

Imperial College London
Imperial College of Science, Technology and Medicine
Department of Computing

Interactive Database Integration Tool

by

Tudor-Alexandru Dobrila

Supervisor: Dr. Peter McBrien

Second marker: Prof. Sophia Drossopoulou

Submitted in part fulfilment of the requirements for the degree of
MSc. in Computing Science / Software Engineering of Imperial College London
September 2011

Abstract

This thesis presents the *Interactive Database Integration Tool*, a software application developed for *database integration*, which, together with *view integration*, form *schema integration*. The goal of the tool is to guide the user through the integration process of several SQL schemas. In the end, only one virtual schema is produced that can be queried. The tool is built on top of the AutoMed framework for data integration.

Integration is performed by transforming the structure of the schemas, until they can be merged into a single schema. In the research literature, transformation patterns have been proposed, that are encountered frequently when integrating schemas. The tool aims to implement some of the most common patterns and simplify the integration process as much as possible.

This paper presents related work that has been done in this area, the features of the application and a description of its architecture, something that is of interest when extending the implementation.

Consideration has also been given to discovering new transformation patterns from the actions that the user performs over schemas. A new method for pattern discovery in the context of data integration is presented in this thesis.

Acknowledgements

I would like thank the following people:

My supervisor, Dr. Peter McBrien, for his guidance and patience, for the numerous meetings and good ideas that have lead to the successful completion of the project.

Prof. Sophia Drossopoulou, for agreeing to be the second marker of my project and for teaching me the Advanced Issues in Object Oriented Programming course.

My parents, for the tremendous moral and financial support they have given me throughout the whole period of my studies.

My girlfriend, Cecilia, and my friends, for being by my side throughout this difficult year and encouraging me, even when I was feeling down.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	3
1.3	Contributions	3
1.4	Report Structure	4
2	Background	5
2.1	Model Management	5
2.2	Data Integration	9
2.2.1	Data Integration in Theory	9
2.2.2	Data Integration in Practice	13
2.3	The SQL Metamodel	13
2.4	Intermediate Query Language (IQL)	15
2.5	Data Integration Systems	18
2.5.1	Clio	18
2.5.2	Tukwila	19
2.5.3	AutoMed	20
2.5.4	DB-Main	21
3	The Interactive Database Integration Tool	22
3.1	The Main Window	23
3.2	SQL Schema Diagrams	23
3.3	Integration Projects	24
3.3.1	Pre-integration	24
3.3.2	Schema Transformation	25
3.4	Adding a Schema to the Repository	26
3.5	Querying a Schema Using IQL	27
3.6	Summary	27
4	Architecture of the Application	28
4.1	Overview of the Architecture	28
4.2	The Transformation Patterns Framework	29
4.3	The Model of the Application	30
4.4	The View of the Application	34
4.5	Adding a New Transformation Pattern	37
4.6	Unit Testing	40
4.7	Implementation Statistics	41
4.8	Summary	41
5	Transformation Patterns	42
5.1	Schema Conforming	42
5.1.1	Table Normalisation Equivalence	44
5.1.2	Mandatory Column and Total Generalisation Equivalence	47
5.1.3	Optional Column/Child Table Equivalence	49
5.1.4	Column Generalisation Equivalence	50
5.1.5	Column/Table Equivalence	51
5.1.6	Introduction of Total Generalisation Equivalence	52
5.2	Schema Merging	54
5.2.1	Addition of Subset	54
5.2.2	Addition of Union	55
5.2.3	Addition of Intersection	55
5.2.4	Addition of Foreign Key	56
5.2.5	Addition of Many-To-Many Table	58
5.3	Schema Improvement	59

5.3.1	Redundant Column Removal	59
5.3.2	Optional Column To Child Table	60
5.3.3	Column Generalisation	60
5.3.4	Redundant Foreign Key Removal	60
5.4	Summary	62
6	Pattern Discovery in BAV Transactions	63
6.1	Overview of the Method	64
6.2	Graph Isomorphism	68
6.2.1	The Schmidt-Druffel Algorithm	68
6.2.2	The Vento-Foggia Algorithm	70
6.2.3	Towards a Hybrid Graph Isomorphism Component	71
6.3	Graph Hashing	71
6.4	Performance Evaluation	72
6.5	The Method in the Database Integration Tool	73
6.6	Summary	74
7	Conclusions and Future Work	76
7.1	Conclusions	76
7.2	Future Work	77
7.2.1	From Manual to Automatic Schema Matching	77
7.2.2	A Language for Specifying Transformation Patterns	78
7.2.3	From the SQL Metamodel to a Generic Tool	78
	Bibliography	80

List of Figures

1.1	View and database integration, as described in [MIR93].	2
2.1	Sample data for the relational schema S_1	6
2.2	The <i>Mp3 Shop</i> schema, S_2 , expressed in the Entity Relationship metamodel	7
2.3	Mediated schema, S_g , expressed in the binary Entity-Relationship metamodel.	10
2.4	Architecture of the CLIO system	18
2.5	Architecture of the Tukwila system	19
2.6	Architecture of the AutoMed Software	20
2.7	The DB-Main integration toolkit. Attributes belonging to two different entities, but having the same semantics, can be merged.	21
3.1	The <i>Interactive Database Integration Tool</i>	22
3.2	The SQL Schema Diagram viewer.	24
3.3	Different types of integration strategy trees	25
4.1	Architecture of the Application. Diagram generated using Structure101 [Str].	28
4.2	Steps in the execution of a transformation pattern.	30
4.3	The <i>AbstractPattern</i> class diagram.	31
4.4	Save diagram strategies.	35
5.1	Schema used throughout this chapter. Diagram generated using the tool.	42
5.2	User interface for the normalisation pattern.	46
5.3	Schema in 3NF after the application of the normalisation pattern.	46
5.4	Mandatory Column and Total Generalisation Equivalence.	47
5.5	User interface for the mandatory column and total generalisation pattern.	48
5.6	Optional Column/Child Table Equivalence.	49
5.7	User interface for the optional column/child table transformation pattern.	50
5.8	Column Generalisation Equivalence.	50
5