m_1 &= \langle\langle \bigwedge_{1 \leq i \leq p} (m_1.e_i^1 = m.e_i^1) \rangle\rangle, \\ m _ m_2 &= \langle\langle \bigwedge_{1 \leq i \leq q} (m_2.e_i^2 = m.e_i^2) \rangle\rangle \end{aligned}$$

where $m _ m_1$ and $m _ m_2$ are views on m .

Proof: Analogous to that of Example 4.2.14. ■

Example 4.2.15. Let

$$\begin{aligned} m_1 &= \langle\langle R_1(A, B) \rangle\rangle, \\ m_2 &= \langle\langle S_2(A, C) \rangle\rangle, \\ m_1 _ m_2 &= \langle\langle \pi_A(R_1) = \pi_A(S_2) \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m &= \langle\langle R(A, B), S(A, C); \pi_A(R) = \pi_A(S) \rangle\rangle, \\ m _ m_1 &= \langle\langle R_1 = R \rangle\rangle, \\ m _ m_2 &= \langle\langle S_2 = S \rangle\rangle \end{aligned}$$

is a valid result of Merge by Theorem 4.2.4. ■

Example 4.2.16. Let

$$\begin{aligned} m_1 &= \langle\langle R[A, B] \rangle\rangle, \\ m_2 &= \langle\langle S[A, C] \rangle\rangle, \\ m_1 _ m_2 &= \langle\langle \text{SELECT A FROM R} = \text{SELECT A FROM S} \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m &= \langle\langle T[A, B, C] \rangle\rangle, \\ m _ m_1 &= \langle\langle \text{SELECT A, B FROM T} \rangle\rangle, \\ m _ m_2 &= \langle\langle \text{SELECT A, C FROM T} \rangle\rangle \end{aligned}$$

is a valid result of Merge.

Proof: Conditions (i) and (iii) are satisfied trivially. We now show condition (ii), that $m_1 _ m_2 = \text{Invert}(m _ m_1) \circ m _ m_2$. Obviously, $\text{Domain}(m_1 _ m_2) = \text{Domain}(\text{Invert}(m _ m_1)) = \text{Domain}(\text{Invert}(m _ m_1) \circ m _ m_2) = m_1$. Let $x \in m_1$, and let $Y_1 = \{y \mid (x, y) \in m_1 _ m_2\}$, $Y_2 = \{y \mid (z, x) \in m _ m_1 \text{ and } (z, y) \in m _ m_2\}$. Condition (ii) holds if $Y_1 = Y_2$. In fact, Y_1 describes all database states of m_2 such that for each $y \in Y_1$, the S.A column of y equals the R.A column of x and the S.C column is unconstrained. In contrast, if we traverse $\text{Invert}(m _ m_1) \circ m _ m_2$ from x , we first obtain all $z \in m$ that agree with x on columns A and B, and then get all y that agree with x on A. Thus, $Y_1 = Y_2$ and

condition (ii) holds. Finally, consider the mapping $m_1_m_2$. $(x, y) \in m_1_m_2$ implies that x and y have the same number of rows. For each A-column of length k , we can construct a list of k B-values and a list of k C-values. Thus, $|m_1_m_2| = \sum k \cdot |A|^k \cdot |B|^k \cdot |C|^k = \sum k \cdot (|A| \cdot |B| \cdot |C|)^k = |\llbracket T[A, B, C] \rrbracket|$. By Theorem 4.2.3, m, m_m_1, m_m_2 is a valid result of Merge. ■

4.2.5 Diff Operator

The operator Diff is complementary to Extract. It takes a model m and a mapping m_m' between m and some model m' , and returns a subordinate model m_d of m that does not “participate” in the mapping. To explain the intuition behind Diff, we continue with the scenario presented in Example 4.2.6.

Example 4.2.17. Let m be the legacy database schema from Example 4.2.6. The legacy database has been migrated to a new operational database with schema m_x (see Fig. 4.7). Assume that due to data protection regulations, all data in the legacy database has to be preserved indefinitely. For efficiency, the legacy data is split between the new operational database and an archival database. Our goal is to develop a schema m_d for the archival database such that m_d captures only the information needed to reconstruct the legacy database from the new operational database and the archive, and no other information. In addition, we need a database transformation m_m_d that allows us to populate m_d with data from m . Together, the transformations m_m_d and m_m_x describe how the data in the new operational database relates to the data in the archive. The correspondence $m_x_m_d$ is defined by composition as $m_x_m_d = \text{Invert}(m_m_x) \circ m_m_d$. The legacy database can be reconstructed by merging m_x and m_d under the mapping $m_x_m_d$. ■

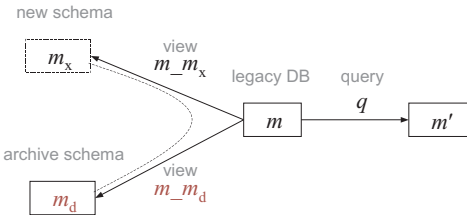


Fig. 4.7. Schematic representation for Example 4.2.17 (Diff)

The formal definition of Diff is given below:

Definition 4.2.5 (Diff). Let $\text{Domain}(m_m') \subseteq m$. $\langle m_d, m_m_d \rangle = \text{Diff}(m, m_m')$ holds if and only if the following conditions are satisfied for some m_x, m_m_x :

- i. $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$.
- ii. $\langle m, m_m_x, m_m_d \rangle = \text{Merge}(m_x, m_d, \text{Invert}(m_m_x) \circ m_m_d)$.
- iii. m_d is a minimal model satisfying (i) and (ii). ■

Operators **Extract** and **Diff** are defined in such a way that for a given pair of instances $x \in m_x$ and $d \in m_d$ we can reconstruct uniquely the instance $y \in m$ from which x and d were obtained by means of m_m_x and m_m_d . If we use the same inputs for **Extract** and **Diff**, we get what we call a *split* (see illustration in Fig. 4.8). Mapping m_m' , which splits the model m into m_x and m_d , is called the *wedge* mapping of a split.

Corollary 4.2.1 (Split). *Let $\langle m_x, m_m_x \rangle = \text{Extract}(m, \text{map})$, $\langle m_d, m_m_d \rangle = \text{Diff}(m, \text{map})$ and $m_x _m_d = \text{Invert}(m_m_x) \circ m_m_d$. Then, we can reconstruct all of m , m_m_x , and m_m_d up to isomorphism from m_x , m_d , and $m_x _m_d$. More precisely, $\langle m, m_m_x, m_m_d \rangle = \text{Merge}(m_x, m_d, m_x _m_d)$ holds.*

Proof: follows from Definition 4.2.5. ■

Definition 4.2.5 specifies a quadrary predicate. Whether it holds or not is hard to verify formally for fixed m_d, m_m_d, m, m_m' , since conditions (i) and (ii) use three complex operators **Extract**, **Merge**, and **Compose**, while condition (iii) requires examining all possible models m_d for which there exist m_x, m_m_x with (i) and (ii). For instance, consider Fig. 4.8. The figure shows a valid result of applying the operator **Diff** to the model m and mapping m_m' of Fig. 4.4. However, the fact that conditions (i)-(iii) are true is not obvious.

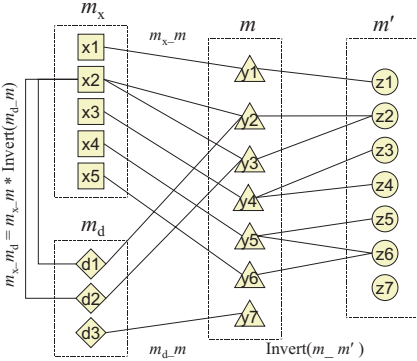


Fig. 4.8. Illustration of Diff operator

Therefore, before giving detailed examples of **Diff**, we present Theorem 4.2.5, which gives an alternative characterization of **Diff** that can be verified easier. The theorem shows how we can reformulate the definition without considering the results of extraction m_x, m_m_x explicitly. We characterize **Diff** using the definition of $\text{ind}(\cdot, \cdot, \cdot)$ from Sect. 4.2.3. Among other things, the theorem states that **Diff** makes all instances of m that are not distinguishable under m_m' distinguishable using m_m_d . Its proof is in Appendix B.

Theorem 4.2.5 (Simplification of Diff). *Let $\text{Domain}(m _ m') \subseteq m$. $\langle m_d, m _ m_d \rangle = \text{Diff}(m, m _ m')$ holds if and only if the following conditions are satisfied:*

1. $m _ m_d$ is a surjective function from m onto m_d .
2. For all $y_1, y_2 \in \text{Domain}(m _ m')$ with $y_1 \neq y_2$ and $\text{ind}(y_1, y_2, m _ m')$ there exist $(y_1, d_1), (y_2, d_2) \in m _ m_d$ with $d_1 \neq d_2$.
3. If $y \in m - \text{Domain}(m _ m')$, then there exists $(y, d) \in m _ m_d$ and $\{y' \mid (y', d) \in m _ m_d\} = \{y\}$.
4. $|m_d| = \text{diffCard}(m, m _ m')$, where $\text{diffCard}(m, m _ m') =_{\text{df}} \max\{|c| : c \in \Pi \cup \{\emptyset\}, |c| \neq 1\} + |m - \text{Domain}(m _ m')|$ and Π is a partitioning of $\text{Domain}(m _ m')$ by $\text{ind}(\cdot, \cdot, m _ m')$. If $m _ m'$ is total, $\text{diffCard}(m, m _ m') = \max\{|c| : c \in \Pi \cup \{\emptyset\}, |c| \neq 1\}$. ■

Condition (2) ensures that the instances of m that are indistinguishable in $m _ m'$ become distinguishable in $m _ m_d$. Condition (3) requires each instance of m that does not participate in $m _ m'$ to have a counterpart in m_d that is not connected to any other instance of m . It ensures that Diff picks up the instances of m that get lost upon extraction. Condition (4) makes the result of Diff minimal.

We illustrate the operator Diff using the following examples.

Example 4.2.18. Consider again the model m and mapping $m _ m'$ of Fig. 4.4. Now, to verify the result, we do not need to construct an auxiliary model obtained by Extract as we did in Fig. 4.8. We use Theorem 4.2.5 instead and depict the result of applying the operator Diff directly in Fig. 4.9. $m _ m_d$ is a surjective function so that condition (1) holds. Instances y_2 and y_3 are indistinguishable under $m _ m'$, therefore y_2 and y_3 are connected to two distinct instances of m_d to satisfy condition (2). Only one instance, y_7 , of m is unconnected in $m _ m'$, and according to condition (3) does have a unique image in m_d . The partitioning Π of $\text{Domain}(m _ m')$ is $\Pi = \{\{y_1\}, \{y_2, y_3\}, \{y_4\}, \{y_5\}, \{y_6\}, \{y_7\}\}$. The largest equivalence class of Π , $\{y_2, y_3\}$, has cardinality 2, while $|m - \text{Domain}(m _ m')| = |\{y_7\}| = 1$. Thus, $\text{diffCard}(m, m _ m') = 2 + 1 = 3 = |m_d|$. ■

Example 4.2.19. Fig. 4.10 illustrates that there may be multiple ways of associating the instances of m with those of m_d . ■

Example 4.2.20. Let

$$\begin{aligned} m &= \langle R(A), S(B) \rangle, \\ m' &= \langle T(B) \rangle, \\ m _ m' &= \langle S=T \rangle. \end{aligned}$$

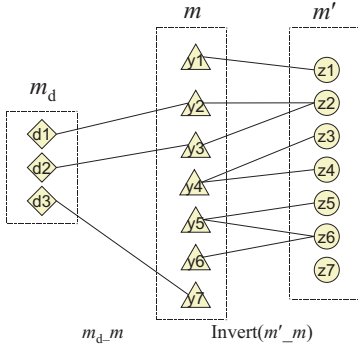


Fig. 4.9. Example of Diff result by Theorem 4.2.5

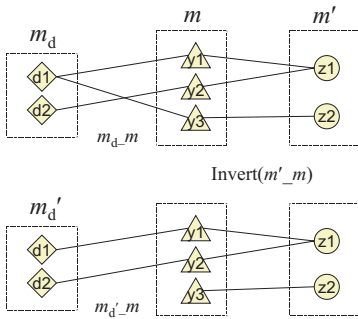


Fig. 4.10. The output mapping in Diff is not determined up to isomorphism

Then,

$$m_d = \langle\langle U(A) \rangle\rangle,$$

$$m_m_d = \langle\langle U=R \rangle\rangle$$

is a valid result of Diff.

Proof: Instances of m_d are obtained by projection, so m_m_d is a surjective function and thus condition (1) holds. Notice that $ind(y_1, y_2, m_m')$ iff $y_1.S = y_2.S$. Assume that $y_1, y_2 \in m$, $y_1 \neq y_2$, and $ind(y_1, y_2, m_m')$. That is, $y_1.S = y_2.S$ and $y_1 \neq y_2$, so y_1, y_2 must differ on R , i.e., $y_1.R \neq y_2.R$. But then, $d_1 = y_1.R \neq y_2.R = d_2$, so that condition (2) holds. All instances of m participate in m_m' , so that condition (3) is satisfied trivially. Finally, notice that m_m' is a total function. By Theorem 4.2.5, $diffCard(m, m_m') = \max\{|c| : c \in \Pi \cup \{\emptyset\}, |c| \neq 1\} = \max_{z \in m'} |\{y : y.S = z.T\}|$. Since for each $z \in m' : |\{y \mid y.S = z.T\}| = |\langle\langle R(A) \rangle\rangle|$, so $diffCard(m, m_m') = |\langle\langle R(A) \rangle\rangle|$. Hence, m_d is minimal since $|m_d| = |\langle\langle R(A) \rangle\rangle|$, and condition (4) holds.

To reiterate the intuition behind Diff, $m_d = \langle\langle U(A) \rangle\rangle$ is a minimal schema that allows us to reconstruct uniquely an instance $y \in \langle\langle R(A), S(B) \rangle\rangle$ from two instances $x \in \langle\langle T(B) \rangle\rangle$ and $d \in \langle\langle U(A) \rangle\rangle$ that were previously obtained from y using the mappings m_m_x and m_m_d produced by the operators Extract and Diff. ■

Example 4.2.21. Let

$$\begin{aligned} m &= \langle\langle R[A,B] \rangle\rangle, \\ m' &= \langle\langle T[A] \rangle\rangle, \\ m_m' &= \langle\langle \text{SELECT A FROM R} \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m_d &= \langle\langle U[B] \rangle\rangle, \\ m_m_d &= \langle\langle \text{SELECT B FROM R} \rangle\rangle \end{aligned}$$

is a valid result of Diff. (Recall that we use the square brackets to indicate multiset semantics for relational tables).

Proof: Instances of m_d are obtained using a SQL query, so m_m_d is a surjective function and condition (1) is satisfied. Under multiset semantics, SELECT A FROM R returns the same number of tuples as the number of tuples in R. Thus, $ind(y_1, y_2, m_m')$ iff y_1 and y_2 agree on the ordered list of A values. Assume that $y_1, y_2 \in m$, $y_1 \neq y_2$, and $ind(y_1, y_2, m_m')$. Although y_1 and y_2 agree on A values, we have $y_1 \neq y_2$, so they must differ on the ordered list of B values. Hence, d_1 obtained using SELECT B FROM $y_1.R$ differs from d_2 obtained using SELECT B FROM $y_2.R$, and condition (2) holds. All instances of m participate in m_m' , so that condition (3) is satisfied trivially.

m_m' is a total function. Hence, by Theorem 4.2.5, $\text{diffCard}(m, m_m') = \max_{z \in m} |\{y : (y, z) \in m_m'\}|$. We have to show that m_d is equipotent with $\{y : (y, z_{max}) \in m_m'\}$, a maximal equivalence class induced by $ind(., ., .)$. We get such maximal class when $z_{max} \in \langle\langle T[A] \rangle\rangle$ is an infinite ordered list of A values. $\text{diffCard}(m, m_m')$ is the number of instances of $\langle\langle R[A,B] \rangle\rangle$ that agree on A with z_{max} . There are $|\langle\langle R[B] \rangle\rangle|$ such instances (since there are finitely many finite lists of B values, there is a bijection from the set of infinite lists to the set of infinite and finite lists). Hence, m_d is minimal because $|m_d| = |\langle\langle R[B] \rangle\rangle|$, and condition (4) holds. ■

Example 4.2.22. Let

$$\begin{aligned} m &= \langle\langle R(\underline{ID}, A, B) \rangle\rangle, \\ m' &= \langle\langle S(\underline{ID}, A) \rangle\rangle, \\ m_m' &= \langle\langle S = \pi_{\underline{ID}, A}(R) \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m_d &= \langle\langle T(\underline{ID}, B) \rangle\rangle, \\ m_m_d &= \langle\langle T = \pi_{\underline{ID}, B}(R) \rangle\rangle \end{aligned}$$

is an invalid result of Diff. There is a “smaller” schema m'_d that satisfies the definition of Diff. We construct m'_d in the proof.

Proof: m_d and m_m_d satisfy conditions (1), (2), (3), but not (4). m_m_d is a surjective function onto m_d since instances of m_d are obtained by projection, so condition (1) holds. Notice that $ind(y_1, y_2, m_m')$ holds iff

$\pi_{ID,A}(y_1.R) = \pi_{ID,A}(y_2.R)$. Let $y_1, y_2 \in m$ be given with $y_1 \neq y_2$ and $ind(y_1, y_2, m_m')$. That is, y_1, y_2 agree on the projection of ID and A values. Since nevertheless $y_1 \neq y_2$, then y_1 and y_2 must differ on B values. Since ID is a primary key, $\pi_{ID,B}$ extracts all values of column B, including the ones that differ. Hence, $d_1 = \pi_{ID,B}(y_1.R) \neq \pi_{ID,B}(y_2.R) = d_2$ and condition (2) holds. Since m_m' is total, condition (3) is satisfied trivially.

However, the minimality condition (4) is not satisfied. Notice that m_m' is a total function. By Theorem 4.2.5, $\text{diffCard}(m, m_m') = \max_{z \in m'} |\{y : (y, z) \in m_m'\}|$. We obtain the maximal equivalence class c_{max} of $ind(., ., .)$ when $z = z_{max}$ has $|ID|$ tuples, i.e., all ID values are used in z_{max} . We can construct all instances y with $(y, z_{max}) \in m_m'$ by adding to z_{max} a column with arbitrary B values. There are $|B|^{|ID|}$ such columns. That is, m_d must have $k = |B|^{|ID|}$ instances to satisfy (3). $\ll T(\underline{ID}, B) \gg$ has however 2^k instances, since the length of table T may vary between 0 and $|ID|$.

To obtain a correct result m'_d and $m_m'_d$, we add to m_d the constraint that the number of tuples in T equals the domain size of ID. We get: $m'_d = \ll T(\underline{ID}, B), |T| = |ID| \gg$, $m_m'_d = \ll T = \pi_{ID,B}(R) \cup \{(id, b_{\text{fixed}}) \mid id \in ID, id \notin \pi_{ID}(R)\} \gg$, i.e., we extend each $\pi_{ID,B}(y.R)$ to have $|ID|$ tuples by assigning the value $b_{\text{fixed}} \in B$ to each unused ID value that is not already contained in $\pi_{ID}(y.R)$. ■

Example 4.2.23. Consider the same setting as in Example 4.2.21 but using set semantics. Let

$$\begin{aligned} m &= \ll R(A, B) \gg, \\ m' &= \ll T(A) \gg, \\ m_m' &= \ll \text{SELECTD A FROM R} \gg. \end{aligned}$$

Then,

$$\begin{aligned} m_d &= \ll U(B) \gg, \\ m_m_d &= \ll \text{SELECTD B FROM R} \gg \end{aligned}$$

is an invalid result of Diff.

Proof: m_m_d violates condition (2). As a counterexample, consider $y_1 = \{(1, 2), (3, 4)\}$ and $y_2 = \{(1, 4), (3, 2)\}$. Obviously, $y_1 \neq y_2$. However, $ind(y_1, y_2, m_m')$ holds, since $\pi_A(y_1.R) = \pi_A(y_2.R) = \{1, 3\}$. d_1 and d_2 with $(y_1, d_1), (y_2, d_2) \in m_m_d$ are determined as $d_1 = \pi_B(y_1.R) = \{2, 4\}$, $d_2 = \pi_B(y_2.R) = \{2, 4\}$. That is, $d_1 = d_2$ and condition (2) does not hold. In other words, we cannot reconstruct uniquely an instance $y \in m$ from the instances $\{1, 3\}$ and $\{2, 4\}$.

If $\ll U(B) \gg$ is not a valid output, how else can we then describe the result of Diff in this example? To do this, we find a maximal equivalence class c_{max} of the disjoint decomposition Π induced by $ind(., ., .)$, just as in Example 4.2.22. c_{max} is a set of ordered lists of B values whose length lies between $|A|$ and $|A| \cdot |B|$. Thus, a schema $m'_d = \ll T[B]; |A| \leq |T| \leq |A| \cdot |B| \gg$ could

be a valid schema produced by Diff. The corresponding mapping $m_m'_d$ is however difficult to describe using a closed formula. We know that it exists due to Theorem 4.2.5. Note that schema $\langle\langle U(A, B) \rangle\rangle$ with a bijection $\langle\langle U = R \rangle\rangle$ satisfies (1), (2), and (3), but cannot be a valid result, since $\langle\langle U(A, B) \rangle\rangle$ contains too many instances and violates condition (4). ■

Example 4.2.24. This example illustrates Diff when m_m' contains a join. It uses the setting of Example 4.2.10. Let

$$\begin{aligned} m &= \langle\langle R(A, B), S(B, C) \rangle\rangle, \\ m' &= \langle\langle T(A, B, C) \rangle\rangle, \\ m_m' &= \langle\langle T = R \bowtie S \rangle\rangle. \end{aligned}$$

Then,

$$\begin{aligned} m_d &= \langle\langle P(A, B), Q(B, C); \pi_B(P) \cap \pi_B(Q) = \emptyset \rangle\rangle, \\ m_m_d &= \langle\langle \text{CREATE VIEW P(A,B) AS} \\ &\quad \text{SELECTD * FROM R WHERE} \\ &\quad \text{B NOT IN (SELECT B FROM S),} \\ &\quad \text{CREATE VIEW Q(B,C) AS} \\ &\quad \text{SELECTD * FROM S WHERE} \\ &\quad \text{B NOT IN (SELECT B FROM R)} \rangle\rangle \end{aligned}$$

is a valid result of Diff.

Proof: Mapping m_m_d is defined using a create-view statement, so condition (1) is satisfied. Two different instances y_1, y_2 of $\langle\langle R(A, B), S(B, C) \rangle\rangle$ map to the same instance z of $\langle\langle T(A, B, C) \rangle\rangle$, only if y_1 and y_2 agree on the subset of joining B-values and the tuples of R and S that contain these B-values. Instances y_1 and y_2 may differ only on those R and S tuples that contain non-joining B-values. These tuples are exactly those extracted in the above create-view statement, so that condition (2) holds. m_m' is a view on m , so that $m = \text{Domain}(m_m')$ and condition (3) is satisfied. m_m' is total. By Theorem 4.2.5, $\text{diffCard}(m, m_m') = \max_{z \in m'} |\{y : (y, z) \in m_m'\}|$. In other words, $\text{diffCard}(m, m_m')$ is the maximal number of instances $y \in \langle\langle R(A, B), S(B, C) \rangle\rangle$ that map to the same fixed instance z of $\langle\langle T(A, B, C) \rangle\rangle$. This largest set is the set of all databases y in which the join condition is not satisfied for any tuple, i.e., $\pi_B(R) \cap \pi_B(S) = \emptyset$. All such databases map to $z_{\max} = \{\emptyset\}$. Thus, m_d is minimal. ■

In the above examples we have seen that the definition of Diff is hard to satisfy due to the minimality condition (4), which makes seemingly correct results invalid. We consider this problem in more detail in Sect. 4.3. In the rest of this section, we prove a few important theorems used in the analysis of the model-management scenarios that we present.

Theorem 4.2.6. *If $\langle\langle m_d, m_m_d \rangle\rangle = \text{Diff}(m, m_m')$ and $\text{Invert}(m_m')$ is a surjective function, then $m_d = \emptyset$ and $m_m_d = \emptyset$.* ■

The above theorem states that in the case when all of m participates in m_m' the difference m_d is “zero”; by Proposition 4.2.6, *Extract* would have to pick up all information of m .

Theorem 4.2.7. *Let $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$, $\langle m_d, m_m_d \rangle = \text{Diff}(m, m_m')$ and $m_x_m_d = \text{Invert}(m_m_x) \circ m_m_d$. Then, $\text{Invert}(m_m_d) \circ m_m' = \text{Invert}(m_x_m_d) \circ \text{Invert}(m_m_x) \circ m_m'$.*

Proof: By Proposition 4.2.2, $\text{Invert}(m_x_m_d) = \text{Invert}(m_m_d) \circ m_m_x$. Thus, the right expression of the equality to prove becomes $\text{Invert}(m_x_m_d) \circ \text{Invert}(m_m_x) \circ m_m' = \text{Invert}(m_m_d) \circ m_m_x \circ \text{Invert}(m_m_x) \circ m_m'$. By Definition 4.2.3, $m_m_x \circ \text{Invert}(m_m_x) \circ m_m' = m_m'$. Therefore, $\text{Invert}(m_x_m_d) \circ \text{Invert}(m_m_x) \circ m_m' = \text{Invert}(m_m_d) \circ m_m'$. ■

4.2.6 Confluence Operator

Let $map_1 \subseteq m_1 \times m_2$ and $map_2 \subseteq m_1 \times m_2$ be two mappings between models m_1 and m_2 . map_1 and map_2 could be partial mappings developed independently by two engineers, or could have been obtained by composition as e.g. $map_1 = m_{1_m_a} \circ m_{a_m_2}$, $map_2 = m_{1_m_b} \circ m_{b_m_2}$. The Confluence operator, \oplus , “unifies” the two mappings:

Definition 4.2.6 (Confluence, \oplus).

$$map_1 \oplus map_2 =_{\text{df}} (map_1 \cap map_2) \cup \{(x, y) \in map_1 \mid x \notin \text{Domain}(map_2) \wedge y \notin \text{Range}(map_2)\} \cup \{(x, y) \in map_2 \mid x \notin \text{Domain}(map_1) \wedge y \notin \text{Range}(map_1)\}$$

The Confluence operator extracts the “submapping” on which map_1 and map_2 agree and adds to it the correspondences between all those instances of m_1 and m_2 that participate either only in map_1 or only in map_2 . Obviously, confluence is commutative.

Theorem 4.2.8. *If $\text{Range}(map_1) \subseteq \text{Range}(map_2)$, then $map_1 \oplus map_2 = map_1 \cap map_2$.*

Proof: follows from Definition 4.2.6. ■

Corollary 4.2.2. *If map_1 and map_2 are both total or both surjective, then $map_1 \oplus map_2 = map_1 \cap map_2$.* ■

Example 4.2.25. Let

$$\begin{aligned} m_1 &= \langle\langle R(A, B), S(B, C) \rangle\rangle, \\ m_2 &= \langle\langle T(A, B, C) \rangle\rangle, \\ map_1 &= \langle\langle T = R \bowtie S \rangle\rangle, \\ map_2 &= \langle\langle \pi_A(R) = \pi_A(T) \rangle\rangle. \end{aligned}$$

Mappings map_1 and map_2 are both total, therefore, by Corollary 4.2.2, $map_1 \oplus map_2 = map_1 \cap map_2$. Hence,

$$map_1 \oplus map_2 = \ll T = R \bowtie S \text{ and } \pi_A(R) = \pi_A(T) \gg.$$

Notice that $map_1 \oplus map_2$ is not total. ■

Theorem 4.2.9. *If $Domain(map_1) \cap Domain(map_2) = Range(map_1) \cap Range(map_2) = \emptyset$, or equivalently, $map_1 \circ Invert(map_2) = Invert(map_1) \circ map_2 = \emptyset$, then $map_1 \oplus map_2 = map_1 \cup map_2$.*

Proof: follows from Definition 4.2.6. ■

Theorem 4.2.10. *If $Invert(map_1)$ is injective or $Invert(map_2) \circ map_3 = \emptyset$, then $map_1 \circ (map_2 \oplus map_3) = (map_1 \circ map_2) \oplus (map_1 \circ map_3)$.* ■

In general, however, the distributive law does not hold. To see that choose $map_1 = \{(x, y_1), (x, y_2)\}$, $map_2 = \{(y_1, z_1), (y_2, z_2)\}$, $map_3 = \{(y_2, z_2)\}$. Then, $map_1 \circ (map_2 \oplus map_3) = \{(x, y_1), (x, y_2)\} \circ \{(y_1, z_1), (y_2, z_2)\} = \{(x, z_1), (x, z_2)\}$. In contrast, $(map_1 \circ map_2) \oplus (map_1 \circ map_3) = \{(x, z_1), (x, z_2)\} \oplus \{(x, z_2)\} = \{(x, z_2)\}$.

The following theorem illustrates an important use case of the Confluence operator.

Theorem 4.2.11 (Mirror Merge). *Let $\langle m, m_{_}m_1, m_{_}m_2 \rangle = Merge(m_1, m_2, m_{_}m_2)$, and $m_{_}n_1$ and $m_{_}n_2$ be bijective mappings. Then, $\langle n, n_{_}n_1, n_{_}n_2 \rangle = Merge(n_1, n_2, Invert(m_{_}n_1) \circ m_{_}m_2 \circ m_{_}n_2)$ such that $m_{_}n = (m_{_}m_1 \circ m_{_}n_1 \circ Invert(n_{_}n_1)) \oplus (m_{_}m_2 \circ m_{_}n_2 \circ Invert(n_{_}n_2))$ is a bijection. Furthermore, $n_{_}n_1 = Invert(m_{_}n) \circ m_{_}m_1 \circ m_{_}n_1$ and $n_{_}n_2 = Invert(m_{_}n) \circ m_{_}m_2 \circ m_{_}n_2$.*

Proof: Models m and n are equipotent since they are obtained from isomorphic pairs of models. The non-trivial part of the theorem is the construction of the bijection $m_{_}n$ using Confluence. We sketch the proof in Fig. 4.11. The key observation is that either the “upper” path over $m_{_}m_1 \circ m_{_}n_1 \circ Invert(n_{_}n_1)$ or the “lower” path over $m_{_}m_2 \circ m_{_}n_2 \circ Invert(n_{_}n_2)$ allows obtaining a unique image $y \in n$ for each instance $x \in m$. For example, although the relationship between the instances x_1, x_2 and y_1, y_2 in the figure is ambiguous in the lower path, we can restore it by following the upper path. ■

4.2.7 Match Operator

Match returns a mapping $m_{_}m_2$ that describes how the instances of m_1 and m_2 relate to each other. Often, we can find infinitely many semantically valid mappings between two models. Each of these mappings makes sense in a specific application context. Therefore, we do not put any restrictions on the result of Match other than it is a mapping between m_1 and m_2 :

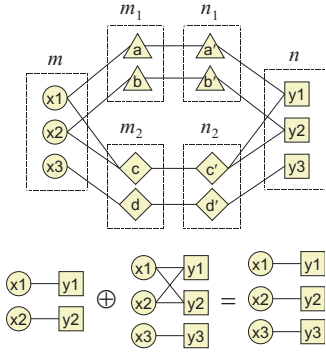


Fig. 4.11. Illustration of Theorem 4.2.11 (Mirror Merge)

Definition 4.2.7 (Match). $m_1 _ m_2 = Match(m_1, m_2)$ holds if and only if $m_1 _ m_2 \subseteq m_1 \times m_2$. ■

Computing the result of **Match** requires human input, so that it can be considered a blocking, black-box operator.

4.3 Materialization

To make the formalization presented in this chapter useful for real applications, the results of model-management scripts need to be computed effectively. As we explained in Sect. 4.1.4, computing the results of a script amounts to finding a variable substitution that satisfies the script. We showed using examples that it is often impossible to find the desired variable substitution due to the limited expressiveness of concrete schema and mapping languages. In this section, we explain how we can extend the scripts in a controlled fashion so that computing the results and deploying them in applications becomes possible. We refer to this problem as *materialization*.

The intuition that we exploit is that it is acceptable to generate more expressive models and mappings as long as we can reconstruct the exact results if necessary. First, we discuss materialization of models using an example.

Example 4.3.1. Imagine that a model m produced by a script can be exactly specified as

$$m = \langle R(A, B), S(B, C); \pi_B(R) = \pi_B(S) \rangle$$

using the classical relational model and the constraint expressed as a relational algebra expression. We argue that models

$$\begin{aligned} &\langle R[A, B], S[B, C] \rangle, \\ &\langle R[A, B], S[B, C] \rangle, \\ &\langle T[A, B, C] \rangle, \end{aligned}$$

and some other more expressive models can be used as an adequate “approximation” of m in SQL DDL. Each of these models dominates m , i.e., there is a total surjective function from m' onto m . For example, for $m' = \llcorner T[A, B, C] \gg$ the function m'_m can be specified as

```

m'_m = <<CREATE VIEW R(A,B) AS
        SELECT DISTINCT A,B FROM T
        WHERE A NOT NULL AND B NOT NULL
CREATE VIEW S(B,C) AS
        SELECT DISTINCT B,C FROM T
        WHERE B NOT NULL AND C NOT NULL>>

```

We can round-trip each instance of m to m' and back without information loss. For example, we can generate an instance of m' from an instance of m using the total injective function

```

map = <<CREATE VIEW T(A,B,C) AS
        SELECT A, B, NULL FROM R UNION
        SELECT NULL, B, C FROM S>>

```

It is easy to see that $map \circ m'_m = \text{ld}(m)$. ■

To materialize m as m' we extend the script that defines m by adding to it the following conditions:

```

Invert(m'_m) o m'_m = ld(m); // m'_m is surjective onto m
m' = Domain(m'_m);

```

The above constraints on m' are quite weak, i.e., there are substantial degrees of freedom in computing m' . As we demonstrate below, materializing the mappings in which m participates places additional constraints on m' .

If a model has been obtained from a relational schema in the script, then casting it into a relational schema may be a reasonable default assumption (in fact, in Rondo the result of merge is assumed to be a model of the same type as the input models). Alternatively, the target meta-model could be specified explicitly by declaring the “type” of the model variable m' as say SQL DDL. We expect that the tuning knobs and implicit policies of the model-management environment, such as syntactic minimality requirements on schemas or efficiency (Cosmadakis and Papadimitriou 1984; Spaccapietra and Parent 1994), can be deployed to drive materialization.

Next, we discuss materialization of mappings. Consider the setting of Fig. 4.12(a). Assume that model m_1 is expressed using a constant, e.g., m_1 is a fixed relational schema, while model m_2 and mapping map are defined using a script, say as results of change propagation. Our goal is to materialize m_2 and map . For m_2 we can proceed as above: we materialize m_2 as a more expressive model m'_2 , as witnessed by a total surjective function m'_2_m2 . Now, mapping map needs to be materialized as a mapping $m_1_m'_2$ between m_1 and m'_2 . To be able to reconstruct the original mapping map , we require

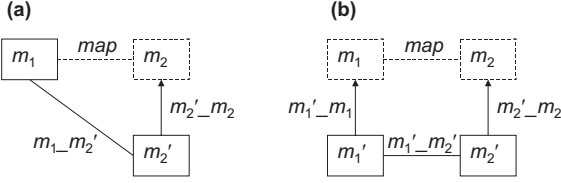


Fig. 4.12. Materialization of models and mappings

that $map = m_1-m_2' \circ m_2'-m_2$. That is, m and map are materialized as m' and m_1-m_2' such that

$$\begin{aligned} \text{Invert}(m_2'-m_2) \circ m_2'-m_2 &= \text{Id}(m_2); \\ m_2' &= \text{Domain}(m_2'-m_2); \\ map &= m_1-m_2' \circ m_2'-m_2; \end{aligned}$$

To illustrate, consider Example 4.2.22. There, we showed that a quite intuitive result of Diff was invalid because it violated the minimality condition. For convenience, we restate the example: let $m = \langle\langle R(\underline{\text{ID}}, A, B) \rangle\rangle$, $m' = \langle\langle S(\underline{\text{ID}}, A) \rangle\rangle$, and $m'-m = \langle\langle S = \pi_{\text{ID},A}(\text{R}) \rangle\rangle$. Then, $m_d = \langle\langle T(\underline{\text{ID}}, B) \rangle\rangle$, $m_d-m = \langle\langle T = \pi_{\text{ID},B}(\text{R}) \rangle\rangle$ is an invalid result of Diff. However, we can demonstrate that this result is a valid materialization of the exact result of Diff.

The materialization constraints suggested above offer substantial degrees of freedom for choosing m_1-m_2' , but they do not guarantee the existence of m_1-m_2' (nor that of m_2'). For example, if map is not a function but our target mapping language is a view definition language, we cannot materialize map as a view m_1-m_2' , since we know that a composition of two functions must yield a function.

Just as in the case of schemas, various tuning knobs can be used to steer materialization of mappings. One metric is view minimization (Ullman 1997). Efficiency is another example: the work in (Shanmugasundaram et al. 2001b; Bohannon et al. 2002; Fernandez et al. 2002; Fan et al. 2003) presents various algorithms that make the mappings between relational and XML data more efficient, while Theodoratos et al. (2001) consider various metrics for selecting views in data warehousing, such as query evaluation cost, view maintenance cost, or storage space.

As another example of materialization, consider Fig. 4.12(b). Here, m_1 is not a constant but needs to be computed as well. We materialize m_1 , m_2 , and map respectively as m_1' , m_2' , and $m_1'-m_2'$ such that:

$$\begin{aligned} \text{Invert}(m_1'-m_1) \circ m_1'-m_1 &= \text{Id}(m_1); \\ m_1' &= \text{Domain}(m_1'-m_1); \\ \text{Invert}(m_2'-m_2) \circ m_2'-m_2 &= \text{Id}(m_2); \\ m_2' &= \text{Domain}(m_2'-m_2); \\ map &= \text{Invert}(m_1'-m_1) \circ m_1-m_2' \circ m_2'-m_2; \end{aligned}$$

In a general case, we are given a set of model and mapping variables with constraints established by a script. We suggest that a valid materialization

should allow us to reconstruct the exact variable substitutions using a set of total surjective functions, one for each model, which can be composed with the materialized mappings to obtain the exact mappings. It is sufficient to ensure the existence of such total surjective functions; they may not be expressible in concrete mapping languages.

In an indirect way, materialization lifts constraints on models and mappings. Since the presence of such constraints, e.g., of integrity constraints, may be essential for applications, constraint lifting should be done in scripts explicitly, as we suggested in the above discussion. The dropped constraints have to be maintained by applications to ensure that the materialized schemas keep only those database instances that would be representable in the exact schemas. That is, in Example 4.3.1, the application would need to maintain the constraint $\pi_B(R) = \pi_B(S)$ for the materialization $\langle\langle R[A, B], S[B, C] \rangle\rangle$, or the constraint: if A is NULL then B and C NOT NULL; if C is NULL then A and B NOT NULL, for the materialization $\langle\langle T[A, B, C] \rangle\rangle$. An approach to automating application-based constraint management is discussed in (Peckham et al. 1995).

We expect that there may be more than one approach to materializing schemas and mappings. For instance, in (Fagin et al. 2003) the authors examine the problem of data exchange, which can be viewed as a problem of materializing a non-functional mapping between two models as a view, i.e., as a function. In this case, the exact mapping could be reconstructed using a different kind of composition, which allows obtaining the so-called certain answers for queries. In general, materialization can be stated as a constraint satisfaction problem, as exemplified in Sect. 10.4.

Materialization of schemas and mappings is associated with information loss, since the total surjective functions utilized for materialization may themselves not be expressible in concrete mapping languages. Therefore, materialization should only be done for models and mappings that need to be deployed by applications, but not for the intermediate results of scripts. As a matter of fact, in (Buneman et al. 1992) the authors found that materializing the intermediate merge results leads to information loss due to the limited expressiveness of the schema language they used, so that merging becomes a non-associative operation. To fix the problem, they introduce a more expressive auxiliary language and materialize the schemas only at the very last step.

In general, it may be beneficial to preserve the original models and mappings, and the scripts used to obtain the intermediate materialized result. In this way, the exact results of scripts are available for later use in other scripts. In addition, keeping the original inputs and scripts facilitates migration to new standards and data management systems. For example, should we at some point of time decide to use a more expressive language for our schemas, e.g., XML Schema instead of SQL DDL, and migrate our data to a new DBMS, we may be able to compute “tighter” models for our results, i.e., the schemas may be materialized more accurately, with extra integrity constraints that were not expressible earlier.

5. Change Propagation Scenario

“Nothing endures but change.”

– Heraclitus (540-480 BC)

In this chapter, we revisit the change propagation scenario. We present a solution for this scenario using the operators of Chap. 4. We argue the correctness of our solution by examining several special cases and by showing that the scripts that we developed match the intuition in these special cases. We cannot formally prove that our solution is correct. To do so, we would need a formal specification of what change propagation means. Instead, we argue that the script that we present provides a major part of such specification in first place. This specification can be used to drive and verify implementations for concrete schema and mapping languages. In fact, in Chap. 6 we examine to what extent the implementation presented in Chap. 2 conforms to the specification of change propagation presented below.

A general outline of the change propagation scenario is the following (see Sect. 2.1 for a more detailed discussion). Assume that we are given models s_1 and d_1 , and a mapping s_1-d_1 between them. Now, s_1 changes into s_2 . The changes are described by the mapping s_1-s_2 , which may have been obtained by matching s_1 and s_2 . We want these changes to be propagated to d_1 , i.e., we look for a model d_2 and a new mapping s_2-d_2 that describes how the new model d_2 relates to s_2 .

We begin with a simple variation of the change propagation scenario and work our way towards a general case. First, we consider additions only in Sect. 5.1. Then, we consider deletions in Sect. 5.2, present a general solution in Sect. 5.3 and examine schema evolution as a special case of change propagation in Sect. 5.4. We conclude the chapter and discuss several other possible variations of the change propagation scenario in Sect. 5.5.

Before we dive into the scenario, one note is due. What we call “addition” corresponds to an abstract model modification that extends the set of possible instances, i.e., it adds information capacity to the model. Addition may be caused by adding new elements to the model or by dropping certain existing constraints. Similarly, “deletion” is another abstract manipulation that

reduces the set of possible instances, i.e., produces a less expressive model. Deletion may be caused by removing certain model elements or by adding constraints to the model.

5.1 Propagating Additions

Consider the schematic representation in Fig. 5.1. We continue using the convention that the names of the mapping variables identify the left and right models participating in the mappings. Thus, $s_{1_s2} \subseteq s_1 \times s_2$.

Assume that s_1 is transformed into a more expressive model s_2 , such that $\text{Invert}(s_{1_s2})$ is a view on s_2 . The direction of the arrow labeled s_{1_s2} in the figure indicates that the instances of s_1 are functionally determined by the instances of s_2 . To propagate additions from s_1 to d_1 means to construct a model d_2 that can express all information of d_1 plus all extra information of s_2 . The extra information of s_2 , which is not captured by s_1 , can be obtained using the operator Diff . The resulting model is then merged with the model d_1 . This approach is described in the following script:

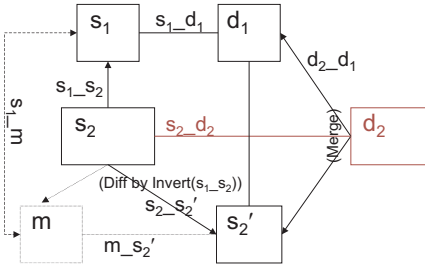


Fig. 5.1. Propagating additions

1. $\langle s'_2, s_{2_s'_2} \rangle = \text{Diff}(s_2, \text{Invert}(s_{1_s2}))$;
2. $d_{1_s'_2} = \text{Invert}(s_{1_d1}) \circ s_{1_s2} \circ s_{2_s'_2}$;
3. $\langle d_2, d_{2_d1}, d_{2_s'_2} \rangle = \text{Merge}(d_1, s'_2, d_{1_s'_2})$;
4. $s_{2_d2} = (\text{Invert}(s_{1_s2}) \circ s_{1_d1} \circ \text{Invert}(d_{2_d1})) \oplus (s_{2_s'_2} \circ \text{Invert}(d_{2_s'_2}))$;

To argue the correctness of the above solution, we consider the following special case:

Proposition 5.1.1. *If $\text{Invert}(s_{1_s2})$ is a surjective function and s_{1_d1} is a bijection, then s_{2_d2} is a bijection too. That is, if s_1 and d_1 are equipotent and we add a certain amount of information to s_1 , then our script ensures that the same amount of information is added to d_1 to obtain d_2 .*

Proof: The idea of the proof is to show that merging s'_2 and d_1 produces a result equivalent to merging of s'_2 and the model m obtained by extraction

from s_2 . The auxiliary mappings and model m that are used in the proof are represented using dotted lines and rectangles in Fig. 5.1 and are highlighted below in bold; all other variables are defined in the script or are input variables.

To construct the alternative merge, we proceed as follows. Let $\langle \mathbf{m}, \mathbf{s}_2\text{-}\mathbf{m} \rangle = \text{Extract}(s_2, \text{Invert}(s_1\text{-}s_2))$ and $\mathbf{m}\text{-}\mathbf{s}'_2 = \text{Invert}(s_2\text{-}m) \circ s_2\text{-}s'_2$. By Corollary 4.2.1 we get $\langle s_2, s_2\text{-}m, s_2\text{-}s'_2 \rangle = \text{Merge}(m, s'_2, m\text{-}s'_2)$.

Let $\mathbf{s}_1\text{-}\mathbf{m} = s_1\text{-}s_2 \circ s_2\text{-}m$. By Proposition 4.2.7, $s_1\text{-}m$ is a bijection. Therefore, $\mathbf{m}\text{-}\mathbf{d}_1 = \text{Invert}(s_1\text{-}m) \circ s_1\text{-}d_1$ is also a bijection. Thus, we have a Merge of m and s'_2 and a Merge of d_1 and s'_2 , where m and d_1 are equipotent. By Definition 4.2.3, $\text{Invert}(s_1\text{-}s_2) = s_2\text{-}m \circ \text{Invert}(s_2\text{-}m) \circ \text{Invert}(s_1\text{-}s_2) = s_2\text{-}m \circ \text{Invert}(s_1\text{-}m)$. Hence, by Proposition 4.2.2, $s_1\text{-}s_2 = s_1\text{-}m \circ \text{Invert}(s_2\text{-}m)$. Notice that $(s_1\text{-}m \circ \text{Invert}(s_2\text{-}m)) \circ s_2\text{-}s'_2 = s_1\text{-}m \circ m\text{-}s'_2$. Therefore, $s_1\text{-}s_2 \circ s_2\text{-}s'_2 = s_1\text{-}m \circ m\text{-}s'_2$, and we obtain that $d_1\text{-}s'_2 = \text{Invert}(m\text{-}d_1) \circ m\text{-}s'_2$.

Now, we are ready to apply Theorem 4.2.11, which entails that d_2 is isomorphic to s_2 with a bijection $(s_2\text{-}m \circ m\text{-}d_1 \circ d_1\text{-}d_2) \oplus (s_2\text{-}s'_2 \circ \text{Invert}(d_2\text{-}s'_2))$ between them.

The second part of the above expression is equivalent to that of the last line of the script, line 4, which defines $s_2\text{-}d_2$. To obtain the proposition, all we need to show is that $s_2\text{-}m \circ m\text{-}d_1 = \text{Invert}(s_1\text{-}s_2) \circ s_1\text{-}d_1$. We have demonstrated above that $\text{Invert}(s_1\text{-}s_2) = s_2\text{-}m \circ \text{Invert}(s_1\text{-}m)$. By composing both parts of the expression with $s_1\text{-}d_1$, we obtain $\text{Invert}(s_1\text{-}s_2) \circ s_1\text{-}d_1 = s_2\text{-}m \circ \text{Invert}(s_1\text{-}m) \circ s_1\text{-}d_1 = s_2\text{-}m \circ m\text{-}d_1$. ■

Analogously, we could show that if $s_1\text{-}d_1$ is a view on s_1 , then $s_2\text{-}d_2$ is a view on s_2 . For that, we would need an extension of Theorem 4.2.11, which uses surjective functions instead of bijections.

5.2 Propagating Deletions

Consider the scenario of Fig. 5.2. Assume that s_1 is transformed into a less expressive model s_2 , such that $s_1\text{-}s_2$ is a view on s_1 . Propagating deletion from s_1 to d_1 means that we discard all instances of d_1 that are not relevant for representing the information in s_2 . In other words, we keep the instances of d_1 that are relevant for s_2 and those that do not participate in $s_1\text{-}d_1$ in the first place, since the latter are not affected by the change. This effect can be achieved using the following script:

1. $d_1\text{-}s_2 = \text{Invert}(s_1\text{-}d_1) \circ s_1\text{-}s_2$;
2. $\langle m, d_1\text{-}m \rangle = \text{Extract}(d_1, d_1\text{-}s_2)$; // still in s_2
3. $\langle n, d_1\text{-}n \rangle = \text{Diff}(d_1, \text{Invert}(s_1\text{-}d_1))$; // to keep in d_1
4. $\langle d_2, d_2\text{-}m, d_2\text{-}n \rangle = \text{Merge}(m, n, \text{Invert}(d_1\text{-}m) \circ d_1\text{-}n)$;
5. $d_1\text{-}d_2 = (d_1\text{-}m \circ \text{Invert}(d_2\text{-}m)) \oplus (d_1\text{-}n \circ \text{Invert}(d_2\text{-}n))$;
6. $s_2\text{-}d_2 = \text{Invert}(d_1\text{-}s_2) \circ d_1\text{-}d_2$;

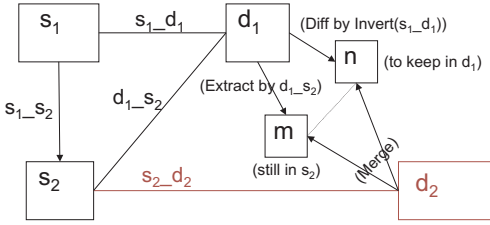


Fig. 5.2. Propagating deletions

In line 2, we extract the model m , which captures all information of s_2 “visible” through s_1 . In line 3, we determine the portion n of d_1 that does not participate in $s_1_d_1$, using the operator Diff. Model n represents the information that needs to be preserved in d_2 no matter what the mapping $s_1_s_2$ looks like.

To substantiate the correctness claim for the above solution, we examine the following two special cases.

Proposition 5.2.1. *If $s_1_s_2$ is a bijection, then $d_1_d_2$ is a bijection and $s_2_d_2$ is isomorphic to $s_1_d_1$. That is, if the information capacity of s_1 remains unchanged, so does that of d_1 .*

Proof: Since $s_1_s_2$ is a bijection, so $d_1_s_2$ is isomorphic to $\text{Invert}(s_1_d_1)$. Hence, Extract and Diff in lines 2-3 form a split (see Corollary 4.2.1). By Theorem 4.2.11, $d_1_d_2$ is a bijection. Therefore, by composition in line 6, $s_2_d_2$ is isomorphic to $s_1_d_1$. ■

Proposition 5.2.2. *If $s_1_s_2$ is a surjective function and $s_1_d_1$ is a bijection, then $s_2_d_2$ is a bijection. That is, if s_1 and d_1 are equipotent and we delete a certain amount of information from s_1 , then our script ensures that the same amount of information is deleted from d_1 to obtain d_2 .*

Proof: Since $s_1_d_1$ is a bijection, it is also a surjective function, and by Theorem 4.2.6, $n = \emptyset$ and $d_1_n = \emptyset$. Followingly, $m \cong d_2$ and the above script is equivalent to the following script (with respect to $d_2, s_2_d_2$):

$$\begin{aligned}
 d_1_s_2 &= \text{Invert}(s_1_d_1) \circ s_1_s_2; \\
 \langle d_2, d_1_d_2 \rangle &= \text{Extract}(d_1, d_1_s_2); \\
 s_2_d_2 &= \text{Invert}(d_1_s_2) \circ d_1_d_2;
 \end{aligned}$$

This script is shown schematically in Fig. 5.3.

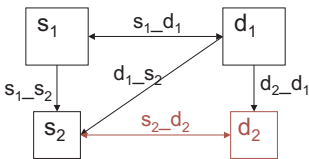


Fig. 5.3. Propagating deletions over bijection

By construction, $d_{1_s_2}$ is a surjective function. By Proposition 4.2.6, $s_{2_d_2}$ is a bijection. ■

Notice however that if $s_{1_d_1}$ is a view on s_1 then the resulting mapping $s_{2_d_2}$ need not be a view on s_2 . At first glance, this result does not seem to match our intuition. As we illustrate in the following example, the reason is that deletion is not limited to removing attributes from s_1 , but could have a variety of other causes, such as restricting the domain of an attribute using an arithmetic function.

Example 5.2.1. Let

$$\begin{aligned} s_1 &= \langle\langle R[\text{val} : \text{byte}] \rangle\rangle, \\ d_1 &= \langle\langle S[\text{val} : \text{byte}(0..128)] \rangle\rangle, \\ s_{1_d_1} &= \langle\langle \text{SELECT val} / 2 \text{ FROM } R \rangle\rangle \end{aligned}$$

Assume integer division and multiset semantics for schemas. Let the deletion be specified by the mapping

$$s_{1_s_2} = \langle\langle \text{SELECT val} / 3 \text{ FROM } R \rangle\rangle.$$

$s_{1_d_1}$ is a surjective function, so the propagation of deletion is equivalent to the 3-line script used in the proof above. The mapping $d_{1_s_2}$ obtained by composition can be specified as follows: $(y, x) \in d_{1_s_2}$ iff x and y have the same number of tuples and for each i -th tuple t_x of x and t_y of y the following condition holds: ($t_y.\text{val} = t_x.\text{val} \cdot 2/3$ or $t_y.\text{val} = (t_x.\text{val} \cdot 2 + 1)/3$). The extracted schema d_2 is obtained as $d_2 = \langle\langle T(\text{val} : \text{byte}[0..128]) \rangle\rangle$, where $d_{1_d_2}$ is a bijection. Thus, the mapping $s_{2_d_2}$ is isomorphic to $\text{Invert}(d_{1_s_2})$, i.e., can be expressed using a similar condition as above. $s_{2_d_2}$ is not a function. In fact, the instance $x = \{4\} \in s_2$ maps by way of $s_{2_d_2}$ to two instances of d_2 , $y_1 = \{2\}$ and $y_2 = \{3\}$, since $4 \cdot 2/3 = 2$ and $(4 \cdot 2 + 1)/3 = 3$. ■

Observe that the “deletion” done in the example is somewhat unorthodox. One can show that in simpler cases, in which s_2 is obtained by removing one or more attributes from s_1 , the result $s_{2_d_2}$ remains a view.

The example illustrates that certain changes of the source schema may break the view in such a way that it cannot be “repaired” fully automatically. Intervention of a human designer may be necessary to define an updated view, which otherwise becomes a non-functional transformation.

5.3 A General Solution

In a general solution for change propagation, which is depicted schematically in Fig. 5.4, we propagate the additions and deletions simultaneously. We combine the scripts of Sections 5.1 and 5.2 and obtain the following solution:

1. $d_{1_s_2} = \text{Invert}(s_{1_d_1}) \circ s_{1_s_2}$;
2. $\langle m, d_{1_m} \rangle = \text{Extract}(d_1, d_{1_s_2})$; // still in s_2

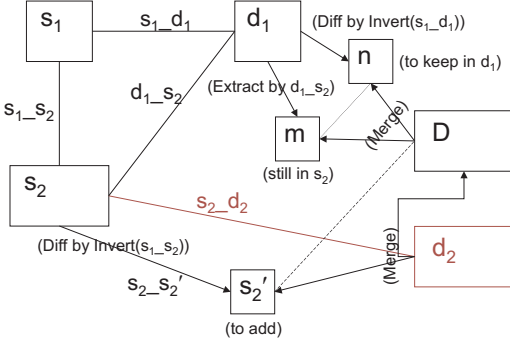


Fig. 5.4. Change propagation: a general solution

3. $\langle n, d_{1-n} \rangle = \text{Diff}(d_1, \text{Invert}(s_{1-d_1}))$; // to keep in d_1
4. $\langle D, D_m, D_n \rangle = \text{Merge}(m, n, \text{Invert}(d_{1-m}) \circ d_{1-n})$;
5. $d_{1-D} = (d_{1-m} \circ \text{Invert}(D_m)) \oplus (d_{1-n} \circ \text{Invert}(D_n))$;
6. $s_{2-D} = \text{Invert}(d_{1-s_2}) \circ d_{1-D}$;
7. $\langle s'_2, s_{2-s'_2} \rangle = \text{Diff}(s_2, \text{Invert}(s_{1-s_2}))$;
8. $D_{-s'_2} = \text{Invert}(d_{1-D}) \circ d_{1-s_2} \circ s_{2-s'_2}$;
9. $\langle d_2, d_{2-D}, d_{2-s'_2} \rangle = \text{Merge}(D, s'_2, D_{-s'_2})$;
10. $s_{2-d_2} = (\text{Invert}(d_{1-s_2}) \circ d_{1-D} \circ \text{Invert}(d_{2-D})) \oplus (s_{2-s'_2} \circ \text{Invert}(d_{2-s'_2}))$;

The lines 1-6 correspond to the deletion script of Sect. 5.2, with the only difference that d_2 is replaced by D . The remaining lines 7-10 deal with propagating additions. Line 8 can be simplified as $D_{-s'_2} = \text{Invert}(s_{2-D}) \circ s_{2-s'_2}$ by exploiting the result computed in line 6.

By construction, the above script is equivalent to either the deletion or addition script if s_{1-s_2} or $\text{Invert}(s_{1-s_2})$ is surjective function, respectively. In fact, if s_{1-s_2} is a surjective function, then $s'_2 = \emptyset$ and the script can be rewritten as the deletion script of Sect. 5.2. If $\text{Invert}(s_{1-s_2})$ is a surjective function, then $\text{Extract}(d_1, \text{Invert}(s_{1-d_1}) \circ s_{1-s_2})$ yields the same result as $\text{Extract}(d_1, \text{Invert}(s_{1-d_1}))$. Hence, m and n form a split over the wedge morphism $\text{Invert}(s_{1-d_1})$, and $d_1 \cong D$ by Corollary 4.2.1. Thus, in this case the script can be rewritten as the addition script of Sect. 5.1.

We suggest that the above script describes the intended state-based semantics of change propagation. To support this claim, we have shown that its semantics matches the intuition in the special cases discussed above.

5.4 Schema Evolution Scenario

The schema evolution problem arises when a change to a database schema breaks a view that is defined on it. The schema evolution scenario is a special case of the change propagation scenario of Sect. 5.3, when s_{1-d_1} is a total surjective function. In this case, by Theorem 4.2.6, $\text{Diff}(d_1, \text{Invert}(s_{1-d_1}))$

yields an empty model n , and the model D in Fig. 5.4 is equipotent with m . Thus, the solution of Sect. 5.3 can be simplified as illustrated in Fig. 5.5. The respective script is shown below:

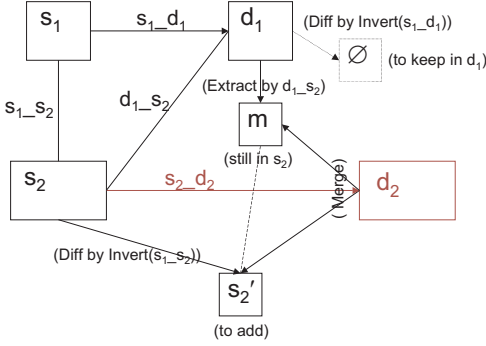


Fig. 5.5. Schema evolution: a special case of change propagation

$$\begin{aligned}
 d_1_s_2 &= \text{Invert}(s_1_d_1) \circ s_1_s_2; \\
 \langle m, d_1_m \rangle &= \text{Extract}(d_1, d_1_s_2); \\
 \langle s_2', s_2_s_2' \rangle &= \text{Diff}(s_2, \text{Invert}(s_1_s_2)); \\
 s_2'_m &= \text{Invert}(s_2_s_2') \circ \text{Invert}(d_1_s_2) \circ d_1_m; \\
 \langle d_2, d_2_s_2', d_2_m \rangle &= \text{Merge}(s_2', m, s_2'_m); \\
 s_2_d_2 &= (\text{Invert}(d_1_s_2) \circ d_1_m \circ \text{Invert}(d_2_m)) \oplus (s_2_s_2' \circ \text{Invert}(d_2_s_2'));
 \end{aligned}$$

The resulting mapping $s_2_d_2$ corresponds to the updated view definition, whereas d_2 is the updated schema. As we demonstrated in Sect. 5.2, $s_2_d_2$ may not be a function, i.e., certain changes to the database schema may make it impossible to define an updated view on the new schema without human decision-making.

In many schema evolution scenarios involving SQL views, it may be desirable to use the above script even if $s_1_d_1$ is a total function, but not surjective. In this case, model n is not empty, but can be safely ignored. This approach is justified by the fact that the SQL view definition language allows the view schema to have instances that are impossible to obtain by executing the view query over the source database. For example, consider the view defined as `CREATE VIEW V(age: int) AS SELECT S.age ≤ 20 FROM S`. This view is not surjective onto the view schema $\langle\langle V(\text{age} : \text{int}) \rangle\rangle$, since the view schema fails to capture the constraint that age values never exceed the value 20. However, due to this constraint the difference schema $n = \langle\langle V(\text{age} : \text{int}, \text{age} > 20) \rangle\rangle$ has no instances that can be obtained by executing the view definition and can be safely ignored in the result.

5.5 Variants of Change Propagation

Change propagation is a very rich scenario. In this section, we highlight several aspects of it that need to be considered in future work, such as conversion, splitting, batching, chaining, etc.

The state-based approach and the solution for change propagation that we presented above abstracts out the fact that models s_1, s_2 and d_1, d_2 may be expressed in different schema languages. For example, line 9 of the script of Sect. 5.3 may describe a merge of a relational and an XML schema. In Chap. 2, we assumed that all operators return their results expressed in the same schema language as the input schemas. Therefore, an explicit conversion step is required in Rondo before we can merge a relational schema with an XML schema. In (Bernstein 2003), a special operator called **ModelGen** was introduced to implement conversion. Ideally, conversion should return an equipotent model and a bijection between the input and the output model. In this case, conversion is a “transparent” operation in terms of state-based semantics and can be introduced at any step of a model-management script without changing its semantics. However, in many cases conversion is bound to yield a strictly more expressive model due to the limited expressiveness of the target schema language. In such cases, the state-based semantics of the script may differ depending on where the conversion step is introduced into the script. To illustrate, consider the following example.

Example 5.5.1. Consider propagation of additions illustrated in Fig. 5.1. If we assume that conversion yields equipotent models, then the solutions in Fig. 5.6 and in Fig. 5.7 are equivalent to that shown in Fig. 5.1. In Fig. 5.6, we convert s_2 to c before applying the Diff operators, just as we did in Sect. 2.1 (see Figures 2.2 and 2.3), using the expression $\langle c, s_2 _c \rangle = \text{ModelGen}(s_2)$. In Fig. 5.7, we first apply the Diff operator and then convert, $\langle c', s'_2 _c' \rangle = \text{ModelGen}(s_2)$. If the mappings $s_2 _c$ and $s'_2 _c'$ are not bijections, the semantics of all three solutions may differ from each other. ■

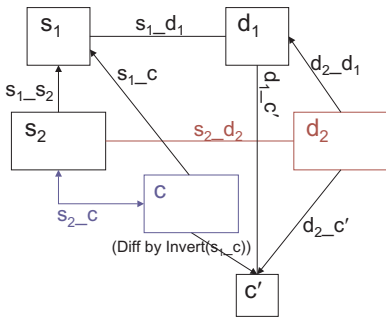


Fig. 5.6. Addition only, convert first then Diff

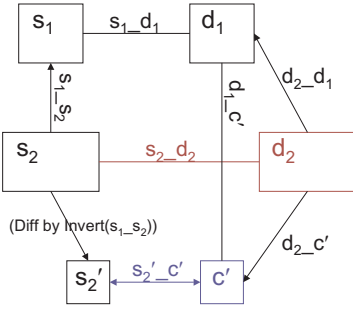


Fig. 5.7. Addition only, Diff first, then convert

The script of Sect. 5.3 presents one possible solution for change propagation. No doubt, many other scripts may achieve the same effect as the one that we developed. However, due to the script complexity, it may be quite hard to determine whether two or more alternative solutions are equivalent. For example, in Fig. 5.4 we effectively specify a 3-way Merge between the models m , n , and s'_2 . We do so by first merging m and n , and then merging the result with s'_2 . What if we first merge s'_2 and m ?

In the script of Sect. 5.3 we assumed that all information added to s_1 by means of s_2 is new and is not covered by d_1 . Thus, we simply merge D and s'_2 using the mapping $D_s'_2$ obtained by composition. However, in a most general case, D and s'_2 may have a greater overlap than that suggested by $D_s'_2$. This can happen when the information added to s_1 by way of s_2 is already contained in d_1 . To determine that, we would need to match the portions of D and s'_2 that do not participate in $D_s'_2$. These portions can be obtained using the operator Diff.

Another question of great practical importance is whether we can propagate the changes in small fragments rather than all at once, and still obtain an equivalent solution. In fact, some change management tools face the choice of either propagating each atomic modification one by one or as a set of batch operations of larger granularity. Splitting or batching the modifications may be necessary for efficiency reasons, for example, if d_1 is a model of a code base of several million lines that needs to be updated. A related question is whether chaining of change propagation operations can be simplified. The chaining takes place when the changes propagated from one model to another trigger change propagation to a third, fourth, etc. model.

We conclude this chapter by making the following remarks:

- Change propagation is a complex scenario that we are only starting to understand.
- Conversion needs to be modeled explicitly if it produces more expressive models.
- An automated theorem prover could be instrumental in analyzing the equivalence or subsumption of alternative solutions, or more elaborate change propagation scenarios.

6. State-Based Semantics in Rondo

“Although nature commences with reason and ends in experience it is necessary for us to do the opposite, that is to commence with experience and from this to proceed to investigate the reason.”

– Leonardo da Vinci (1452-1519)

In this chapter, we discuss the relationship between the structural operator definitions given in Chap. 2 and the state-based definitions presented in Chap. 4. To avoid ambiguity, we refer to the operators and scripts used in Rondo (Chap. 2) as structural and to those of Chap. 4 as state-based. The discussion that we present illustrates how the behavior of Rondo and other complex metadata management systems can be analyzed in terms of state-based semantics.

In Rondo, we use subsets of the standard schema languages such as SQL DDL or XML Schema. The state-based semantics for these languages is well-known. The definitions in Chap. 2 give a precise specification of the output schemas and morphisms returned by the structural operators. Hence, to define the state-based semantics of the structural operators, we merely have to provide a formal specification of the state-based semantics for morphisms and selectors. Once we have the state-based semantics for models, morphisms, and selectors, we effectively obtain a precise state-based semantics for the structural operators. Consequently, the state-based semantics of scripts containing a mix of state-based and structural operators becomes well-defined. We will use this observation in Sect. 6.3 to relate the structural operators to the state-based ones.

6.1 Semantics of Morphisms

Morphisms are a graphical language. The arcs that connect individual schema elements constitute the elementary expressions of this language. Although this language has been used in various tools, to our knowledge no previous

work defined its semantics precisely. Moreover, it is likely that the semantics differs from tool to tool, i.e., there is no best definition that suits each use case. In this section, we discuss several alternatives for the meaning of morphisms and select one of them as our working interpretation. We focus on morphisms connecting the attributes of simple relational schemas under multiset (i.e., SQL) semantics without integrity or key constraints. In the case of complex schemas, such as relational schemas with constraints or XML Schema with nested types, the behavior of Rondo scripts is hard to characterize using the state-based operators. The reason is that the structural operators have been developed prior to the state-based semantics of morphisms that is presented in this section. Hence, the properties of the structural operators do not match exactly the requirements stated in Chap. 4. We illustrate this point in more detail in Sect. 6.3.

Consider Fig. 6.1. It shows a simple morphism that connects the schemas m_1 and m_2 using three arcs. Observe that the schema m_2 appears to be a normalized representation of m_1 . The foreign key dependency between the tables S and T in m_2 is intentionally omitted, i.e., we do not know whether S.ID is a foreign key for T.ID or the other way around, or whether S.ID and T.ID are keys at all. We examine three interpretations of this morphism, denoted as map_1 , map_2 , and map_3 . To give them names, we call these interpretations the *tuple-list*, *tuple-set*, and *value-set* interpretation, respectively.

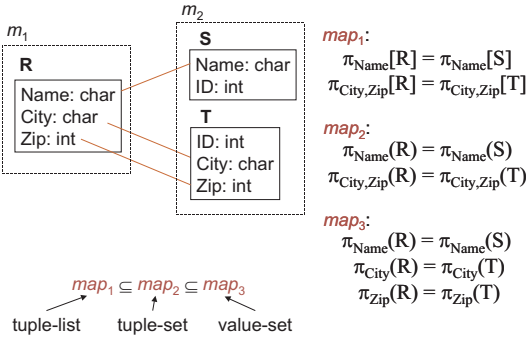


Fig. 6.1. Three alternative semantics for a morphism

The tuple-list interpretation map_1 is motivated by version management. Assume that one of the schemas is a new version of the other schema. In this case, an instance of the new version of the schema can be obtained by copying the values from the instance of the old schema. Each arc indicates what attribute values are copied. For example, the arc connecting R.Name and S.Name indicates that the attribute values of R.Name are selected and inserted into S.Name, or vice versa. This relationship can be expressed as a constraint `SELECT Name FROM R = SELECT Name FROM S`, which we abbreviate as $\pi_{Name}[R] = \pi_{Name}[S]$. Similarly, for the attributes City and Zip we obtain the constraints

$\pi_{\text{City}}[\mathbf{R}] = \pi_{\text{City}}[\mathbf{T}]$ and $\pi_{\text{City}}[\mathbf{R}] = \pi_{\text{City}}[\mathbf{T}]$, respectively. Thus, our first interpretation of the morphism yields a mapping that can be described as $map_1 = \ll \pi_{\text{Name}}[\mathbf{R}] = \pi_{\text{Name}}[\mathbf{S}], \pi_{\text{City}}[\mathbf{R}] = \pi_{\text{City}}[\mathbf{T}], \pi_{\text{Zip}}[\mathbf{R}] = \pi_{\text{Zip}}[\mathbf{T}] \gg$. Notice that due to multiset semantics map_1 is equivalent to $\ll \pi_{\text{Name}}[\mathbf{R}] = \pi_{\text{Name}}[\mathbf{S}], \pi_{\text{City,Zip}}[\mathbf{R}] = \pi_{\text{City,Zip}}[\mathbf{T}] \gg$.

Should m_2 indeed represent a normal form for m_1 , then map_1 does not characterize the mapping adequately. Copying all R.City and R.Zip tuples into T including the duplicate ones is not what we want; we are interested in distinct tuples only. That is, the tuple-list interpretation is too restrictive: it is consistent with the semantics of normalization only when (R.Name) and (R.City, R.Zip) are keys in m_1 . To embrace the more general case, we examine another possible interpretation for morphisms, the tuple-set interpretation. For the morphism of Fig. 6.1, the tuple-set interpretation is $map_2 = \ll \pi_{\text{Name}}(\mathbf{R}) = \pi_{\text{Name}}(\mathbf{S}), \pi_{\text{City,Zip}}(\mathbf{R}) = \pi_{\text{City,Zip}}(\mathbf{T}) \gg$. An expression such as $\pi_{\text{City,Zip}}(\mathbf{R})$, with parentheses instead of brackets, refers to the set of tuples City, Zip selected from R. Observe that map_2 holds even if (R.Name), (R.City, R.Zip) are not keys in R.

In the tuple-set interpretation, we assume that whenever two or more lines connect the attributes of two tables, the relationship between the respective attribute values remains preserved, as in $\pi_{\text{City,Zip}}(\mathbf{R}) = \pi_{\text{City,Zip}}(\mathbf{T})$. This assumption turns out to be too strong for the morphisms used in Rondo. To illustrate, consider Fig. 6.2. The figure depicts the composition of two simple morphisms, whose semantics is $m_1 _ m_2 = \ll \pi_{\text{City}}(\mathbf{R}) = \pi_{\text{City}}(\mathbf{S}), \pi_{\text{Zip}}(\mathbf{R}) = \pi_{\text{Zip}}(\mathbf{T}) \gg$ and $m_2 _ m_3 = \ll \pi_{\text{City}}(\mathbf{S}) = \pi_{\text{City}}(\mathbf{U}), \pi_{\text{Zip}}(\mathbf{T}) = \pi_{\text{Zip}}(\mathbf{U}) \gg$. Observe that the relationship between cities and zip codes in m_2 is vacuous. That is, if we associate the instances of m_1 and m_3 by way of mappings $m_1 _ m_2$ and $m_2 _ m_3$, we cannot expect the relationship between cities and zip codes to be preserved in the composed mapping $m_1 _ m_2 \circ m_2 _ m_3$. In other words, the constraint $\pi_{\text{City,Zip}}(\mathbf{R}) = \pi_{\text{City,Zip}}(\mathbf{U})$ is not guaranteed to be satisfied given the constraints imposed by the mappings $m_1 _ m_2$ and $m_2 _ m_3$. For example, consider the instances $i_1 = \ll \{(\text{Seattle}, 001), (\text{Berlin}, 002)\} \gg$, $i_2 = \ll \{\text{Seattle}, \text{Berlin}\}, \{001, 002\} \gg$, $i_3 = \ll \{(\text{Seattle}, 002), (\text{Berlin}, 001)\} \gg$. Although $(i_1, i_2) \in m_1 _ m_2$ and $(i_2, i_3) \in m_2 _ m_3$, the constraint $\pi_{\text{City,Zip}}(\mathbf{R}) = \pi_{\text{City,Zip}}(\mathbf{U})$ does not hold for the instances (i_1, i_3) . In contrast, the constraint $\pi_{\text{City}}(\mathbf{R}) = \pi_{\text{City}}(\mathbf{U}), \pi_{\text{Zip}}(\mathbf{R}) = \pi_{\text{Zip}}(\mathbf{U})$ does hold. It corresponds to our third interpretation, the value-set interpretation map_3 of Fig. 6.1.

The interpretations that we considered relate to each other as follows: $map_1 \subseteq map_2 \subseteq map_3$. We argued that map_1 and map_2 do not characterize the semantics of Rondo morphisms correctly. A weaker interpretation map_3 is a compromise and we will use it as our working assumption for morphism semantics. A number of alternative interpretations of morphisms could be constructed by e.g. combining the set-based and list-

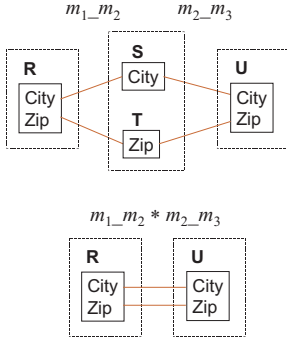


Fig. 6.2. Relationship between cities and zip codes is not preserved on composition

based projection or using the subset operator instead of equality, as in $\ll \pi_{\text{Name}}[\mathbf{R}] \subseteq \pi_{\text{Name}}[\mathbf{S}], \pi_{\text{City,Zip}}(\mathbf{R}) = \pi_{\text{City,Zip}}(\mathbf{T}) \gg$, etc. However, since the morphisms in Rondo are plain sets of arcs, such variants cannot be distinguished in the syntax, i.e., in the graphical representation; among other things, the interpretation of morphisms needs to be symmetric with respect to m_1 and m_2 . These variants could be used as another useful mapping language in future work.

The value-set semantics is broad enough to subsume a number of useful transformations. Some of them are listed below (referring to the schemas of Fig. 6.1).

$\mathbf{S} \bowtie \mathbf{T} \rightarrow \mathbf{R}$:

```
R = SELECT [DISTINCT] Name, City, Zip
      FROM S,T WHERE S.ID=T.ID
```

$\mathbf{R} \rightarrow \mathbf{S}, \mathbf{T}$:

```
S = SELECT DISTINCT Name, Sk(Name) FROM R;
T = SELECT DISTINCT Sk(City, Zip), City, Zip FROM R
```

$\mathbf{R} \rightarrow \underline{\mathbf{S}} \bowtie \mathbf{T}$:

```
S = SELECT DISTINCT Name, Sk(Name) FROM R;
T = SELECT [DISTINCT] Sk(Name), City, Zip FROM R
```

$\mathbf{R} \rightarrow \underline{\mathbf{T}} \bowtie \mathbf{S}$:

```
S = SELECT [DISTINCT] Name, Sk(City, Zip) FROM R;
T = SELECT DISTINCT Sk(City, Zip), City, Zip FROM R
```

For example, the first SELECT clause, labeled with $\mathbf{S} \bowtie \mathbf{T} \rightarrow \mathbf{R}$, represents two transformations obtained by including or omitting the DISTINCT sub-clause; they define m_1 as a view on m_2 . The remaining five transformations define **S** and **T** in terms of **R**. $Sk()$ denotes a Skolem function. For instance, in $\mathbf{R} \rightarrow \underline{\mathbf{S}} \bowtie \mathbf{T}$, **S.ID** is intended to be the primary key in **S**, whereas **T.ID** is the foreign key. In fact, since **S** contains distinct tuples only and the attribute **Name** is in the domain of the Skolem function used to compute **S.ID**, so the functional dependency $\mathbf{S.ID} \rightarrow \mathbf{S.Name}$ holds. The inclusion dependency

$T.ID \subseteq S.ID$ is satisfied because T is generated from the same relation R using the same Skolem function. In $R \rightarrow \underline{T} \bowtie S$, $T.ID$ is the primary key in T while $S.ID$ is its foreign key, i.e., $T.ID \rightarrow \{T.City, T.Zip\}$ and $S.ID \subseteq T.ID$. In $R \rightarrow S, T$ the relations S and T are obtained as independent collections of names and addresses. All of the above view definitions are contained in map_3 , i.e., for each view v we have $v \subseteq map_3$.

In the subsequent sections we assume the value-set semantics for morphisms: each arc in a morphism establishes equality of the value sets of the connected schema elements and is independent of other arcs. Thus, morphisms do not describe structural or value transformations of schemas. They contain no joins and no WHERE clause. Although the subset of morphisms that we discuss is a fairly weak mapping language, it is instrumental for describing the relationships between schemas when the exact transformation is not known. For example, in Clio (Popa et al. 2002), the set of initial correspondences between the elements of two schemas is represented graphically similarly to the morphism of Fig. 6.1. These correspondences are subsequently refined into precise view definitions using an elaborate user interface. Hence, a morphism could be seen as a “rough” mapping that covers a variety of possible view definitions.

We define the semantics of the empty morphism between m_1 and m_2 as $m_1 \times m_2$. That is, the empty morphism does not place any constraints on the mapping between m_1 and m_2 . An empty relational schema contains exactly one database instance, the empty set. Finally, we assume that relational schemas may contain NULL values, which are treated as absent values. In particular, $\pi_A(R)$ is guaranteed to contain only non-NULL values and is the empty set when all $R.A$ values are NULLs. We explain why this assumption is important when we discuss composition in Sect. 6.3.

6.2 Semantics of Selectors

A selector identifies a set of model elements. We define the state-based semantics of selector s as that of the corresponding identity morphism $\text{Id}(s)$. If s contains a set of relation attributes, so $\text{Id}(s)$ is a one-to-one correspondence between the attributes s of two identical relational schemas. The state-based semantics of such correspondence has been defined in the previous section.

To illustrate, consider the model m_2 of Fig. 6.1. Let $s = \{S.Name, T.City, T.Zip\}$ and let m'_2 be a model identical to m_2 . The state-based semantics of the selector s is that of the mapping $s_{map} = \ll \pi_{Name}(m_2.S) = \pi_{Name}(m'_2.S), \pi_{City}(m_2.T) = \pi_{City}(m'_2.T), \pi_{Zip}(m_2.T) = \pi_{Zip}(m'_2.T) :: m_2 :: m'_2 \gg$. Notice that $\text{Invert}(s_{map}) = s_{map}$.

6.3 Structural vs. State-Based Operators

In this section, we compare the state-based semantics of the structural operators used in Rondo with that of the operators of Chap. 4. As we will see, the state-based semantics of some structural operators, such as `Compose` and `Invert`, is identical to that of the respective state-based operators. Other structural operators, such as `Extract` and `Merge`, return materializations of the exact results (compare Sect. 4.3), i.e., have a weaker state-based semantics. This fact is not surprising given that the definitions of the state-based operators `Extract` and `Merge` contain minimality requirements that are very hard to meet in concrete schema and mapping languages. We also show that one of the operators, `Diff`, produces results that violate the desired conditions that we postulated in Chap. 4; specifically, under the value-set interpretation for morphisms, `Diff` loses information.

We summarized the signatures of the structural and state-based operators in Table 2.1 on page 23 and Table 4.1 on page 64, respectively. The structural operators are defined for models represented as directed labeled graphs and for a simple concrete mapping language, the morphisms. Selectors can be viewed as syntactic sugar; they can be replaced in all operator signatures by (identity) morphisms. In contrast, the state-based operator definitions apply to mappings expressed in arbitrary languages and do not rely on a particular representation of models and mappings. Although the signatures of the state-based operators such as `Id`, `Domain`, or `Invert`, are very similar to those of the respective structural operators, a key difference to keep in mind is that the mappings taken as parameters identify binary relations on instances rather than binary relations on model elements.

We assume that in accordance with the operator signatures, the free variables used below range over simple relational schemas, value-set morphisms, and selectors (i.e., identity morphisms) rather than over arbitrary models and mappings, so that we do not have to quantify the variables in each expression. We start with the structural operator `Compose` and the state-based operator `Compose`, denoted as \circ . The state-based semantics of `Compose` is exactly the one specified in Definition 4.2.1:

$$\text{Compose}(map_1, map_2) = map_1 \circ map_2;$$

Fig. 6.3 shows three representative examples of composition, in the rows. The leftmost column contains the source schemas and mappings. The second column depicts the result of composition produced by the structural operator `Compose`. The third column presents the result that satisfies the conditions of the state-based operator `Compose` under the assumption that `NULL` values are allowed in relations (our working assumption). In the rightmost column, `NULL`s are disallowed. Observe that the morphisms shown in column 2 have exactly the semantics of the mappings in column 3. For instance, in the second example structural composition yields an empty morphism. And indeed, for any two given instances $i_1 \in m_1$ and $i_3 \in m_3$ it is always possible to find

an instance $i_2 \in m_2$ such that $\pi_A(i_1.R) = \pi_A(i_2.S)$ and $\pi_B(i_3.T) = \pi_A(i_2.S)$ by constructing a relation S whose A values are drawn from $\pi_A(i_1.R)$ and B values are drawn from $\pi_B(i_3.T)$. Should either $\pi_A(i_1.R) = \emptyset$ or $\pi_B(i_3.T) = \emptyset$ hold, the respective attribute values can be simply filled with NULLs. However, if NULL values are disallowed, which is the case in the classical relational model, then such an instance i_2 can only be found if either both R and T are empty, or both are non-empty. Hence, in this case composition yields the mapping $\langle R = \emptyset \leftrightarrow T = \emptyset \rangle$ shown in column 4. The example illustrates that it is critical to specify the state-based semantics of schemas precisely, down to such details as whether NULLs are supported or not. The third example of Fig. 6.3 is analogous.

Input morphisms		Result of composition		
m_1, m_2	m_2, m_3	$m_1, m_2 * m_2, m_3$ (in Rondo)	$m_1, m_2 \circ m_2, m_3$ (with NULLs)	$m_1, m_2 \circ m_2, m_3$ (no NULLs)
			$\pi_A(R) = \pi_A(T),$ $\pi_B(R) = \pi_B(T)$	$\pi_A(R) = \pi_A(T),$ $\pi_B(R) = \pi_B(T)$
			empty constraint	$R = \emptyset \leftrightarrow$ $T = \emptyset$
			$\pi_A(R_1) = \pi_A(T_1),$ $\pi_B(R_2) = \pi_B(T_2)$	$\pi_A(R_1) = \pi_A(T_1),$ $\pi_B(R_2) = \pi_B(T_2),$ $(R_1 = \emptyset \leftrightarrow$ $R_2 = \emptyset \leftrightarrow$ $T_1 = \emptyset \leftrightarrow$ $T_2 = \emptyset)$

Fig. 6.3. Structural composition vs. state-based composition (the latter with and without NULLs; predicate \leftrightarrow denotes if-and-only-if)

For the operator `Invert` we obtain:

$$\text{Invert}(map) = \text{Invert}(map);$$

`Invert` simply swaps the left and right sides of the morphism; this is exactly what `Invert` does to arbitrary mappings.

The relationship between `Extract` and `Extract` is more subtle. We can show that the result of `Extract` is a valid materialization of the result of `Extract`, as we discussed in Sect. 4.3. Formally:

$$\begin{aligned} \langle m_c, m_m_c \rangle &= \text{Extract}(m, s) \rightarrow \\ &\exists m_x, m_m_x, m_c_m_x : \\ &(\langle m_x, m_m_x \rangle = \text{Extract}(m, s); \\ & m_m_x = m_m_c \circ m_c_m_x; \\ & m_c = \text{Range}(m_m_c) = \text{Domain}(m_c_m_x); \\ & \text{Invert}(m_c_m_x) \circ m_c_m_x = \text{Id}(m_x);) \end{aligned}$$

The predicate \rightarrow above denotes the logical implication. For clarity, we quantify the free variables $m_x, m_m_x, m_c_m_x$ occurring in the implied part of the

statement explicitly. The above formula says that we can get the (exact) output of `Extract` by defining a view on the (materialized) result produced by `Extract`. This view is $m_c _ m_x$. The condition $\text{Invert}(m_c _ m_x) \circ m_c _ m_x = \text{Id}(m_x)$ requires $m_c _ m_x$ to be a surjective function onto m_x (see Proposition 4.2.4).

To illustrate the relationship between m_x and m_c , consider Fig. 6.4. Let s be the identity morphism on the attributes Name, City, and Zip in m . The schemas extracted from m by the structural operator and by Definition 4.2.3 are depicted in Fig. 6.4 as m_c and m_x , respectively. Schema m_x contains three tables with a single key attribute each. The mapping $m_c _ m_x$ is depicted using arrows to distinguish it from morphisms; both $m_c _ m_x$ and $m _ m_x$ can be defined as views as follows ($m _ m_c$ is shown for convenience):

$$\begin{aligned}
 m_c _ m_x &= \langle\langle m_x.U_1 = \text{SELECT DISTINCT Name FROM } m_c.S, \\
 &\quad m_x.U_2 = \text{SELECT DISTINCT City FROM } m_c.T, \\
 &\quad m_x.U_3 = \text{SELECT DISTINCT Zip FROM } m_c.T \rangle\rangle \\
 m _ m_x &= \langle\langle m_x.U_1 = \text{SELECT DISTINCT Name FROM } m.S, \\
 &\quad m_x.U_2 = \text{SELECT DISTINCT City FROM } m.T, \\
 &\quad m_x.U_3 = \text{SELECT DISTINCT Zip FROM } m.T \rangle\rangle \\
 m _ m_c &= \langle\langle m_c.S = \pi_{\text{Name}}(m.S), \\
 &\quad \pi_{\text{City}}(m_c.T) = \pi_{\text{City}}(m.T), \\
 &\quad \pi_{\text{Zip}}(m_c.T) = \pi_{\text{Zip}}(m.T) \rangle\rangle
 \end{aligned}$$

It is straightforward to verify that $m _ m_x = m _ m_c \circ m_c _ m_x$.

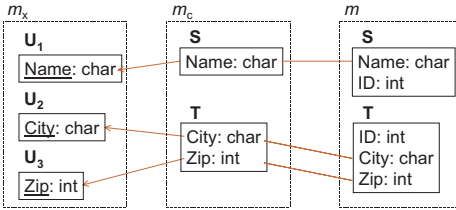


Fig. 6.4. Structural extraction yields materialization of the state-based operator

This example illustrates that extraction implemented in Rondo does not produce a minimal schema, but nevertheless contains all information necessary to derive the result required by Definition 4.2.3. The example also shows a tradeoff in using weak mappings such as morphisms: on extraction they yield schemas that are not very expressive (m_x).

Using similar considerations we can verify that the following relationship holds for merging:

$$\begin{aligned}
 \langle m_c, m_c _ m_1, m_c _ m_2 \rangle &= \text{Merge}(m_1, m_2, m_1 _ m_2) \rightarrow \\
 &\exists m, m _ m_1, m _ m_2, m_c _ m : \\
 &\langle m, m _ m_1, m _ m_2 \rangle = \text{Merge}(m_1, m_2, m_1 _ m_2);
 \end{aligned}$$

$$\begin{aligned}
m_{c_}m_1 &= m_{c_}m \circ m_m_1; \\
m_{c_}m_2 &= m_{c_}m \circ m_m_2; \\
m_c &= \text{Domain}(m_{c_}m); \\
\text{Invert}(m_{c_}m) \circ m_{c_}m &= \text{Id}(m); \quad)
\end{aligned}$$

That is, the schema and morphisms produced by Merge are materializations of the exact results of Merge.

Unfortunately, we cannot characterize the operator Delete as easily in terms of Diff. Delete is defined as a derived operator, $\text{Delete}(m, s) := \text{Extract}(m, \text{All}(m) - s)$, and does not guarantee that we can reconstruct each instance of m given an instance of the schema obtained by extraction and an instance obtained by deletion. For a counterexample, see Example 4.2.23. That is, as we explain in Sect. 10.5, the operator Delete is not suitable for computing the view complement in data warehousing scenarios.

Interestingly, if we assume a different semantics for morphisms, the tuple-list semantics discussed in Sect. 6.1, we find that the results of Delete are in fact materializations of the results of Diff. This fact reiterates the importance of formal specification of semantics: the operators in Rondo may have been developed with different morphism semantics in mind. Under value-set semantics of morphisms, Delete can be viewed as a variant of the Extract operator.

Other structural operators used in Rondo can be characterized as follows:

$$\begin{aligned}
\text{Range}(map) &= \text{Invert}(map) \circ map; \\
\text{Domain}(map) &= map \circ \text{Invert}(map); \\
\text{RestrictDomain}(map, s) &= s \circ map; \\
\text{RestrictRange}(map, s) &= map \circ s; \\
\text{Union}(s_1, s_2) &= s_1 \oplus s_2; \\
\text{All}(m) &\supseteq \text{Id}(m);
\end{aligned}$$

6.4 Revisiting Change Propagation

The script that implements change propagation, which we presented in Chap. 2, differs from the script developed in Chap. 5. The principal difference lies in the way that deletion is propagated. In this section we demonstrate that the state-based solution of Chap. 5 can be used to obtain a different solution for propagating deletion in Rondo, which turns out to be equivalent to the original structural script of Chap. 2. Note that in general, a structural script is not equivalent to a state-based script created simply by replacing the structural operators by the corresponding state-based operators.

In Chap. 2, propagation of deletion is implemented using the script

```

operator PropagateDeletionsA( $s_1, d_1, s_{1\_}d_1, s_{1\_}s_2$ )
   $\langle d'_1, d_{1\_}d'_1 \rangle = \text{Delete}(d_1, \text{ Traverse}(\text{All}(s_1) - \text{Domain}(s_{1\_}s_2), s_{1\_}d_1));$ 
  return  $\langle d'_1, d_{1\_}d'_1 \rangle;$ 

```

In Chap. 5 we argued that the following state-based specification of propagating deletions is correct:

```
operator PropagateDeletionsB( $s_1, d_1, s_{1-d_1}, s_{1-s_2}$ )
  1.  $d_{1-s_2} = \text{Invert}(s_{1-d_1}) \circ s_{1-s_2}$ ;
  2.  $\langle d_{1d}, d_{1-d_{1d}} \rangle = \text{Diff}(d_1, \text{Invert}(s_{1-d_1}))$ ; // added in  $d_1$ 
  3.  $\langle d_{1x}, d_{1-d_{1x}} \rangle = \text{Extract}(d_1, d_{1-s_2})$ ; // kept in  $d_1$ 
  4.  $\langle d'_1, d'_{1-d_{1x}}, d'_{1-d_{1d}} \rangle = \text{Merge}(d_{1x}, d_{1d}, \text{Invert}(d_{1-d_{1x}}) \circ d_{1-d_{1d}})$ ;
  5.  $d_{1-d'_1} = d_{1-d_{1d}} \circ \text{Invert}(d'_{1-d_{1d}}) \oplus d_{1-d_{1x}} \circ \text{Invert}(d'_{1-d_{1x}})$ ;
  6. return  $\langle d'_1, d_{1-d'_1} \rangle$ ;
```

Now we show that the implementation of PropagateDeletionsB using a one-to-one operator translation into structural operators is equivalent to PropagateDeletionA. Whenever a morphism is passed as a parameter to a structural operator that expects a selector, we simply take the Domain of the morphism. For example, $\text{Extract}(d_1, d_{1-s_2})$ of line 3 becomes $\text{Extract}(d_1, \text{Domain}(d_{1-s_2}))$. If we replace Diff by Delete and expand the definition of Delete as $\text{Delete}(m, s) = \text{Extract}(\text{All}(m) - s)$ according to the definition of Sect. 2.3.3, then lines 2-3 translate to:

$$\begin{aligned} \langle d_{1d}, d_{1-d_{1d}} \rangle &= \text{Extract}(d_1, \text{All}(d_1) - \text{Domain}(\text{Invert}(s_{1-d_1}))); \\ \langle d_{1x}, d_{1-d_{1x}} \rangle &= \text{Extract}(d_1, \text{Domain}(d_{1-s_2})); \end{aligned}$$

The expression $\text{Domain}(\text{Invert}(s_{1-d_1}))$ can be simplified as $\text{Range}(s_{1-d_1})$ (see Sect. 2.3.2). The algorithm used in Chap. 2 to implement the Merge operator computes the union of models with renaming. In line 4, no conflicts arise from using the operator Merge, since the schemas to be merged have been extracted from the same model. Therefore, the above two extractions followed by a Merge are equivalent to a single extraction over the union of mappings. Thus, lines 1-5 translate to:

$$\langle d'_1, d_{1-d'_1} \rangle = \text{Extract}(d_1, (\text{All}(d_1) - \text{Range}(s_{1-d_1})) + \text{Range}(\text{Invert}(s_{1-s_2}) * s_{1-d_1}));$$

It remains to show that the above expression is equivalent to:

$$\langle d'_1, d_{1-d'_1} \rangle = \text{Delete}(d_1, \text{Traverse}(\text{All}(s_1) - \text{Domain}(s_{1-s_2}), s_{1-d_1}));$$

which can be expanded into

$$\langle d'_1, d_{1-d'_1} \rangle = \text{Extract}(d_1, \text{All}(d_1) - \text{Traverse}(\text{All}(s_1) - \text{Domain}(s_{1-s_2}), s_{1-d_1}));$$

It is sufficient to show the equality:

$$\begin{aligned} (\text{All}(d_1) - \text{Range}(s_{1-d_1})) + \text{Range}(\text{Invert}(s_{1-s_2}) * s_{1-d_1}) = \\ \text{All}(d_1) - \text{Traverse}(\text{All}(s_1) - \text{Domain}(s_{1-s_2}), s_{1-d_1}) \end{aligned}$$

We prove the equality using the schematic representation of Fig. 6.5, which uses the following definitions:

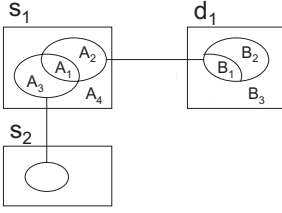


Fig. 6.5. Schematic representation for structural change propagation script

$$\begin{aligned}
 A_1 &= \text{Domain}(s_1_s_2) \cap \text{Domain}(s_1_d_1) \\
 A_2 &= \text{Domain}(s_1_d_1) - \text{Domain}(s_1_s_2) \\
 A_3 &= \text{Domain}(s_1_s_2) - \text{Domain}(s_1_d_1) \\
 A_4 &= \text{All}(s_1) - A_1 - A_2 - A_3 \\
 B_1 &= \text{Traverse}(A_1, s_1_d_1) \\
 B_2 &= \text{Traverse}(A_2, s_1_d_1) \\
 B_3 &= \text{All}(d_1) - B_1 - B_2
 \end{aligned}$$

The selectors A_1, A_2, A_3, A_4 and B_1, B_2, B_3 yield disjoint decompositions of $\text{All}(s_1)$ and $\text{All}(d_1)$, respectively. The following properties hold:

$$\begin{aligned}
 \text{All}(s_1) &= A_1 + A_2 + A_3 + A_4 \\
 \text{All}(d_1) &= B_1 + B_2 + B_3 \\
 \text{Domain}(s_1_d_1) &= A_1 + A_2 \\
 \text{Range}(s_1_d_1) &= B_1 + B_2 \\
 \text{Domain}(s_1_s_2) &= A_1 + A_3
 \end{aligned}$$

Thus, we obtain

$$\begin{aligned}
 (\text{All}(d_1) - \text{Range}(s_1_d_1)) + \text{Range}(\text{Invert}(s_1_s_2) * s_1_d_1) &= \\
 B_1 + B_2 + B_3 - (B_1 + B_2) + \text{Traverse}(A_1, s_1_d_1) &= \\
 B_1 + B_3 &
 \end{aligned}$$

and

$$\begin{aligned}
 \text{All}(d_1) - \text{Traverse}(\text{All}(s_1) - \text{Domain}(s_1_s_2), s_1_d_1) &= \\
 B_1 + B_2 + B_3 - \text{Traverse}(A_2 + A_4, s_1_d_1) &= \\
 B_1 + B_2 + B_3 - (\text{Traverse}(A_2, s_1_d_1) + \text{Traverse}(A_4, s_1_d_1)) &= \\
 B_1 + B_2 + B_3 - (B_2 + \emptyset) &= \\
 B_1 + B_3 &
 \end{aligned}$$

Hence, both realizations of propagating deletions are equivalent.

Although strictly speaking the structural scripts do not implement the state-based scripts, the above example backs the intuition for the following conclusions that we expect to hold in general.

On the one hand, the example illustrates that it may be possible to find alternative realizations, or “physical plans”, for a given state-based script. Here, a different equivalent realization allows us to use a single complex operator *Extract* instead of three invocations of complex operators *Extract*, *Diff*, and *Merge*. Propagating deletions using selectors does not require computing

the results of `Diff` and `Merge`. The rest of the selector-oriented realization `PropagateDeletionsA` uses the primitive operators, which can be optimized by the SQL engine of the underlying DBMS. In other scripts, further potential optimization can be obtained by storing precomputed results that are used at several places in a script.

On the other hand, the example shows that it may be possible to optimize the execution of a given state-based script even if the script itself cannot be further simplified into an equivalent state-based script. This is similar to database query processing: we first optimize the logical query plan, then the physical query plan. We expect the above observations to hold for proper implementations of state-based scripts as well.

6.5 Conclusions

In Part II, we explored a state-based semantics for the set of operators put forth in model management. We derived the properties of the operators from established metadata management scenarios and verified their applicability using numerous examples. We obtained a simplified characterization of the operators and presented an initial study of their properties.

The state-based semantics proved instrumental for clarifying the meaning of the conceptual structures used in the prototype Rondo. Moreover, the analysis that we presented helped us realize a deficiency in the implementation of the operator `Diff`.

A noticeable feature of the operators `Extract`, `Merge`, and `Diff` are the minimality conditions on the output models. These conditions are critical for expressing the intended semantics of the operators. For example, if we remove the minimality condition (iii) from the definition of `Extract`, then the operator becomes vacuous: for any input model m we could easily find a trivial output model, the model m itself. Analogously, removing the minimality condition (iv) from `Merge` would allow the operator to produce a virtually arbitrary very expressive model as output. Finally, eliminating condition (ii) from the definition of `Diff` would make `Diff` a derived operator that could be specified in terms of the operators `Extract`, `Merge`, `Invert`, and `Compose`. The minimality conditions help us make the operators non-redundant, but necessarily contribute to the complexity of computing their results. The completeness and non-redundancy of the suggested set of operators is an open problem, which we discuss in more detail in Sect. 11.3.3.

A major strength of the state-based characterization is its ability to specify model-management operations in an abstract fashion, without appealing to any idiosyncratic schema, constraint, or transformation languages. However, applying such an abstract characterization to concrete languages and developing practical algorithms for computing the results of operators effectively can be extremely hard. For example, as we demonstrate in Sect. 10.1.2, the problem of answering queries using views can be defined quite easily using

a state-based characterization, whereas developing rewriting algorithms for specific query languages proved to be very challenging. We expect that computing the results of model-management scripts for concrete languages, the problem which we called materialization, will prove at least as hard. Extensive future research is required for solving this problem.

By following a state-based approach we recognize that a model is more than just syntax: it is a template for instances. On the other hand, a model is more than just a template for instances: its syntax is important for developers and applications. For example, although many relational schemas might be able to express identical information, applications rely on the fact that they contain tables with certain names and certain attributes, whose order in the table definition may be important. Therefore, considering both state-based and structural semantics is critical for specifying the effects of model-management scripts.

We believe that the state-based semantics should apply not only to Rondo, but also to other systems that will be built in the future. In particular, it provides guidelines for implementing the operators for much more powerful schema and mapping languages as compared to those that we utilized in our programming platform. We expand the discussion of future work on state-based semantics in Chap. 11.

7. Similarity Flooding Algorithm

“Mr. Martin: . . . You know, in my bedroom there is a bed, and it is covered with a green eiderdown. This room, with the bed and the green eiderdown, is at the end of the corridor between the w.c. and the bookcase, dear lady!

Mrs. Martin: What a coincidence, good Lord, what a coincidence! My bedroom, too, has a bed with a green eiderdown and is at the end of the corridor, between the w.c., dear sir, and the bookcase!

Mr. Martin: How bizarre, curious, strange! Then, madam, we live in the same room and we sleep in the same bed, dear lady.”

– Eugene Ionesco (1958), “The Bald Soprano”

Finding correspondences between models is required in many application scenarios. This task is often referred to as *matching*. In generic model management, matching is embodied in the operator `Match`, which plays a critical role in many model-management scripts. The operator `Match` takes two models as input and returns a mapping between the models as output. Of all operators that we examined in the previous chapters, `Match` is the only one that lacks a formal definition and, in a way, enjoys a special status. The reason for its speciality is that matching typically involves information that is not contained in the input models. Uncovering how two models relate to each other requires reading documentation, examining instances of models, and talking to the engineers who designed or deploy the models.

Matching is a complex and time-consuming design task. Techniques used to automate this task often differ substantially. For example, for matching relational schemas one could use SQL data types to determine which columns are possibly related. On the other hand, in XML schema matching, hierarchical relationships between schema elements can be exploited. Because of this diversity, applications that rely on matching are often built from scratch and require significant amount of thought and programming. We address this problem by proposing a matching algorithm that allows quick development of matchers for a broad spectrum of different scenarios. We are not trying

to outperform custom matchers that use highly tuned, domain-specific heuristics.

In this chapter we suggest a simple structural algorithm that can be used for matching of diverse data structures including schemas, instances, and other kinds of models. The algorithm that we suggest is based on the following idea. The models to be matched, which are represented as directed labeled graphs, are used in an iterative fixpoint computation whose results tell us what nodes in one graph are similar to nodes in the second graph. For computing the similarities, we rely on the intuition that elements of two distinct models are similar when they occur in similar contexts, i.e., when their adjacent elements are similar. In other words, a part of the similarity of two elements propagates to their respective neighbors. The spreading of similarities in the matched models is reminiscent to the way how IP packets flood the network in broadcast communication. For this reason, we call our algorithm the *Similarity Flooding* algorithm. The result produced by the algorithm is a morphism, a simple kind of mapping that we presented in Chap. 2. Depending on the particular matching goal, we then choose a subset of the resulting mapping using adequate filters.

After our algorithm runs, we expect a human to check and if necessary adjust the results. As a matter of fact, we evaluate the “accuracy” of the algorithm by counting the number of needed adjustments. We designed a graphical tool which helps human developers to inspect and post-process the suggestions delivered by the algorithm. In this tool, the user adjusts the proposed match result by removing or adding lines connecting the elements of two schemas. As we stressed above, the correct match often depends on the information only available or understandable by humans. For example, even matches as plausible as `ZipCode` to `zip_code` can be doomed as incorrect by a data warehouse designer who knows that zip codes from a given relational source should not be collected due to poor data quality. In such cases, the suggested mappings may be incorrect or incomplete.

This chapter is structured as follows:

- In Sect. 7.1, we give an overview of the approach. The Similarity Flooding algorithm is introduced in Sect. 7.2.
- In Sect. 7.3, we present a generalized formula for the algorithm and discuss its convergence and complexity in Sect. 7.4.
- In Sect. 7.5, we demonstrate the applicability of the algorithm for diverse matching tasks.

In subsequent chapters, we address the filtering of the results delivered by the SF algorithm (Chap. 8) and its evaluation and tuning (Chap. 9). We conclude Part III in Sect. 9.6.

7.1 Overview of the Approach

Before we go into details of our matching algorithm, let us briefly walk through an example that illustrates matching of two relational database schemas. Please keep in mind that the technique we describe is not limited to relational schemas. Consider schemas S_1 and S_2 depicted in Fig. 7.1. The elements of S_1 and S_2 are tables and columns. Assume for now that our goal is to obtain exactly one matching element for every element in S_1 . A part of the matching result could be, for example, the correspondence of column `Personnel/Pname` to column `Employee/EmpName`. A sequence of steps that allows us to determine the correspondences between tables and columns in S_1 and S_2 can be expressed as the following script:

```

CREATE TABLE Personnel (
  Pno int,
  Pname string,
  Dept string,
  Born date,
  UNIQUE perskey(Pno)
)
      (S1)

CREATE TABLE Employee (
  EmpNo int PRIMARY KEY,
  EmpName varchar(50),
  DeptNo int REFERENCES Department,
  Salary dec(15,2),
  Birthdate date
)

CREATE TABLE Department (
  DeptNo int PRIMARY KEY,
  DeptName varchar(70)
)
      (S2)

```

Fig. 7.1. Matching two relational schemas: Personnel and Employee-Department

1. $G_1 = \text{ReadSQLDDL}(S_1)$; $G_2 = \text{ReadSQLDDL}(S_2)$;
2. $\text{initialMap} = \text{StringMatch}(G_1, G_2)$;
3. $\text{product} = \text{SFJoin}(G_1, G_2, \text{initialMap})$;
4. $\text{result} = \text{SelectThreshold}(\text{product})$;

As a first step, we translate the schemas from their native format into graphs G_1 and G_2 . In our example, the native format of the schemas are ASCII files containing table definitions in SQL DDL. A portion of the graph G_1 is depicted in Fig. 7.2. The translation into graphs is done using an import filter `ReadSQLDDL` that understands the definitions of relational schemas. We do not insist on choosing a particular graph representation for relational schemas. The representation used in Fig. 7.2 is based on the Open Information Model specification (Bernstein et al. 1999). The nodes in the graph are shown as ovals and rectangles. The labels inside the ovals denote the identifiers of the nodes, whereas rectangles represent literals, or string values. For example, node `&1` represents the table `Personnel` in graph G_1 , whereas nodes `&2`, `&4`, and `&6` denote columns `Pno`, `Pname`, and `Dept`, respectively. Column `Born` and unique key `perskey` are omitted from the figure for clarity. Tables `Employee`

and `Department` from schema S_2 are represented in a similar manner in graph G_2 . In our example, G_1 has a total of 31 nodes while G_2 has 55 nodes.

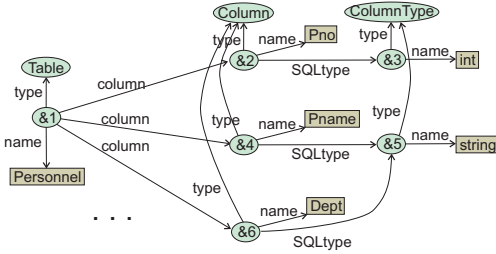


Fig. 7.2. A portion of graph representation G_1 for relational schema S_1

As a second step, we obtain an initial mapping *initialMap* between G_1 and G_2 using operator `StringMatch`. The mapping *initialMap* is obtained using a simple string matcher that compares common prefixes and suffixes of literals. A portion of the initial mapping is shown in Table 7.1. Literal nodes are highlighted using apostrophes. The second column of the table lists similarity values between nodes in G_1 and G_2 computed on the basis of their textual content. The similarity values range between 0 and 1 and indicate how well the corresponding nodes in G_1 match their counterparts in G_2 . Notice that the initial mapping is still quite imprecise. For instance, it suggests mapping column names onto table names (e.g. column `Dept` in S_1 onto table `Department` in S_2 , line 9), or names of data types onto column names (e.g., SQL type `date` in S_1 onto column `Birthdate` in S_2 , line 8).

Table 7.1. A portion of *initialMap* obtained by string matching (10 of total 26 entries are shown)

Line#	Similarity	Node in G_1	Node in G_2
1.	1.0	Column	Column
2.	0.66	ColumnType	Column
3.	0.66	'Dept'	'DeptNo'
4.	0.66	'Dept'	'DeptName'
5.	0.5	UniqueKey	PrimaryKey
6.	0.26	'Pname'	'DeptName'
7.	0.26	'Pname'	'EmpName'
8.	0.22	'date'	'Birthdate'
9.	0.11	'Dept'	'Department'
10.	0.06	'int'	'Department'

As a third step, operator `SFJoin` is applied to produce a refined mapping called *product* between G_1 and G_2 . In this chapter we propose an iterative “similarity flooding” (SF) algorithm based on a fixpoint computation that is used for implementing operator `SFJoin`. The SF algorithm has no knowledge of node and edge semantics. As a starting point for the fixpoint computa-

tion the algorithm uses an initial mapping like *initialMap*. Our algorithm is based on the assumption that whenever any two elements in models G_1 and G_2 are found to be similar, the similarity of their adjacent elements increases. Thus, over a number of iterations, the initial similarity of any two nodes propagates through the graphs. For example, in the first iteration the initial textual similarity of strings “Pname” and “EmpName” adds to the similarity of columns `Personnel/Pname` and `Employee/EmpName`. In the next iteration, the similarity of `Personnel/Pname` to `Employee/EmpName` propagates to the SQL types `string` and `varchar(50)`. This subsequently causes increase in similarity between literals “string” and “varchar”, leading to a higher resemblance of `Personnel/Dept` to `Department/DeptName` than that of `Personnel/Dept` to `Department/DeptNo`. The algorithm terminates after a fixpoint has been reached, i.e. the similarities of all model elements stabilize. In our example, the refined mapping *product* returned by SFJoin contains 211 node pairs with positive similarities (out of a total of $31 \cdot 55 = 1705$ entries in the G_1, G_2 cross-product).

As a last operation in the script, operator `SelectThreshold` selects a subset of node pairs in *product* that corresponds to the “most plausible” matching entries. We discuss this operator in Chap. 8. The complete mapping returned by `SelectThreshold` contains 12 entries and is listed in Table 7.2. For readability, we substituted numeric node identifiers by the descriptions of the objects they represent. For example, we replaced node identifier &2 by `[Column:Personnel/Pno]`.

Table 7.2. The mapping after applying `SelectThreshold` on result of SFJoin

Sim.	Node in G_1	Node in G_2
1.0	Column	Column
0.81	[Table: Personnel]	[Table: Employee]
0.66	ColumnType	ColumnType
0.44	[ColumnType: int]	[ColumnType: int]
0.43	Table	Table
0.35	[ColumnType: date]	[ColumnType: date]
0.29	[UniqueKey: perskey]	[PrimaryKey: on EmpNo]
0.28	[Column: Personnel/Dept]	[Column: Department/DeptName]
0.25	[Column: Personnel/Pno]	[Column: Employee/EmpNo]
0.19	UniqueKey	PrimaryKey
0.18	[Column: Personnel/Pname]	[Column: Employee/EmpName]
0.17	[Column: Personnel/Born]	[Column: Employee/Birthdate]

As we see in Table 7.2, the SF algorithm was able to produce a good mapping between S_1 and S_2 without any built-in knowledge about SQL DDL by merely using graph structures. For example, table `Personnel` was matched to table `Employee` despite the lack of textual similarity. Notice that the table still contains correspondences like the one between node `Column` in G_1 to node `Column` in G_2 , which are hardly of use given our goal of matching the specific

tables and columns. We discuss the filtering of match results in more detail in Chap. 8. The similarity values shown in the table may appear relatively low. As we will explain, in presence of multiple match candidates for a given model element, relative similarities are often more important than absolute values.

7.2 Similarity Flooding Algorithm

The internal data model that we use for models and mappings is based on directed labeled graphs. Every edge in a graph is represented as a triple (s, p, o) , where s and o are the source and target nodes of the edge, and the middle element p is the label of the edge. For a more formal definition of our internal data model please refer to Sect. 2.2.1. In this section, we explain our algorithm using a simple example presented in Fig. 7.3. The top left part of the figure shows two models A and B that we want to match.

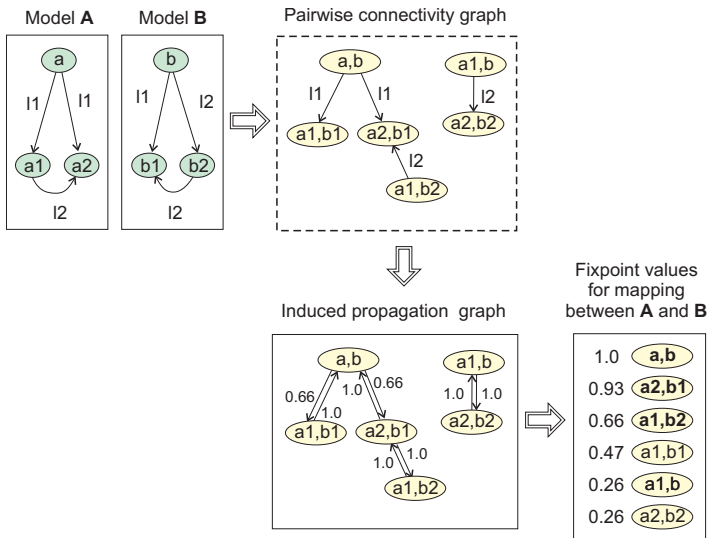


Fig. 7.3. Example illustrating the Similarity Flooding algorithm

7.2.1 Similarity Propagation Graph

A *similarity propagation graph* is an auxiliary data structure derived from models A and B that is used in the fixpoint computation of our algorithm. To illustrate how the propagation graph is computed from A and B , we first define a *pairwise connectivity graph* (PCG) as follows:

$$((x, y), p, (x', y')) \in \text{PCG}(A, B) \iff (x, p, x') \in A \text{ and } (y, p, y') \in B$$

Each node in the connectivity graph is an element from $A \times B$. We call such nodes *map pairs*. The connectivity graph for our example is enclosed in a dashed frame in Fig. 7.3. The intuition behind arcs that connect map pairs is the following. Consider for example map pairs (a, b) and (a_1, b_1) . If a is similar to b , then probably a_1 is somewhat similar to b_1 . The evidence for this conclusion is provided by the l_1 -edges that connect a to a_1 in graph A and b to b_1 in graph B . This evidence is captured in the connectivity graph as an l_1 -edge leading from (a, b) to (a_1, b_1) . We call (a_1, b_1) and (a, b) *neighbors*.

The induced propagation graph for A and B is shown next to the connectivity graph in Fig. 7.3. For every edge in the connectivity graph, the propagation graph contains an additional edge going in the opposite direction. The weights placed on the edges of the propagation graph indicate how well the similarity of a given map pair propagates to its neighbors and back. These so-called *propagation coefficients* range from 0 to 1 inclusively and can be computed in many different ways. The approach illustrated in Fig. 7.3 is based on the intuition that each edge type makes an equal contribution of 1.0 to spreading of similarities from a given map pair. For example, there is exactly one l_2 -edge out of (a_1, b) in the connectivity graph. In such case we set the coefficient $w((a_1, b), (a_2, b_2))$ in the propagation graph to 1.0. The value 1.0 indicates that the similarity of a_1 to b contributes fully to that of a_2 and b_2 . Analogously, the propagation coefficient $w((a_2, b_2), (a_1, b))$ on the reverse edge is also set to 1.0, since there is exactly one incoming l_2 -edge for (a_2, b_2) . In contrast, two l_1 -edges are leaving map pair (a, b) in the connectivity graph. Thus, the weight of 1.0 is distributed equally among $w((a, b), (a_1, b_1)) = 0.5$ and $w((a, b), (a_2, b_1)) = 0.5$. In Sect. 7.3 we analyze several alternative ways of computing the propagation coefficients.

7.2.2 Fixpoint Computation

Let $\sigma(x, y) \geq 0$ be the similarity measure of nodes $x \in A$ and $y \in B$ defined as a total function over $A \times B$. We refer to σ as a mapping. The similarity flooding algorithm is based on an iterative computation of σ -values. Let σ^i denote the mapping between A and B after i^{th} iteration. Mapping σ^0 represents the initial similarity between nodes of A and B , which is typically obtained using string comparisons of node labels. In our example we assume that no initial mapping between A and B is available, i.e. $\sigma^0(x, y) = 1.0$ for all $(x, y) \in A \times B$.

In every iteration, the σ -values for a map pair (x, y) are incremented by the σ -values of its neighbor pairs in the propagation graph multiplied by the propagation coefficients on the edges going from the neighbor pairs to (x, y) . For example, after the first iteration $\sigma^1(a_1, b_1) = \sigma^0(a_1, b_1) + \sigma^0(a, b) \cdot 0.5 = 1.5$. Analogously, $\sigma^1(a, b) = \sigma^0(a, b) + \sigma^0(a_1, b_1) \cdot 1.0 + \sigma^0(a_2, b_1) \cdot 1.0 = 3.0$. Then, all values are normalized, i.e., divided by the maximal σ -value (of

current iteration) $\sigma^1(a, b) = 3.0$. Thus, after normalization we get $\sigma^1(a, b) = 1.0$, $\sigma^1(a_1, b_1) = \frac{1.5}{3.0} = 0.5$, etc. In general, mapping σ^{i+1} is computed from mapping σ^i as follows (normalization is omitted for clarity):

$$\sigma^{i+1}(x, y) = \sigma^i(x, y) + \sum_{(a_u, p, x) \in A, (b_u, p, y) \in B} \sigma^i(a_u, b_u) \cdot w((a_u, b_u), (x, y)) + \sum_{(x, p, a_v) \in A, (y, p, b_v) \in B} \sigma^i(a_v, b_v) \cdot w((a_v, b_v), (x, y))$$

The above computation is performed iteratively until the Euclidean length of the residual vector $\Delta(\sigma^n, \sigma^{n-1})$ becomes less than ε for some $n > 0$. If the computation does not converge, we terminate it after some maximal number of iterations. In Chap. 9, we study the convergence properties of the algorithm. The right part of Fig. 7.3 displays the similarity values for the map pairs in the propagation graph. These values have been obtained after five iterations using the above equation. In the figure, the top three matches with the highest ranks are highlighted in bold. These map pairs indicate how the nodes in A should be mapped onto nodes in B .

Taking normalization into account, we can rewrite the above equation to obtain the “basic” fixpoint formula shown in Table 7.3. The function φ increments the similarities of each map pair based on similarities of their neighbors in the propagation graph. The variations A , B , and C of the fixpoint formula are studied in Chap. 9. Our experiments suggest that formula C performs best with respect to quality of match results and convergence speed. In the next section we explain how the fixpoint formulas are derived and present a more general formulation of the flooding algorithm.

Table 7.3. Variations of the fixpoint formula

Identifier	Fixpoint formula
Basic	$\sigma^{i+1} = \text{normalize}(\sigma^i + \varphi(\sigma^i))$
A	$\sigma^{i+1} = \text{normalize}(\sigma^0 + \varphi(\sigma^i))$
B	$\sigma^{i+1} = \text{normalize}(\varphi(\sigma^0 + \sigma^i))$
C	$\sigma^{i+1} = \text{normalize}(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))$

7.3 Generalized Version of the Algorithm

The core of the formal definition of the algorithm is based on the function φ that takes a mapping σ as input parameter and produces mapping θ as output. For any two given models A and B , φ is defined as follows:

$$\varphi(\sigma) = \theta \iff \forall (a, b) \in A \times B: \theta(a, b) = \sum_{(a, p, x) \in A, (b, q, y) \in B} \sigma(x, y) \cdot \pi_r(\langle x, p, A \rangle, \langle y, q, B \rangle) + \sum_{(x, p, a) \in A, (y, q, b) \in B} \sigma(x, y) \cdot \pi_l(\langle x, p, A \rangle, \langle y, q, B \rangle)$$

Function φ describes how the similarity of the neighbor pairs of (a, b) “flows” into the similarity of (a, b) . Function π defines the propagation coefficients for a map pair (x, y) with respect to p -labeled edges in A and q -labeled edges in B . The π -function that corresponds to the example described in Sect. 7.2 is based on inverse-product number of equilabeled edges in A and B computed for each map pair:

$$\pi_{\{l,r\}}(\langle x, p, A \rangle, \langle y, q, B \rangle) = \begin{cases} \frac{1}{\text{card}_{\{l,r\}}(x,p,A) \cdot \text{card}_{\{l,r\}}(y,q,B)}, & \text{if } p = q \\ 0, & \text{if } p \neq q \end{cases}$$

where $\text{card}(x, p, M)$ delivers the number of outgoing or incoming edges of node x that carry label p in model M :

$$\begin{aligned} \text{card}_l(x, p, M) &= |\{(x, p, t) \mid \exists t : (x, p, t) \in M\}| \\ \text{card}_r(x, p, M) &= |\{(t, p, x) \mid \exists t : (t, p, x) \in M\}| \end{aligned}$$

The definitions of functions φ and π use A and B directly without relying on the pairwise connectivity graph. This is a more general approach, since the propagation graph typically contains more information than the connectivity graph. For example, the propagation coefficients obtained using a π -function based on inverse average (described below) cannot be computed using just the connectivity graph. Finally, in the definition of our algorithm we rely on summation and normalization of mappings. These two operations are defined as follows. The sum of mappings σ and ν is a mapping θ such as:

$$\forall (x, y) \in A \times B : \theta(x, y) = \sigma(x, y) + \nu(x, y)$$

The function *normalize* projects all similarity values of a mapping into the range $[0, 1]$. That is, normalization corresponds to dividing vector σ by a scalar value that represents the highest similarity value in σ :

$$\theta = \text{normalize}(\sigma) \iff \forall (a, b) \in A \times B : \theta(a, b) = \frac{\sigma(a, b)}{\max\{s \mid \exists x, y : \sigma(x, y) = s\}}$$

Now we can define the main iteration step of our algorithm. In the version of the algorithm illustrated in Sect. 7.2, on every iteration, a set of new similarity values is computed as follows:

$$\sigma^{i+1} = \text{normalize}(\sigma^i + \varphi(\sigma^i))$$

The above computation is performed iteratively until $\Delta(\sigma^n, \sigma^{n-1})$ satisfies a chosen precision goal for some $n > 0$. To ensure convergence and efficiency (compare Table 9.3), we use a variation of the algorithm shown below:

$$\sigma^{i+1} = \text{normalize}(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))$$

The rationale behind this modification is discussed in Sect. 7.4. Our user study suggests that the faster converging version of the algorithm does not negatively impact the quality of the results.

7.4 Convergence and Complexity of the Algorithm

The fixpoint computation of the similarity flooding algorithm can be expressed as the following eigenvector computation. Let T be the square matrix corresponding to the similarity propagation graph G obtained from models A and B . If there is an edge going from map pair $j = (x, y)$ to $i = (x', y')$ with propagation coefficient c , then let the matrix entry t_{ij} have the value c . Let all other entries have the value 0. Notice that the propagation coefficients in G correspond to transition probabilities if T is a transition matrix.

The fixpoint computation converges when T is an aperiodic, irreducible matrix (Ergodic theorem). Matrix T is irreducible if and only if the associated graph G is strongly connected (every node is reachable from every other node). To ensure these properties, we can introduce self-loops in G by including the summand σ^0 in the fixpoint equation, for example as $\sigma^{i+1} = \text{normalize}(\sigma^0 + \varphi(\sigma^i))$. This approach is also referred to in the literature as dampening. If σ^0 assigns a non-zero value to each map pair in $A \times B$, then adding σ^0 is equivalent to modifying G into G' in which all nodes are interconnected with certain propagation coefficients. Let T' be the matrix associated with G' .

Now the eigenvector computation can be expressed as follows. Let S be a map pair vector that at every position contains a similarity value from σ for a fixed order of map pairs. One iteration of the fixpoint computation corresponds to the matrix-vector multiplication $T' \times S$. Repeatedly multiplying S by T' yields the dominant eigenvector S^* of the matrix T' such as $T' \times S^* = \lambda S^*$, where λ is the dominant eigenvalue of T' . In the fixpoint equation, normalization corresponds to dividing $T' \times S^*$ by λ .

The fixpoint computation corresponds to computing Markov chains over T . This fact provides an interesting insight into the algorithm. Because T corresponds to the transition matrix over the graph G , the obtained similarity measure can be viewed as the stationary probability distribution over map pairs induced by a random walk from pair to pair. This random walk corresponds to a manual matching process performed by a human designer on models A and B . Suppose that only structural information is available to the designer. Starting with a given map pair, the designer infers the similarity of another map pair based on the structural properties of A and B . Consider that A and B are models of relational schemas. If the designer concludes that table t_1 in A matches table t_2 in B , then there is a certain probability that his or her next step will be matching the columns of t_1 to those of t_2 .

The conversion rate of the fixpoint computation depends on the ratio between the dominant and the second eigenvalue of T , which are determined by the structural properties¹ of G' . Higher dampening values contribute to a faster conversion rate of the matrix. For a given precision, using both σ^0

¹ Asymptotic rate of convergence coincides with the so-called spectral radius of the matrix T'

and σ^i in the variation $\sigma^{i+1} = \text{normalize}(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))$ of the fixpoint formula improves the convergence speed by up to a factor of 5 without impeding the quality of the result.

The convergence of the iterations can be measured using the residual vector $R_i = \frac{T' \times S_i}{\lambda_{S_i^*}} - S_i$. We can treat $|R_i|$ as an indicator for how well S_i approximates S^* . For many practical purposes we are only interested in the resulting order of map pairs and not in the absolute values of the similarity coefficients. In such cases, the iterations can be interrupted when the order in a certain subset of a mapping with the highest similarity values has stabilized, i.e. does not change from σ^{n-1} to σ^n . In many practical scenarios, this criterion is already satisfied when $|R_i| < 0.05$.

Let us now turn to the complexity of the algorithm. The number of operations in every iteration of the fixpoint computation is proportional to the number of edges in the propagation graph G . This number is in turn proportional to the product of edge numbers in models A and B . Let N_A and N_B be the number of nodes in A and B , respectively. If nodes in A and B are fully interconnected (every node is directly connected to every other node), the edge numbers in A and B are $O(N_A^2)$ and $O(N_B^2)$. If all these edges are equilabeled, the number of edges in G is $O(N_A^2 \cdot N_B^2)$. That means, the worst case complexity of every iteration is $O(N_A^2 \cdot N_B^2)$, or $O(|A| \cdot |B|)$, where $|A|$ and $|B|$ are the numbers of edges in A and B . However, in many common scenarios, the average complexity of every iteration is $O(N_A \cdot N_B)$. For typical relational or XML schemas the fixpoint computation converges within 5-30 iterations. That means that the running time of the flooding algorithm is comparable to that of a nested loop join in relational databases (multiplied with a small factor).

A straight-forward implementation of the fixpoint computation requires two occurrences of σ -vectors in memory besides σ^0 . The memory usage is important for very large models that may contain parts of dictionaries or classification schemas.

7.5 Features of the Algorithm by Example

In this section we discuss the features and limitations of the similarity flooding algorithm using four matching problems. In Sect. 7.1 we demonstrated how the algorithm performs on two sample relational schemas encoded as directed labeled graphs. Our next example deals with matching of semistructured data instances. After that, we illustrate matching of XML schemas. The third example addresses matching XML schemas using XML instance data. The last example deals with the task of finding related data elements in a database. The goal of our discussion in this section is to illustrate the usefulness of the algorithm and the threshold-based filter defined in the previous section for different application scenarios.

7.5.1 Semistructured Data

Detecting changes by comparing data snapshots in an important task in difference queries, version and configuration management. Fig. 7.4 shows an example borrowed from (Chawathe and García-Molina 1997) that illustrates change detection in two labeled trees. The numbers inside the circles are node identifiers. The tree T_2 on the right has been obtained from the tree T_1 on the left by applying a series of transformation operations. First, all node identifiers have been replaced. In addition, some subtrees have been copied and moved, and a new node (60) has been inserted. In this example, we are interested in finding a best match candidate for every node of T_2 (i.e. a mapping between T_2 and T_1 that satisfies the cardinality constraint $[0, n] - [1, 1]$). We can express the matching procedure using the following script:

1. $product = \text{SFJoin}(T_2, T_1)$;
2. $result = \text{SelectLeft}(product)$;

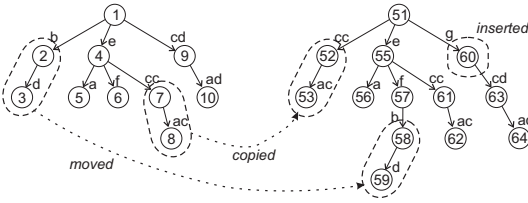


Fig. 7.4. Matching of semi-structured data

Since no initial mapping is passed to SFJoin, the initial similarities between all nodes are set to 1.0. We are using operator SelectLeft instead of SelectThreshold to ensure that all nodes of T_2 are present in the resulting mapping (we discuss filtering in detail in Chap. 8). For every “left” node of the mapping, SelectLeft returns the match candidate with the highest absolute similarity. The *result* of matching is shown in Table 7.4. The fourth column in the table describes the transformation operations performed on the nodes (this information it is not part of the resulting mapping and is provided for illustration only). As the table suggests, the algorithm could correctly map every node in the modified tree T_2 to its previous version in T_1 . Notice a heavy drop in similarity for copied, moved and inserted nodes. This result supports the intuition that exact structural matches should yield higher similarity values.

The right-most columns of the table show the relative similarities of the nodes in T_1 and T_2 . For instance, node 62 is the top candidate for node 8, so $\overleftarrow{\sigma}_{rel}(62, 8) = 1$. For node 53, i.e. the second best candidate, $\overleftarrow{\sigma}_{rel}(53, 8) = \frac{\sigma(53, 8)}{\sigma(62, 8)} = \frac{0.05}{0.30} = 0.16$. If instead of SelectLeft we applied SelectThreshold with any $t_{rel} \in (0.16, 1]$ to the result of SFJoin, we would get all map pairs

Table 7.4. The mapping after applying $SFJoin \circ SelectLeft$ to semistructured data in Fig. 7.4

Similarity	Node $t_2 \in T_2$	Node $t_1 \in T_1$	Operation	$\vec{\sigma}_{rel}(t_2, t_1)$	$\overleftarrow{\sigma}_{rel}(t_2, t_1)$
1.0	55	4		1	1
0.63	61	7		1	1
0.58	51	1		1	1
0.48	56	5		1	1
0.48	57	6		1	1
0.30	62	8		1	1
0.07	52	7	copied	1	0.11
0.07	58	2	moved	1	1
0.07	63	9	moved	1	1
0.05	53	8	copied	1	0.16
0.05	59	3	moved	1	1
0.05	60	1	inserted	1	0.09
0.05	64	10	moved	1	1

for T_1 -nodes that have been either just renamed or moved. Lowering t_{rel} to 0.10 causes all copied nodes to appear additionally in the result. Finally, setting t_{rel} to a value like 0.05 includes the inserted node (but still filters out the rest of total 130 map pairs returned by $SFJoin$). This example illustrates that in certain scenarios undesired results can be pruned quickly by modifying threshold values interactively.

7.5.2 XML Schemas

The next example that we discuss illustrates how our algorithm copes with different choices of graph-based representation for the models to be matched. Consider two XML schemas in Fig. 7.5. The schemas are specified using the

```

<Schema name="Schema 1"
  xmlns="urn:schemas-microsoft-com:xml-data">
  <ElementType name="AccountOwner">
    <element type="Name"/>
    <element type="Address"/>
    <element type="Birthdate"/>
    <element type="TaxExempt"/>
  </ElementType>
  <ElementType name="Address">
    <element type="Street"/>
    <element type="City"/>
    <element type="State"/>
    <element type="ZIP"/>
  </ElementType>
</Schema>

<Schema name="Schema 2"
  xmlns="urn:schemas-microsoft-com:xml-data">
  <ElementType name="Customer">
    <element type="Cname"/>
    <element type="CAddress"/>
  </ElementType>
  <ElementType name="CustomerAddress">
    <element type="Street"/>
    <element type="City"/>
    <element type="USState"/>
    <element type="PostalCode"/>
  </ElementType>
</Schema>

```

Fig. 7.5. Matching of two XML schemas: AccountOwner (S_1) vs. Customer (S_2)

XML schema language deployed on the website biztalk.org designed for electronic documents used in e-business.

As in the example of matching relational schemas (Sect. 7.1), both XML data structures are first converted algorithmically into graphs. Fig. 7.6 shows portions of two different graph-based representations that are frequently used for manipulating XML data structures. The XML graph representation on the left corresponds to that of OEM/Lore (Papakonstantinou et al. 1995), while the representation on the right is based on the XML/DOM standard. In the OEM representation, element tags are treated as edge labels, whereas in DOM representation hierarchical relationships between elements are captured using a uniform edge labels `child`.

The result of matching `AccountOwner` and `Customer` schemas is depicted in Table 7.6. Two left-most columns show the similarity values for computed map pairs. Omitted values indicate that the corresponding map pair does not appear in the match result. For readability, we substituted numeric node identifiers by the descriptions of the objects they represent (in square brackets).

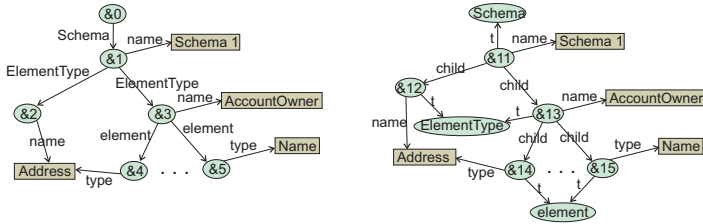


Fig. 7.6. Two different representations of XML data: OEM/Lore-like vs. XML/DOM-like

The mapping for the OEM representation was obtained by executing the script

1. $G_1 = \text{XML2OEMGraph}(S_1)$; $G_2 = \text{XML2OEMGraph}(S_2)$;
2. $\text{initialMap} = \text{StringMatch}(G_1, G_2)$;
3. $\text{product} = \text{SFJoin}(G_1, G_2, \text{initialMap})$;
4. $\text{result} = \text{SelectThreshold}(\text{product})$;

For exploiting the DOM representation, the first line is replaced by

1. $G_1 = \text{XML2DOMGraph}(S_1)$; $G_2 = \text{XML2DOMGraph}(S_2)$;

This example illustrates two features of the algorithm. First, the algorithm produces similar results for different choices of graph-based representation. Second, the example shows that graph-based representations for models that use a wider spectrum of edge labels contributes to a faster iterative computation. The sizes of the graphs in both representations are presented in Table 7.5. Notice that although the graphs for S_1 and S_2 have similar sizes in both representations, the propagation graph in the OEM representation is 50% smaller than that of the DOM-like representation. Thus, every fixpoint iteration takes less time (we discuss the complexity of the algorithm in detail in Sect. 7.4). Also note that the only extra code required for adapting the algorithm for matching XML schemas is the implementation of the `XML2OEMGraph` or `XML2DOMGraph` operator.

Table 7.5. Parameters of the fixpoint computation for S_1 and S_2

Nodes in S_1	Nodes in S_2	Nodes in propagation graph	Iterations
37	39	128	7
40	38	267	6

7.5.3 Matching XML Schemas Using Instance Data

Two previous examples illustrated matching of instance data and matching of schema data. The third example that we discuss in this section deals with

Table 7.6. Match results for XML schemas in Fig. 7.5 using two different graph representations

σ using OEM	σ using DOM	Node in S_1	Node in S_2
1.0		XMLARC	XMLARC
0.82	1.0	[Schema: Schema M1]	[Schema: Schema M2]
0.81	0.55	[Element Type: Address]	[Element Type: CustomerAddress]
0.40	0.25	[element: Street]	[element: Street]
0.40	0.25	[element: City]	[element: City]
0.24	0.33	[Element Type: AccountOwner]	[Element Type: Customer]
0.15	0.13	[element: Name]	[element: Cname]
0.14	0.11	[element: State]	[element: USState]
0.11	0.10	[element: Address]	[element: CAddress]
0.05	0.06	[element: ZIP]	[element: PostalCode]
	0.75	element	element
	0.40	Element Type	Element Type
	0.32	XMLDOM	XMLDOM
	0.32	urn:schemas-microsoft-com:xml-data	urn:schemas-microsoft-com:xml-data
	0.32	Schema	Schema

yet another matching problem, matching XML schemas using instance data. Consider two XML instances depicted in Fig. 7.7. Suppose that the XML tags used in the instances are defined in some schemas (not shown in the figure) and our goal is to establish the correspondences between the tags. The data on the left contains information about a Sony camcorder on the `amazon.com` website. The data on the right shows similar information from the `yahoo.com` website. XML tag names for both schemas were derived from the actual vocabulary terms used on both sites. For example, Amazon site uses term `review`, whereas Yahoo site talks about `rating`. Notice that many text pieces in both XML files are different.

```

<amazon>
  <item>
    <title>Sony DCR-PC100 Digital HandyCam
      Camcorder</title>
    <listPrice>1899.99</listPrice>
    <ourPrice>1699.00</ourPrice>
    <youSave>200.00</youSave>
    <review>
      <avgReview>4.5</avgReview>
      <numOfReviews>20</numOfReviews>
    </review>
    <availability>On Order; usually ships
      within 1-2 weeks</availability>
    <features>
      <zoom>10x optical zoom</zoom>
      <zoom>120x digital zoom</zoom>
      <lcd>2.5 inch LCD</lcd>
      <other>4 MB Memory Stick included</other>
    </features>
  </item>
</amazon>

<yahoo>
  <productInfo>
    <id>Sony DCR-PC100</id>
    <merchantPrice>1799.94</merchantPrice>
    <rating>
      <userRating>3.5</userRating>
      <userReviews>7</userReviews>
    </rating>
    <description>
      <LCDScreenSize>2.5in</LCDScreenSize>
      <opticalZoom>10 X</opticalZoom>
      <special>4MB Memory Stick</special>
    </description>
  </productInfo>
</yahoo>

```

Fig. 7.7. Matching of two XML schemas using instance data in DOM graph representation

Table 7.7 shows how XML tags used in `amazon` and `yahoo` match. This result was determined by running our algorithm on XML/DOM graphs corresponding to both data instances. After that, the match candidates that do not correspond to XML tags were filtered out using a custom operator `XMLMapFilter`:

1. $G_1 = \text{XML2DOMGraph}(db_1); G_2 = \text{XML2DOMGraph}(db_2);$
2. $initialMap = \text{StringMatch}(G_1, G_2);$
3. $product = \text{SFJoin}(G_1, G_2, initialMap);$
4. $result = \text{XMLMapFilter}(product, G_1, G_2);$

Setting the minimal similarity t_{abs} to 0.05 returns a set of correspondences shown above the horizontal bar in the table. Notice that the only additional code required for using the algorithm for matching XML schemas on the basis of instance data was the implementation of operator `XMLMapFilter`.

Table 7.7. Match results for XML element tags in Fig. 7.7 using similarity threshold 0.05

Similarity	Tag in db1	Tag in db2
0.27	item	productInfo
0.20	amazon	yahoo
0.18	zoom	opticalZoom
0.12	features	description
0.11	ourPrice	merchantPrice
0.11	listPrice	merchantPrice
0.09	title	id
0.08	numOfReviews	userReviews
0.07	other	special
0.06	lcd	LCDScreenSize
0.05	review	userReviews
0.04	avgReview	userReviews
0.04	review	rating
0.03	youSave	id
0.03	avgReview	userRating
...

7.5.4 Finding Related Data

One last application that we illustrate in this section deals with finding related data instances. The relatedness information can be computed using the same instance graph for both inputs of the algorithm. Consider the instance graph in Fig. 7.8. This graph captures a piece of information about four faculty members of the Stanford Database Group. The data says that Jennifer works with Hector on the project WHIPS and that she wrote a textbook together with Jeff. Table 7.8 shows the relative similarities between the faculty members. The match result was obtained using the trivial script $result =$

SFJoin(G, G). “Perfect” match candidates with $\vec{\sigma}_{rel} = 1$ like (Gio, Gio) are omitted in the table for brevity. Also, we substituted the identifiers of the faculty members by their names, e.g. &5 by Jennifer. Since relative similarity is not symmetric, Jeff is related to Jennifer closer than Jennifer to Jeff.

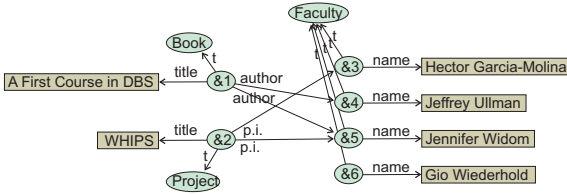


Fig. 7.8. Excerpt of relationships in the Stanford DB Group

Table 7.8. Relatedness of faculty members in the DB group based on data in Fig. 7.8

Faculty	Relative similarity ($\vec{\sigma}_{rel}$)	Faculty
Hector	0.40	Jennifer
	0.14	Jeff, Gio
Jeff	0.40	Jennifer
	0.14	Hector, Gio
Jennifer	0.32	Jeff, Hector
	0.11	Gio
Gio	0.19	Hector, Jennifer, Jeff

Other applications that we used in our experiments include matching of ER, UML and RDFS schemas, comparing product catalogs, approximate queries, matching of service invocations, and matching of mappings. To summarize, notice that the examples that we discussed in this section differ quite a lot from each other. They illustrate diverse application scenarios, the semantics of the nodes in the respective graph representations is different, even the matching goals vary. Common to all these examples is, however, that different matching tasks could be addressed in a uniform fashion using a very limited amount of custom code. In all scenarios, the similarity flooding algorithm could be deployed by providing converters into graph representation for native formats and selecting the desired subsets from the result of SFJoin. These selection techniques, or filtering, is the subject of the next chapter.

The purpose of the examples presented above is to illustrate that the algorithm is applicable to a broad range of matching problems. Of course, the examples do not substitute a comprehensive evaluation. In this thesis, we focus on schema matching and evaluate our algorithm using several schema matching problems in Chap. 9. The effectiveness of the algorithm for instance matching or finding related data remains to be investigated in future work.

8. Filters

“The more alternatives, the more difficult the choice.”

– Abbé d’Allainval (1695-1753)

In this chapter we examine several filters that can be used for choosing the best match candidates from the list of ranked map pairs returned by the Similarity Flooding algorithm. Usually, for every element in the matched models, the algorithm delivers a large set of match candidates. Hence, the immediate result of the fixpoint computation may still be too voluminous for many matching tasks. For instance, in a schema matching application the choice presented to a human user for every schema element may be overwhelming, even when the presented match candidates are ordered by rank. We refer to the immediate result of the iterative computation as *multimapping*, since it contains many potentially useful mappings as subsets.

It is not evident which criteria could be useful for selecting a desirable subset from a multimapping. An additional complication is that as many as 2^n different subsets can be formed from a set of n map pairs. To illustrate the selection problem, consider the match result obtained for two tiny models A and B that is shown on the left in Fig. 8.1 (the models themselves are omitted in the figure for clarity). The multimapping M contains four map pairs with similarities $\sigma(a_1, b_1) = 1.0$, $\sigma(a_2, b_1) = 0.54$, etc. From the set of 4 pairs, $2^4 = 16$ distinct subsets can be selected. Every one of these 16 subsets may be a plausible alternative for the final match result presented to the user.

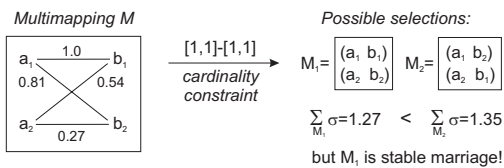


Fig. 8.1. Cumulative similarity vs. “stable marriage”

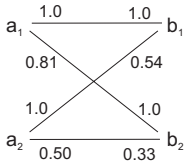


Fig. 8.2. Relative similarities for the example in Fig. 8.1

We address the selection problem using a three-step approach.

- First, as discussed in Sect. 8.1, we use the available application-specific constraints to reduce the size of the multimapping. As exemplified below, typing and cardinality constraints may help to eliminate many map pairs from the multimapping.
- As a second step, presented in Sect. 8.2, we use selection techniques developed in context of matching in bipartite graphs to pick out the subset that is finally delivered to the user.
- At last, we evaluate the usefulness of particular selection techniques for a given class of matching tasks (e.g. schema matching) and choose the technique with empirically best results.

In this chapter, we discuss the first two steps in more detail. In Sect. 8.3, we present an efficient algorithm for computing one of the best-performing filters and discuss its SQL implementation in Sect. 8.4. The evaluation of the selection techniques is presented in Chap. 9.

8.1 Constraints

Frequently, matching tasks include application-specific constraints that can be used for pruning of a large portion of possible selections. Recall our relational schemas S_1 (`Personnel`) and S_2 (`Employee`) from Sect. 7.1. At least two useful constraints are conceivable for this matching scenario. First, we could use a *typing* constraint to restrict the result to only those matches that hold between columns or tables, i.e., we can ignore matches of keys, data types etc. Second, if our goal were to populate the `Personnel` table with data from the `Employee` table, we could deploy a *cardinality* constraint that requires exactly one match candidate for every element of schema S_1 . In this case, the cardinality of the resulting mapping would have to satisfy the restriction $[0, n] - [1, 1]$ (using the UML notation). The right expression $[1, 1]$ limits the number of S_2 -elements that may match each element of S_1 to exactly one (between a lower limit of 1 and an upper limit of 1). Conversely, the left expression $[0, n]$ specifies the valid number of S_1 -match candidates (between 0 and n) for each element of S_2 , i.e., elements of S_2 may remain unmatched or may have one or more match candidates.

Unfortunately, in many matching tasks typing or cardinality constraints do not narrow down the match result sufficiently. To illustrate, consider the

multimapping in Fig. 8.1. If the definition of the matching task implies a cardinality constraint $[0, n] - [1, 1]$ (i.e., the mapping is required to contain exactly one match candidate for every element in A), 4 of 16 selections remain possible. A stricter cardinality constraint $[1, 1] - [1, 1]$ (i.e. one-to-one mapping) limits our choice to two sets of map pairs M_1 and M_2 shown on the right in Fig. 8.1. Even after applying tight constraints in this simple matching task we are still left with more than one choice. Below we examine several strategies for making the decision between the remaining alternatives M_i .

8.2 Selection Metrics

To make an educated choice between M_i 's we need an intuition of what constitutes a “better” mapping. Fortunately, our selection dilemma is closely related to well-known matching problems in bipartite graphs, so that we can build on intuitions and algorithms developed for solving this class of problems (see e.g. (Lovász and Plummer 1986; Gusfield and Irving 1989)). In the graph matching literature, a *matching* is defined as a mapping with cardinality $[0, 1] - [0, 1]$, i.e., a set of edges no two of which are incident on the same node. A *bipartite* graph is one whose nodes form two disjoint parts such that no edge connects any two nodes in the same part. Thus, a mapping can be viewed as an undirected weighted bipartite graph.

A helpful intuition that we will predominantly use for explaining alternative selection strategies for multimappings is provided by the so-called *stable marriage* problem. To remind, in an instance of the stable marriage problem, each of n women and n men lists the members of the opposite sex in order of preference. The goal is to find the best match between men and women. A stable marriage is defined as a complete matching of men and women with the property that there are no two couples (x, y) and (x', y') such that x prefers y' to y and y' prefers x to x' . For obvious reasons, such a situation would be regarded as unstable. Imagine that in Fig. 8.1 elements a_1 and a_2 correspond to women. Then, men b_1 and b_2 would be the primary and the secondary choice for woman a_1 . Obviously, mapping M_1 satisfies the stable marriage condition, whereas M_2 does not. In M_2 , woman a_1 and man b_1 favor each other over their actual partners, which puts their marriages in jeopardy.

The stable-marriage property provides a plausible criterion for selecting a desired mapping from a multimapping. Further candidates for desired mappings can be drawn from the following selection criteria and well-known matching problems:

- The *assignment problem* consists in finding a matching M_i in a weighted bipartite graph M that maximizes the total weight (cumulative similarity) $\sum_{(x,y) \in M_i} \sigma(x,y)$. Viewed as a marriage, such matching maximizes the total satisfaction of all men and women. In Fig. 8.1, $\sum_{M_2} \sigma = 0.81 + 0.54 =$

- 1.35, whereas $\sum_{M_1} \sigma = 1.0 + 0.27 = 1.27$. Thus, M_2 maximizes the total satisfaction of all men and women even though M_2 is not a stable marriage.
- Another group of selection candidates are maximal, maximum and perfect matchings. A *maximal* matching is a matching that is not properly contained in any other matching. A *maximum* matching is a matching of maximum cardinality, i.e., with the most number of married couples. A *perfect* (or *complete*) matching is one containing an edge incident of every node, i.e., the one in which every man and woman is married. Obviously, a perfect matching is achievable only if both parts of a mapping contain the same number of elements. M_1 and M_2 in Fig. 8.1 are maximal, maximum and perfect matchings. All of the above-mentioned matching problems produce $[0, 1] - [0, 1]$ mappings, i.e., monogamous marriages, and can be solved using polynomial-time algorithms (Lovász and Plummer 1986; Motwani and Raghavan 1995).
 - Under *polygamy*, multiple matching counterparts for every element are allowed. Polygamy is useful for matching tasks in which many-to-many mappings are desirable. In schema matching, for instance, an element of one schema may have multiple counterparts in another schema. A polygamous variant of perfect matching corresponds to an *outer match*, i.e., a minimal mapping in which every element in both models has at least one counterpart. When multiple partners are allowed, the number of candidates for every element can be used as an additional factor for selecting the desired subset. For example, we may favor a subset M_i of the multimapping that maximizes function $\sum_{M_i} \frac{\sigma(x,y)}{|(x,?)| \cdot |(? ,y)|}$, analogously to the optimum function used in the assignment problem. Terms $|(x, ?)|$ and $|(?, y)|$ denote the number of partners for woman x and man y in M_i .
 - The flooding algorithm produces at most one similarity value for any map pair (x, y) . We call this value *absolute* similarity. Absolute similarity is symmetric, i.e., x is similar to y exactly as y to x . Under the marriage interpretation, this means that any two prospective partners like each other to the same extent. Considering *relative* similarities suggests a more diversified interpretation. Relative similarities are asymmetric and are computed as fractions of the absolute similarities of the best match candidates for any given element. In the example in Fig. 8.1, b_1 is the best match candidate for a_2 , so we set $\vec{\sigma}_{rel}(a_2, b_1) := 1.0$. The relative similarity for all other match candidates of a_2 is computed as a fraction of $\sigma(a_2, b_1)$. Thus, $\vec{\sigma}_{rel}(a_2, b_2) := \frac{\sigma(a_2, b_2)}{\sigma(a_2, b_1)} = \frac{0.27}{0.54} = 0.5$. All relative similarities for this example are summarized in Fig. 8.2. A multimapping based on relative similarities corresponds to a *directed* weighted bipartite graph. The previously mentioned selection strategies can be adapted to relative similarities in a straightforward way.
 - Some matching tasks require finding a connected subgraph in the target model that matches best the one in the source model. In such case, the

number of edit operations needed to transform one subgraph to another may be included in the selection metric.

- Similarity *thresholds* are the last criteria that we discuss. For a given absolute-similarity threshold t_{abs} we select a subset of a multimapping, in which all map pairs carry an absolute similarity value of at least t_{abs} . For example, for $t_{abs} = 0.5$, Fig. 8.1 suggests that woman a_2 finds man b_1 acceptable (0.54), and would rather not go out with man b_2 at all (0.27). The relative-similarity threshold t_{rel} is used analogously. In the same example, for a relative-similarity threshold $t_{rel} = 0.5$ woman a_2 would still accept man b_2 as a partner, but man b_2 would reject woman a_2 since $\overleftarrow{\sigma}_{rel}(a_2, b_2) = 0.33 < 0.5$.

To summarize, the filtering problem can be characterized by providing a set of constraints and a selection function that picks out the “best” subset of the multimapping under a given selection metric. Conceptually, the selection function assigns a value to every subset of the multimapping. The subset for which the function takes the largest/smallest value is selected as the final result. For example, using the assignment problem as selection metric, we can construct a filter that applies a cardinality constraint $[0, 1] - [0, 1]$ and utilizes a selection function $\sum_{(x,y) \in M_i} \sigma(x, y)$ to choose the best subset. Some selection metrics (e.g., threshold-based ones) can be described in terms of a boolean selection function that assigns the value 1 for one subset of the multimapping, and 0 to all others. In concrete implementations of selection functions, we can often find algorithms that avoid enumerating all subsets of the multimapping and determine the desired subset directly.

In the remainder of this section we describe a filter that produced empirically best results in a variety of schema matching tasks, as we show later in Chap. 9. This approach is implemented in our testbed as the `SelectThreshold` operator. The intuition behind this approach is based on a *perfectionist egalitarian polygamy*, which means that no male or female is willing to accept any partner(s) but the best. This criterion corresponds to using relative-similarity threshold $t_{rel} = 1.0$.

`SelectThreshold` operator selects a subset of the multimapping which is guaranteed to satisfy the stable-marriage property. However, this selection strategy sacrifices the happiness of those individuals who are not number one on the list of at least one person of the opposite sex. Such individuals are left unmarried, i.e., excluded from the mapping. Most of the time, `SelectThreshold` with $t_{rel} = 1.0$ yields matchings, or monogamous societies. In a less picky version of the operator with $t_{rel} < 1.0$, more persons have a chance to find a partner, and polygamy is more likely. In the examples presented in the following section we demonstrate the impact of threshold value t_{rel} in several practical scenarios.

8.3 FilterBest Algorithm

For large graphs, the immediate result produced by the flooding algorithm can be very large. For example, given two graphs with 5,000 equilabeled edges each, the resulting similarity vector contains 25,000,000 elements. Therefore, filtering needs to be done efficiently. In this section, we discuss the efficient implementation of the SelectThreshold filter.

A straightforward approach is to sort all pairs in the order of decreasing similarity and extract the best matching candidates in a single pass over the sorted list of pairs. However, sorting has $O(n \log n)$ complexity. Fortunately, there is a simple algorithm that we call FilterBest, which extracts the desired subset of the mapping in $O(n)$ time.

The algorithm FilterBest is presented below. It takes as input morphism map represented as a list of triples $\langle l, r, \sigma \rangle$ where l and r denote the nodes in the matched graphs and σ is the computed absolute similarity value. The algorithm returns as output a morphism that satisfies the stable-marriage property, i.e., no candidate match is included for a given node if a better candidate is available. Notice that the algorithm FilterBest does not produce a matching in the sense used in (Lovász and Plummer 1986), since some nodes may remain unmatched or have multiple top match candidates.

Algorithm FilterBest(map)
 // map is represented as array of pairs
 $cmap :=$ empty hash table;
 // $cmap$ maps each node to a linked list
 // of candidate nodes of equal similarity
 $rejected :=$ boolean array of size $length(map)$
for $i := 1$ **to** $length(map)$ **do**
 ProbeCandidate($map[i].left, map[i].right, i$);
end for
 clear $cmap$ hash table;
for $i := 1$ **to** $length(map)$ **do**
 ProbeCandidate($map[i].right, map[i].left, i$);
end for
 $result :=$ empty list of pairs;
for $i := 1$ **to** $length(map)$ **do**
 if not $rejected[i]$ **then** add pair $map[i]$ to $result$;
end for
return $result$;

Procedure ProbeCandidate($node, candidate, i$)
 // uses hash table $cmap$ and array $rejected$ from above
 $clist :=$ retrieve linked list of candidates for $node$ from $cmap$;
if $clist$ is empty **then**
 append position i to $clist$;
else if $candidate$ is more similar to $node$ than those in $clist$

```

for each  $j$  in  $clist$  do  $rejected[j] := \mathbf{true}$ ; end for
else if  $candidate$  is less similar
     $rejected[i] := \mathbf{true}$ ;
else //  $candidate$  is equally similar
    append position  $i$  to  $clist$ ;
endif
return;

```

FilterBest computes the result using two passes over the input morphism map , and two auxiliary data structures: a hash table ($cmap$) and a boolean array ($rejected$). The first pass makes sure that the stable-marriage property is satisfied for the left nodes of map , while the second pass verifies the right nodes of map . During each pass, the top seen candidates are stored in the hash table $cmap$, which associates each left/right node with a linked list of top right/left candidate nodes. All candidates contained in each linked list have identical (absolute) similarity. Once a better candidate has been spotted, the currently best candidates are marked as rejected. The rejected candidates will not appear in the final result.

To maintain the rejected candidates efficiently, a single boolean array $rejected$ is used in both passes. Instead of keeping track of lists of rejected candidates, we can simply mark the pairs of the input morphism map as rejected, since whenever a is not a top match candidate for b , b cannot be a match candidate for a due to the perfectionist egalitarian polygamy principle explained above. Since all pairs are passed as a list, they can be marked using a boolean array of the same length as map . A given pair may be rejected in either pass, or in both passes.

After both passes have been done, the resulting morphism can be obtained as a list of non-rejected pairs. The left and right nodes of each pair are guaranteed to be mutually best candidates, otherwise the pair would have been rejected. It is easy to see that the algorithm has the asymptotic complexity of $O(n)$ in the number of pairs of the input morphism. To see that, notice that the procedure ProbeCandidate is called $2n$ times. Appending of the candidates in ProbeCandidate corresponds to pushing elements on a stack, whereas the internal for-loop can be seen as a series of pop operations. Since each pair can be “pushed” and “popped” at most once, the total number of these operations is bound by $n + n = 2n$ in each pass. The last for-loop of FilterBest does another n operations. That is, in the worst case, the algorithm performs $2n + 2n + 2n + n = 7n$ steps.

If the set of left elements of map is disjoint with the set of right elements, both probing for-loops of the FilterBest algorithm can be merged into one as follows:

```

for  $i := 1$  to  $length(map)$  do
    ProbeCandidate( $map[i].left, map[i].right, i$ );
    ProbeCandidate( $map[i].right, map[i].left, i$ );
end for

```

This modification allows saving one iteration over the *map* pairs (yielding $6n$ steps in worst case), but increases the amount of memory used temporarily, since the auxiliary hash table *cmap* has to keep both the candidates for the left nodes and the candidates for the right nodes.

8.4 Expressing FilterBest in SQL

If the input morphism *map* is stored as a single relational table with the attributes *left*, *right*, and *sim*, the filtering can be expressed using a nested SQL query shown below:

```

SELECT map.left, map.right
FROM map,
    (SELECT left AS L, max(sim) AS M
     FROM map
     GROUP BY left) AS T1,
    (SELECT right AS R, max(sim) AS M
     FROM map
     GROUP BY right) AS T2
WHERE map.left = T1.L AND map.right = T2.R AND
      map.sim = T1.M AND map.sim = T2.M
    
```

To understand why this query works, consider a small example depicted in Fig. 8.3. The example shows the input morphism *map* and two intermediate tables, *T1* and *T2*, which correspond to the nested `SELECT` clauses shown above. The table *T1* defined in the first clause uses a group-by statement to extract the maximal similarity values for each left element of the morphism *map*. The second nested `SELECT` clause yields the table *T2* which associates each right element of *map* with its maximal similarity value. In the example, *T1* and *T2* have 2 and 3 rows each, according to the domain and range sizes of *map*.

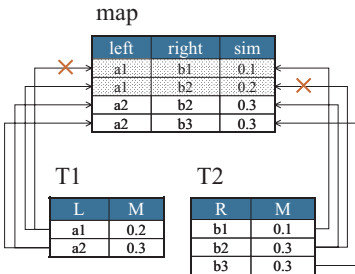


Fig. 8.3. Example illustrating execution of FilterBest in SQL

In the top `SELECT` statement, the tables *map*, *T1*, and *T2* are joined to obtain the final result. The important portion is the `WHERE` clause, which

ensures that each pair in *map* appears in the result only if its similarity value is the maximal similarity value for both the left and right element. In Fig. 8.3, the first row of *map* does not join with the first row of *T1*, because the similarity value *map.sim* is unequal to the value of *T1.M* ($0.1 < 0.2$). Therefore, the pair $(a1, b1)$ does not appear in the result, because there must be a better candidate for *a1* than *b1*, namely *b2*. Tuples $(a2, b2)$, $(a2, b3)$ are produced as the result of the query.

The declarative specification of the SQL query shown above can be executed very efficiently by the query optimizer. In fact, it suggests an alternative linear-time filtering algorithm, which yields the result identical to that of the FilterBest algorithm of Sect. 8.3. First, the maximal similarity values are computed in a single pass over *map* using two hash tables, which play the role of tables *T1* and *T2*. After that, a single pass over *map* is done, during which a lookup in *T1* and *T2* is performed for each pair of *map*. This alternative in-memory algorithm has similar performance characteristics as the FilterBest algorithm.

The SQL specification presented in this section supports filtering the match results backed by the secondary storage. Thus, even very large match results can be filtered efficiently using a database system. The SQL approach could be particularly useful if the Similarity Flooding algorithm is implemented using a set of SQL statements and the results already reside in a database.

9. Evaluation and Tuning

“Evaluation is creation: hear it, you creators! Evaluating is itself the most valuable treasure of all that we value. It is only through evaluation that value exists: and without evaluation the nut of existence would be hollow. Hear it, you creators!”

– Friedrich Nietzsche (1844-1900)

In this chapter, we suggest an accuracy metric for evaluating automatic schema matching algorithms and evaluate the effectiveness of the SF algorithm on the basis of a user study that we conducted.

A crucial issue in evaluating matching algorithms is that a precise definition of the desired match result is often impossible. In many applications the goals of matching depend heavily on the intention of the users, much like the users of an information retrieval system have varying intentions when doing a search. Typically, a user of an information retrieval system is looking for a good, but not necessarily perfect search result, which is generally not known. In contrast, a user performing say schema matching is often able to determine the perfect match result for a given match problem. Moreover, the user is willing to adjust the result manually until the intended match has been established. Thus, we feel that the quality metrics for matching tasks that require tight human quality assessment need to have a slightly different focus than those developed in information retrieval.

The quality metric that we suggest below is based upon *user effort needed to transform a match result obtained automatically into the intended result*. We assume a strict notion of matching quality i.e., being close is not good enough. For example, imagine that a matching algorithm comes up with five equally plausible match candidates for a given element, then decides to return only two of them, and misses the intended candidate(s). In such case, we give the algorithm zero points despite the fact that the two returned candidates might be very similar to what we are looking for. Moreover, our metric does not address iterative matching, in which the user repeatedly adjusts the result and invokes the matching procedure. Thus, the accuracy results we obtain

here can be considered “pessimistic”, i.e., our matching algorithm may be “more useful” than what our metric predicts.

This chapter is structured as follows:

- In Sect. 9.1, we define the metric that we use for evaluating the matching quality, called matching accuracy.
- In Sect. 9.2, we argue that the intended match result needs to be known precisely to evaluate the matching quality.
- The user study in which we gathered intended match results for several tasks is presented in Sect. 9.3.
- The evaluation of the SF algorithm and filters is described in Sect. 9.4.
- In Sect. 9.5, we study the impact of different ways of computing propagation coefficients on overall matching accuracy in the user study.

We conclude the chapter, and Part III, in Sect. 9.6.

9.1 Matching Accuracy

Our goal is to estimate how much effort it costs the user to modify the proposed match result $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$ into the intended result $I = \{(a_1, b_1), \dots, (a_m, b_m)\}$. The user effort can be measured in terms of additions and deletions of map pairs performed on the proposed match result P . One simplified metric that can be used for this purpose is what we call *match accuracy*. Let $c = |P \cap I|$ be the number of correct suggestions. The difference $(n - c)$ denotes the number of false positives to be removed from P , and $(m - c)$ is the number of false negatives, i.e., missing matches that need to be added. For simplicity, let us assume that deletions and additions of match pairs require the same amount of effort, and that the verification of a correct match pair is free. If the user performs the whole matching procedure manually (and does not make mistakes), m add operations are required. Thus, the portion of the manual clean-up needed after applying the automatic matcher amounts to $\frac{(n-c)+(m-c)}{m}$ of the fully manual matching.

We approximate the labor savings obtained by using an automatic matcher as accuracy of match result, defined as $1 - \frac{(n-c)+(m-c)}{m}$. In a perfect match, $n = m = c$, resulting in accuracy 1. Notice that $\frac{c}{m}$ and $\frac{c}{n}$ correspond to recall and precision of matching (Li and Clifton 2000). Hence, we can express match accuracy as a function of recall and precision as follows:

$$\text{Accuracy} = 1 - \frac{(n-c)+(m-c)}{m} = \frac{c}{m} \left(2 - \frac{n}{c}\right) = \text{Recall} \left(2 - \frac{1}{\text{Precision}}\right)$$

In the above definition, the notion of accuracy only makes sense if precision is not less than 0.5, i.e. at least half of the returned matches are correct. Otherwise, the accuracy is negative. Indeed, if more than a half of the matches are wrong, it would take the user more effort to remove the false positives and

Table 9.1. Three plausible intended match results for matching problem in Fig. 7.1

Sparse	Expected	Verbose	Node in G_1	Node in G_2
	+	+	[Table: Personnel]	[Table: Employee]
	+	+	[Table: Personnel]	[Table: Department]
+	+	+	[UniqueKey: perskey]	PrimaryKey: on EmpNo]
		+	[Column: Personnel/Dept]	[Column: Department/DeptName]
		+	[Column: Personnel/Dept]	[Column: Department/DeptNo]
		+	[Column: Personnel/Dept]	[Column: Employee/DeptNo]
		+	[Column: Personnel/Pno]	[Column: Employee/EmpNo]
+	+	+	[Column: Personnel/Pname]	[Column: Employee/EmpName]
+	+	+	[Column: Personnel/Born]	[Column: Employee/Birthdate]

add the missing matches than to do the matching manually from scratch. As expected, the best accuracy 1.0 is achieved when both precision and recall are equal to 1.0. Notice that accuracy is biased towards precision. For example, recall/precision measure (0.7, 0.9) corresponds to accuracy 0.62. This accuracy value is higher than that for (0.9, 0.7), which amounts to 0.51.

9.2 Intended Match Result

Accuracy, as well as recall and precision, are relative measures that depend on the *intended match result*. For a meaningful assessment of match quality, the intended match result must be specified precisely. Recall our example dealing with relational schemas that we examined in Sect. 7.1. Three plausible match results for this example (that we call Sparse, Expected, and Verbose) are presented in Table 9.1. A plus sign (+) indicates that the map pair shown on the right is contained in the corresponding desired match result. For example, map pair (`[Table: Personnel]`, `[Table: Employee]`) belongs to both Expected and Verbose intended results. The Expected result is the one that we consider the most natural one. The Verbose result illustrates a scenario where matches are included due to additional information available to the human designer. For example, the data in table `Personnel` is obtained from both `Employee` and `Department`, although this is not apparent just by looking at the schemas. Similarly, the Sparse result is a matching where some correspondences have been eliminated due to application-dependent semantics. Keep in mind that in the Sparse and Verbose scenarios, the human selecting the “perfect” matchings has more information available than our matcher. Thus, clearly we cannot expect our matching algorithm to do as well as in the Expected case.

Accuracy, precision, and recall obtained for all three intended results using version *C* of the flooding algorithm (see Table 7.3) are summarized in Fig. 9.1. For each diagram, we executed a script like the one presented in Sect. 7.1. The `SelectThreshold` operator was parameterized using t_{rel} -threshold values ranging from 0.6 to 1.0. As an additional last step in the script, we applied operator `SQLDDLMapFilter` that eliminates all matches except those between tables, columns, and keys. As shown in the figure, match accuracy 1.0 is achieved for $0.95 \leq t_{rel} \leq 1.0$ in the Expected match, i.e., no manual adjustment of the result is required from the user. In contrast, if the intended result is Sparse, the user can save only 50% of work at best. Notice that the accuracy quickly becomes negative (precision goes below 0.5) with decreasing threshold values. Using no threshold filter at all, i.e. $t_{rel} = 0$, yields recall of 100% but only 4% precision, and results in a disastrous accuracy value of -2144% (not shown in the figure). Increasing threshold values corresponds to the attempt of the user to quickly prune undesired results by adjusting a threshold slider in a graphical tool.

Fig. 9.1 indicates that the quality of matching algorithms may vary significantly in presence of different matching goals. As mentioned earlier, our definition of accuracy is pessimistic, i.e., the user may save more work as indicated by the accuracy values. The reason for that is twofold. On the one hand, if accuracy goes far below zero, the user will probably scrap the proposed result altogether and start from scratch. In this case, no additional work (in contrast to that implied by negative accuracy) is required. On the other hand, removing false positives is typically less labor-intensive than finding the

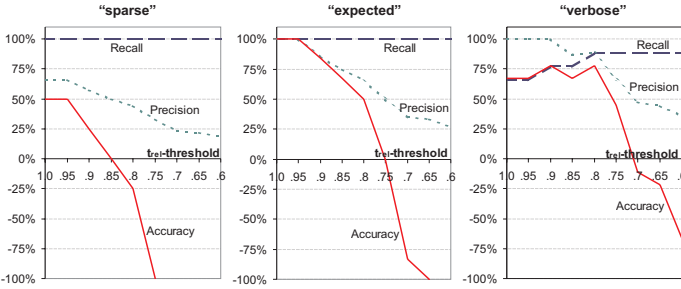


Fig. 9.1. Matching accuracy as a function of t_{rel} -threshold for intended match results Sparse, Expected, and Verbose from Table 9.1

missing match candidates. For example, consider the data point $t_{rel} = 0.75$ in the Expected diagram. The matcher found all 6 intended map pairs (100% recall), and additionally returned 6 false positives (50% precision) resulting in an accuracy of 0.0. Arguably, removing these false positives requires less work than starting with a blank screen.

9.3 User Study

To evaluate the performance of the algorithm for schema matching tasks, we conducted a user study with help of eight volunteers in the Stanford Database Group. The study also helped us to examine how different filters and parameters of the algorithm affect the match results. For our study we used nine relatively simple match problems. The complete specification of the match tasks handed out to the users is in Appendix A. Some of the problems were borrowed from research papers (Miller et al. 2000; Doan et al. 2001; Rahm and Bernstein 2001). Others were derived from data used on the websites like Amazon.com or Yahoo.com. Every user was required to solve tasks of three different kinds (shown along the x -axis of Fig. 9.2):

1. matching of XML schemas (Tasks 1,2,3)
2. matching of XML schemas using XML data instances (Tasks 4,5,6)
3. matching of relational schemas (Tasks 7,8,9)

The information provided about the source and target schemas was intentionally vague. The users were asked to imagine a plausible scenario and to map elements in both schemas according to the scenario they had in mind. No cardinality constraints were given (any $[0, n] - [0, n]$ mapping was accepted). Noteworthy is that almost no two users could agree on the intended match result for a given matching task, even when examples of data instances were provided (tasks 4,5,6). Therefore, we could hardly expect any automatic procedure to produce excellent results. From eight users, one outlier (i.e., the user with highly deviating results) was eliminated. The accuracy in percent

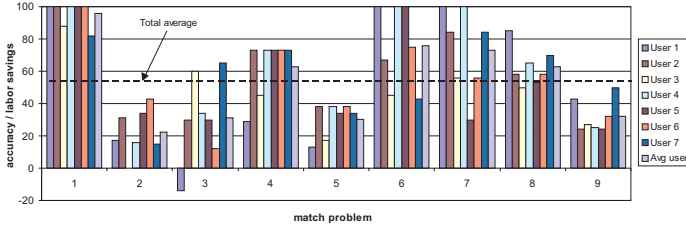


Fig. 9.2. Average matching accuracy for 7 users and 9 matching problems

achieved by our algorithm (using fixpoint formula C) for each of the seven users and every task is summarized in Fig. 9.2. The accuracy metric was used to estimate the amount of work that a given user could save by using our algorithm. The accuracy data was obtained after applying `SelectThreshold` operator with $t_{rel} = 1$. Negative accuracy of -14% in Task 3 indicates that User 1 would have spent 14% more work adjusting the automatic match result than doing the match manually.

Note that in Task 1 the algorithm performed very well, while in Task 2 the results were poor. It turned out that the models used in Task 2 had very simple structure, so that the algorithm was mainly driven by the initial textual match. We did not use any dictionaries for string matching in any of the experiments reported in this chapter. Hence, the synonyms used in Task 2 were considered as plausible matches by humans but were not recognized by the algorithm. The matching accuracy over 7 users and 9 problems averaged to 52%. Hence, our study suggests that for many matching tasks, as much as a half of manual work can be saved using very little application-specific code. This figure is typically even higher in simpler tasks, e.g., when matching two XML documents conforming to the same DTD. Using synonyms may further improve the results of matching. For completeness, the sizes of graphs obtained from schemas used in the study are summarized in Table 9.2.

Table 9.2. Sizes of graphs in the user study

Task	Edges in propagation graph	Edges/arcs in left model	Edges/arcs in right model
T_1	128	35/39	32/37
T_2	313	37/43	40/46
T_3	376	46/46	49/52
T_4	383	55/62	39/44
T_5	309	36/41	48/48
T_6	571	66/55	54/45
T_7	339	33/31	69/55
T_8	1222	113/78	66/51
T_9	594	113/78	32/30

9.4 Evaluation of Algorithm and Filters

Using matching accuracy as the quality measure, we utilized the data collected in the user study to drive our evaluation and tuning of the algorithm for schema matching. As a result of this evaluation, we determined the parameters of the algorithm and the filter that performed best on average for all users and matching problems in our study. The variations of the fixpoint formula that we used are depicted in Table 7.3 (compare Sect. 7.3). Using distinct fixpoint formulas results in different multimappings produced by the algorithm as well as different convergence speed. We then applied different filters to choose the best subsets of multimappings. Fig. 9.3 summarizes the accuracy (averaged over all tasks) obtained for every version of the algorithm and filter that we used. The filters were defined as follows:

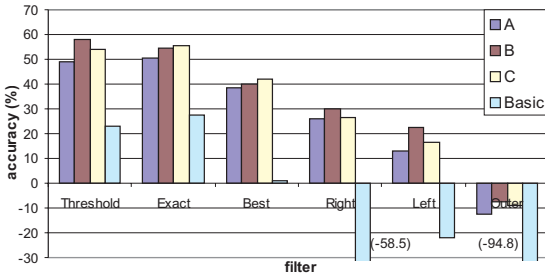


Fig. 9.3. Matching accuracy for different filters and four versions of the algorithm

- *Threshold* filter corresponds to the `SelectThreshold` operator described in Chap. 8. It produces mappings of cardinality $[0, n] - [0, n]$ using relative-similarity threshold $t_{rel} = 1.0$.
- *Exact* is a $[0, 1] - [0, 1]$ version of *Threshold*, which yields monogamous societies.
- *Best* returns a $[0, 1] - [0, 1]$ mapping using a selection metric that corresponds to the assignment problem. The implementation of the filter uses a greedy heuristic. For the next unmatched element, a best available candidate is chosen that maximizes the cumulative similarity.
- *Left* yields a $[0, 1] - [1, 1]$ mapping, in which every node on the left is assigned a match candidate that maximizes the cumulative similarity. *Right* is a $[1, 1] - [0, 1]$ counterpart of *Left*.
- *Outer* filter delivers a $[1, n] - [1, n]$ mapping, in which every node on the left and on the right is guaranteed to have at least one match candidate.

As suggested by Fig. 9.3, the best overall accuracy of 57.9% was achieved using *Threshold* filter with the fixpoint formula *B*. The accuracy of *Threshold* and *Exact* filters lie very close to each other. This is not surprising, since *Threshold* with $t_{rel} = 1.0$ typically produces $[0, 1] - [0, 1]$ mappings. In

our study, *Right* consistently outperforms *Left*, since in most matching tasks the right schemas were smaller; nodes in right schemas were therefore more likely to appear in the intended match results supplied by the users. *Outer* performed worst, since in many tasks only small portions of schemas were intended to have matching counterparts.

We tried to estimate the usefulness of other filters, which are either hard to implement or require extensive computation, by using sampling. For example, a filter that returns a maximal matching (a $[0, 1] - [0, 1]$ mapping with the most map pairs) is apparently not an optimal one for schema matching. Under formula *B*, the total number of map pairs in all tasks after applying the *Best* filter is 101, with associated accuracy of 40%. This accuracy value is lower than 54% obtained using the *Exact* filter that yields only 73 map pairs. Overall, our study suggests that preserving the stable-marriage property is desirable for selecting subsets of multimappings.

Notice that the fixpoint formulae *A*, *B*, and *C* yield comparable matching accuracy for each filter. However, formula *C* has much better convergence properties, as suggested by Table 9.3. The table shows the number n of iterations that were required in every task to obtain a residual vector $|\Delta(\sigma^n, \sigma^{n-1})| < 0.05$. For every fixpoint formula, we executed the algorithm in two versions, “as is” and “strongly connected”. Strongly connected versions guarantee convergence. This effect is achieved by making σ^0 contain positive similarity values (e.g., at least 0.001) for each map pair in the cross-product of nodes of left and right schemas. We found experimentally that the strongly connected versions of the algorithm yielded approximately the same overall accuracy for the filters that preserve the stable-marriage property (*Threshold*, *Exact*, and *Best*). In contrast, enforcing convergence had a substantial negative impact on accuracy for the filters *Left*, *Right*, and *Outer*. For a detailed discussion of convergence criteria please refer to Sect. 7.4.

Table 9.3. Illustration of convergence properties of variations of fixpoint formula for tasks T_1, \dots, T_9 in the user study. Shows iterations needed until length of residual vector got below 0.05.

Fixpoint formula	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	Total
A (as is)	18	48	122	78	∞	12	37	25	25	∞
A (strongly conn'd)	15	56	89	81	1488	18	48	25	31	1851
B (as is)	8	428	17	39	8	13	10	24	21	568
B (strongly conn'd)	7	268	21	32	13	15	14	21	53	444
C (as is)	7	9	9	11	7	7	9	10	9	78
C (strongly conn'd)	7	9	8	11	7	5	9	7	9	72

The formula for computing the propagation coefficients in the induced propagation graph is another important configuration parameter of the flooding algorithm. We experimented with seven distinct formulae and determined the one that performed best in our user study. For the details of this

experiment please refer to Sect. 7.3. The best-performing formula is based on the *inverse average* of equilabeled edges in the graphs to be matched. This approach is similar to the one illustrated in Sect. 7.2, which corresponds to inverse product, and performs only slightly better.

As a last experiment in this section, we study the impact of the initial similarity values (σ^0) on the performance of the algorithm. For this purpose, we randomly distorted the initial values computed by the string matcher. The initial similarities were computed using two versions of a string matcher, one of which took term frequencies into account. Fig. 9.4 depicts the influence of randomization on matching accuracy across all users and matching tasks. For example, randomization of 50% means that every initial similarity value was randomly increased or decreased by x percent, $x \in [-50\%, 50\%]$. Negative similarity was adjusted to zero. It is noteworthy that a randomization factor of 100% introduced accuracy penalty of just about 15%. This result indicates that the similarity flooding algorithm is relatively robust against variations in seed similarities. The dotted lines show another radical modification of initial similarities, in which each non-zero value in σ^0 was set to the same number computed as the average of all positive similarity values. In this case, the accuracy dropped to 30%, which still saves the users on average one third of the manual work.

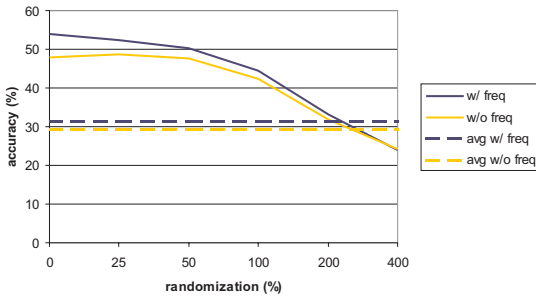


Fig. 9.4. Impact of randomizing initial similarities on matching accuracy

9.5 Propagation Coefficients

The similarity flooding algorithm offers several tuning parameters. One such parameter is the definition of the function π that computes the propagation coefficients in the propagation graph. Above we presented a product-based definition of π that we used to illustrate the algorithm in Sect. 7.2. In our user study we found empirically that an average-based definition of π slightly outperformed the product-based one. The average-based π -formula as well as

another six approaches to computing the propagation coefficients that we examined are summarized in Table 9.4.

For example, the stochastic formula ensures that the sum of propagation coefficients on all edges originating from each map pair in the propagation graph is 1.0. Hence, the transition matrix that corresponds to the propagation graph (see Sect. 7.4) becomes a stochastic matrix, i.e., the entries in each column sum to 1. We evaluated the performance of each π -function listed in the table using the data obtained in the user study. Fig. 9.5 summarizes the accuracy values obtained using different π -functions. In this experiment, we used the fixpoint formula B of Table 7.3 and filters *Threshold* and *Best* to determine the overall average accuracy. We found that the constant π -function, which places weights of 1.0 on each edge of the propagation graph, performs surprisingly well as compared to more sophisticated approaches. We did not extensively examine other π -functions that take into account edge-label similarities, i.e., those that return a non-zero value when $p \neq q$.

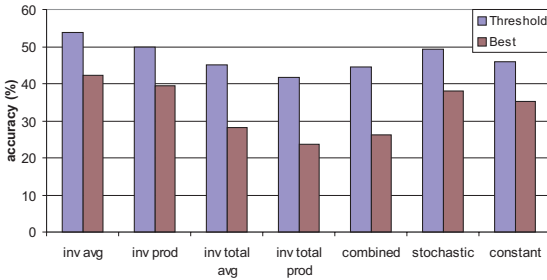


Fig. 9.5. Impact of different ways of computing propagation coefficients on overall matching accuracy in the user study

9.6 Conclusions and Open Issues

In Part III, we presented a simple structural algorithm based on fixpoint computation that is usable for matching of diverse data structures. We illustrated the applicability of the algorithm to a variety of scenarios. We defined several filtering strategies for pruning the immediate result of the fixpoint computation. We suggested a novel quality metric for evaluating the performance of matching algorithms, and conducted a user study to determine which configuration of the algorithm and filters performs best in chosen schema matching scenarios. We discussed the convergence and complexity of the algorithm.

The main results of our study were the following:

- For an average user, overall labor savings across all tasks were above 50%. Recall from Chap. 9 that our accuracy metric gives a pessimistic estimate, i.e., actual savings may be even higher.

Table 9.4. Different approaches to computing the propagation coefficients $\pi_{\{l,r\}}(\langle x, p, A \rangle, \langle y, q, B \rangle)$

Approach	$p = q$	$p \neq q$
inverse average	$\frac{2}{card_{\{l,r\}}(x, p, A) + card_{\{l,r\}}(y, q, B)}$	0
inverse product	$\frac{card_{\{l,r\}}(x, p, A) \cdot card_{\{l,r\}}(y, q, B)}{1}$	0
inverse total average	$\frac{card_{\{l,r\}}(p, A) + card_{\{l,r\}}(q, B)}{2}$	0
inverse total product	$\frac{card_{\{l,r\}}(p, A) \cdot card_{\{l,r\}}(q, B)}{1}$	0
combined inverse average	$\frac{(card_{\{l,r\}}(p, A) + card_{\{l,r\}}(q, B)) \cdot (card_{\{l,r\}}(x, p, A) + card_{\{l,r\}}(y, q, B))}{4}$	0
stochastic	$\frac{\sum_{\forall p'} (card_{\{l,r\}}(x, p', A) \cdot card_{\{l,r\}}(y, p', B))}{1}$	0
constant	1.0	0

- A quickly converging version of the fixpoint formula (C) did not introduce accuracy penalties.
- *Threshold* filter performed best.
- The best formula for computing the propagation coefficients was the one based on inverse average (Sect. 7.3).
- The flooding algorithm is relatively insensitive to “errors” in initial similarity values.

By studying various model-management scenarios, we found that the SF algorithm performs particularly well if the input schemas are two versions of the same schema, with minor variations. In such cases, the algorithm often produces an output that does not need any manual post-processing.

Below we summarize the limitations of the algorithm and several open issues that need to be investigated. This list is by no means exhaustive:

1. The algorithm works for directed labeled graphs only. It degrades when labeling is uniform or undirected, or when nodes are less distinguishable. For example, the algorithm does not perform well for solving the graph isomorphism problem on undirected graphs having no edge labels.
2. Applicability of the algorithm is limited to equityped models. While matching of an XML schema against another XML schema delivers usable results, matching of a relational schema against an XML schema fails.
3. An important assumption behind the algorithm is that adjacency contributes to similarity propagation. Thus, the algorithm will perform unexpectedly in cases when adjacency information is not preserved. For example, in HTML pages nodes that are structurally far away from each other may be displayed visually close. Thus, two cells in an HTML table that are vertically adjacent may be far apart in the document and won't contribute to similarity propagation.
4. The algorithm tends to favor superstructures. Consider graph A containing subgraph A_1 . Let graph B contain a superstructure B_1 such that $A \subset B_1$ and a substructure B_2 such that $B_2 \subset A$. The algorithm would favor B_1 as a match candidate for A , i.e., similarity values between nodes in A and B_1 will be higher than those between A and B_2 .
5. Currently, we do not consider order and aggregation in the algorithm. It is possible that matching of XML schemas could benefit from taking XML features into account.
6. The distribution of the similarity values produced by the fixpoint computation is non-uniform. It may be difficult to combine the algorithm with the matching techniques that rely on absolute similarity values such as those presented in (Do and Rahm 2002).
7. It is unlikely that a standalone version of the algorithm could outperform custom matchers developed for a particular domain. Custom matchers may deploy domain-specific heuristics that are not available to the similarity flooding algorithm (e.g., value ranges, cardinalities, classifiers

etc.). However, the algorithm can make use of custom import and export filters, such as `XMLMapFilter` mentioned in Sect. 7.5, to prototype a first-cut version of a specialized matcher quickly.

A detailed examination of the related work on matching algorithms and evaluation of matching techniques is presented in Sect. 10.2.

10. Related Work

“The secret to creativity is knowing how to hide your sources.”

– Albert Einstein (1879-1955)

The work on metadata management looks back onto over three decades of prolific research efforts ranging from the invention of database schemas (McGee 1959) to database design (Wiederhold 1977), from storing schemas and queries as first-class objects (Stonebraker et al. 1984; den Bussche et al. 1993; Lakshmanan et al. 2001), to transforming them using complex algorithms (Abiteboul et al. 1995; Halevy 2001). Generic model management is, however, a quite recent approach to metadata management. Its goal is to factor out the similarities of the metadata problems studied in the literature and develop a set of high-level operators that can be utilized in various scenarios. The set of operators that we examined was inspired by the vision and model management scenarios presented in (Bernstein et al. 2000b; Bernstein and Rahm 2000; Bernstein 2003) and can be traced back to the early work (Wiederhold 1994).

In this chapter, we survey the literature that motivated the operator definitions, algorithms, and scenarios presented in the dissertation:

- In Sections 10.1-10.5 we examine the major metadata problems that underpin the operators **Merge**, **Match**, **Compose**, **Extract**, and **Diff**. These problems are data integration, schema matching, mapping composition, view selection, and view complement, respectively.
- In Sect. 10.6, we discuss the work that exploited state-based semantics and show how our approach can be viewed in category-theoretic terms.
- In Sect. 10.7, we discuss the metadata management capabilities of today’s repository systems.
- In Sect. 10.8, we review two metadata-intensive applications, declarative mediation and change propagation.
- In Sect. 10.9, we briefly cover other related work such as data translation, mapping tables, and the Z method.

10.1 Data Integration and Merge

Data integration is probably the most widely known metadata-centric area of database research. Data integration comprises a whole range of problems that arise when heterogeneous data needs to be stored or manipulated in a uniform fashion. A characteristic property of data integration scenarios is the presence of a number of heterogeneous schemas, called *local* schemas, and a unified, integrated schema, called *global* schema (Lenzerini 2002). The data itself can be stored either in local databases, in a global database, or at both places. Either the local schemas or the global schema can be used to query or update the data. Depending on where the data actually resides, query and update rewriting may be necessary. When the data is stored in the local databases, the latter are often called (*data*) *sources*.

At least three major data integration scenarios can be distinguished in the literature depending on the integration goals, location of data, target of queries, etc. (Wiederhold 1977; Batini et al. 1986; Davidson et al. 1995a):

- In *view integration*, a global schema is produced for a number of local schemas. The local schemas capture the individual requirements of different user groups. The global schema must be capable of representing the complete information of each local schema to satisfy the requirements of each user group. In other words, each local schema must be definable as a view on the global schema. The users do not adopt the global schema for their applications but run queries and updates through the local schemas defined as views over the global schema, i.e., query rewriting is required. Moreover, in some cases the local databases already contain data before the integration takes place. This data needs to be physically migrated to the global database.
- The aim of *database integration* is to provide a uniform view on a set of local databases, or sources. That is, the data remains at the sources but is queried and updated via the global schema. The global schema may cover all information of the local schemas or, more typically, only a fragment relevant for a particular application.
- In *data warehousing*, data is stored both at the sources, which are called operational databases, and in the global database, or data warehouse. The content of the data warehouse typically contains a portion of the operational data but may also store historical information that is not present in the sources any more. Online analytical processing (OLAP) queries are run against the global database and need not be rewritten. The updates on sources are either propagated to the warehouse in batches or are rewritten and executed directly on the warehouse if the warehouse is incrementally updatable.

The properties of the aforementioned scenarios are summarized in Table 10.1. Letters L and G stand for “local” and “global”, respectively. The

Table 10.1. Data integration scenarios

Scenario	Location of data	Target of queries or updates	Direction of mappings	Coverage of global schema
View integration	(L \rightarrow) G	L	LAV	complete
Database integration	L	G and L	GLAV	partial
Data warehousing	L and G	G and L	GAV	partial

terms LAV, GAV, and GLAV abbreviate local-as-view, global-as-view, and global-local-as-view, respectively. They are discussed in Sect. 10.1.2.

Across all these scenarios, several common tasks can be identified:

1. Constructing a global schema for a given set of local schemas, known as *schema integration*.
2. Constructing mappings between the given global schema and local schemas or among the local schemas, studied in the context of *schema matching*.
3. Answering queries and performing updates on the local databases via the global view or on the global database via the local views. This task has been studied in the context of *answering queries using views* and the *view update* problem.

In this section, we focus on schema integration and answering queries using views. We review the schema matching problem in Sect. 10.2. In Sect. 10.8.1, we consider another aspect closely related to data integration, namely integration of information processing services.

10.1.1 Schema Integration

The operator *Merge* is designed to be used as a principal schema integration operator. We derived the signature and the definition of the semantics of the operator from various approaches to schema integration suggested in the literature.

Inputs and Outputs of Schema Integration. One of the key observations that surfaced in most of approaches is that schema integration is driven by a formal description of the relationship between the local schemas (input schemas). This relationship is called “interrelational dependencies” in (Casanova and Vidal 1983), “integration constraints” in (Biskup and Convent 1986), and “inter-schema correspondences” in (Spaccapietra and Parent 1994). Davidson et al. (1995a) argued that the relationship between the input schemas needs to be specified using a complex mapping language. Thus, our operator *Merge* takes as input the schemas and a mapping between the schemas. The output of the operator includes the mappings between the integrated schema and the input schemas, just as in (Spaccapietra and Parent 1994).

Some approaches to schema integration, such as (Buneman et al. 1992), make a simplifying assumption that the input mapping is given implicitly through syntactic equality of the elements (classes, attributes, etc.) that occur in the input models. Yet other techniques, e.g., (Gotthard et al. 1992; Noy and Musen 2000), allow the engineer to construct the input mapping during the merging process, i.e., include a built-in matching step. Taking a mapping as an input parameter to *Merge* offers a more general approach, which facilitates the development of independent algorithms for *Match* and *Merge* and helps formalizing implicit assumptions. Still, as we point out below, the two operations may sometimes be hard to separate from each other.

Complete vs. Partial Global Schema. Batini et al. (1986) observe that the process of constructing the global schema is often performed in two phases, by first obtaining a union of the source schemas and then performing restructuring operations on the result. The idea behind “unioning” the source schemas is to ensure that the global schema preserves *all* information of the source schemas. As we explained above, this is precisely the objective in the view integration scenarios. We followed this intuition in defining the semantics of the operator *Merge*.

In contrast, in database integration the goal is to provide access to several existing, autonomous databases. In this setting, the global schema may provide any conceivable (partial) view on the local schemas, i.e., its construction is driven by the application requirements and cannot be fully automated. The trend to focus on importing and integrating selected portions of source databases, as in the mediation and federated architectures, has been observed in (Hull 1997). Notice that the selected portions of local schemas can be characterized by defining a view v_i on each local schema s_i such that v_i exposes all information of s_i relevant for the given application. Then, the extracted view schemas can be taken as inputs of the *Merge* operator. Hence, it seems that the objective of capturing all information is fundamental to schema integration and can be exploited even in mediation and federated architectures. In several other schema integration tasks the objective is to construct a global schema that exposes only the “overlapping” information of the source databases. This case corresponds to the operator *Intersect* that we define in Sect. 11.3.3. The distinction between integrating all vs. overlapping information was emphasized and studied in (Buneman et al. 1992; Wiederhold 1994).

Minimality of Global Schema. The purpose of restructuring of the global schema that is done in various methodologies is to eliminate the redundancy caused by the fact that the local schemas are not disjoint and to obtain a “minimal” global schema that still captures the complete content of the local schemas. We formalized this idea in condition (iv) of Definition 4.2.4. A variety of transformation heuristics, rules, or restructuring primitives have been suggested to obtain “smaller” or “better” schemas (Casanova and Vidal 1983; Buneman et al. 1992). A common requirement on such transformation

rules is information preservation. For example, in (Spaccapietra and Parent 1994) the rule definitions are based on the principle that whenever there is a conflict between two structures of the schemas, the integrated schema holds the more unconstrained structure.

Casanova and Vidal (1983) describe a heuristic optimization procedure that tries to reduce redundancy and the size of the output schema produced by merging. They consider a schema language with a rich set of constraints such as inclusion, exclusion and functional dependencies and observe that constructing minimization procedures for such a language is very hard due to the interaction of the dependencies and the complexity of the inference problem for inclusion dependencies.

Uniqueness of Merge Result. Ideally, the outcome of the **Merge** operation should not depend on the particular merge algorithms, on the choice of schema restructuring primitives, or on the order in which the local schemas are merged. In other words, the input schemas and mapping should uniquely determine the merged schema and the mappings from the merged schema to the input schemas. Indeed, the definition of **Merge** given in Sect. 4.2.4 suggests that **Merge** could be a fully automatic operation, provided that all potential conflicts are resolved in the input mapping. And yet, as observed in (Rosenthal and Reiner 1994), much of the research literature on schema integration consists of careful case-by-case heuristics for resolving particular types of mismatches between schemas, such as conditionally mergeable relationships. That is, the result of **Merge** may vary substantially from one methodology to another.

The reason for this discrepancy becomes clearer if we examine the work such as (Casanova and Vidal 1983; Biskup and Convent 1986; Buneman et al. 1992; Rosenthal and Reiner 1994), which give a rigorous theoretical treatment of schema integration. Assuming that the mapping between the input schemas has been agreed upon, a primary source of difficulties seems to be due to the limitations of the schema language used for representing the merged schema.

For example, Biskup and Convent (1986) suggested a notation quite similar to our **Merge** operator to denote the immediate result of schema integration: $Comb(v_1, \dots, v_n, I)$, where v_i are view schemas and I is a set of integration constraints (input mapping). Essentially, $Comb()$ describes a trivial way of merging the schemas by simply including the integration constraints specified in the input mapping into the definition of the merged schema, just as suggested in Theorem 4.2.4. The authors observed that $Comb()$ is in general not a valid database schema because it may contain constraints that cannot be represented in the schema language. The goal of the subsequent manipulation of $Comb()$ is to find a valid schema G that contains no integration constraints and is equally expressive as $Comb()$. They present a pseudocode algorithm for eliminating integration constraints step by step. Since it may not be possible to rewrite all integration constraints into integrity constraints

in G , the authors acknowledge that G may have to be more expressive than $Comb()$. Obviously, there is substantial flexibility in choosing G , which in part explains various heuristics and restructuring primitives that have been examined in the literature.

As another example, consider commutativity and associativity of *Merge*, highly desirable properties (see e.g., (ElMasri 1980)) that are hardly ever satisfied in existing schema integration techniques. Associativity of *Merge* was discussed in (Buneman et al. 1992) for a simple object-oriented schema language with generalization. The authors argued that the merged schemas may contain so-called implicit classes in addition to the classes of the schemas being merged. If these implicit classes are made explicit in the result (similar to replacing an existential formula by a constant), *Merge* becomes non-associative. To overcome this problem, the authors extended their schema language to accommodate the implicit classes. In this example, the limited expressiveness of the schema language was a major obstacle in making merge associative.

Across many techniques, the variability of results of schema integration is often due to the need to materialize the merged schema in some target schema language. The evaluation of the merge algorithms and heuristics proposed in the literature is an open problem, which is largely due to the lack of established quality metrics and formal requirements.

Schema and Mapping Languages. A variety of schema and mapping languages have been used for schema integration. Relational schemas are considered in (Casanova and Vidal 1983; Biskup and Convent 1986). Buneman et al. (1992) use a simple object-oriented schema language. The work in (Miller et al. 1994; Spaccapietra and Parent 1994; Pottinger and Bernstein 2003) uses other quite different schema languages that are reminiscent to the Entity-Relationship (ER) schema language. Noy and Musen (2000) focus on merging of ontologies.

The spectrum of utilized mapping languages is also quite broad. In (Casanova and Vidal 1983; Biskup and Convent 1986; Spaccapietra and Parent 1994), the mapping language consists of a set of element correspondence assertions on entities or relational attributes. The assertions include equality of value set, containment (inclusion dependencies), non-empty intersection, disjointness, or functional dependencies. The morphisms, our mapping language discussed in Sections 2.2.2 and 6.1, can be seen as a simple variant of the languages suggested in these approaches, in which only equality of value sets is utilized.

The mapping language of Pottinger and Bernstein (2003) allows describing the structural relationships between the elements of the input schemas and can be utilized for specifying *a priori* how the structural conflicts are to be resolved. Davidson et al. (1995a) suggested a very powerful mapping language WOL (Well-founded Object Language). They motivate the need for

such a language using schema integration scenarios but do not address the computation of the merge result.

Separability of Match and Merge. The order of the unioning and restructuring phases used in schema integration approaches varies between integration methods. For example, in (Motro 1987) the union is carried out first to obtain a “superview” of the sources, which is then restructured into a final shape. On the other hand, in (Buneman et al. 1992; Spaccapietra and Parent 1994) the information on how the schemas are to be reshaped is part of the input, i.e., in a way restructuring happens before unioning of schemas. In (Gotthard et al. 1992; Noy and Musen 2000), the user is asked questions such as whether two given structures are similar and whether they conflict during the merging process. In (Pottinger and Bernstein 2003) and in our GraphMerge algorithm of Sect. 3.2.7 restructuring information comes in part from the input mapping and in part from the information obtained during semiautomatic conflict resolution.

The input mapping used for schema integration is obtained by matching the input schemas using the operator `Match`. The benefit of separating matching and merging is that these two operators can be studied independently, and the algorithms developed for matching or merging can be used across different approaches to schema integration. However, this separation is not always possible. As we argue in Sect. 11.3.4, in some cases two schemas can be only related by way of a third schema and two mappings. That is, the problem of constructing the global schema may be very closely intertwined with that of matching the local schemas. Moreover, binary mappings may be insufficient to describe the relationships between more than two input schemas. Furthermore, in practice it may be easier to merge schemas step by step, resolving conflicts on demand, as compared to specifying the complete conflict resolution information *a priori* in a matching step. It may however be possible to specify such a stepwise schema integration approach using a model-management script that uses the elementary operators `Extract`, `Match` and `Merge`.

Merging Algorithms. Much of the work on schema integration focused on the design of algorithms that satisfy certain desirable properties, such as information preservation (see e.g. (Spaccapietra and Parent 1994)). In contrast, the approach exploited in Rondo and in (Pottinger and Bernstein 2003) aims at simplifying the *implementation* of the `Merge` operator for various kinds of models. In the GraphMerge algorithm of Sect. 3.2.7 and the algorithm presented in (Pottinger and Bernstein 2003), the input schemas are represented and manipulated as graphs. The attractiveness of this approach is that the algorithms can be easily adapted for new kinds of models, either by tuning the conflict resolution rules (function `conflictsWith` in Sect. 3.2.7) or by encoding the conflict-resolution strategy in the input mapping (e.g., using direction of morphism arcs in Rondo or structure of input mappings in (Pottinger and Bernstein 2003)).

The generic merging algorithm proposed by Pottinger and Bernstein (2003) is similar in spirit to the GraphMerge algorithm. Instead of simple morphisms, their algorithm takes as input a mapping with a complex internal structure, which can be exploited for describing how the structural conflicts of the input models are to be resolved. The authors demonstrate that the algorithms developed in (Spaccapietra and Parent 1994; Noy and Musen 2000) and the GraphMerge algorithm can be implemented by adapting their generic merging algorithm. They observe that an approach based on a common meta-meta model is very flexible and can accommodate virtually all merging techniques proposed in the literature.

Indeed, as long as the schemas and merging rules are relatively simple it is easy to see how a structured mapping and simple conflict resolution rules can drive the transformation of the input schemas represented as graphs into an output schema. However, in presence of non-trivial schema constraints or an expressive mapping language the conflict resolution rules may become arbitrarily complex and the benefit of a generic merging algorithm diminishes. For example, it is unlikely that the algorithm and the axiom system presented in (Casanova and Vidal 1983) can be expressed using simple structural primitives.

10.1.2 Answering Queries Using Views

When the global schema, the local schemas, and the mappings between them are given, the major remaining task is to rewrite the queries stated in terms of the global schema into queries on local schemas, or the other way around. The problem of answering queries using views can be stated in a relatively straightforward fashion in terms of state-based semantics. However, computing the rewritten queries can be extremely hard for concrete mapping languages. The purpose of this section is to illustrate that materializing the mappings that are produced in model-management scripts can be very challenging, yet there has been substantial research that we can build upon. A recent survey of approaches to answering queries using views is presented in (Halevy 2001).

Equivalent and Maximally-Contained Rewritings. In its simplest setting, the problem of answering queries using views can be formulated as follows. Given a view m_x on m , rewrite query q on m into a query q' on m_x such that the result of q' is identical to the result of q or, if this is not possible, the result of q' is a maximal result that is contained in the result of q . Thus, the two major problems studied in the context of answering queries using views are the equivalent rewriting problem and the maximally-contained rewriting problem. Equivalent rewriting is a prerequisite of query optimization (see e.g. (Goldstein and Larson 2001)), whereas in a data integration setting, equivalent rewriting is rarely achievable and one frequently has to settle for maximally-contained rewritings.

An equivalent rewriting exists only when the condition $q = m_{_m_x} \circ \text{Invert}(m_{_m_x}) \circ q$ holds. In this case, the rewritten query q' on m_x is $q' = \text{Invert}(m_{_m_x}) \circ q$. However, in general q' may be non-functional and only a weaker condition $q \subseteq m_{_m_x} \circ \text{Invert}(m_{_m_x}) \circ q$ is satisfied (assuming that $m_{_m_x}$ is total). That is, for a fixed $x \in m_x$ there are multiple *possible* results r_1, \dots, r_k of q' such that $(x, r_i) \in q'$. Each possible result r_i corresponds to executing q on some source state $y_i \in m$ with $(y_i, x) \in m_{_m_x}$. The intersection $r = r_1 \cap \dots \cap r_k$ of all possible results of q' yields the *certain* result for x . The certain result is the maximal result that is contained in each result $q[y_i]$. The query q_c that returns the certain result for each $x \in m_x$ is a maximally-contained rewriting of q .

The concept of a certain answer was originally introduced in (Abiteboul and Duschka 1998) for a relational setting. A certain answer is a tuple that occurs in each possible result r_i . What we call a certain result above corresponds to the set of all certain answers in the relational case. A certain result $r_1 \cap \dots \cap r_k$ may not be equal to one of the possible results r_i , i.e., in general $q_c \not\subseteq \text{Invert}(m_{_m_x}) \circ q = q'$.

Notice that the notion of a certain result depends on a concrete schema language, whereas the notion of a maximally-contained rewriting depends, in addition, on a concrete mapping language. For example, Halevy (2001) observes that a rewriting is maximally-contained only with respect to a specific query language; there can sometimes be a maximally-contained query in a more expressive language that provides more answers. Thus, for certain query languages a maximally-contained rewriting may not yield all certain answers. Similarly, the notion of a certain result depends on the instance-containment relationship which varies among schema languages, as we stress below when we discuss query containment. Thus, just as with other model-management scenarios that we considered, the hardness of the problem of answering queries using views is due to materialization of rewritten queries in concrete mapping languages. For many important schema and mapping languages, the problem is NP-hard or even undecidable (see complexity summary in (Abiteboul and Duschka 1998; Lenzerini 2002)).

In practice, many factors need to be considered for choosing an “optimal” rewriting (if it is computable). For example, instead of a certain result we may be more interested in computing a maximal partial result that can be obtained from a set of distributed sources under given time constraints. Or, if the goal is equivalent rewriting, we may want to find the “cheapest” equivalent rewriting, especially in the context of query optimization.

Query Containment. The problem of finding a maximally-contained query rewriting q_c is closely related to the *query containment* problem (Ullman 1997). A maximally-contained query q_c is defined as a query on m_x that is “contained” in q , denoted as $q_c \subseteq_c q$, and is maximal with respect to \subseteq_c , i.e., $q_c \subseteq_c t \subseteq_c q$ implies $t = q_c$ for each query t . The algorithms for query

containment do not provide a means for computing q_c but help verify whether a given q_c is a candidate rewriting.

Strictly speaking, the query containment problem deals with the containment of query results, rather than queries. Formally, $q_1 \subseteq_c q_2$ iff $\forall x \in m : q_1[x] \subseteq_i q_2[x]$. The result-containment relationship \subseteq_c is defined in terms of the instance-containment relationship \subseteq_i . The relationship \subseteq_i is a transitively closed reflexive relation on $m \times m$ and can be defined in various ways for concrete schema languages. For instances of relational schemas, $x_1 \subseteq_i x_2$ typically indicates that the set of tuples of x_1 is contained in the set of tuples of x_2 . However, \subseteq_i has to be defined differently for relations with multiset semantics, instances of object-oriented database schemas, or XML documents. In these cases, \subseteq_i may be based on a sublist, subtree, or graph embedding relationship on instances.

Several approaches in the literature use alternative, more general notions of query containment. For example, Li et al. (2001) suggest the notion of “p-containment” meaning that the result of a query q_1 p-contained in q_2 can be computed from the result of q_2 using a third query f , i.e., $q_1 = q_2 \circ f$. In a simplest case, which is often assumed for relational schemas, f is a selection query. A similar idea is presented in (Bancilhon and Spyrtos 1981). There, the authors use the condition $\forall x, y \in m : q_2[x] = q_2[y] \Rightarrow q_1[x] = q_1[y]$. Whenever it holds, they say that q_2 “determines” q_1 . Their definition is subsumed by the condition $q_1 = q_2 \circ f$. The advantage of using these more general notions of query containment is that they can be characterized in state-based semantics without appealing to a containment relationship on instances that needs to be defined separately for each concrete schema language.

The complexity of the query containment problem for different query languages, such as conjunctive queries, queries with negation, Datalog, etc. is summarized in (Ullman 1997). For example, containment for conjunctive queries is NP-complete, whereas containment for Datalog programs is undecidable.

GAV, LAV, and GLAV Mappings. In a data integration setting, the relationship between the local schemas and the global schema may be expressed using the so-called LAV, GAV, and GLAV mappings, which are distinguished based on the functional properties of the mappings (compare Table 10.1 on page 165):

- Global-as-view (GAV): there is a view that defines the content of the global schema based on the content of the sources, as in data warehousing.
- Local-as-view (LAV): for each local schema L , there is a view on the global schema that defines the content of L , as in view integration.
- Global-local-as-view (GLAV): the relationship between the local schemas and the global schema is established using a combination of GAV and LAV assertions, i.e., the mappings between the local schemas and the global

schema and their inverse mappings may be non-functional. This is a general case in database integration.

Distinguishing between GAV, LAV, and GLAV mappings led to different query rewriting algorithms with varying computational properties.

Sound, Complete, and Exact Views. Another distinction often made in the literature is that between sound, complete, and exact views (Lenzerini 2002). Strictly speaking, this terminology finesses the fact that different, non-functional mapping languages are used to characterize the relationship between the global schema and the local schemas. To illustrate, let v be a view on m , i.e., v is a total functional mapping from m to m' . Assume that we are given a valid “snapshot” of the application state, i.e., we have an instance $x \in m$ of the source schema and an instance $y \in m'$ of the view. The view v is said to be exact when for all such snapshots $v[x] = y$, sound when $v[x] \supseteq_i y$ and complete when $v[x] \subseteq_i y$. The relationship \subseteq_i describes containment of instances, as explained above. (The case $v[x] \supseteq_i y$ is referred to as “open-world assumption” in (Abiteboul and Duschka 1998).)

Technically, each of the above “views” can be represented as a mapping *map* defined as $map = v$ for exact views, $(x, y) \in map \iff v[x] \supseteq_i y$ for sound views, and $(x, y) \in map \iff v[x] \subseteq_i y$ for complete views. For example, let $v = \langle\langle \pi_A(R) = S \rangle\rangle$. A mapping that describes v as a sound view is $map = \langle\langle \pi_A(R) \supseteq S \rangle\rangle$, i.e., the mapping holds if and only if all tuples of S are contained in $\pi_A(R)$. This mapping is non-functional.

In this section, we have shown that several well-known hard problems can be characterized relatively easily using the state-based approach. What makes these problems hard is the need to express the mappings and models whose properties are specified in an abstract fashion using concrete languages. This is the objective of materialization (see Sect. 4.3). Materialization seems to be one of the major upcoming challenges for model management. Fortunately, there is excellent prior work to build on.

10.2 Schema Matching and Match

Similarity Flooding. In the model-management prototype that we presented in this dissertation, the operator Match is implemented using the Similarity Flooding (SF) algorithm of Chap. 7. In designing the SF algorithm and the filters, we borrowed ideas from three research areas. The fixpoint computation corresponds to random walks over graphs (Motwani and Raghavan 1995), as explained in Sect. 7.4. A well-known example of using fixpoint computation for ranking nodes in graphs is the PageRank algorithm used in the Google search engine (Brin and Page 1998). Unlike PageRank, our algorithm has two source graphs and extensively uses and depends on edge labeling. The filters that we proposed for choosing subsets of multimappings are based on the

intuition behind the class of stable marriage problems (Gusfield and Irving 1989). General matching theory and algorithms are comprehensively covered in (Lovász and Plummer 1986).

Since the SF algorithm was published (Melnik et al. 2002), a number of other research efforts that exploit a similar idea have emerged in the literature. For example, Jeh and Widom (2002) examine the case where all arcs bear the same label and represent document citations links. Anyanwu and Sheth (2002) use the intuition of contextual similarity to discover associations on the Semantic Web. Noy and Musen (2002) present an algorithm for comparing ontology versions, which uses fixpoint computation. Goldstone and Rogosky (2002) exploit unlabeled relations in graphs to translate across conceptual systems using the intuition of contextual similarity that underlies the SF algorithm. They draw very intriguing conclusions regarding cognitive processes that take place in establishing relationships between different conceptual models. In particular, their work supports the claim made by some philosophers and cognition scientists that it is often possible to translate between the concepts of two conceptual systems by exploiting only intrinsic, within-system relations and no or little extrinsic grounding.

We investigated the application of the SF algorithm for schema matching. However, SF is a general-purpose graph matching algorithm and graph matching has many other applications. As we observed in Chap. 7, the algorithm may be utilized for computing schema correspondences using instance data, and for finding related elements in data instances. In fact, matching of data instances is a promising application area. For example, consider two CAD files or program scripts that have been independently modified by several developers. In this scenario, matching helps to identify moved or modified elements in these complex data structures. In bioinformatics, matching has been used for network analysis of molecular interactions (Ogata et al. 2000; Kanehisa 2000). In this domain, data instances represent e.g. metabolic networks of chemical compounds, or molecular assembly maps. Matching of molecular networks and biochemical pathways may help predict metabolism of an organism given its genome sequence. We are aware of ongoing work by other researchers who are applying a variation of the SF algorithm to data structures used in computer graphics, semantic integration of spatial data, and bioinformatics.

Related Approaches. The SF algorithm is only one possible implementation for the `Match` operator. Various systems and approaches have recently been developed to determine mappings between schemas (semi-)automatically, e.g., Autoplex (Berlin and Motro 2001), Automatch (Berlin and Motro 2002), Clio (Yan et al. 2001; Miller et al. 2001), COMA (Do and Rahm 2002), Cupid (Madhavan et al. 2001), Delta (Clifton et al. 1997), DIKE (Palopoli et al. 2003), EJX (Embley et al. 2001), GLUE (Doan et al. 2002), LSD (Doan et al. 2001), MOMIS (and ARTEMIS) (Bergamaschi et al. 2001; Castano and Antonellis 1999), SemInt (Li and Clifton 2000), SKAT (Mitra et al. 1999), and

TranScm (Milo and Zohar 1998). While most of them have emerged from the context of a specific application, a few approaches (Clio, COMA, Cupid) try, just as SF, to address the schema matching problem in a generic way that is suitable for different applications and schema languages. A taxonomy of automatic match techniques and a comparison of the match approaches followed by the various systems is provided in (Rahm and Bernstein 2001). According to their taxonomy, Similarity Flooding can be classified as a structural, 1:1-local, m:n-global matching algorithm.

To identify a solution for a particular match problem, it is important to understand which of the proposed techniques performs best, i.e., can reduce the manual work required for the match task at hand most effectively. To show the effectiveness of their system, the authors have usually demonstrated its application to some real-world scenarios or conducted a study using a range of schema matching tasks. Unfortunately, the system evaluations were done using diverse methodologies, metrics, and data making it difficult to assess the effectiveness of each single system, not to mention to compare their effectiveness. Furthermore, the systems are usually not publicly available making it virtually impossible to apply them to a common test problem or benchmark in order to obtain a direct quantitative comparison.

To obtain a better overview of the current state of the art in evaluating schema matching approaches, we reviewed the recently published evaluations of the schema matching systems in (Do et al. 2002). There, we introduced and discussed the major criteria influencing the effectiveness of a schema matching approach, e.g., the chosen test problems, the design of the experiments, the metrics used to quantify the match quality and the amount of saved manual effort. Apart from the Cupid evaluation, which represents the first ever effort to evaluate multiple systems on uniform test problems, the problems used in other approaches originate from very different domains of varying complexity. While some evaluations used simple match tasks with small schemas and few correspondences to be identified, several systems also showed high match quality for somewhat more complex real-world schemas (COMA, LSD, GLUE, SemInt). Some evaluations, such as Autoplex, Automatch, completely lack the description of their test schemas. Unlike other systems, Autoplex, Automatch and LSD perform matching against a previously constructed global schema. All systems return correspondences at the element level with similarity values in the range of $[0,1]$. Except for SemInt, correspondences are of 1:1-local cardinality (using the taxonomy of (Rahm and Bernstein 2001)), providing a common basis for determining match quality.

As we have discussed in Chap. 9, matching is a subjective operation and there is not always a unique result. The evaluation that we presented in this dissertation is the only matching evaluation we are aware of that took into account the subjectivity of the user perception about required match correspondences. The schemas utilized in the user study are in Appendix A.

Previously proposed metrics for measuring the matching accuracy (Li and Clifton 2000; Doan et al. 2001) did not consider the extra work caused by wrong match proposals. Our quality metric, matching accuracy, which we used for evaluating the SF algorithm is related to the precision/recall metrics developed in the context of information retrieval. It has been used in subsequent work by (Do and Rahm 2002) under the name “Overall” metric. Our metric is similar in spirit to measuring the length of edit scripts as suggested in (Chawathe and García-Molina 1997). However, we are counting the edit operations on mappings, rather than those performed on models to be matched. Another source of extra work is additional preparing and training effort. SF and SemInt do not require any such pre-match effort, unlike other approaches such as the use of neural networks (Li and Clifton 2000) or machine learning techniques (Doan et al. 2001).

Open Issues and Promising Techniques. Schema matching is an AI-complete problem, i.e., it is as hard as simulating human intelligence. Thus, there is little hope that we will be able to automate it fully anytime soon. Although semiautomatic techniques can be quite useful in many scenarios, in many other applications automated schema matching is less effective and matching can rather be compared to a complex design task. Development of powerful GUI tools is essential to support such design tasks.

One of the hardest open problems in schema matching is the computation of *mapping expressions* that describe value transformations and structural manipulations of data, such as aggregation or transposition. Mapping expressions are needed to make the mappings operational, i.e., to map instances of schemas from one representation into another. They can be used to generate SQL views, XSL transformations, or Java programs that can be directly executed. Specifying mapping expressions is typically a much more expensive step as compared to finding schema correspondences. In addition, it is very hard to automate. With the exception of Clio (Miller et al. 2000; Miller et al. 2001), an overwhelming majority of schema matching approaches focus on determining schema correspondences. In Clio, after the element correspondences have been uncovered, the engineer is presented an exhaustive list of possibilities of computing the joins over the source schema and can select the desired ones by examining the data samples that are judiciously chosen by the system. The formulas describing value transformations have to be entered manually by the engineer. The recent work by Brown and Haas (2003) may be helpful for identifying such formulas.

One of the most promising techniques in schema matching is *reuse* of existing mappings by composition. The effectiveness of this approach for finding schema correspondences was first studied in (Do and Rahm 2002). A major benefit of reuse is that the technique could also be deployed to compute mapping expressions. To our best knowledge, reuse of mapping expressions has not been addressed in the literature yet. We illustrate and define the problem of reuse by composition at the end of this section. Mapping adap-

tation, which can be seen as a variant of reuse, is considered in (Velegarakis et al. 2003). We discuss that approach in Sect. 10.8.2. The problem of how the schema element labels, which are reused across different websites, can be utilized to drive the schema matching task between n schemas is examined in (He and Chang 2003). Their work considers the schema matching problem in the context of integrating the so-called Hidden Web databases.

Exploiting instance data offers another range of promising schema matching techniques. In addition to Bayesian learners (LSD/GLUE) or neural networks (SemInt), schema induction can be utilized to derive a more detailed description of the underlying data. This approach is especially valuable if the schemas are generic, such as entity-attribute-value tables that store heterogeneous information using just a few tables (Agrawal et al. 2001b), or when schemas are missing entirely, e.g., in semistructured databases. Data mining techniques such as those presented in (Nestorov et al. 1998) can help derive schemas from instance data. A recent work exploiting statistical correlation of instance data is presented in (Kang and Naughton 2003).

Reuse in Schema Matching. We illustrate and formalize the problem of mapping reuse. The key ideas that we exploit were presented in (Rahm and Bernstein 2001; Do and Rahm 2002; Halevy et al. 2003; Kementsietsidis et al. 2003). Informally, the problem of mapping reuse can be stated as follows: given a repository of schemas and mappings, derive the mapping between two given schemas using the mappings contained in the repository. Below, we specify the problem using two operators, composition and confluence.

Do and Rahm (2002) observed that given two mappings $m_1_m_2$ and $m_2_m_3$, we can compute the mapping between m_1 and m_3 by composing the two mappings. They suggested the use of composition as a heuristic and argued that composition may yield incorrect results when the transitivity assumption of element correspondences does not hold. Their conclusion, however, describes a property of the mapping language they used, and not a property of the composition operation.

The result of composition $m_1_m_2 \circ m_2_m_3$ defines a correct mapping between m_1 and m_3 , although this mapping is typically incomplete. In general, any k -way composition “path” that connects m_p and m_q describes a partial mapping between m_p and m_q . A more precise mapping between m_p and m_q can be obtained by aggregating several partial mappings using the Confluence operator. To illustrate, consider the following example.

Example 10.2.1. Assume that the following schemas and mappings are contained in the repository:

$$\begin{aligned} m_1 &= \langle R_1, S_1, T_1 \rangle \\ m_2 &= \langle R_2, S_2 \rangle \\ m_3 &= \langle T_3 \rangle \\ m_4 &= \langle R_4, S_4, T_4 \rangle \\ m_1_m_2 &= \langle R_1 = R_2, S_1 \subseteq S_2 \rangle \end{aligned}$$

$$\begin{aligned} m_2_m_4 &= \langle\langle R_2 = R_4, S_4 \subseteq S_2 \rangle\rangle \\ m_1_m_3 &= \langle\langle T_1 \subseteq T_3 \rangle\rangle \\ m_3_m_4 &= \langle\langle T_3 \subseteq T_4 \rangle\rangle \end{aligned}$$

For example, mapping $m_1_m_2$ states that relations R_1 and R_2 have the same extensions, while the tuples of S_1 are a subset of tuples of S_2 . The goal is to infer the “best” mapping between m_1 and m_4 based on the information in the repository. Observe that one path between m_1 and m_4 can be obtained via m_2 as

$$m_1_m_2 \circ m_2_m_4 = \langle\langle R_1 = R_4 \rangle\rangle$$

The composition over m_3 yields another partial mapping

$$m_1_m_3 \circ m_3_m_4 = \langle\langle T_1 \subseteq T_4 \rangle\rangle$$

The confluence of both composition paths yields a “maximal” mapping between m_1 and m_4 derivable from the repository

$$\begin{aligned} map &= (m_1_m_2 \circ m_2_m_4) \oplus (m_1_m_3 \circ m_3_m_4) \\ &= \langle\langle R_1 = R_4, T_1 \subseteq T_4 \rangle\rangle \quad \blacksquare \end{aligned}$$

Formally, we state the reuse problem as follows: given a repository of schemas and mappings, infer a maximal mapping between two given schemas m_p and m_q , defined as

$$m_p_m_q = \bigoplus_i (map_{i_1} \circ \dots \circ map_{i_k})$$

where $\text{Domain}(map_{i_1}) \subseteq m_p$ and $\text{Range}(map_{i_k}) \subseteq m_q$.

Notice that $m_p_m_q$ is in general still a partial mapping. For instance, in Example 10.2.1 it may well be the case that the exact mapping between m_1 and m_4 is $\langle\langle R_1 = R_4, T_1 = T_4 \rangle\rangle$. The equality $T_1 = T_4$ may not be derivable from the repository, but only a weaker condition $T_1 \subseteq T_4$. The maximal inferable mapping may still require post-processing by a human engineer, since it is guaranteed to be correct albeit not complete.

The reuse scenario presents a number of computational challenges. Examples are avoiding the computation of redundant composition paths or finding the best order of executing the composition, similar to finding an optimal query plan for multiple joins. Also, some mappings stored in the repository may have been obtained using model management scripts, in which case it may be possible to optimize the computation of the maximal inferable mapping even further.

10.3 Mapping Composition and Compose

Composition is a fundamental algebraic operation. It plays a key role in many different areas of mathematics and theoretical computer science and is considered a basic axiomatic abstraction in category theory (compare Sect. 10.6.3).

In database research, mapping composition has been primarily considered for queries, i.e., functional mappings. For example, answering a query stated against a virtual, non-materialized view amounts to composing the query with the view definition. In relational algebra, each relational operator such as projection or selection describes a database transformation. Thus, a formula expressed in relational algebra, such as $\pi_A(\sigma_{B=5}(R \bowtie S))$, can be viewed as a composition of several elementary transformations. Composition of data transformations was considered as early as in (Shu et al. 1977; Paolini and Pelagatti 1977; Borkin 1978; Bancilhon and Spyrtatos 1981). Commutativity of elementary transformations provides a foundation for query optimization. Abiteboul et al. (1995) note that relational algebra queries are closed under composition, i.e., the result of composing two relational algebra queries can always be expressed as another relational algebra query.

Composition is likely to be the most frequently used operation in model-management scripts. Composition implemented in Rondo respects the state-based semantics of the operator suggested in Sect. 4.2.1. Although our implementation focuses on morphisms, composition is a truly generic operation, which has been utilized in many scenarios and for various kinds of mapping languages. For example, in GUI tools composition of the operations recorded in the undo history can be viewed as a compensating transformation for user editing operations. In database recovery, the effects of multiple updates are composed to speed up and ensure the correctness of recovery. In (Spaccapietra and Parent 1994; Rosenthal and Reiner 1994; Atzeni and Torlone 1996) composition is performed on schema transformations rather than on instance transformations and is exploited to study soundness and completeness of rewriting rules. Bernstein and Rahm (2000) illustrate how composition can be exploited in data warehousing scenarios.

A significant application of composition, which has gained importance with the advent of XML, XQuery, and XSLT, consists in composing XQuery or XSLT queries over XML views on relational data with the views and pushing it down for query optimization in the relational engine. As reported in (Fernandez et al. 2002), three commercial XML publishing systems, Oracle XML SQL Utility, IBM DB2 XML Extender, and Microsoft SQL Server, support such query composition. Although querying XML publishing views can be considered a query optimization problem, the work on this application produced algorithms that could be utilized for implementing composition (and decomposition) of expressions in different mapping languages in a model-management system.

We explain the general principles behind the aforementioned application using the work (Shanmugasundaram et al. 2001a) as an example. Their approach is illustrated in Fig. 10.1 (presentation in the figure differs from that used in the paper). There, a publishing view ($v_{\text{def}} \circ v_{\text{user}}$) over relational data is defined using a composition of a so-called default XML view v_{def} and a user-defined view v_{user} expressed in XQuery. The queries q are run against

such publishing views. To push down the queries, the following approach is used. First, a query is composed with the publishing view. That is, a 3-way composition of the query with a user view and a default view is computed as $v_{\text{def}} \circ v_{\text{user}} \circ q$. The result of this 3-way composition is represented internally as a so-called XML Query Graph Model, q_{XQGM} . Then, this internal mapping is decomposed into two transformations: a SQL query q_{SQL} and a so-called tagger graph v_{tag} (expression in yet another language that generates an XML document from the query results delivered by the relational engine). The purpose of this decomposition is to push down data and memory intensive computation to the underlying relational engine. Thus, a user query is processed by creating an intermediate mapping using a 3-way composition and then translating it into an equivalent mapping expressed as a 2-way composition. The examined subset of the XQuery language supports nested expressions and nested order, while the presented view composition technique is shown to be complete and produce minimal SQL queries. Fernandez et al. (2002) present a very similar approach (there, default views are called canonical views).

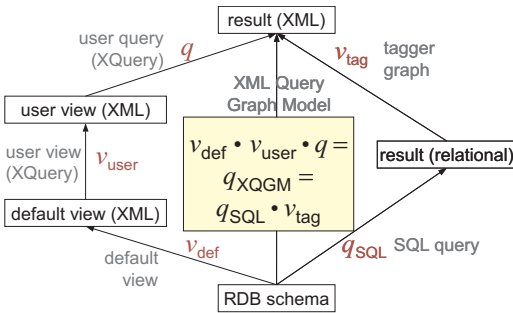


Fig. 10.1. Use of composition in (Shanmugasundaram et al. 2001a)

More recently, Li et al. (2003) investigated composition of XSL transformations, instead of XQuery transformations, with XML publishing views. Composition of transformations on semistructured data was explored in (Papakonstantinou and Vassalos 1999).

Madhavan and Halevy (2003) depart from composing purely functional mappings and study composition for a GLAV language, which combines the global-as-view (GAV) and local-as-view (LAV) formalisms. In the mapping language they consider, a mapping is expressed as a set of formulas of the kind $Q_A \subseteq Q_B$, where Q_A and Q_B are conjunctive queries over two schemas. They show that in this setting the mapping produced as result of composition may not be representable using a finite expression. However, the composition turns out to be finite for a useful subset of the considered language. The authors present an algorithm for obtaining a minimal composition and establish its complexity bounds.

The works (Bernstein and Rahm 2000; Bernstein 2003) give a structural definition of composition, which assumes that mappings are represented as directed acyclic graphs. They distinguish several variants of composition, such as left/right and outer composition. It is difficult to assess how their definitions relate to the state-based composition (Definition 4.2.1), since the focus of their work is on structural semantics and no instance-level characterization of the mapping language that they deploy is presented.

10.4 View Selection and Extract

The intuition that we exploit in defining the operator **Extract** is closely related to the problem of *view selection* in data warehouse design. The view selection problem can be stated as follows (Theodoratos et al. 2001; Chirkova et al. 2001): given a set of workload queries over a database schema, select a view to materialize in the data warehouse such that:

1. All queries can be answered using the materialized view,
2. The warehouse design is optimal with respect to a certain cost metric (e.g., combination of query evaluation and maintenance cost),
3. All operational constraints are satisfied (e.g., the warehouse fits into the available storage space).

If the set of queries to be supported is limited to one query, condition (1) can be stated using the operator **Extract**. More precisely, it corresponds to the materialization of **Extract** (compare Sect. 4.3), since it lacks the minimality constraint. Conditions (2)-(3) can be seen as tuning knobs that drive the materialization. In fact, condition (1) has been considered in more depth in the context of answering queries using views (see Sect. 10.1.2), while the warehouse design literature focused primarily on practicability of the design, i.e., conditions (2)-(3).

Various kinds of algorithms and query rewriting techniques have been suggested for view selection, including randomized algorithms and heuristic approaches. In (Theodoratos et al. 2001), queries are combined into a so-called multiquery graph. Multiquery graphs are rewritten using a set of transformation rules, such that each rule preserves condition (1), i.e., is sound. The authors also prove completeness of the presented rules. Chen et al. (2002) introduce another data structure, a so-called merging tree that is used to combine the candidate views derived from the workload. The algorithms for obtaining minimal views were presented in (Li et al. 2001).

Important theoretical results were presented by (Chirkova et al. 2001). In particular, they study the cardinality of resulting view configurations and establish lower and upper complexity bounds for the problem. Many, if not most, approaches to view selection focus on select-project-join queries for relational schemas under set semantics. Multiset semantics and group-by queries

are considered in (Agrawal et al. 2001a; Chen et al. 2002). In a more recent work, Gupta et al. (2003) addressed the view selection problem for XML schemas in content-based routing.

Although using materialized views for query optimization is a relatively old idea, it has only recently been adopted in commercial database systems. Agrawal et al. (2001a) developed a tool that recommends materialized views and indexes and examines tradeoffs between using indexes and materialized views for a given query workload. The tool ships with Microsoft SQL Server 2000.

A major source of complexity of the view selection problem originates from the fact that workload queries can interact in various ways. The operator **Extract** that we presented takes only one mapping as input. Although it would be easy to extend the definition of the operator for n input mappings, we conjecture that the n -ary case can be expressed by a combination of existing operators. We consider this aspect in more detail in Sect. 11.3.3.

10.5 View Complement and Diff

The operator **Diff** generalizes the notion of a *view complement* studied extensively in the context of data warehousing. The notion of view complement was introduced in the groundbreaking work by Bancilhon and Spyrtos (1981) who used it as a vehicle for a formal treatment of the view update problem (Dayal and Bernstein 1978). Two views are complementary if given the state of each view, there is a unique corresponding state of the source database. That is, the two views are sufficient to reconstruct the database. Recently, it has been shown (Laurent et al. 2001) that view complements can be exploited to guarantee desirable data warehouse properties such as independence and self-maintainability (a data warehouse view is called self-maintainable if the updated warehouse can be computed directly given the reported changes in the sources without additional maintenance queries). The theoretical properties of view complements were studied in (Keller and Ullman 1984; Hegner 1994). The computability of view complements was examined in (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001; Lechtenbörger and Vossen 2003). In the remainder of this section we expand the summary given in this paragraph.

The *view update problem* consists in finding a correct translation of an update on the view into an update on the source database. One of the main correctness criteria suggested in (Dayal and Bernstein 1978) requires that a view update translation have no side effects on the view. Since a view typically does not preserve all information in the source database, such translation is in general non-unique. Bancilhon and Spyrtos (1981) suggested that the desired update policy could be characterized by choosing a certain view complement and making sure that it remains invariant under updates.

Bancilhon and Spyrtos (1981) used a state-based formal framework. In the definition of the view complement g of f they require the set of pairs of instances of the respective view schemas, $\{(f(x), g(x) \mid x \in m)\}$, to be equipotent with the source schema m . Notice that this set is equipotent with the mapping $f \circ \text{Invert}(g)$. Essentially, their requirement corresponds to condition (ii) of Definition 4.2.5. They also reformulate their definition of view complement similarly to what we did in Theorem 4.2.5 by requiring the instances of the source schema that were indistinguishable to become distinguishable in the view complement (their Theorem 4.2, p. 564). The authors note that a view can have many different complements and that the identity view $\text{Id}(m)$ is a trivial complement for each view. This fact suggests that without some sort of minimality conditions the definition of the view complement can always be satisfied trivially. Thus, (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001) focus on “small” or “minimal” complements.

Our definition of Diff (Definition 4.2.5) corresponds to that of a minimal view complement. As we noticed in Sect. 4.3, the minimality condition can be relaxed by looking for a materialization of the mapping, or view complement, of interest. That is, we can obtain any view complement v_c from the minimal view complement v_{min} by composing v_c with some function h such that $v_c \circ h = v_{min}$. Also, note that our definition of Diff and the results with respect to the alternative formulation are more general than those in (Bancilhon and Spyrtos 1981) in that they apply to arbitrary mappings and not only total surjective functions.

Our definition and the formalization of Bancilhon and Spyrtos (1981) are decoupled from concrete schema and view languages. The materialization of view complements for database schemas has been first addressed by Keller and Ullman (1984). In particular, they considered the classical relational schemas whose instances are subsets of finite powersets. They showed that a so-called monotonic view has at most one complement that is independent and monotonic. They also presented an illuminating way of visualizing view complements in a tabular form. Hegner (1994) considered a less restrictive setting and showed that under certain conditions the complemented views form a Boolean algebra.

The algorithms for computing view complements were first studied by Cosmadakis and Papadimitriou (1984) for the views expressed in relational algebra using projection, selection, and join. They showed that for relational schemas with arbitrary functional dependencies computing a minimal complement is NP-complete, but did so using a peculiar notion of minimality that differs from the one used in our work and other approaches cited above. More recently, Lechtenböcker and Vossen (2003) argued that it may not even be necessary to look for the minimal complements; instead, reasonably small yet non-minimal complements may be more useful in practice. The authors study the problem for a large class of relational views that include the relational difference operator.

Most approaches dealing with view complements focused on relational databases and relational views under set semantics. As we demonstrated using examples in Sect. 4.2.5, the results of Diff may look quite differently in the case of multiset semantics. The work (de Amo and Halfeld Ferrari Alves 2000) presents algorithms for computing minimal view complements for temporal databases by translating views that are expressed in temporal algebra to first order expressions over non-temporal relations.

Stated as a view complement problem, computing the results of Diff has always been considered as a standalone operation. The problem becomes much harder if Diff is combined in scripts with other operators. As we demonstrated using the change propagation scenario in Chap. 5, studying the properties of such complex scripts can be non-trivial.

10.6 Approaches to Specifying Semantics

10.6.1 Semantics of Models and Mappings

The approach presented in Chap. 4 follows a standard way of specifying semantics used in databases and formal logic. For example, in model theory, which provides standard semantics for mathematical logic, the semantics of logical expressions is explained in terms of all possible worlds or interpretations that are consistent with these expressions. Similarly, semantics of database schemas is traditionally expressed in terms of all possible database states or instances that are consistent with the schema (Borkin 1978; Bancilhon and Spyrtatos 1981; Atzeni et al. 1982; Abiteboul et al. 1995). In (Madhavan et al. 2002), the semantics of models and mappings is characterized directly in terms of model-theoretic interpretations. In all these approaches, the semantics of a formal artifact is described by a set (or family) of other formal artifacts. We call this universal concept instantiation and use it as a foundation for defining the state-based semantics of model-management operators.

In contrast to many other techniques, the notion of state-based semantics exploited in this dissertation is not tied to a specific data model, such as the relational model, or language, such as the first-order logic or SQL. By raising the level of abstraction, we characterize the semantics of operators in a generic fashion. That is, we consider the instances of models as opaque entities and characterize the operators that manipulate models and mappings without considering the internal structure of the instances. In principle, the instances of models can be treated as models themselves, i.e., can have their own sets of instances. This flexibility is required in the approaches such as (Atzeni and Torlone 1996; Cluet et al. 1998) which exploit multiple levels of instantiation. (The fact that instances can be treated as models might have contributed to the apparent overloading of the term “model” used in

the literature on databases, AI, and formal logic, which was observed in (Madhavan et al. 2002.)

We illustrated the operators using relational algebra and SQL views, and examined in more detail a very simple mapping language, morphisms. A multitude of other mapping languages have been utilized in the literature for addressing various metadata management scenarios (Bergamaschi et al. 1999; Bernstein 2003; Claypool 2002; Cluet et al. 1998; Davidson et al. 1995a; Fan et al. 2003; Halevy 2001; Kementsietsidis et al. 2003; Li et al. 2003; Madhavan and Halevy 2003; Melnik et al. 2003b; Mitra et al. 2000; Popa et al. 2002; Pottinger and Bernstein 2003; Velegrakis et al. 2003) – their number can probably compete only with the number of data models, or schema languages, developed for the same purpose. The state-based approach characterizes the semantics of these mapping languages in a uniform fashion as a relationship on instance sets. One of the earliest manifestations of this approach can be found in (Paolini and Pelagatti 1977), where databases were represented by many-sorted algebras and mappings were treated as homomorphisms. The authors demonstrated how update operations on databases can be treated as mappings. A similar technique was developed in (Maibaum 1977). Treating mappings as relationships between instances allows us to specify and study mapping containment, mapping composition, and mapping confluence for heterogeneous mapping languages. This capability is especially important in scenarios that deploy more than one mapping language (Li et al. 2003; Shanmugasundaram et al. 2001a).

Lenzerini (2002) suggested a very general way of specifying mappings as $Q_1 \sim Q_2$, where \sim is some predicate that holds between the results of queries Q_1 and Q_2 . We notice however that this mapping specification makes use of a third common model in which the results of Q_1 and Q_2 are represented. Thus, effectively $Q_1 \sim Q_2$ describes a ternary mapping that holds between three models (compare Sect. 11.3.4).

An interesting observation is that many schema languages such as XML Schema or almost every semantic database model, e.g., Schema Intention Graphs (Miller et al. 1994), support quite expressive constraints and, in fact, can be used as mapping languages: if we assume that the entities used in schema m are defined in several other schemas m_1, \dots, m_n , then effectively m describes an n -ary mapping between the schemas m_1, \dots, m_n .

The mapping languages vary substantially in their expressive power and some are better suited for a given purpose than others. For example, the mapping tables of (Kementsietsidis et al. 2003), which we discuss in Sect. 10.9, is a mapping language that addresses the needs of quite different applications than does SQL or XQuery. Hence, there may not be a best language for each model management scenario. If the mapping map between two models is expressed using several partial mappings map_i , each potentially in a different mapping language, the confluence operator can be used to specify the semantics of map as $map = \oplus_i map_i$.

10.6.2 Information Capacity

State-based semantics is closely related to the work on information capacity (Hull 1986; Miller et al. 1994). The information capacity of schema m is the cardinal number of its instance set, $|\text{Inst}(m)|$ or simply $|m|$ in our simplified notation. The relative information capacity of two schemas m_1 and m_2 refers to the relationship that holds between $|m_1|$ and $|m_2|$, such as \leq or $=$. The relationship \leq is called dominance and is characterized by the existence of a surjective function map from m_2 onto m_1 .

A key question in the work on information capacity has been whether a given database schema is more, less, or equally expressive than another database schema, i.e., whether there exists a surjective or bijective function between m_1 and m_2 . In contrast, the model-management approach focuses on obtaining the actual mappings between m_1 and m_2 that can be deployed by applications. Such mappings are specified by means of model-management scripts and may be non-functional (for example, morphisms and GLAV mappings are typically non-functional).

In (Hull 1986), four progressively more restrictive notions of dominance are studied (absolute, internal, generic, and query dominance). For example, query dominance means that the function map can be specified using a first-order predicate calculus expression. Still, even this restrictive notion of dominance turns out too liberal to accurately measure whether an underlying semantic connection exists between database schemas. The underlying line of argument is to present a transformation between two schemas that establishes query dominance but appears semantically vacuous. In (Miller et al. 1994), it is suggested that more restrictive notions of dominance need to be developed. That is, more constraints should be placed on the mappings. And yet, the mappings deployed in real applications can be arbitrarily complex. For example, some wrappers of legacy systems squeeze multiple attributes of structured XML messages into a single general-purpose ASCII field.

We argue that dominance and equivalence are inadequate measures of semantic connection between schemas. In fact, we intensionally use the term equipotence rather than equivalence to avoid implying such semantic connection. The fact that dominance or equipotence holds provides no guarantees as to whether and how the schemas are semantically related. For example, schema $m_1 = \langle\langle R(\text{Name} : \text{char}, \text{Age} : \text{int}) \rangle\rangle$ is equipotent with $m_2 = \langle\langle S(\text{ID} : \text{int}, \text{Name} : \text{char}) \rangle\rangle$, i.e., $|m_1| = |m_2|$. However, m_1 and m_2 are semantically incommensurate; the mapping $m_1 _ m_2 = \langle\langle \pi_{\text{Name}}(R) = \pi_{\text{Name}}(S) \rangle\rangle$, which may reflect the relationship between the schemas in a particular application context, is not even a function.

In fact, it seems that most schemas used in practice are related in an inherently non-functional way. Frequently, it is simply irrelevant whether dominance or equipotence holds. Part of the reason for that is that schemas rarely specify all valid application constraints because they were not known at the time of schema design or are not expressible in the schema constraint

language. That is, the schemas often allow “irrelevant” instances, i.e., states that can never be reached due to application constraints. The presence of such instances makes the transformations on schemas non-functional. From this perspective, the undecidability results of (Miller et al. 1994) concerning equipotence may look less discouraging. Still, they provide an illuminating insight in the nature of difficulties that may have to be addressed in the state-based approach to model management. Another instructive result, presented in (Hull 1986), is that equipotence of relational schemas without constraints implies that the schemas must be structurally identical (up to renaming and isomorphism). This result may generalize to other schema and mapping languages.

Considering the information capacity of schemas alone is insufficient for defining the state-based semantics of model-management operators precisely. For example, in Sect. 2.3.5 we require that the schema produced by the *Merge* operator be at least as expressive as each of the input schemas. In Sect. 2.3.3 we argue that the schema delivered by *Extract* must be at most as expressive as the input schema. However, these requirements do not specify what the operators do to the instances of schemas: the relationship between the instances of the output schemas and the instances of the input schemas remains unspecified. In Chap. 4, we define this relationship precisely for all key operators. Further limitations of the information capacity approach are studied in (Davidson et al. 1995a).

Our viewpoint is that the semantic relationship between two schemas is determined by the mappings that hold between them (as we argue in Sect. 11.3.4, the mappings may be n -ary and may involve other schemas). Such mappings may or may not be functions and may be arbitrarily complex. In Chap. 4, for all key operators we state the conditions on the mappings between the output schemas and the input schemas. Combining the operators into scripts helps us establish precise criteria on mappings produced in various model-management scenarios.

10.6.3 Category Theory

A principle underlying our work is that the essence of formal artifacts, such as models and instances, is to be sought primarily in the nature of their relationships with other artifacts of the same kind rather than in their internal constitution. This idea has achieved its fullest expression in category theory (Mac Lane 1998), an axiomatic framework within which the notions of transformation (as morphism or arrow), composition, and structure (as object) are fundamental, i.e., are not defined in terms of anything else. Note that the notion of morphism in category theory, which we consider in this section, is not to be confused with a concrete mapping language discussed in Sect. 6.1.

One of the first applications of category theory to data management was studied by Maibaum (1977). The author considered database states of a specific database schema as a category, in which the updates transforming one

state into another are treated as morphisms between the objects in the category.

The first category-theoretic approach to semantics in model management has been investigated in (Alagic and Bernstein 2001). There, signatures of database schemas form a category **Sig**, while mappings between schemas correspond to morphisms. The relationship between schemas and instances is captured by a functor *Db* (a functor is a morphism between categories), which corresponds to our instantiation function *Inst*. A minor terminological difference is that *Db* maps each database signature **Sig** to a *category* of instances of **Sig**, whereas *Inst* maps a model to a *set* of instances. Just as all signatures form the category **Sig**, all categories of instances form another category, which we henceforth denote as **Inst**. In contrast to (Alagic and Bernstein 2001), the formalization presented in Chap. 4 unfolds in the category **Inst** rather than in **Sig**.

The approach in (Alagic and Bernstein 2001) distinguishes between schemas and schema signatures. Schemas are schema signatures with associated integrity constraints. The authors define the category of schemas **Sch**, which differs from **Sig** by adding integrity constraints, with the understanding that all instances of schemas are guaranteed to satisfy the integrity constraints defined on the schemas. Explicit treatment of constraints makes their discussion somewhat verbose, since the constraints surface in all definitions and theorems. In their case, this was necessary because their main results concerned the mapping of constraints across morphisms. In our approach, constraints are an integral part of the schema language, i.e., each set of instances of a model (i.e., each category in **Inst**) is guaranteed to satisfy all integrity constraints.

In the definition of a category, objects are “just things” for which no internal structure is observable by categorical means (composition, identities, morphisms, and typing). A key challenge in working with a category of objects is to develop a set of axioms that characterize the relationships between the objects of the category as precisely as possible. In (Alagic and Bernstein 2001), the authors characterize two operations, called schema integration and schema join, using commutative diagrams. However, they rely on the notion of a “matching part” defined intuitively which makes it hard to study their operations formally and relate them to the operator **Merge**.

The playground of state-based semantics is the category **Inst**. Working with **Inst**, rather than **Sch** or **Sig**, enables us to deal with non-atomic objects, i.e., models as sets of instances, and to understand the semantics of the key model-management operators more deeply. Well-known scenarios help us analyze the properties of the operators and may enable us to derive their axiomatic characterization in **Sch**. For example, the operator **Merge**, $\langle M, f, g \rangle = \text{Merge}(A, B, h)$, could be characterized in **Sch** as: $f \circ f^{-1} = 1_A$, $g \circ g^{-1} = 1_B$, $M = \text{dom}(h)$, $h = f \circ g^{-1}$. The minimality condition of Definition 4.2.4 cannot be stated in **Sch** directly. However, it may be possi-

ble to find a set of theorems in **Inst** involving the operator **Merge**, such as Theorem 4.2.11, Corollary 4.2.1, or Conjecture 11.3.1, that provide further restrictions on the operator and can be stated as axioms in **Sch**. Alternatively, the operators **Extract**, **Merge**, and **Diff** could be viewed as new fundamental operators, equal peers of the operators **Compose** and **Id**. Notice that the minimality conditions in our definitions are essential. Without them, we could always extract m from m , or get arbitrarily expressive models as a result of **Merge**.

Another relevant category-theoretic notion is that of a topos (Bell 1988). Topos is a Cartesian closed category in which for each object there exists an object of its “subobjects”, which can be regarded as instances. In a topos, as in set theory, every object and every arrow can be considered as the extension $\{x \mid P(x)\}$ of some predicate P . This view corresponds closely to our treatment of models and mappings as predicate variables. The category **Inst** seems to be closely related to toposes.

10.7 Metadata Repositories

Over the years, a number of research prototypes and commercial products have been developed to support metadata management, including Rational Rose tools¹ and Microsoft Repository² (Bernstein et al. 1999). Such tools do an excellent job in providing a design environment or persistent storage for metadata artifacts. However, the existing tools do not go far enough to support the developers of metadata applications, which may be one of the reasons that limited their broad adoption and commercial success.

Do and Rahm (2000) reviewed several commercial metadata repository systems that are specifically targeted at metadata management for data warehousing. Typically, repository systems use a relational database for storing metadata. The metadata can be accessed and manipulated using SQL, assuming that the database schema of the repository is known. Some tools provide query templates to speed up the construction of frequently used queries, such as data lineage and impact analysis queries.

A SQL or SQL-like query interface to metadata artifacts offers substantial help to the developers of metadata applications. However, such approach still has significant limitations:

- A thorough understanding of the relational representation chosen for particular metadata artifacts is required in order to write the queries. In other words, the developers need to be proficient in the meta-meta model and the meta-models of the artifacts that they deploy.

¹ www.rational.com

² Currently shipped with Microsoft SQL Server 2000 under the name Meta Data Services

- The queries are often quite complex, since they operate on the individual model elements.
- It is hard to migrate the applications between different repository systems because the applications are tied to particular meta-models and meta-meta models.

In contrast, the model-management operators offer a much higher-level, generic interface for application developers. Still, as we pointed out in Sect. 3.3, a SQL-like querying capability has proved instrumental for implementing the model-management operators in Rondo. Such capability may be exposed to the developers of metadata applications to complement the model-management operators in dealing with special-purpose metadata transformations such as schema normalization. Furthermore, low-level, performance-sensitive metadata management functionality, such as versioning at the level of individual model elements (Bernstein et al. 1999), will likely to continue relying on SQL-like APIs.

The model-management approach has a great potential to boost the capabilities of today's repository systems and simplify their use. However, even the vendors of repository systems themselves might benefit from the availability of model-management operators. In fact, many repository systems include pre-packaged metadata applications, such as configuration management or impact analysis applications, whose development and maintenance using low-level APIs is costly.

The books (Marco 2000; Tannenbaum 2001) offer a survey of commercial repository systems and discuss implementation options for several metadata management tasks.

10.8 Metadata-Intensive Applications

As we illustrated in Sect. 1.1, metadata problems arise in a variety of applications and scenarios. In this section we discuss in more detail the work related to two such scenarios, integration of heterogeneous information processing services and change propagation.

10.8.1 Declarative Mediation

In the late 90's, a multitude of information processing services started to become available online and supersede the static Web content. Such services accept data, process it, and return results. A variety of services went on air, such as search engines, digital libraries, flight/hotel/car reservation systems, e-shops, tax filing services, or calendar managers. As more such components were deployed, the diversity of program-level interfaces had emerged as an important stumbling block providing a challenging research opportunity.

In the work (Melnik et al. 2000; Melnik 2000; Melnik and Decker 2000; Decker et al. 2000a; Decker et al. 2000b) we focused on interoperation of heterogeneous information processing services. Management of metadata turned out to be a heavy component of this work and motivated in part the subject of this dissertation. In fact, the systems and frameworks that we developed provided us a hands-on case study for metadata management. The code developed in these projects served as a seed and inspiration for the programming platform that has been prototyped as part of this dissertation. The remainder of this section gives a brief overview of this work and summarizes the key conclusions.

The *mediation* architecture has been used in numerous information integration projects (Wiederhold 1992). It introduces two key elements, wrappers and mediators. The wrappers hide a significant portion of the heterogeneity of services, whereas the mediators perform a dynamic brokering function in a relatively homogeneous environment created by the wrappers. A mediator typically receives a request (e.g., a query), submits a translated version of the request to several services, collects and merges the responses, and presents them to the user. Mediators that were developed in previous research efforts had some important shortcomings, which are in fact still present in most of today's systems:

- Mediators are often hard to extend beyond the initial set of services they were designed for.
- It is difficult to incorporate into a mediator components that were developed elsewhere. For example, once a particular query translation algorithm has been implemented in a mediator, it is very hard to replace it by some other query translation package.
- Most often mediators do not tackle protocol differences. For instance, many mediators assume that all their targets communicate via HTTP.
- Usually it is not easy to extend a mediator to non-search tasks. For example, if a mediator is designed to query multiple search engines, it is hard to make it mediate among different payment mechanisms or among different document summarization services.

In (Melnik 2000), we proposed a mediation framework that addressed these shortcomings. In (Melnik et al. 2000), we studied an application of this framework to the domain of digital libraries. The framework presents a very flexible environment where different components, which we call “blades”, can be combined to address a specific mediation task. One of the components, in particular, is responsible for translating protocols. For example, this component may receive a single synchronous message from a user, and in turn issue a sequence of asynchronous messages to perform the requested task.

In our approach, all data conversion and protocol translation logic is embodied in mediators, whereas wrappers are as simple as possible and thus can be developed with moderate effort. Such wrappers, which we call *canonical*,

capture information contained in the messages of the component by means of logical descriptions and are not required to do any processing beyond trivial syntactic transformation of the messages of the wrapped component. A logical description of the request makes it possible to abstract out factors irrelevant to the purpose of the service, e.g., whether it is implemented as a distributed object or a CGI script. The logical descriptions of the messages are encoded as directed labeled graphs and can be manipulated using algebraic operations, transformation rules etc. The interface descriptions of the canonical wrappers are represented as finite-state automata that accept and send messages of different kinds.

Even though mediators are shielded from the native components by the wrappers, they still have to deal with the semantic heterogeneity of information exposed by the wrappers. The major integration problems include data, query, and protocol translation. Thus, in general, a number of different formalisms are required to describe the mediator logic. For example, manipulations of the content of the messages can be expressed using Datalog rules, XSLT, or other data manipulation languages such as YATL (Cluet et al. 1998). Transformations of message sequences can be described using finite-state machines, Petri nets etc. For query rewriting, powerful approaches have been developed in the database literature (Halevy 2001). In (Melnik et al. 2000) we describe in detail which formal techniques we picked for implementing a declarative mediation system for digital libraries.

To facilitate mixing and reuse of different formalisms, the declarative languages used by our mediators are represented using a meta-meta-model that is capable of capturing and linking expressions in different languages. Our meta-meta-model is based on directed labeled graphs, similarly to how the messages themselves are encoded (in Rondo, we use an extended version of this meta-meta-model that supports ordered relationships). To execute mediators that deploy different formalisms and languages, we developed a comprehensive runtime environment, which makes sure that the appropriate interpreters, or blades, are invoked for the declarative languages used in the specifications.

Lessons Learned. One of the key lessons that we learned from our work on declarative mediation is that developing and maintaining an environment which uses a variety of complex metadata artifacts is really hard. A first step that we took to address the metadata management issues was to store mediator specifications and interface descriptions as first-class objects in a database system. In this way, we had a reliable persistence mechanism which also allowed us to query mediator and wrapper specifications. However, the key challenge turned out to be the implementation of *operations* on these complex structures. Examples of such operations are tracing the changes of evolving interface descriptions and updating the mediators accordingly, generating wrapper skeletons from interface specifications, or determining algorithmically the compatibility of wrappers and mediators. We realized that

to handle the complexity of descriptions, support for mediator *composition* is required (Melnik 2000). Supporting composition is hard, since multiple formalisms may be used throughout the mediator. For example, a complex mediator may be specified as a finite-state machine and may invoke Petri nets for executing subtasks that require concurrency control. Service composition is an open problem that recently has attracted significant attention (Hull et al. 2003).

We also learned that to be able to deal with a variety of metadata artifacts and formal languages, we needed some way of abstracting out the key properties of their representation and manipulation. We grew even more convinced of the importance of a *generic* approach for metadata management in our subsequent work (Melnik and Decker 2000). In (Melnik and Decker 2000), we presented a layered approach to information modeling and interoperability on the Web. The key idea of the approach is to automate the translation of messages exchanged between heterogeneous components by attaching metadata to the messages that describes the features of the meta-model utilized for representing the message content. The message metadata is split into “layers”, each of which describes a certain set of modeling features, such as object identity, ordered relationships, aggregation, etc. Our work on declarative mediation and layering indicated that there was a great potential in approaching metadata management in a generic fashion.

10.8.2 Change Propagation

Change propagation is a pervasive problem that has attracted substantial attention in the database research literature. For example, Roddick (1992) lists an impressive annotated bibliography of work done in the 1970-80’s. (Roddick et al. 2000) classify manifestations of change by subject, causes, effects, temporal and spatial issues, etc. The economical factors of software evolution and maintenance have been considered in (Wiederhold 2003).

Various aspects of the change propagation problem have been studied in the database literature including view adaptation, view synchronization, view maintenance, and, most recently, mapping adaptation. The general theme is to examine the effect of changes of some *source* metadata artifacts or data on *target* metadata artifacts or data. The subjects of change, i.e., sources and targets, can differ substantially. We distinguish some of them below:

- Changes to a schema affect an existing schema instance. Banerjee et al. (1987) study close to two dozen kinds of changes that can occur in object-oriented databases, such as adding/dropping instance variables to a class, changing default values of variables, changing the order of superclasses of a class, etc. They suggest a set of rules of how instance data should be adapted to schema changes, and address the soundness and completeness of elementary changes. Essentially, soundness and completeness guarantee that all operations produce valid object lattices and any object lattice

can be obtained using a sequence of operations. The work in (Peters and Özsu 1997) provides an axiomatization of schema evolution in a similar perspective. In (Lerner 2000), complex changes spanning multiple classes are considered.

- Changes to a source schema affect a view defined on the schema. This change propagation problem is referred to as *view synchronization* and has been studied, e.g., in (Lee et al. 2002).
- Changes to the instance of a source schema affect the target instance. This issue was studied in the context of *maintenance of materialized views* (see e.g. (Mumick et al. 1997)). Claypool et al. (1998) considers schema evolution using primitives expressed in OQL, while Claypool and Rundensteiner (2003) define a cross-algebra that helps propagating changes on instances between two different data models, such as XML and relational.
- Changes to the mapping (and the target schema) affect the instance of the target schema. This kind of change has also been termed as *view adaptation* in (Gupta et al. 1995): once the view definition changes, the materialized view needs to be updated, ideally, without recomputing the view from the base relations. This problem is closely related to answering queries using views (Halevy 2001), where the query corresponds to the new view definition, and the view is the original materialized view. In general, the new materialized view can be computed using a portions of data from the old view and another portion recomputed from old relations.
- Changes to instance data yield changes to the schema. This way of propagating changes takes a reverse path compared to the first item and is important for cases when instances are decoupled from schemas (Parsons and Wand 2000), description logics (Borgida 1995), or for incremental maintenance of schemas extracted from semi-structured data (Nestorov et al. 1998).
- Changes to the source or target schema affect the mapping between the source and the target (Velegrakis et al. 2003) and the target schema (Bernstein 2003).

When schemas are the source of changes, the way that changes are specified is another dimension in which approaches to change propagation differ. A typical assumption is that the changes are characterized using a finite set of primitive operations on schemas, often accompanied by a corresponding set of primitive instance transformations. This approach was used, e.g., in (Banerjee et al. 1987; Peters and Özsu 1997; Lerner 2000; Velegrakis et al. 2003). The advantage of using a fixed set of elementary schema changes is that we can specify precisely how to handle each individual change. The disadvantage is that the way in which the schema can evolve is restricted.

Alternatively, the changes can be described using a mapping between the old and new source schema, the approach presented in (Bernstein 2003), which we also followed in our work on change propagation (see Sect. 2.1 and Chap. 5). A mapping is capable of accommodating virtually all kinds of

conceivable changes such as schema normalization, changing attribute type, applying a user defined function, transposing the schema, etc. In fact, in the general case such a mapping need not even be functional. To our knowledge, propagation of changes described by an arbitrary mapping has not been examined previously.

The mapping that describes the changes can be obtained in various ways. One way is to allow schemas to evolve and then find the changes that took place by comparing the modified schema to the original version using schema matching. Another way is to compose a number of elementary changes and thereby leverage the specification of changes developed and studied in previous approaches. A history of changes can be produced, e.g., by schema manipulation tools.

In Sect. 5.4, we discussed schema evolution as a special case of change propagation. A formal specification of schema evolution offers precise guidelines for computing the effects of changes, in contrast to heuristic rules that are deployed in some of the approaches in the literature. For example, Velegrakis et al. (2003) consider the problem of adapting the mapping upon a schema change and suggest to “make the minimum changes necessary to achieve a mapping that is consistent with the new schema”. However, the lack of formalization of what constitutes a minimal change makes the adaptation techniques that they propose debatable: specifically, we argue that neither removal of schema constraints nor addition of new schema elements should impact the existing mappings. By modifying the mappings in such cases, new ways of relating data instances are “invented” whereas the old mapping still holds – a minimal change should arguably leave the mapping intact.

An important aspect of change propagation is efficiency (see e.g. (Banerjee et al. 1987; Gupta et al. 1995; Mumick et al. 1997)). In approaches that focus on instance data, a primary concern has been batching and delaying the updates, and minimizing their impact on the DBMS performance. Batching updates can be expressed as a composition of several individual updates. For applications that access old versions of data that has been reorganized in the course of schema evolution, reverse transformations need to be computed.

10.9 Other Related Work

Data Translation. Data translation was one of first hot topics in the database research field (Bernstein 1999). In fact, before ACM SIGMOD gained its name in 1975 (“Management Of Data”), it was previously called SIGFIDET, for “File Description and Translation”. Data translation is the problem of transforming data when moving it from one application to another. The EXPRESS project at IBM Research was one of the foremost data translation projects of its day (Shu et al. 1977).

A number of rigorous formal techniques for data translation have been developed. For example, Kalinichenko (1990) presents a formal definition of

data models and manipulates them as formal objects in the process of development of mappings between data models. The author also explores a methodology for synthesizing a “unifying generalized data model” for a given set of data models. Markowitz and Shoshani (1992) discuss a formal technique for translating Entity-Relationship structures into a relational representation. Rosenthal and Reiner (1994) examine equipotence-preserving transformations of schemas and give formal proofs of correctness of schema rearrangements. They argue for a combination of heuristics and rigorous transformations.

Generic approaches to data translation across different schema languages have been explored, e.g., in (Atzeni and Torlone 1996; Cluet et al. 1998). The techniques presented there could be used for implementing a generic operator for generating one model from another. Such an operator, called *ModelGen*, was suggested in (Bernstein 2003). In our prototype, we are using a less general approach, in which each converter is implemented as a custom, non-generic operator. In Sect. 5.5, we discussed the impact of data translation on model-management scripts and their state-based semantics.

More recently, the problem of data translation has undergone a resurgence of interest in the context of data warehousing (ETL tools) and integration of heterogeneous Web sources, in particular, translating relational data to XML (Shanmugasundaram et al. 2001b; Fan et al. 2003).

Mapping Tables. The recently proposed mapping language of (Kementsietsidis et al. 2003) generalizes the notion of value transformations. It allows specifying the dependencies between the entities of two schemas using a so-called mapping table, an extensionally defined table of value correspondences. Value transformations between entities in schemas can be quite intricate. A trivial example is concatenation of first name and last name to obtain full name. A more involved example is the transformation of a planar circle represented by three points into a circle represented by a center and a radius. Sometimes, however, value correspondences cannot be represented using a formula and need to be defined extensionally. For example, the correspondences between gene and protein identifiers in biochemical databases, or the mapping from Zip and City to State are specified as lists of value tuples. In (Kementsietsidis et al. 2003), the authors use a state-based approach to define the semantics of mapping tables. They also consider operations on them, such as AND (\wedge), OR (\vee), and negation (\neg). Operation \wedge generalizes to \cap in our formalization, \vee corresponds to \cup , and negation can be expressed as $\neg m_1 _ m_2 = m_1 \times m_2 - m_1 _ m_2$.

Confluence. Kementsietsidis et al. (2003) also considered an operation that corresponds to our *Confluence* operator (personal communication). They note that sometimes a small part of two mapping tables is inconsistent. In this case, ANDing the information of two tables yields a contradiction and renders the whole result unusable. An equivalent of the *Confluence* operator can be used to AND a mutually consistent part of two mapping tables and OR

the non-overlapping parts. The advantage of defining confluence in a generic fashion is that it can also be used to combine a mapping that consists in part of say an SQL view and a mapping table. Another interesting observation is that a mapping table is actually an instance of a relational schema. That is, we have an example of a mapping that is itself an instance of some model.

As noted above, the Confluence operator can be used to deal with inconsistent mappings and thus may be useful integration of mutually inconsistent data sources (Lenzerini 2002). By Definition 4.2.6, combination of queries using the Confluence operator ensures that only mutually consistent answers appear in the result. That is, information loss is possible when the input queries or views are inconsistent with each other (Agarwal et al. 1995).

In database literature, the notion of confluence has been used in the context of active rules and triggers (see e.g., (Aiken et al. 1992)). There, confluence is a property of a set of active rules or triggers that holds if the effect of rule execution is invariant with the order of their execution. We use the term confluence differently, to denote an operator on mappings. See Sect. 10.1 for further discussion.

Z, B-Method, AMN. Z (pronounced: “zed”), B-Method and AMN (Abstract Machine Notation) are languages designed for specification and verification of computer systems. These languages have formal semantics that is based on the set theory and predicate calculus. The schemas in Z describe the possible states of a system, its operators, and relationships between its parts (Davies and Woodcock 1996). Basically, a schema defines a number of n -ary relations with pre-conditions and post-conditions. The language Z is an ISO standard (ISO 2002).

The so-called *schema calculus* used in Z shows an interesting parallel to model management. The schema calculus introduces several operators, such as And, Or, Iff, Compose, Implies, Pipe, and Project, for building bigger schemas out of smaller ones. The operators represent set operations with special signature translations. For example, And and Or are similar to ANDing and ORing of logical formulae.

The schema calculus in Z is bound to a specific schema language and its operators are more low-level than in generic model management. However, it would be interesting to see whether the operators in Z can be generalized for other kinds of models.

11. Conclusions and Outlook

“The significant problems we face cannot be solved at the same level of thinking we were at when we created them.”

– Albert Einstein (1879-1955)

11.1 Summary of Contributions

Many problems facing data management and other areas of computer-aided engineering involve the manipulation of models. Yet applications that manipulate models are complicated and hard to build. The goal of generic model management is to reduce the cost of developing such applications by raising the level of abstraction of model manipulation operations.

This dissertation presents an initial study of the concepts and algorithms for generic model management. To demonstrate that model management operators are implementable and useful, we developed a prototype of a programming platform, called Rondo, in which high-level algebraic operators are deployed for manipulating models and mappings. The prototype helped us experiment with various representations of models, alternative definitions of operators, and different algorithms used for implementing the operators. Using Rondo, we developed scripts for several practically relevant scenarios, such as change propagation and reintegration. We have shown that one can solve practical problems using the model management operators, and that these solutions require a relatively small amount of code.

To implement one of the most challenging model-management operators, the operator Match, we devised a general-purpose matching algorithm called Similarity Flooding. The algorithm can be applied for matching various kinds of models in metadata management scenarios as well as for other data structures and applications. We examined the computational properties of the algorithm and evaluated its quality using a novel accuracy metric and a user study that we conducted.

We presented a detailed survey of the related work that helped us factor out the common aspects of metadata applications and specify the structural and state-based semantics of the operators. Specifically, we considered data integration, schema matching, mapping composition, view selection, and view complement problems. The state-based semantics describes the effect of the operators on instances of models. It provides guidelines for implementing the operators for complex schema and mapping languages and is independent of a particular meta-meta-model. Both structural and state-based semantics is critical for specifying the effects of model-management scripts.

Our implementation experience, backed by the in-depth investigation of the individual operations in the research literature, suggests that the question raised in the panel discussion (Bernstein et al. 2000a) is likely to have a positive answer, i.e., generic metadata management is in fact feasible. Even if we cannot handle subtle and complex cases, if we can solve a large class of non-trivial problems then we are offering a useful programming platform. Still, resolving this debate to the full extent can be done only by writing scripts for a substantial number of real applications, which use practically relevant schema and mapping languages, and demonstrating that they work.

In this first dissertation on generic model management we only scratched the surface of this emerging field of research. In Sect. 11.2, we attempt to give an assessment of the current state of the field and provide a roadmap for developing the next generation of model-management systems. Our work uncovered many hard technical challenges and exciting new research opportunities, which are reviewed in Sect. 11.3. A salient non-technical challenge is acceptance by the developer community. As with each new programming paradigm, the willingness of engineers to learn a new way of approaching old problems is a critical ingredient for success of generic model management.

11.2 Concluding Discussion

In this section, we examine the achieved state of the art in model management and the gaps that need to be filled in order to build the next generation of more powerful and versatile model-management systems.

In the core of the model-management approach is a set of *generic* operators on models and mappings that simplify application programming. To what extent can the techniques developed in the literature and in this dissertation be called generic? How far can we push the model-management approach while claiming genericity? These questions are critical for laying out a roadmap for future work and understanding how far we are from achieving our goals.

Generic model management techniques address the following three aspects:

- *Generic applicability*: The operators can be applied to various kinds of models and mappings.
- *Generic use*: The operators are useful for a broad range of model-management tasks.
- *Generic implementation*: A single implementation of the operators is applicable for various kinds of models and mappings.

Generic Applicability. This aspect refers to the ability of engineers to write scripts without worrying about the nature of metadata artifacts they work with. To ensure generic applicability, the operators need to provide guarantees to the engineers that hold for all relevant kinds of models and mappings, including database schemas, workflow definitions, interface specifications, etc. Obviously, the less is known about the metadata artifacts under consideration, the fewer guarantees can be provided to the engineers with respect to the properties of the operators. In other words, the semantics of the operators can be stated only in very general terms or otherwise sacrifice genericity.

Three distinct ways of achieving generic applicability of model-management operators have been suggested. One way is to consider models and mappings as syntactic objects represented in a common meta-meta-model, for example, as graphs. This approach has been pursued in almost all prior work on generic model management, including the prototype developed as part of this thesis (Bernstein et al. 2000b; Bernstein and Rahm 2000; Bernstein 2003; Melnik et al. 2003b; Pottinger and Bernstein 2003). In essence, the operators are specified by means of graph transformations. As long as the graph transformations do not exploit any knowledge of what the graphs actually represent, the operators can be considered truly generic. Unfortunately, there are very few useful operations that can be defined in such an agnostic fashion. Largely, they are limited to Subgraph, Copy, and the set operations on graphs. In our experience, specification of most if not all other operations needs to be adapted to the individual meta-models for the operators to produce meaningful results. For example, most operators in Rondo (see Table 2.1 on page 23) are defined assuming a concrete mapping language, the morphisms. Analogously, the operators presented in (Bernstein 2003; Pottinger and Bernstein 2003) exploit the properties of a specific mapping language, though a more general one.

A second way to achieve generic applicability is by using state-based semantics. In this approach, the properties of the operators are characterized in terms of instances of models and mappings that are taken as input and produced as output. Under the assumption that models possess well-defined sets of instances, all key operators can be characterized in a truly generic fashion, as we demonstrated in Chap. 4. Such characterization is applicable to very complex kinds of models and mappings that are used in real applications, including XML Schemas, XQuery, and SQL. Although state-based characterization does not provide a detailed implementation blueprint, it is sufficiently specific so that the effect of the operators can be worked out for

concrete languages. A weakness of the state-based approach is that it says nothing about the syntax of models and mappings. Yet, the syntax of models (e.g., their structure and naming of model elements) is important for applications. Moreover, for certain kinds of models, such as make scripts and other program-like models, specifying the sets of instances formally can be non-trivial.

A third way of addressing generic applicability is an axiomatic one, e.g., using a category-theoretic approach (compare Sect. 10.6.3). The idea of the approach is to define the operators using axioms that are expressed in terms of the operators to be defined. Commutativity of `Compose` or associativity of `Merge` are examples of such axioms. This approach seems to be the most challenging, both in terms of determining a useful set of axioms and implementing the operators in such a way that the axioms hold when the operators are applied to concrete languages.

From the current perspective, it seems that our best bet for achieving generic applicability of operators is to combine state-based semantics with a syntax-oriented specification based on a common meta-meta-model. Such a combined specification of operator semantics may provide enough guarantees to the engineers to deploy the operators for manipulating various kinds of models and mappings without having a detailed knowledge of the operator implementation. Working out the details of such a combined specification is one of the gaps to be filled. It is possible that its syntax-oriented part turns out relatively simple: for example, one condition could be that the element names of the output models have to be drawn from the corresponding element names of the input models.

Of course, it is conceivable that we face hard limits to the generic applicability of operators. Most model-management scenarios examined so far in the literature focus on schema-like models, e.g., database schemas, ER/UML diagrams, or ontologies. To stand to the claim of generic applicability, the model-management operators should be applicable to workflow definitions, interface specifications, computational models, and other artifacts. Nevertheless, manipulation of schema-like models makes up a lion share of today's metadata management applications. Even if we limit the scope of a model-management system to schema-like models, the ability of manipulating such artifacts in a generic fashion could yield a dramatic increase in programmer productivity.

Generic Use. The usefulness of the model-management operators for implementing real applications is probably the most challenging claim in model-management research. There seem to be two complementary ways of justifying this claim: an empirical and a theoretical one.

The previous work on model management and this dissertation followed the empirical path. That work started with a solid intuitive understanding of the operator semantics and substantiated the generic use of the operators by examining detailed walkthroughs of various model-management problems

(Bernstein and Rahm 2000; Bernstein 2003). The prototype Rondo developed as part of this thesis helped prove that such abstract programs are indeed executable. To address the requirements of industry-strength applications, future model-management systems need to support complex mapping languages such as SQL, XQuery, or transformation languages used in software engineering applications. The ultimate empirical proof of generic use could be provided by turning a model-management system into a successful product.

A complementary way of justifying generic use is a theoretical one. The idea is to show that the proposed set of operators is complete with respect to the chosen operator semantics. For example, if the operators are specified in terms of graph transformations, completeness would ensure that all meaningful graph transformations can be realized using a combination of the operators. If state-based semantics is assumed, one could attempt to verify whether the operators can be used to define output models and mappings that describe any chosen set of instances and relations on instances. We consider the completeness question in more detail in Sect. 11.3.3. Currently, there is no good understanding of what completeness of model management operators could mean. This is an important gap to be filled.

Generic Implementation. Using a single implementation for various kinds of models and mappings has been considered a primary objective in most existing literature on generic model management (Bernstein et al. 2000b; Bernstein and Rahm 2000; Bernstein 2003; Melnik et al. 2003b; Pottinger and Bernstein 2003). Generic implementation helps extend a model-management system quickly for new kinds of models and mappings. For example, **Extract** and **Merge** are implemented in Rondo using a single algorithm for each of the operators, and a simple callback function to encapsulate meta-model specific behavior. **Match** has a truly generic implementation that does not exploit any properties of the underlying meta-models.

In Rondo, a largely generic implementation was possible due the simplicity of morphisms, the utilized mapping language. For more complex mapping languages generic implementation is unlikely. For example, it is hard to see how the **Compose** algorithm of Madhavan and Halevy (2003) or the **Merge** algorithm of Casanova and Vidal (1983) can be embedded into generic operators without actually implementing the algorithms. Moreover, these and many other algorithms are specialized to concrete schema and mapping languages. It seems unlikely that the algorithm of Madhavan and Halevy (2003) can be used with little changes to compose mapping tables (Kementsietsidis et al. 2003) or expressions in other mapping languages.

Although generic implementation is a desirable feature, it does not seem critical for the success of generic model management. The greatest benefit of the model-management approach is expected from *using* the operators for effective application development. Ideally, the developers of metadata applications should not be concerned with operator implementation, as long as each implementation satisfies the desired operator semantics.

Even if generic implementation cannot be achieved, it is still possible and desirable to utilize a *generic representation* of models and mappings to simplify the implementation of model-management operators. Generic representation amounts to rendering all features of individual meta-models in a common “data structure”, the meta-meta-model. Earlier in this section we discussed the use of a common meta-meta-model for specifying the semantics of the operators in a generic fashion. While we think that using a common meta-meta-model alone for specifying semantics is problematic, it may certainly facilitate the implementation. Low-level transformations of metadata artifacts that are necessary to support operator execution can be carried out using a SQL-like declarative language that operates on the common meta-meta-model. This approach worked very well in Rondo. Moreover, many commercial metadata repository systems offer a SQL interface to metadata artifacts stored in a relational representation (compare Sect. 10.7). Hence, future model-management systems may leverage the low-level capabilities of existing metadata repositories. Finding a convenient generic representation for complex mappings is another gap to be bridged.

A Research Agenda for Model Management. To summarize the above discussion, we outline a high-level research agenda for developing the next generation of model-management systems. We believe that the following research directions are among the most promising and challenging ones:

1. Developing a formal semantics for the operators that combines the state-based and structural approach while preserving generic applicability of operators.
2. Developing practical materialization algorithms, i.e., algorithms that compute the results of operators effectively, for model and mapping languages used in real applications. Existing algorithms suggested in the literature for the individual operations should be exploited to implement and optimize the execution of complex scripts.
3. Finding appropriate architectures and techniques for coupling model-management systems with applications, tools, and conventional programming languages. The capabilities of existing metadata repositories should be exploited for implementing the operators and algorithms.
4. Developing powerful user interfaces for building model-management solutions and supporting user feedback during script execution. Ultimately, we envision a tool for building model-management solutions graphically using Venn-like diagrams like the ones that we used throughout the dissertation. In this way, the engineer can simply “draw” a script using a graphical development environment and materialize the desired models and mappings using a single click.

Bringing the model-management capability to novel and promising domains, such as design and management of business processes or network management, may have a great impact on the way applications are developed and maintained today and in the future.

11.3 Open Technical Challenges

The work presented in this dissertation raised many hard technical issues. In this section, we review some of them. We believe that resolving these issues is instrumental for advancing the state of the art in model management.

11.3.1 Decidability and Complexity

The state-based operator definitions are decoupled from any concrete schema or mapping languages. However, to make the scripts executable we have to consider the operators in the context of specific languages. For example, Madhavan and Halevy (2003) study decidability and complexity of a single operation, composition. They consider a GLAV mapping language which consists of expressions of the form $Q_A \subseteq Q_B$, where Q_A and Q_B are conjunctive queries. A similar in-depth investigation may be necessary to obtain decidability and complexity results for each operator and each concrete language of interest.

It might be possible to state certain general conditions under which the operators are guaranteed to be computable, but it is unlikely. Even very simple conditions expressed using a state-based characterization are known to be undecidable for particular schema and mapping languages. Examples are the query containment problem for Datalog programs (Ullman 1997), or the question whether there exists a bijection between m_1 and m_2 for the SIG schema language (Miller et al. 1994). However, for simpler languages such questions may become decidable. Thus, query containment has an NP-complete decision procedure for conjunctive queries.

One way of implementing the state-based semantics is by developing what we call a *closed language system*, i.e., a set of sufficiently expressive schema and mapping languages that is closed under all model-management operators. That is, the result of each operator can be represented explicitly within the language system. It is relatively easy to find very simple languages that form a closed language system. The problem seems much harder for more expressive languages. For example, it would be interesting to investigate whether relational schemas with relational algebra (or WOL language (Davidson et al. 1995a)) used as a constraint and mapping language yields a closed language system.

If the results of each operator are computable and finite, then we can obtain the exact results for each script. However, even if certain intermediate results of scripts are not representable in finite form, it may still be possible to compute the final results or their materializations by script rewriting.

11.3.2 Equivalence and Entailment of Scripts

Studying equivalence and entailment of scripts provides the foundation for script rewriting and optimization. For example, it may be desirable to rewrite

a script into an equivalent script which uses fewer operators or favors operators of one kind over another and can therefore be executed more efficiently. Script optimization may be critical for practical deployment given that computing the results of a single operator may be NP-hard (Kementsietsidis et al. 2003; Madhavan and Halevy 2003) and some models can be very large, such as models of executable code. Moreover, since the results of certain operators may not be computable or representable in finite form, script rewriting may help us translate an infeasible script into a feasible one.

Testing entailment and equivalence may however be quite hard. For example, we hypothesize that the following conjecture holds (see illustration in Fig. 11.1):

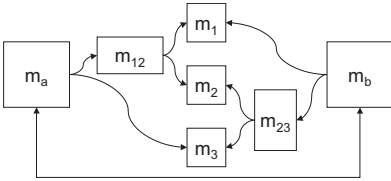


Fig. 11.1. Schematic representation for Conjecture 11.3.1 (Associative Merge)

Conjecture 11.3.1 (Associative Merge). The Merge operator is associative up to isomorphism. Formally, the following entailment holds:

$$\begin{aligned}
 \langle m_{12}, m_{12_m1}, m_{12_m2} \rangle &= \text{Merge}(m_1, m_2, m_{1_m2}); \\
 \langle m_{23}, m_{23_m2}, m_{23_m3} \rangle &= \text{Merge}(m_2, m_3, m_{2_m3}); \\
 m_{12_m3} &= m_{12_m2} \circ m_{2_m3}; \\
 m_{23_m1} &= m_{23_m2} \circ \text{Invert}(m_{1_m2}); \\
 \langle m_a, m_a_m_{12}, m_a_m_3 \rangle &= \text{Merge}(m_{12}, m_3, m_{12_m3}); \\
 \langle m_b, m_b_m_{23}, m_b_m_1 \rangle &= \text{Merge}(m_{23}, m_1, m_{23_m1}); \\
 m_a_m_b &= (m_a_m_{12} \circ m_{12_m1} \circ \text{Invert}(m_b_m_1)) \oplus \\
 &\quad (m_a_m_{12} \circ m_{12_m2} \circ \text{Invert}(m_{23_m2}) \circ \text{Invert}(m_b_m_{23})) \oplus \\
 &\quad (m_a_m_3 \circ \text{Invert}(m_{23_m3}) \circ \text{Invert}(m_b_m_{23})); \\
 \rightarrow \\
 \text{Invert}(m_a_m_b) \circ m_a_m_b &= \text{ld}(m_b); \\
 m_a_m_b \circ \text{Invert}(m_a_m_b) &= \text{ld}(m_a); \quad // \text{ i.e., } m_a_m_b \text{ is a bijection} \quad \blacksquare
 \end{aligned}$$

We made some initial progress on a simple theorem prover that uses a technique similar to “freezing” of (Ullman 1997) to test equivalence and entailment of scripts. Our prover was not able to find a contradiction to the above conjecture, but it is currently unable to provide a complete proof.

11.3.3 Completeness and Redundancy

Another vital question is that of completeness and redundancy: do we have a “complete” algebra with the operators `Invert`, `Compose`, `Extract`, `Merge`,

Diff, and Confluence? What could be suitable completeness criteria? Are the operators that we suggest non-redundant, i.e., is it true that none of the operators can be expressed using a combination of others? Are our operators the best? What other operators are conceivable?

We think that it may not be possible to characterize completeness other than by definition, similarly to the completeness of relational algebra. We do not know yet whether we succeeded to identify all key operators, whether the auxiliary operators such as `Domain` and `Id` should be considered part of the algebra, or whether more operators are needed.

It may be desirable to introduce other fundamental operators into the algebra. For example, operator `Hom(m)` could return a mapping that establishes a homomorphism relationship on instances of m . Such an operator could be used to characterize the data exchange scenarios (Fagin et al. 2003). Another useful operator could be an instance inclusion operator `Incl(m)`. It returns a mapping in which, say, right instances are entirely included in the associated right instances. This operator could be used for characterizing the certain answers for queries (compare Sect. 10.1.2). The operators such as `Hom` and `Incl` cannot be defined in a language-independent fashion, but there seems to be a good understanding of how to specify them precisely for each schema language of interest so that these operators may be of generic value.

To illustrate some other possible operators, consider the operator `Merge`. We defined the semantics of this operator based on a data integration scenario in which a unified database needs to be constructed. Another important data integration scenario is the one where we construct a virtual view of several databases to give them the appearance of a single database. This scenario could lead to a different operator definition, which figuratively speaking integrates only the overlapping part of two databases, whereas `Merge` integrates all information. It seems possible to define this operator as a derived operator `Intersect` (see illustration in Fig. 11.2):

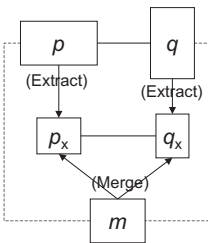


Fig. 11.2. Illustration of `Intersect` operator

Definition 11.3.1 (Intersect). $\langle m, m_p, m_q \rangle = \text{Intersect}(p, q, p_q)$ if and only if the following script holds:

$$\begin{aligned} \langle p_x, p_{p_x} \rangle &= \text{Extract}(p, p_q); \\ \langle q_x, q_{q_x} \rangle &= \text{Extract}(q, \text{Invert}(p_q)); \end{aligned}$$

$$\begin{aligned} \langle m, m_{p_x}, m_{q_x} \rangle &= \text{Merge}(p_x, q_x, \text{Invert}(p_{p_x}) \circ p_{q_x} \circ q_{q_x}); \\ m_{p_x} &= m_{p_x} \circ \text{Invert}(p_{p_x}); \\ m_{q_x} &= m_{q_x} \circ \text{Invert}(q_{q_x}); \quad \blacksquare \end{aligned}$$

Whether the above definition describes the intended semantics correctly or not is however an open question.

As another example, consider the operator **Extract**. In Definition 4.2.3, the operator takes a single mapping as input. In a more general setting, we may be interested in extracting a view that allows us to answer a *set* of given queries q_1, \dots, q_n rather than a single query (compare Sect. 10.4). That is, it must be possible to reformulate each of the input queries against the view schema. In other words, condition (ii) of Definition 4.2.3 has to be stated for each of the input queries. The question is though, whether the definition of **Extract** needs to be extended or whether it is possible to express the desired semantics using a script. Again, we do not have an answer to this question. However, we think that we do not need to extend the operator and we postulate the following hypothesis to be verified:

Conjecture 11.3.2 (Extract for two queries). Let q_1 and q_2 be two queries over m , i.e., $q_i \subseteq m \times s_i$. Further, let Definition 4.2.3 of **Extract** be extended for two mappings,

$$\langle m_x, m_{m_x} \rangle = \text{Extract}(m, q_1, q_2);$$

such that

$$\begin{aligned} m_{m_x} \circ \text{Invert}(m_{m_x}) \circ q_1 &= q_1; \\ m_{m_x} \circ \text{Invert}(m_{m_x}) \circ q_2 &= q_2; \end{aligned}$$

and minimality of m_x is guaranteed. Then, the script

$$\begin{aligned} \langle s, s_{s_1}, s_{s_2} \rangle &= \text{Merge}(s_1, s_2, \text{Invert}(q_1) \circ q_2); \\ \langle m_x, m_{m_x} \rangle &= \text{Extract}(m, (q_1 \circ \text{Invert}(s_{s_1})) \oplus (q_2 \circ \text{Invert}(s_{s_2}))); \end{aligned}$$

has the same effect on m_x and m_{m_x} as the application of the extended operator **Extract**(m, q_1, q_2). \blacksquare

In the conjecture we exploit the intuition from the view selection problem in data warehousing that the view m_x can be computed using a so-called multiquery (Theodoratos et al. 2001), which corresponds to the expression $(q_1 \circ \text{Invert}(s_{s_1})) \oplus (q_2 \circ \text{Invert}(s_{s_2}))$. Speaking informally, a multiquery combines several queries into one. In fact, if queries are represented as graphs, a multiquery can be obtained by “merging” the individual query graphs. In this process, the view schema induced by the queries changes. The schema induced by the multigraph corresponds to the schema s in the script.

We verified the conjecture in a preliminary form using our simple automated prover. The prover could not find a contradiction for the implication

$$\begin{aligned}
\langle s, s_{s_1}, s_{s_2} \rangle &= \text{Merge}(s_1, s_2, \text{Invert}(q_1) \circ q_2); \\
\langle m_x, m_{m_x} \rangle &= \text{Extract}(m, (q_1 \circ \text{Invert}(s_{s_1})) \oplus (q_2 \circ \text{Invert}(s_{s_2}))); \\
\rightarrow \\
m_{m_x} \circ \text{Invert}(m_{m_x}) \circ q_1 &= q_1; \\
m_{m_x} \circ \text{Invert}(m_{m_x}) \circ q_2 &= q_2;
\end{aligned}$$

for arbitrary mappings q_1 and q_2 , not only functions. Moreover, replacing $\text{Invert}(q_1) \circ q_2$ by $s_1 \times s_2$ in the premise did not cause the prover to find a contradiction either, so that the conjecture may hold even if we simply take a union of signatures of s_1 and s_2 : $\langle s, s_{s_1}, s_{s_2} \rangle = \text{Merge}(s_1, s_2, s_1 \times s_2)$ (compare Theorem 4.2.4).

No matter whether or not the conjectures that we presented hold, there may be other fundamental model management scenarios that cannot be expressed using a combination of the operators that we defined in this dissertation.

Notice that we stated Conjecture 11.3.2 for two queries and not for n queries. In fact, it turns out that generalizing the conjecture for $n > 2$ using binary mappings is not that easy. That leads us to a more general question, whether binary mappings are sufficient to address all model management scenarios of interest.

11.3.4 N -ary Mappings

An elegant and intriguing extension of the formalization that we presented is obtained by considering n -ary mappings, such as $map \subseteq m_1 \times m_2 \times \dots \times m_n$. (For $n = 1$, we call a mapping a model.) To motivate n -ary mappings, consider the following example. Imagine that we are given three models m_1, m_2, m_3 each with a single class definition, class A in m_1 , B in m_2 , C in m_3 . Now we want to establish the fact that $C = A \cup B$ (e.g., to subsequently merge all three models). This is impossible to do if the mappings are limited to two models at a time: we can state that A is a subclass of C, and B is a subclass of C, but not the condition we want. The desired relationship can be specified using a ternary mapping $map \subseteq m_1 \times m_2 \times m_3$, $map = \langle\langle C = A \cup B \rangle\rangle$. Analogously, we can argue for the need of mappings of higher arity by examining the condition $A_n = A_1 \cup A_2 \cup \dots \cup A_{n-1}$ such that each of A_i is defined in a different model m_i .

The fact that the relationship between two models can only be specified using a third model, so-called “helper” model, has been recognized in (Madhavan et al. 2002). The definition of mappings that the authors suggest can be viewed as a ternary relationship on model instances. A similar argument for ternary mappings was presented in (Pottinger and Bernstein 2003) in the context of the Merge operator. The database transformation language WOL can be used to express constraints that span multiple databases (Davidson et al. 1995a), just as the languages utilized for answering queries using views (Halevy 2001). In (Kementsietsidis et al. 2003), n -ary mappings

between peer-to-peer sources were considered. Therefore, we think that n -ary mappings provide a practically important generalization of the theory that we presented.

All operators that we discussed can be generalized for n -ary mappings. For example, the operator `Compose` becomes very similar to the relational equijoin operator, except that the join is performed on entire database states rather than attribute values. For example, for $map_1 \subseteq m_1 \times m_2 \times m_3$, $map_2 \subseteq m_1 \times m_3 \times m_4$, we write $map_1 \circ_{m_1, m_3} map_2 \subseteq m_2 \times m_4$. Operators `Domain` and `Range` can be generalized as the operator Π , which is similar to the relational projection operator: $\Pi_{m_1, m_3}(map_1) \subseteq m_1 \times m_3$.

Each n -ary mapping map can be viewed as a binary mapping between a k -ary and a $(n-k)$ -ary mapping, i.e., $map \subseteq (m_1 \times \dots \times m_k) \times (m_{k+1} \times \dots \times m_n)$, $1 \leq k \leq n$. In this way, we can adapt the definitions of the operators `Extract` and `Diff` of Sect. 4.2 with very little change. Both operators yield mappings of a smaller dimensionality for a given mapping, e.g., allow us to get $m_1_m_2$ from $m_1_m_2_m_3$. The grouping of an n -ary mapping into such quasi-binary mappings can be done in various ways. The operator `Invert` becomes obsolete, since the mapping positions that participate in composition, extraction, etc. need to be specified explicitly in each operator anyway.

The operator `Match` in general returns an n -ary mapping to reflect the fact that we may need one or more helper models to relate k given input models, $k \leq n$. For example, for $k = 2$, $n = 3$ we can write $m_1_m_2_H = \text{Match}(m_1, m_2)$. Model H and mappings m_1_H , m_2_H , and $m_1_m_2$ are implicitly contained in $m_1_m_2_H$ and can be obtained using the operator Π . To merge n models, we write $\text{Merge}(m_1, m_2, \dots, m_n, map)$, where map is an n -ary mapping. Extending the `Merge` operator for n -ary mappings and studying its properties may help analyze the associativity of `Merge` for binary mappings (see Conjecture 11.3.1).

n -ary mappings can be used to characterize dynamic scenarios, such as mediation between distributed services. For example, consider that answering a query q requires first consulting a data source m_1 , then formulating a query against m_2 using the data obtained from m_1 , and finally combining the results for the final answer. This mediation scenario can be characterized by a ternary mapping $q \subseteq m_1 \times m_2 \times r$, where r is the result schema for q . Specific execution strategies, such as caching subsets of results from m_1 and using them for later query processing are abstracted out in the ternary mapping.

11.3.5 Formalization of Model-Management Problems

A set of high-level operators with well-understood state-based semantics may be instrumental for finding agreement on a number of long-standing model-management problems and scenarios that have traditionally been addressed using heuristic or intuitive approaches. Data integration and schema evolution are two prominent examples of such problems. We suggested that the

two predominant kinds of data integration, database and view integration (Davidson et al. 1995a), can be described formally using the operator **Merge** (Sect. 4.2.4) and the operator **Intersect** (Sect. 11.3.3). We proposed a formal specification of schema evolution in Sect. 5.4. However, our work makes only a first step in understanding these scenarios precisely.

Many more important scenarios are outstanding. Examples include data exchange, mediation, or answering queries in a data integration setting. Characterizing these scenarios using model-management scripts is a promising direction for future research. Such scripts provide implementation guidelines for system developers and do so independently of the concrete schema and mapping languages deployed by the developers. The scripts can be used as formal specifications for driving customized solutions, i.e., they can be valuable even without a generic model-management system that executes them.

A. User Study: Gathering Intended Match Results

The user study was handed out to nine members of Stanford Database Group in February 2001. The task specifications have been reformatted to fit the page size used in the dissertation. The tables for entering the answers are omitted for brevity.

This user study attempts to collect various intended match results for a set of schema matching problems. General remarks:

1. The information provided about the source and target schemas is intentionally vague. Imagine a plausible scenario and try to map elements in both schemas according to the scenario you have in mind.
2. You don't have to match every element on the left and every one on the right, partial mappings are fine (if consistent with the scenario you have in mind).
3. $m : n$ correspondences between schema elements are welcome.
4. No mapping expressions are required.
5. The elements in the left and right schemas are numbered. Please fill out the table following every problem as shown in the example below.

When you are finished, please return your results to my office (438). Thanks a lot!

Example

This example shows schematically two XML schemas.

1	Cust	a	Customer
2	C#	b	CustID
3	CName	c	Company
4	FirstName	d	Contact
5	LastName	e	Phone

Many possible match results are conceivable for the schemas. Two of them are depicted below:

Left element(s)	Right element(s)
1	a
2	b
3	c
4,5	d

Left element(s)	Right element(s)
2	b
3	c,d

A.1 BizTalk Schemas (XML)

Left schema.

```

1 <Schema name="Schema 1"
   xmlns="urn:microsoft-com:xml-data">
2 <ElementType name="AccountOwner">
3   <element type="Name"/>
4   <element type="Address"/>
5   <element type="Birthdate"/>
6   <element type="TaxExempt"/>
   </ElementType>
7 <ElementType name="Address">
8   <element type="Street"/>
9   <element type="City"/>
10  <element type="State"/>
11  <element type="ZIP"/>
   </ElementType>
</Schema>

```

Right schema.

```

a <Schema name="Schema 2"
   xmlns="urn:microsoft-com:xml-data">
b <ElementType name="Customer">
c   <element type="Cname"/>
d   <element type="CAddress"/>
e   <element type="CPhone"/>
   </ElementType>
f <ElementType name="CustomerAddress">
g   <element type="Street"/>
h   <element type="City"/>
i   <element type="USState"/>
j   <element type="PostalCode"/>
   </ElementType>
</Schema>

```

A.2 Property Listing Schemas (XML)

1	HOUSE	a	listing
2	ADDRESS	b	location
3	COUNTY	c	area
4	PRICE	d	price
5	DESCRIPTION	e	comments
6	CONTACT-INFO	f	contact
7	OFFICE-INFO	g	agent
8	OFFICE-NAME	h	name
9	OFFICE-PHONE	i	office
10	AGENT-INFO	j	brokerage
11	AGENT-NAME	k	name
12	AGENT-PHONE	l	phone
		m	house-style

A.3 Library Schemas (XML)

1	<E n="Library">	a	<E n="Collection">
2	<E n="Item">	b	<E n="Document">
3	<e n="ISBN"/>	c	<e n="Identifier"/>
4	<e n="Author"/>	d	<e n="Creator"/>
5	<e n="Title"/>	e	<e n="Contributor"/>
6	<e n="Year"/>	f	<e n="Publisher"/>
	</E>	g	<e n="Title"/>
7	<E n="Author">	h	<e n="Year"/>
8	<e n="FirstName"/>		</E>
9	<e n="LastName"/>	i	<E n="Creator">
	</E>	j	<e n="Name"/>
10	<E n="BorrowedItems">		</E>
11	<e n="Item"/>	k	<E n="Name">
12	<e n="Borrower"/>	l	<e n="first"/>
	</E>	m	<e n="last"/>
13	<E n="Borrower">		</E>
14	<e n="FirstName"/>	n	<E n="Publisher">
15	<e n="LastName"/>	o	<e n="Address"/>
	</E>	p	<e n="Name"/>
</E>			</E>
			</E>

A.4 Product Schemas with Data Instances (XML)

In this problem, XML tags in both schemas need to be matched given two instances of schemas. The numbering enumerates all different tags on the left and on the right. Remember, you are matching the tag names, the particular instance values provide the hints for the matching process.

Left schema.

```

1 <amazon>
2   <item>
3     <title>Sony DCR-PC100
         Digital HandyCam Camcorder</title>
5     <listPrice>1899.99</listPrice>
6     <ourPrice>1699.00</ourPrice>
7     <youSave>200.00</youSave>
8     <review>
9       <avgReview>4.5</avgReview>
10      <numOfReviews>20</numOfReviews>
11    </review>
12    <availability>On Order;
        usually ships within 1-2
        weeks</availability>
13    <features>
14      <zoom>10x optical zoom</zoom>
15      <zoom>120x digital zoom</zoom>
16      <lcd>2.5inch LCD</lcd>
17      <other>4 MB Memory Stick included</other>
18    </features>
19  </item>
20 </amazon>

```

Right schema.

```

a <yahoo>
b   <productInfo>
c     <id>Sony DCR-PC100</id>
d     <merchantPrice>1799.94</merchantPrice>
e     <rating>
f       <userRating>3.5</userRating>
g       <userReviews>7</userReviews>
h     </rating>
i     <description>
j       <LCDScreenSize>2.5in</LCDScreenSize>
k       <opticalZoom>10x</opticalZoom>
l       <special>4MB Memory Stick</special>
m     </description>
n   </productInfo>
o </yahoo>

```

A.5 University Schemas with Data Instances (XML)

Same problem as the previous one: XML tags in both schemas need to be matched given two instances of schemas. The numbering enumerates all different tags on the left and on the right. Remember, you are matching the tag names, the particular instance values provide the hints for the matching process.

Left schema.

```

1  <db1>
2    <Faculty>
3      <SSN>234-56-7890</SSN>
4      <Facu_Name>Richie Solomon</Facu_Name>
5      <Salary>170000</Salary>
6    </Faculty>
7    <Student>
8      <Stud_ID>7206362</Stud_ID>
9      <Stud_Name>Teresa Lista</Stud_Name>
10     <Stipend>23000</Stipend>
11     <Tel>408-973 0110</Tel>
12   </Student>
13 </db1>

```

Right schema.

```

a  <db2>
b    <Personnel>
c      <ID>234-56-7890</ID>
d      <Name>Solomon, Richie</Name>
e      <Address>Sand Hill Road, Menlo Park, CA</Address>
f      <W_phone>(408) 495 8423</W_phone>
g      <H_phone>(650) 923 4193</H_phone>
h    </Personnel>
i    <Personnel>
j      <ID>7206362</ID>
k      <Name>Lista, Teresa</Name>
l      <Address>Cotton St, Palo Alto, CA</Address>
m      <W_phone>(408) 973 0110</W_phone>
n      <H_phone>(650) 198 2424</H_phone>
o    </Personnel>
p  </db2>

```

A.6 Catalogs with Data Instances (XML)

In this problem, catalog entries in both schemas need to be matched given two instances of schemas. The numbering enumerates all different catalog categories on the left and on the right.

Left schema.

```

<yahoo>
1  <cat id="Home">
2    <cat id="Electronics and Photography">
3      <cat id="Television and Video">
4        <cat id="Camcorders">
5          <cat id="By Format - DV">
6            <product name="SONY DCR-PC100"/>
7          </cat>
8        </cat>
9      </cat>
10     </cat>
11 </cat>

```

```

6      <cat id="Photography">
7      <cat id="Brands - Polaroid">
        <product name="POLAROID PDC 3000"/>
      </cat>
    </cat>
  </cat>
8  <cat id="Movies">
9  <cat id="Comedy">
10 <cat id="Parody">
11 <cat id="Science Fiction">
    <product name="Mars Attacks!"/>
  </cat>
</cat>
12 <cat id="Satire">
    <product name="The Graduate"/>
  </cat>
</cat>
</cat>
</cat>
</yahoo>

```

Right schema.

```

<epinions>
a  <cat id="Home">
b  <cat id="Electronics">
c  <cat id="Video">
d  <cat id="Camcorders">
    <product name="Sony DCR-PC100"/>
  </cat>
</cat>
e  <cat id="Photo">
f  <cat id="Cameras">
    <product name="Minolta Maxxum 9"/>
  </cat>
</cat>
g  <cat id="Arts and Entertainment">
h  <cat id="Movies">
i  <cat id="Video">
    <product name="The Graduate"/>
    <product name="Mars Attacks!"/>
  </cat>
</cat>
</cat>
</cat>
</epinions>

```


A.7 Personnel Schemas (Relational)

The numbering enumerates tables and columns in both schemas.

```

1 CREATE TABLE Personnel (      a CREATE TABLE Employee (
2   Pno int,                    b   EmpNo int PRIMARY KEY,
3   Pname string,              c   EmpName varchar(50),
4   Dept string,              d   DeptNo int REFERENCES
5   Born date,                Department,
   UNIQUE perskey (Pno)      e   Salary dec(15,2),
   );                          f   Birthdate date
                               );
                               g CREATE TABLE Department (
                               h   DeptNo int PRIMARY KEY,
                               i   DeptName varchar(70)
                               );

```

A.8 University Schemas (Relational)

Table on the right presents a previous version of the schema shown on the left. The left schema is the evolved schema. Match the new version of the schema onto the old one!

Left schema.

```

1 CREATE TABLE Address (
2   Id int PRIMARY KEY,
3   Street string,
4   City string,
5   PostalCode int
   );

6 CREATE TABLE Professor (
7   Id int PRIMARY KEY,
8   Name string,           # name
9   Sal double,           # salary
10  addr int              # address
   );

11 CREATE TABLE Student (
12  Name string,          # name
13  GPA double,          # grade point avg
14  Yr int               # year of studies
   );

15 CREATE TABLE PayRate (
16  Rank int PRIMARY KEY, # project rank
17  HrRate double        # hourly pay rate
   );

```

```

18 CREATE TABLE WorksOn (
19   Name string,           # name of student
20   Proj string,          # project name
21   Hrs int,              # hours spent
22   ProjRank int         # project rank
   );

```

Right schema.

```

a CREATE TABLE Professor (
b   Id int PRIMARY KEY,
c   Name string,
d   Salary double,
e   Address string
   );

f CREATE TABLE Student (
g   Name string,
h   GradePointAverage double,
i   Year int
   );

j CREATE TABLE WorksOn (
k   StudentName string,
l   Project string,
m   Expenses double
   );

```

A.9 Personnel/University Schemas (Relational)

Left schema. is the same as in the previous example. It deals with professors, students, and provides the information about who worked on which project for how long. Moreover, the schema contains information about payment of professors and students.

Right schema. captures general personnel information:

```

a CREATE TABLE Personnel (
b   Id int PRIMARY KEY,
c   Name string,
d   Sal double,           # salary
e   Addr string         # address
   );

```

Hints. for interpretation of schemas:

- WorksOn(ProjRank) may or may not be foreign key for PayRate(Rank)
- WorksOn(Name) may or may not be foreign key for Professor(Name) or Student(Name)
- Student(Yr) may or may not be foreign key of PayRate(Rank)
- Pay rate of a student may or may not depend on the year and/or his/hers grades

B. Proofs of Simplification Theorems

In this appendix, we prove the Theorems 4.2.1, 4.2.3, and 4.2.5, which provide simplified characterization of operators `Extract`, `Merge`, and `Diff`, respectively. For convenience, we repeat the definition of the equivalence relation $ind(\cdot, \cdot, m_m')$:

$$ind(y_1, y_2, m_m') =_{df} (\{z_1 \mid (y_1, z_1) \in m_m'\} = \{z_2 \mid (y_2, z_2) \in m_m'\})$$

If $ind(y_1, y_2, m_m')$, we say that y_1 and y_2 are indistinguishable under m_m' .

B.1 Extract Operator

Theorem 4.2.1 (from page 69). Let $\text{Domain}(m_m') \subseteq m$. $\langle m_x, m_m_x \rangle = \text{Extract}(m, m_m')$ holds if and only if the following conditions are satisfied:

1. $m_x = \text{Range}(m_m_x)$.
2. $\text{Domain}(m_m_x) = \text{Domain}(m_m')$.
3. For all $(y_1, x_1), (y_2, x_2) \in m_m_x$: $x_1 = x_2$ iff $ind(y_1, y_2, m_m')$.

Condition (2) makes sure that exactly those instances of m participate in m_m_x that are connected in m_m' . Condition (3) requires collapsing any two instances y_1 and y_2 of m into a single instance of m_x if and only if y_1 and y_2 are indistinguishable under m_m' .

Proof: First, we simplify condition (ii), i.e., the equality of mappings $m_m_x \circ \text{Invert}(m_m_x) \circ m_m'$ and m_m' . Notice that for any two mappings map_1 and map_2 , $map_1 = map_2$ holds if and only if: $\text{Domain}(map_1) = \text{Domain}(map_2)$ and for each $x \in \text{Domain}(map_1)$: $\{y \mid (x, y) \in map_1\} = \{y \mid (x, y) \in map_2\}$.

In the composition $m_m_x \circ \text{Invert}(m_m_x)$, the range of m_m_x is identical with the domain of $op\text{Invert}(m_m_x)$. Thus, the composition does not drop instances from the domain of $m_m_x \circ \text{Invert}(m_m_x)$. Therefore, $\text{Domain}(m_m_x \circ \text{Invert}(m_m_x) \circ m_m') = \text{Domain}(m_m')$ iff $\text{Domain}(m_m_x) = \text{Domain}(m_m')$.

Let $y \in \text{Domain}(m_m_x)$. If we traverse y over m_m_x to m_x and back, we obtain the set of round-tripped images $Rt(y) = \{y' \mid (y, x) \in m_m_x \text{ and } (y', x) \in m_m_x\}$, with $y \in Rt(y)$. Traversing from y over m_m'

directly must give us the identical set of m' -images as by first round-tripping y to $Rt(y)$ and traversing each $y' \in Rt(y)$ over m_m' . That is, for each $y \in \text{Domain}(m_m_x)$ and each $y' \in Rt(y)$: $\{z \mid (y, z) \in m_m'\} = \{z \mid (y', z) \in m_m'\}$. In other words: for all $y \in \text{Domain}(m_m_x)$, $y' \in Rt(y) : \text{ind}(y, y', m_m')$. Now we expand the definition of Rt again and get an equivalent expression: for all $y_1, y_2 \in \text{Domain}(m_m_x)$ with $(y_1, x), (y_2, x) \in m_m_x : \text{ind}(y_1, y_2, m_m')$. This expression can be further simplified as: if $(y_1, x), (y_2, x) \in m_m_x$, then $\text{ind}(y_1, y_2, m_m')$.

That is, condition (ii) is equivalent to the conjunction: $\text{Domain}(m_m_x) = \text{Domain}(m_m')$ and for all $(y_1, x), (y_2, x) \in m_m_x : \text{ind}(y_1, y_2, m_m')$.

Now we turn to the actual proof. First we show that the conditions (1)-(3) stated in Theorem 4.2.1 are necessary, i.e., they follow from Definition 4.2.3. Then, we demonstrate that they are also sufficient.

(\rightarrow) Let conditions (i)-(iii) hold. Conditions (1) and (2) are satisfied trivially.

To prove condition (3), let $(y_1, x), (y_2, x) \in m_m_x$ with $y_1 \neq y_2$. Then, $\text{ind}(y_1, y_2, m_m')$ follows immediately from (ii). Now, let $(y_1, x_1), (y_2, x_2) \in m_m_x$ with $y_1 \neq y_2$ and $x_1 \neq x_2$. Assume that $\text{ind}(y_1, y_2, m_m')$ holds. Since $\text{ind}(., ., .)$ is transitive, then for all y_1, y_2 with $(y_1, x_1), (y_2, x_2) \in m_m_x : \text{ind}(y_1, y_2, m_m')$. Observe that condition (ii) remains true when we set $x_1 = x_2$. We can construct a smaller model $m'_x = m_x - \{x_2\}$ and a mapping $m_m'_x$, in which x_2 is substituted by x_1 . m'_x and $m_m'_x$ satisfy (i)-(ii), but $m_x \leq m'_x$ does not hold. This yields a contradiction to (iii), so that our assumption is false and $\text{ind}(y_1, y_2, m_m')$ does not hold. We have shown that all conditions (1)-(3) stated in Theorem 4.2.1 follow from Definition 4.2.3.

(\leftarrow) Now we prove the reverse. Let conditions (1)-(3) hold. Trivially, if (1) then (i). Conjunction of (2) and (3) is obviously more restrictive than condition (ii), so (ii) holds. We show the minimality condition (iii) using the following approach. First, we establish a lower bound on the number of instances that m_x must have as $|m_x| \geq k$ using conditions (i) and (ii). Then, we show that if (1)-(3) are satisfied, then m_x necessarily has k instances, so it is a minimal model with (i)-(ii).

The lower bound is established by condition (ii). Recall that $\text{ind}(., ., m_m')$ is an equivalence relation. It yields a disjoint decomposition Π of instances in $\text{Domain}(m_m')$, i.e., of all instances of m that are connected in m_m' . By condition (ii), each equivalence class $c \in \Pi$ must be associated with a distinct instance in m_x . The proof is by contradiction: let $c_1, c_2 \in \Pi$, $y_1 \in c_1$, $y_2 \in c_2$, and $(y_1, x), (y_2, x) \in m_m_x$, i.e., instance x is shared among c_1 and c_2 . Then, by condition (ii), $\text{ind}(y_1, y_2, m_m')$ holds and thus c_1, c_2 are not disjoint – we obtained a contradiction. That is, $|m_x| \geq k = |\Pi|$.

Now, we demonstrate that conditions (1)-(3) imply $|m_x| = |\Pi|$. By condition (3), if $(y_1, x_1), (y_2, x_2) \in m_m_x$ and $\text{ind}(y_1, y_2, m_m')$, then $x_1 = x_2$. That is, all instances from the same equivalence class $c \in \Pi$ must be associated with an identical instance $x \in m_x$. By condition (2),

$\text{Domain}(m_{_}m_x) = \text{Domain}(m_{_}m')$, i.e., each instance of m_x is associated with some $y \in c$, $c \in \Pi$. Hence, $|m_x| \leq |\Pi|$. We already have $|m_x| \geq |\Pi|$, since (i)-(iii) imply (1)-(3). Therefore, $|m_x| = |\Pi|$, and (i)-(iii) from Definition 4.2.3 follow from (1)-(3) of Theorem 4.2.1. ■

B.2 Merge Operator

Lemma B.2.1. *Let $m_{1_}m_2 \subseteq m_1 \times m_2$. $\langle m, m_{_}m_1, m_{_}m_2 \rangle = \text{Merge}(m_1, m_2, m_{1_}m_2)$ holds if and only if the conditions (i)-(iii) of Definition 4.2.4 are satisfied and*

- $S_1 = m - \text{Domain}(m_{_}m_2) \cong m_1 - \text{Domain}(m_{1_}m_2)$,
- $S_2 = m - \text{Domain}(m_{_}m_1) \cong m_2 - \text{Range}(m_{1_}m_2)$,
- $S_3 = \text{Domain}(m_{_}m_1) \cap \text{Domain}(m_{_}m_2) \cong m_{1_}m_2$

for disjoint partitions S_1, S_2, S_3 of m , $S_1 \cup S_2 \cup S_3 = m$. ■

Proof: We show that each model with (i)-(iv) can be partitioned as suggested in the proposition. The partitioning determines m uniquely up to isomorphism for fixed m_1, m_2 , and $m_{1_}m_2$. This implies that each model m' with (i)-(iii) that is isomorphic to m is minimal and so satisfies (iv).

Let conditions (i)-(iv) hold. We partition m into four sets, S_0, S_1, S_2, S_3 :

- $S_0 = \{z \mid z \in m \text{ and } z \notin \text{Domain}(m_{_}m_1) \text{ and } z \notin \text{Domain}(m_{_}m_2)\}$.
- $S_1 = \{z \mid z \in m \text{ and } z \in \text{Domain}(m_{_}m_1) \text{ and } z \notin \text{Domain}(m_{_}m_2)\}$.
- $S_2 = \{z \mid z \in m \text{ and } z \notin \text{Domain}(m_{_}m_1) \text{ and } z \in \text{Domain}(m_{_}m_2)\}$.
- $S_3 = \{z \mid z \in m \text{ and } z \in \text{Domain}(m_{_}m_1) \text{ and } z \in \text{Domain}(m_{_}m_2)\}$.

By construction, S_0, S_1, S_2, S_3 are pairwise disjoint with $S_0 \cup S_1 \cup S_2 \cup S_3 = m$. First, we show that $S_0 = \emptyset$. Let $z \in m$ and $z \notin \text{Domain}(m_{_}m_1)$ and $z \notin \text{Domain}(m_{_}m_2)$. Then, obviously the model $m' = m - \{z\}$ with the same $m_{1_}m, m_{2_}m$ satisfies (i)-(iii) and thus (iv) is violated. By contradiction, there is no z with these properties, and $S_0 = \emptyset$.

Due to condition (iii), $\text{Domain}(m_{_}m_1) \subseteq m$ and $\text{Domain}(m_{_}m_2) \subseteq m$. Therefore, we can simplify the definitions of S_1, S_2 and S_3 as follows:

- $S_1 = m - \text{Domain}(m_{_}m_2)$
- $S_2 = m - \text{Domain}(m_{_}m_1)$
- $S_3 = \text{Domain}(m_{_}m_1) \cap \text{Domain}(m_{_}m_2)$.

Let $x \in m_1 - \text{Domain}(m_{1_}m_2)$. Then, $x \in \text{Range}(m_{_}m_1)$ and there exists z with $(z, x) \in m_{_}m_1$. Since $\text{Domain}(m_{_}m_1)$ is fully contained in m by condition (iii), so $z \in m$. Assume that there exists y with $(z, y) \in m_{_}m_2$. Then, by (ii), $(x, y) \in m_{1_}m_2$ and $x \in \text{Domain}(m_{1_}m_2)$. We arrived at a contradiction. Hence, our assumption was false and $z \notin \text{Domain}(m_{_}m_2)$.

Now, assume that there exists $z' \neq z$ with $(z', x) \in m_{_}m_1$. We construct $m' = m - \{z'\}$ and $m'_{_}m_1 = m_{_}m_1 - \{(z', x)\}$. Since $z' \notin \text{Domain}(m_{_}m_2)$ and $x \notin \text{Domain}(m_1_{_}m_2)$, removing (z', x) from $m_{_}m_1$ preserves condition (ii). Conditions (i) and (iii) are satisfied trivially. Hence, we obtained a smaller model m' . By contradiction to (iv), we conclude that z is determined uniquely. That is, there is a function from $m_1 - \text{Domain}(m_1_{_}m_2)$ into m . But since $m_{_}m_1$ is a surjective function, then there is a bijection $f_1 \subseteq m_1_{_}m$ between $m_1 - \text{Domain}(m_1_{_}m_2)$ and $S_1 = m - \text{Domain}(m_{_}m_2)$.

Analogously, we show that there is a bijection $f_2 \subseteq m_2_{_}m$ between $m_2 - \text{Range}(m_1_{_}m_2)$ and $S_2 = m - \text{Domain}(m_{_}m_1)$.

Finally, we demonstrate that $S_3 \cong m_1_{_}m_2$. Let $(x, y) \in m_1_{_}m_2$. By condition (ii) there exists $z \in \text{Domain}(m_{_}m_1)$ with $(z, x) \in m_{_}m_1$ and $(z, y) \in m_{_}m_2$. By (iii), $z \in m$. Now, let $(z_1, x), (z_2, x) \in m_{_}m_1$, and $(z_1, y), (z_2, y) \in m_{_}m_2$. Assume that $z_1 \neq z_2$. We construct $m' = m - \{z_2\}$, $m'_{_}m_1 = m_{_}m_1 - \{(z_2, x)\}$, and $m'_{_}m_2 = m_{_}m_2 - \{(z_2, y)\}$. By condition (i), there is no $x' \neq x$ with $(z_1, x') \in m'_{_}m_1$ or $(z_2, x') \in m'_{_}m_1$, and there is no $y' \neq y$ with $(z_1, y') \in m'_{_}m_2$ or $(z_2, y') \in m'_{_}m_2$. Thus, $m', m'_{_}m_1, m'_{_}m_2$ satisfy (i)-(iii). Hence, we found a smaller model m' that satisfies (i)-(iii). This is a contradiction to (iv). Therefore, our assumption is false and $z_1 = z_2$, i.e., (x, y) determine $z \in m$ uniquely. Now, let $z \in S_3$. By condition (iii), there exists $(x, y) \in m_1_{_}m_2$ with $(z, x) \in m_{_}m_1$ and $(z, y) \in m_{_}m_2$. x and y are determined uniquely by condition (i). That is, there is a bijection g between $m_1_{_}m_2$ and S_3 . Hence, $S_3 \cong m_1_{_}m_2$. ■

Lemma B.2.1 implies that the output model m in **Merge** is determined up to isomorphism. Thus, we can further simplify the definition of **Merge** as follows.

Theorem 4.2.3 (from page 74). Let $m_1_{_}m_2 \subseteq m_1 \times m_2$. $\langle m, m_{_}m_1, m_{_}m_2 \rangle = \text{Merge}(m_1, m_2, m_1_{_}m_2)$ holds if and only if

- the conditions (i)-(iii) of Definition 4.2.4 are satisfied, and
- $|m| = \text{mergeCard}(m_1, m_2, m_1_{_}m_2)$, where $\text{mergeCard}(m_1, m_2, m_1_{_}m_2) =_{\text{df}} |m_1_{_}m_2| + |m_1 - \text{Domain}(m_1_{_}m_2)| + |m_2 - \text{Range}(m_1_{_}m_2)|$.

If $m_1_{_}m_2$ is total and surjective, or if $m_{_}m_1$ and $m_{_}m_2$ are total, then $\text{mergeCard}(m_1, m_2, m_1_{_}m_2) = |m_1_{_}m_2|$.

Proof: The sets S_1, S_2, S_3 of Lemma B.2.1 constitute a disjoint decomposition of m . Thus, each model m with (i)-(iv) has exactly $k = |S_1| + |S_2| + |S_3|$ instances. This implies that any other model with (i)-(iii) has at least k instances. Thus, given a model m' with (i)-(iii) that has k instances we can conclude that it is minimal and so satisfies (iv). By Lemma B.2.1, $k = |S_1| + |S_2| + |S_3| = |m_1_{_}m_2| + |m - \text{Domain}(m_{_}m_1)| + |m - \text{Domain}(m_{_}m_2)| = |m_1_{_}m_2| + |m_1 - \text{Domain}(m_1_{_}m_2)| + |m_2 - \text{Range}(m_1_{_}m_2)| = \text{mergeCard}(m_1, m_2, m_1_{_}m_2)$. If $m_1_{_}m_2$ is total and surjective, or if $m_{_}m_1$ and $m_{_}m_2$ are total, then $S_1 = S_2 = \emptyset$ and hence $k = |m_1_{_}m_2|$. ■

B.3 Diff Operator

Lemma B.3.1. *Let $\text{Domain}(m_m') \subseteq m$. $\langle m_d, m_m_d \rangle = \text{Diff}(m, m_m')$ holds if and only if the following conditions are satisfied:*

1. m_m_d is a surjective function from m onto m_d .
2. For all $y_1, y_2 \in \text{Domain}(m_m')$ with $y_1 \neq y_2$ and $\text{ind}(y_1, y_2, m_m')$ there exist $(y_1, d_1), (y_2, d_2) \in m_m_d$ with $d_1 \neq d_2$.
3. If $y \in m - \text{Domain}(m_m')$, then there exists $(y, d) \in m_m_d$ and $\{y' \mid (y', d) \in m_m_d\} = \{y\}$.
4. m_d is a minimal model with (1)-(3). ■

Condition (2) ensures that the instances of m that are indistinguishable in m_m' become distinguishable in m_m_d . Condition (3) requires each instance of m that does not participate in m_m' to have a counterpart in m_d that is not connected to any other instance of m . It ensures that Diff picks up the instances of m that get lost upon extraction. Condition (4) makes the result of Diff minimal.

Proof: Conditions (iii) and (4) are identical. We show that conditions (i)-(ii) of Definition 4.2.5 are equivalent with (1)-(3). First, we rewrite the conditions (i)-(iii) by expanding the alternative definitions of Extract and Merge from Theorem 4.2.1 and Theorem 4.2.3 and removing tautologies. We obtain the following:

- a. m_m_x and m_m_d are surjective functions onto m_x and m_d , respectively.
- b. $\text{Domain}(m_m_x) = \text{Domain}(m_m')$.
- c. For all $(y_1, x_1), (y_2, x_2) \in m_m_x : x_1 = x_2$ iff $\text{ind}(y_1, y_2, m_m')$.
- d. $m = \text{Domain}(m_m_x) \cup \text{Domain}(m_m_d)$.
- e. $m_x_m_d = \text{Invert}(m_m_x) \circ m_m_d$.
- f. The statements below hold for pairwise disjoint partitions S_1, S_2, S_3 of m , $S_1 \cup S_2 \cup S_3 = m$:
 - $S_1 = m - \text{Domain}(m_m_d) \cong m_x - \text{Domain}(m_x_m_d)$,
 - $S_2 = m - \text{Domain}(m_m_x) \cong m_d - \text{Range}(m_x_m_d)$,
 - $S_3 = \text{Domain}(m_m_x) \cap \text{Domain}(m_m_d) \cong m_x_m_d$.

(\rightarrow) We show that conditions (1)-(3) of Lemma B.3.1 follow from (a)-(f).

Let (a)-(f) hold. Condition (1) follows immediately from (a). Now, let $y_2, y_1 \in \text{Domain}(m_m')$ with $y_1 \neq y_2$ and $\text{ind}(y_1, y_2, m_m')$. By condition (b), $y_1, y_2 \in \text{Domain}(m_m_x)$. Therefore, there exist $(y_1, x_1), (y_2, x_2) \in m_m_x$. Since $\text{ind}(y_1, y_2, m_m')$, then due to condition (c), $x_1 = x_2$. That is, there exist $(y_1, x), (y_2, x) \in m_m_x$. Assume that $y_1 \notin \text{Domain}(m_m_d)$, i.e., $y_1 \in m - \text{Domain}(m_m_d)$. Then, by Theorem 4.2.3, y_1 is determined uniquely by x , a contradiction to $y_1 \neq y_2$. Thus, $y_1 \in \text{Domain}(m_m_d)$. Analogously, $y_2 \in \text{Domain}(m_m_d)$. Let $(y_1, d_1), (y_2, d_2) \in m_m_d$. Assume

that $d_1 = d_2 = d$. Then, by Theorem 4.2.3, (x, d) determine y uniquely, a contradiction to $y_1 \neq y_2$. Hence, $d_1 \neq d_2$ and condition (2) holds.

To show condition (3), let $y \in m - \text{Domain}(m_m')$. Then, by condition (b), $y \in m - \text{Domain}(m_m_x)$. By Theorem 4.2.3, there exist uniquely determined $d \in m_d - \text{Range}(m_x_m_d)$ such that y is the only instance of m with $(y, d) \in m_m_d$.

(\leftarrow) We prove that conditions (1)-(3) of Lemma B.3.1 imply (a)-(f). Let (1)-(3) hold. Conditions (a)-(c), which come from Extract, determine m_x and m_m_x uniquely up to isomorphism. Let m_x and m_m_x be fixed with (a)-(c). Condition (1) also states that m_m_d is a surjective function onto m_d , thus (a) is satisfied.

We show condition (d): $m = \text{Domain}(m_m_x) \cup \text{Domain}(m_m_d) = \text{Domain}(m_m') \cup \text{Domain}(m_m_d)$. Let $y \in m$. If $y \in \text{Domain}(m_m')$, then trivially $m \subseteq \text{Domain}(m_m') \cup \text{Domain}(m_m_d)$. Let $y \notin \text{Domain}(m_m')$. By condition (3), there exists $(y, d) \in m_m_d$, therefore $y \in \text{Domain}(m_m_d)$. By the assumption of Lemma B.3.1, $\text{Domain}(m_m') \subseteq m$. By condition (1), $\text{Domain}(m_m_d) \subseteq m$. Therefore, $\text{Domain}(m_m_d) \cup \text{Domain}(m_m') \subseteq m$, and the equality (d) follows.

Finally, we show condition (f), which can be stated as

- $S_1 = m - \text{Domain}(m_m_d) \cong m_x - \text{Domain}(\text{Invert}(m_m_x) \circ m_m_d)$,
- $S_2 = m - \text{Domain}(m_m_x) \cong m_d - \text{Range}(\text{Invert}(m_m_x) \circ m_m_d)$,
- $S_3 = \text{Domain}(m_m_x) \cap \text{Domain}(m_m_d) \cong \text{Invert}(m_m_x) \circ m_m_d$.

S_1 : Let $y \in m - \text{Domain}(m_m_d)$. Then, there are two possibilities: either $y \in m - \text{Domain}(m_m_x)$ or $y \notin m - \text{Domain}(m_m_x)$. In the former case, we obtain $y \in m - \text{Domain}(m_m_x) = S_2$. This is a contradiction, since S_1 and S_2 are disjoint. Thus, $y \notin m - \text{Domain}(m_m_x)$. In other words, $y \in \text{Domain}(m_m_x)$. Hence, there exists $(y, x) \in m_m_x$. Since m_m_x is a function, x is determined uniquely by y . By (1), $x \in m_x$. Assume that $x \in \text{Domain}(\text{Invert}(m_m_x) \circ m_m_d)$. That is, there must exist $(y, d) \in m_m_d$ for the composition not to drop x . Consequently, $y \in \text{Domain}(m_m_d)$. This is a contradiction to our assumption $y \in m - \text{Domain}(m_m_d)$. Therefore, $x \in m_x - \text{Domain}(\text{Invert}(m_m_x) \circ m_m_d)$. Now, let $x \in m_x - \text{Domain}(\text{Invert}(m_m_x) \circ m_m_d)$. That is, $x \notin \text{Domain}(\text{Invert}(m_m_x) \circ m_m_d)$. By (1), $x \in \text{Range}(m_m_x)$. Let $(z, x) \in m_m_x$. By (d), $z \in m$. For the composition to fail, $z \notin \text{Domain}(m_m_d)$. Thus, $z \in m - \text{Domain}(m_m_d)$. Assume that we have two such instances $z_1 \neq z_2$ with $(z_1, x), (z_2, x) \in m_m_x$. By (b), $z \in \text{Domain}(m_m_x) = \text{Domain}(m_m')$. Hence, by condition (c), $\text{ind}(z_1, z_2, m_m')$ holds. But then, condition (2) applies and there exist $(z_1, d_1), (z_2, d_2) \in m_m_d$ with $d_1 \neq d_2$. However, $z_1, z_2 \notin \text{Domain}(m_m_d)$. Thus, we obtained a contradiction and z is determined uniquely.

S_2 : Let $y \in m - \text{Domain}(m_m_x)$. By (b), $y \in m - \text{Domain}(m_m')$. Then, by condition (3) there exists a unique $d \in \text{Range}(m_m_d)$ with $(y, d) \in$

m_m_d . Assume that $d \in \text{Range}(\text{Invert}(m_m_x) \circ m_m_d)$. Since m_m_d is a function, $d \in \text{Domain}(m_m_d)$ is determined uniquely by y . Therefore, there must exist $(y, x) \in m_m_x$ for the composition to produce d . Thus, $y \in \text{Domain}(m_m_x)$. This contradicts our assumption, therefore, $d \in m_d - \text{Range}(\text{Invert}(m_m_x) \circ m_m_d)$.

Now, let $d \in m_d - \text{Range}(\text{Invert}(m_m_x) \circ m_m_d)$. Since m_m_d is surjective, $d \in \text{Range}(m_m_d)$. Thus, there exists $(z, d) \in m_m_d$ with $z \notin \text{Domain}(m_m_x)$. By (b), $z \in m - \text{Domain}(m_m')$. Assume that we have two such instances, $z_1 \neq z_2$. Since m_m_d is a function, they both map to d . This is a contradiction to (3). Thus, z is determined uniquely.

S_3 : Let $y \in \text{Domain}(m_m_x) \cap \text{Domain}(m_m_d)$. Then, trivially, there exist $(z, x) \in m_m_x$ and $(z, d) \in m_m_d$. By (a), m_m_x and m_m_d are functions. Therefore, $(x, d) \in \text{Invert}(m_m_x) \circ m_m_d$ is determined uniquely. Now, let $(x, d) \in \text{Invert}(m_m_x) \circ m_m_d$. Thus, there exists $z \in \text{Domain}(m_m_x) \cap \text{Domain}(m_m_d)$ with $(z, x) \in m_m_x$ and $(z, d) \in m_m_d$. Assume that $(z', x) \in m_m_x$ and $(z', d) \in m_m_d$. Then, we obtain a contradiction to (1). Hence, instance z is determined uniquely. ■

Although Lemma B.3.1 simplifies Definition 4.2.5 substantially, the presence of the minimality condition is still unsatisfactory. The following theorem substitutes the minimality condition by a precise lower bound and helps us to argue the correctness of the subsequent examples. The construction used in the proof of the theorem can be exploited to find a valid solution for Diff for concrete schema and mapping languages.

Theorem 4.2.5 (from page 78). Let $\text{Domain}(m_m') \subseteq m$. $\langle m_d, m_m_d \rangle = \text{Diff}(m, m_m')$ holds if and only if conditions (1)-(3) of Lemma B.3.1 are satisfied and $|m_d| = \text{diffCard}(m, m_m')$, where $\text{diffCard}(m, m_m') =_{\text{df}} \max\{|c| : c \in \Pi \cup \emptyset, |c| \neq 1\} + |m - \text{Domain}(m_m')|$ and Π is a partitioning of $\text{Domain}(m_m')$ by $\text{ind}(., ., m_m')$. If m_m' is total, $\text{diffCard}(m, m_m') = \max\{|c| : c \in \Pi \cup \emptyset, |c| \neq 1\}$.

Proof: Let Π be a partitioning of $\text{Domain}(m_m')$ by $\text{ind}(., ., m_m')$. By condition (3) of Lemma B.3.1, m_d contains a distinct instance for each $y \in m - \text{Domain}(m_m')$. Moreover, $\{d \mid (y, d) \in m_m_d \text{ and } y \in m - \text{Domain}(m_m')\}$ is disjoint with $\{d \mid (y, d) \in m_m_d \text{ and } y \in \cup \Pi\}$. Let c_{\max} be a maximal equivalence class of Π . Condition (2) requires m_d to have a distinct instance for each $y \in \Pi$. If $|c_{\max}| = 1$, then there are no two distinct indistinguishable instances in m , and condition (2) is satisfied trivially. Otherwise, $|m_d| \geq |c_{\max}| + |m - \text{Domain}(m_m')|$. Taking into account the case when $|c_{\max}| = 1$, we obtain $|m_d| \geq \max\{|c| : c \in \Pi \cup \{\emptyset\}, |c| \neq 1\} + |m - \text{Domain}(m_m')| = k$.

Next we prove by construction that there always exist m_m_d with $|m_d| = k$ that satisfies (1)-(3). That will allow us to conclude that each m_d must have the cardinality of exactly k due to condition (4). We construct m_m_d

as follows. Notice that for each equivalence class $c = \{y_1, \dots, y_p\} \in \Pi$, there exist a total injective function $f_c : c \rightarrow c_{max}$, since c_{max} is maximal. Let f be a mapping defined as $f = \cup\{f_c : c \in \Pi\}$. f is a surjective function onto c_{max} . Let g be a bijection from $m\text{-Domain}(m_m')$ onto some set S , such that $S \cap \text{Range}(f) = \emptyset$. Now, let $m_m_d = f \cup g$ and $m_d = \text{Range}(m_m_d) = c_{max} \cup S$. By construction, m_m_d is a surjective function with $|m_d| = k$ that satisfies (1)-(3). ■

References

- Abiteboul, S. and Duschka, O. M. 1998. Complexity of Answering Queries Using Materialized Views. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. ACM Press, 254–263.
- Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley, Reading, Mass.
- Agarwal, S., Keller, A. M., Wiederhold, G., and Saraswat, K. 1995. Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, P. S. Yu and A. L. P. Chen, Eds. IEEE Computer Society, 495–504.
- Agrawal, R., Somani, A., and Xu, Y. 2001b. Storage and Querying of E-Commerce Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 149–158.
- Agrawal, S., Chaudhuri, S., and Narasayya, V. R. 2001a. Materialized View and Index Selection Tool for Microsoft SQL Server 2000. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Ahn, I. 1994. Database Issues in Telecommunications Network Management. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 37–43.
- Aiken, A., Widom, J., and Hellerstein, J. M. 1992. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 59–68.
- Alagic, S. and Bernstein, P. A. 2001. A Model Theory for Generic Schema Management. In *Proc. DBPL 2001, Springer, LNCS*. 228–246.
- Anyanwu, K. and Sheth, A. 2002. The ρ operator: Discovering and Ranking Associations on the Semantic Web. In *SIGMOD Record*. 42–47.
- Atzeni, P., Ausiello, G., Batini, C., and Moscarini, M. 1982. Inclusion and Equivalence between Relational Database Schemata. *Theoretical Computer Science* 19, 267–285.
- Atzeni, P. and Torlone, R. 1996. Management of Multiple Models in an Extensible Database Design Tool. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*. 79–95.
- Bancilhon, F. and Spyrtatos, N. 1981. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)* 6, 4, 557–575.
- Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. 1987. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 311–322.

- Baral, C., Kraus, S., and Minker, J. 1991. Combining Multiple Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 3, 2, 208–220.
- Barsalou, T. and Gangopadhyay, D. 1992. M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, F. Golshani, Ed. IEEE Computer Society, 218–227.
- Batini, C., Lenzerini, M., and Navathe, S. B. 1986. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys* 18, 4, 323–364.
- Becker, P. 1996. Verteiltes Modell-Management und Objektbanken für diskrete Probleme und diskrete Strukturen. Ph.D. thesis, University of Tübingen.
- Bell, J. 1988. *Toposes and Local Set Theories: An Introduction*. Oxford University Press.
- Bergamaschi, S., Castano, S., and Vincini, M. 1999. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Record* 28, 1, 54–59.
- Bergamaschi, S., Castano, S., Vincini, M., and Beneventano, D. 2001. Semantic Integration of Heterogeneous Information Sources. *Data and Knowledge Engineering* 36, 3, 215–249.
- Berlin, J. and Motro, A. 2001. Autoplex: Automated Discovery of Content for Virtual Databases. In *Proc. Intl. Conf. on Cooperative Information Systems (CoopIS)*. 108–122.
- Berlin, J. and Motro, A. 2002. Database Schema Matching Using Machine Learning with Feature Selection. In *CAiSE*. 452–466.
- Bernstein, P. A. 1999. Review – EXPRESS: A Data EXtraction, Processing, and REStructuring System. *ACM SIGMOD Digital Review* 1.
- Bernstein, P. A. 2003. Applying Model Management to Classical Metadata Problems. In *Proc. of the 1st Biennial Conf. on Innovative Data Systems Research (CIDR)*.
- Bernstein, P. A. and Bergstraesser, T. 1999. Meta-Data Support for Data Transformations Using Microsoft Repository. *IEEE Data Engineering Bulletin* 22, 1, 9–14.
- Bernstein, P. A., Bergstraesser, T., Carlson, J., Pal, S., Sanders, P., and Shutt, D. 1999. Microsoft Repository Version 2 and the Open Information Model. *Information Systems* 24, 2, 71–98.
- Bernstein, P. A., Haas, L. M., Jarke, M., Rahm, E., and Wiederhold, G. 2000a. Panel: Is Generic Metadata Management Feasible? In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 660–662.
- Bernstein, P. A., Halevy, A. Y., and Pottinger, R. 2000b. A Vision of Management of Complex Models. *SIGMOD Record* 29, 4, 55–63.
- Bernstein, P. A. and Rahm, E. 2000. Data Warehouse Scenarios for Model Management. In *Intl. Conf. on Conceptual Modeling (ER) 2000*. LNCS, Springer, 1–15.
- Biskup, J. and Convent, B. 1986. A Formal View Integration Method. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 398–407.
- Blanning, R. W. 1982. Data Management and Model Management: a Relational Synthesis. In *Proc. of the 20th ACM Southeast Regional Conf.* 139–147.

- Blott, S. and Vckovski, A. 1995. Accessing Geographical Metafiles through a Database Storage System. In *Advances in Spatial Databases, 4th Intl. Symposium, SSD'95*. 117–131.
- Bohannon, P., Freire, J., Haritsa, J. R., Ramanath, M., Roy, P., and Simeon, J. 2002. LegoDB: Customizing Relational Storage for XML Documents. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 1091–1094.
- Borgida, A. 1995. Description Logics in Data Management. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 7, 5, 671–682.
- Borkin, S. A. 1978. Data Model Equivalence. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 526–534.
- Bretherton, F. P. and Singley, P. T. 1994. Metadata: A User's View. In *Proc. Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*. 166–174.
- Brin, S. and Page, L. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. Intl. World Wide Web Conf. (WWW)*. Computer Networks.
- Brown, P. G. and Haas, P. J. 2003. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 668–679.
- Buneman, P., Davidson, S. B., and Kosky, A. 1992. Theoretical Aspects of Schema Merging. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Lecture Notes in Computer Science, vol. 580. Springer, 152–167.
- Casanova, M. A. and Vidal, V. M. P. 1983. Towards a Sound View Integration Methodology. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 36–47.
- Castano, S. and Antonellis, V. D. 1999. A Schema Analysis and Reconciliation Tool Environment. In *Proc. Intl. Database Engineering and Applications Symposium (IDEAS)*. 53–62.
- Chawathe, S. S. and García-Molina, H. 1997. Meaningful Change Detection in Structured Data. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 26–37.
- Chen, C., Feng, Y., and Feng, J. 2002. View Merging in the Context of View Selection. In *Proc. Intl. Database Engineering and Applications Symposium (IDEAS)*. 33–43.
- Chirkova, R., Halevy, A. Y., and Suciu, D. 2001. A Formal Perspective on the View Selection Problem. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 59–68.
- Claypool, K. T. 2002. Managing Schema Change in a Heterogeneous Environment. Ph.D. thesis, Worcester Polytechnic Institute.
- Claypool, K. T., Jin, J., and Rundensteiner, E. A. 1998. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Proc. ACM Intl. Conf. on Information and Knowledge Management (CIKM)*. 314–321.
- Claypool, K. T. and Rundensteiner, E. A. 2003. Sangam: A Framework for Modeling Heterogeneous Database Transformations. In *Proc. Intl. Conf. on Enterprise Information Systems (ICEIS)*.

- Clifton, C., Housman, E., and Rosenthal, A. 1997. Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases. In *IFIP 7th Conf. on Database Semantics (DS-7)*.
- Cluet, S., Delobel, C., Siméon, J., and Smaga, K. 1998. Your Mediators Need Data Conversion! In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 177–188.
- Codd, E. F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13, 6, 377–387.
- Cosmadakis, S. S. and Papadimitriou, C. H. 1984. Updates of Relational Views. *Journal of the ACM* 31, 742–760.
- Date, C. J. 1995. *An Introduction to Database Systems*, 6 ed. Addison-Wesley.
- Davidson, S., Buneman, P., and Kosky, A. 1995a. Semantics of Database Transformations. In *LNCS, Springer, Vol. 1358*.
- Davidson, S., Overton, G. C., and Buneman, P. 1995b. Challenges in Integrating Biological Data Sources. *Journal of Computational Biology* 2, 4, 557–572.
- Davies, J. and Woodcock, J. 1996. *Using Z: Specification, Refinement and Proof*. Prentice Hall.
- Dayal, U. and Bernstein, P. A. 1978. On the Updatability of Relational Views. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, S. B. Yao, Ed. IEEE Computer Society, 368–377.
- de Amo, S. and Halfeld Ferrari Alves, M. 2000. Efficient Maintenance of Temporal Data Warehouses. In *Proc. Intl. Database Engineering and Applications Symposium (IDEAS)*. 188–196.
- Decker, S., Jannink, J., Melnik, S., Mitra, P., Staab, S., Studer, R., and Wiederhold, G. 2000a. An Information Food Chain for Advanced Applications on the WWW. In *Proc. European Conf. on Digital Libraries (ECDL), LNCS, Springer*. Vol. 1923. 490–493.
- Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M. C. A., Broekstra, J., Erdmann, M., and Horrocks, I. 2000b. The Semantic Web: The Roles of XML and RDF. *IEEE Internet Computing* 4, 5, 63–74.
- den Bussche, J. V., Gucht, D. V., and Vossen, G. 1993. Reflective Programming in the Relational Algebra. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 17–25.
- Do, H. H., Melnik, S., and Rahm, E. 2002. Comparison of Schema Matching Evaluations. In *Proc. GI-Workshop Web and Databases, LNCS 2593, Springer, 2003*.
- Do, H. H. and Rahm, E. 2000. On Metadata Interoperability in Data Warehouses. Tech. Rep. 01, University of Leipzig.
- Do, H. H. and Rahm, E. 2002. COMA – A System for Flexible Combination of Schema Matching Approaches. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 610–621.
- Doan, A., Domingos, P., and Halevy, A. Y. 2001. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.

- Doan, A., Madhavan, J., Domingos, P., and Halevy, A. Y. 2002. Learning to Map between Ontologies on the Semantic Web. In *Proc. Intl. World Wide Web Conf. (WWW)*. 662–673.
- DOM 1998. XML Document Object Model (DOM), W3C Recommendation. <http://www.w3.org/TR/REC-DOM-Level-1/>.
- ElMasri, R. 1980. On the Design, Use, and Integration of Data Models. Ph.D. thesis, Stanford University. STAN-CS-80-801.
- Embley, D. W., Jackman, D., and Xu, L. 2001. Multifaceted Exploitation of Metadata for Attribute Match Discovery in Information Integration. In *Workshop on Information Integration on the Web (WIIW)*. 110–117.
- Fagin, R., Kolaitis, P. G., Miller, R. J., and Popa, L. 2003. Data Exchange: Semantics and Query Answering. In *Proc. Intl. Conf. on Database Theory (ICDT)*. 207–224.
- Fan, W., Benedikt, M., Chan, C.-Y., Freire, J., and Rastogi, R. 2003. Capturing both Types and Constraints in Data Integration. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Fernandez, M. F., Florescu, D., Kang, J., Levy, A. Y., and Suciu, D. 1997. STRUDEL: A Web-site Management System. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 549–552.
- Fernandez, M. F., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W. C. 2002. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems (TODS)* 27, 4, 438–493.
- Goldstein, J. and Larson, P.-Å. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Goldstone, R. L. and Rogosky, B. J. 2002. Using Relations with Conceptual Systems to Translate Across Conceptual Systems. *Cognition* 84, 295–320.
- Gotthard, W., Lockemann, P. C., and Neufeld, A. 1992. System Guided View Integration for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 4, 1, 1–22.
- Günther, O., Müller, R., Schmidt, P., Bhargava, H. K., and Krishnan, R. 1997. MMM: A Web-Based System for Sharing Statistical Computing Modules. *IEEE Internet Computing* 1, 3, 59–68.
- Gupta, A., Mumick, I. S., and Ross, K. A. 1995. Adapting Materialized Views after Redefinitions. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 211–222.
- Gupta, A. K., Suciu, D., and Halevy, A. Y. 2003. The View Selection Problem for XML Content Based Routing. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 68–77.
- Gusfield, D. and Irving, R. 1989. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Cambridge, MA.
- Halevy, A. Y. 2001. Answering Queries Using Views: A Survey. *VLDB Journal* 10, 4, 270–294.

- Halevy, A. Y., Ives, Z. G., Suciu, D., and Tatarinov, I. 2003. Schema Mediation in Peer Data Management Systems. In *Proc. Intl. Conf. on Data Engineering (ICDE)*.
- He, B. and Chang, K. C.-C. 2003. Statistical Schema Matching across Web Query Interfaces. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 217–228.
- Hegner, S. J. 1994. Unique Complements and Decomposition of Database Schemata. *Journal on Computer Systems Science* 48, 9–57.
- Hull, R. 1986. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal on Computing* 15, 3, 856–886.
- Hull, R. 1997. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 51–61.
- Hull, R., Benedikt, M., Christophides, V., and Su, J. 2003. E-Services: a Look Behind the Curtain. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 1–14.
- ISO 2002. Information technology – Z formal specification notation – Syntax, type system and semantics, ISO/IEC 13568.
- Jannink, J., Mitra, P., Neuhold, E., Pichai, S., Studer, R., and Wiederhold, G. 1999. An Algebra for Semantic Interoperation of Semistructured Data. In *IEEE Knowledge and Data Engineering Exchange Workshop (KDEX), Chicago*. 77–84.
- Jeh, G. and Widom, J. 2002. SimRank: A Measure of Structural-Context Similarity. In *Proc. SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*.
- Kalinichenko, L. A. 1990. Methods and Tools for Equivalent Data Model Mapping Construction. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*. 92–119.
- Kanehisa, M. 2000. *Post-Genome Informatics*. Oxford University Press.
- Kang, J. and Naughton, J. F. 2003. On Schema Matching with Opaque Column Names and Data Values. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 205–216.
- Keller, A. M. and Ullman, J. D. 1984. On Complementary and Independent Mappings on Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 143–148.
- Kementsietsidis, A., Arenas, M., and Miller, R. J. 2003. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Kook, H. J. and Novak, G. S. 1991. Representation of Models for Expert Problem Solving in Physics. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 3, 1, 48–54.
- Lakshmanan, L. V. S., Sadri, F., and Subramanian, G. S. N. 2001. SchemaSQL — An Extension to SQL for Multidatabase Interoperability. In *ACM Transactions on Database Systems (TODS)*. 476–519.
- Lassila, O. and Swick, R. 1998. Resource Description Framework (RDF) Model and Syntax Specification. <http://www.w3.org/TR/REC-rdf-syntax/>.
- Laurent, D., Lechtenbörger, J., Spyratos, N., and Vossen, G. 2001. Monotonic Complements for Independent Data Warehouses. *VLDB Journal* 10, 4, 295–315.

- Lechtenbörger, J. and Vossen, G. 2003. On the Computation of Relational View Complements. *ACM Transactions on Database Systems (TODS)* 28, 2, 175–208.
- Lee, A. J., Nica, A., and Rundensteiner, E. A. 2002. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 14, 5, 931–954.
- Lenzerini, M. 2002. Data Integration: A Theoretical Perspective. In *Proc. ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. 233–246.
- Lerner, B. S. 2000. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Transactions on Database Systems (TODS)* 25, 1, 83–127.
- Li, C., Bawa, M., and Ullman, J. D. 2001. Minimizing View Sets without Losing Query-Answering Power. In *Proc. Intl. Conf. on Database Theory (ICDT)*. 99–103.
- Li, C., Bohannon, P., Korth, H., and Narayan, P. 2003. Composing XSL Transformations with XML Publishing Views. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Li, W.-S. and Clifton, C. 2000. SemInt: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases using Neural Networks. *Data & Knowledge Engineering* 33, 49–84.
- Lovász, L. and Plummer, M. 1986. *Matching Theory*. North-Holland, Amsterdam.
- Mac Lane, S. 1998. *Categories for a Working Mathematician*, 2 ed. Springer.
- Madhavan, J., Bernstein, P. A., Domingos, P., and Halevy, A. Y. 2002. Representing and Reasoning about Mappings between Domain Models. In *AAAI/IAAI 2002*. 80–86.
- Madhavan, J., Bernstein, P. A., and Rahm, E. 2001. Generic Schema Matching with Cupid. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 49–58.
- Madhavan, J. and Halevy, A. 2003. Composing Mappings Among Data Sources. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*.
- Maibaum, T. S. 1977. Mathematical Semantics and a Model for Data Bases. In *Proc. IFIP Congress, Toronto, Canada*. North Holland, 133–138.
- Marco, D. 2000. *Building and Managing the Meta Data Repository: A Full Lifecycle Guide*. Wiley.
- Markowitz, V. M. and Shoshani, A. 1992. Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach. *ACM Transactions on Database Systems (TODS)* 17, 3, 423–464.
- McCarthy, J. L. 1982. Metadata Management for Large Statistical Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 234–243.
- McGee, W. 1959. Generalization — Key to Successful Electronic Data Processing. *Journal of the ACM* 6, 1 (Jan.), 1–23.
- Mecca, G., Atzeni, P., Masci, A., Merialdo, P., and Sindoni, G. 1998. The Araneus Web-Base Management System. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 544–546.
- Melnik, S. 2000. Declarative Mediation in Distributed Systems. In *Proc. Intl. Conf. on Conceptual Modeling (ER), Salt Lake City*. 66–79.

- Melnik, S. and Decker, S. 2000. A Layered Approach to Information Modeling and Interoperability on the Web. In *Proc. ECDL'00 Workshop on the Semantic Web, Lisbon, Portugal*.
- Melnik, S., García-Molina, H., and Paepcke, A. 2000. A Mediation Infrastructure for Digital Library Services. In *Proc. of the 5th ACM Intl. Conf. on Digital Libraries*. 123–132.
- Melnik, S., García-Molina, H., and Rahm, E. 2002. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proc. Intl. Conf. on Data Engineering (ICDE)*. 117–128.
- Melnik, S., Rahm, E., and Bernstein, P. A. 2003a. Developing Metadata-Intensive Applications with Rondo. *Intl. Journal on Web Semantics* 1.
- Melnik, S., Rahm, E., and Bernstein, P. A. 2003b. Rondo: A Programming Platform for Generic Model Management. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Miller, R. J., Haas, L. M., and Hernández, M. A. 2000. Schema Mapping as Query Discovery. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 77–88.
- Miller, R. J., Hernández, M. A., Haas, L. M., Yan, L.-L., Ho, C. T. H., Fagin, R., and Popa, L. 2001. The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30, 1, 78–83.
- Miller, R. J., Ioannidis, Y. E., and Ramakrishnan, R. 1994. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems* 19, 1, 3–31.
- Milo, T. and Zohar, S. 1998. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, Eds. Morgan Kaufmann, 122–133.
- Mitra, P., Wiederhold, G., and Jannink, J. 1999. Semi-automatic Integration of Knowledge Sources. In *Intl. Conf. on Information Fusion*.
- Mitra, P., Wiederhold, G., and Kersten, M. L. 2000. A Graph-Oriented Model for Articulation of Ontology Interdependencies. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, Eds. Lecture Notes in Computer Science, vol. 1777. Springer, 86–100.
- Motro, A. 1987. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering* 13, 7 (July), 785–798.
- Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms*. Cambridge University Press.
- Mumick, I. S., Quass, D., and Mumick, B. S. 1997. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 100–111.
- Nestorov, S., Abiteboul, S., and Motwani, R. 1998. Extracting Schema from Semistructured Data. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.
- Noy, N. F. and Musen, M. A. 2000. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proc. AAAI/IAAI*.
- Noy, N. F. and Musen, M. A. 2002. PromptDiff: A Fixed-Point Algorithm for Comparing Ontology Versions. In *Proc. AAAI/IAAI*.

- Ogata, H., Fujibuchi, W., Goto, S., and Kanehisa, M. 2000. A Heuristic Graph Comparison Algorithm and its Application to Detect Functionally Related Enzyme Clusters. *Nucleic Acids Research*, 4021–4028.
- OMG 2002a. Meta-Object Facility (MOF) Specification, Version 1.4, Object Management Group. <http://www.omg.org/cgi-bin/doc?formal/2002-04-03>.
- OMG 2002b. Unified Modeling Language (UML) Specification, Version 1.5, Object Management Group. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>.
- Palopoli, L., Saccà, D., Terracina, G., and Ursino, D. 2003. Uniform Techniques for Deriving Similarities of Objects and Subschemes in Heterogeneous Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 15, 2, 271–294.
- Paolini, P. and Pelagatti, G. 1977. Formal Definition of Mappings in a Data Base. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 40–46.
- Papakonstantinou, Y., García-Molina, H., and Widom, J. 1995. Object Exchange Across Heterogeneous Information Sources. In *Proc. Intl. Conf. on Data Engineering (ICDE)*. Taipei, Taiwan, 251–260.
- Papakonstantinou, Y. and Vassalos, V. 1999. Query Rewriting for Semistructured Data. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 455–466.
- Parsons, J. and Wand, Y. 2000. Emancipating Instances from the Tyranny of Classes in Information Modeling. *ACM Transactions on Database Systems (TODS)* 25, 2, 228–268.
- Peckham, J., MacKellar, B., and Doherty, M. 1995. Data Model for Extensible Support of Explicit Relationships in Design Databases. *VLDB Journal* 4, 2, 157–191.
- Peters, R. J. and Özsu, M. T. 1997. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions on Database Systems (TODS)* 22, 1, 75–114.
- Popa, L., Velegarakis, Y., Miller, R. J., Hernández, M. A., and Fagin, R. 2002. Translating Web Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*.
- Pottinger, R. and Bernstein, P. A. 2003. Merging Models Based on Given Correspondences. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*.
- Powers, S. 2003. *Practical RDF*, 1 ed. O'Reilly & Associates.
- Rahm, E. and Bernstein, P. A. 2001. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal* 10, 4, 334–350.
- Roddick, J. F. 1992. Schema Evolution in Database Systems - An Annotated Bibliography. *SIGMOD Record* 21, 4, 35–40.
- Roddick, J. F., Al-Jadir, L., Bertossi, L. E., Dumas, M., Estrella, F., Gregersen, H., Hornsby, K., Lufter, J., Mandreoli, F., Mannistö, T., Mayol, E., and Wedemeijer, L. 2000. Evolution and Change in Data Management - Issues and Directions. *SIGMOD Record* 29, 1, 21–25.
- Rosenthal, A. and Reiner, D. S. 1994. Tools and Transformations – Rigorous and Otherwise – for Practical Database Design. *ACM Transactions on Database Systems (TODS)* 19, 2, 167–211.
- Shanmugasundaram, J., Kiernan, J., Shekita, E. J., Fan, C., and Funderburk, J. 2001a. Querying XML Views of Relational Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 261–270.

- Shanmugasundaram, J., Shekita, E. J., Barr, R., Carey, M. J., Lindsay, B. G., Pirahesh, H., and Reinwald, B. 2001b. Efficiently Publishing Relational Data as XML Documents. *VLDB Journal* 10, 2-3, 133–154.
- Shoshani, A., Olken, F., and Wong, H. K. T. 1984. Characteristics of Scientific Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*. 147–160.
- Shu, N. C., Housel, B. C., Taylor, R. W., Ghosh, S. P., and Lum, V. Y. 1977. EXPRESS: A Data EXtraction, Processing, and REStructuring System. *ACM Transactions on Database Systems (TODS)* 2, 2, 134–174.
- Spaccapietra, S. and Parent, C. 1994. View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 6, 2, 258–274.
- Stonebraker, M. 2003. Profiles of Distinguished Database Researchers: An Interview by Marianne Winslett. *SIGMOD Record* 32, 2, 60–67.
- Stonebraker, M., Anderson, E., Hanson, E. N., and Rubenstein, W. B. 1984. Quel as a Data Type. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*. 208–214.
- Subrahmanian, V. S. 1994. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems (TODS)* 19, 2, 291–331.
- Tannenbaum, A. 2001. *Metadata Solutions: Using Metamodels, Repositories, XML, and Enterprise Portals to Generate Information on Demand*. Addison-Wesley.
- Theodoratos, D., Ligoudistianos, S., and Sellis, T. K. 2001. View Selection for Designing the Global Data Warehouse. *Data and Knowledge Engineering (DKE)* 39, 3, 219–240.
- Ullman, J. D. 1997. Information Integration Using Logical Views. In *Proc. Intl. Conf. on Database Theory (ICDT)*, F. N. Afrati and P. G. Kolaitis, Eds. Lecture Notes in Computer Science, vol. 1186. Springer, 19–40.
- Velegrakis, Y., Miller, R. J., and Popa, L. 2003. Mapping Adaptation under Evolving Schemas. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*.
- Wiederhold, G. 1977. *Database Design*. McGraw-Hill Book Company, New York, NY. 2nd Ed., January 1983, 768 pages, republished in the ACM SIGMOD Anthology DVD, available online at <http://www-db.stanford.edu/pub/gio/dbd/acm/toc.html>.
- Wiederhold, G. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer* 25, 38–49.
- Wiederhold, G. 1994. An Algebra for Ontology Composition. In *Proc. of Monterey Workshop on Formal Methods, U.S. Naval Postgraduate School, Monterey CA*. 56–61.
- Wiederhold, G. 2003. The Product Flow Model. In *Proc. 15th Conf. on Software Engineering and Knowledge Engineering (SEKE)*. 183–186. Keynote.
- Yan, L.-L., Miller, R. J., Haas, L. M., and Fagin, R. 2001. Data-Driven Understanding and Refinement of Schema Mappings. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*.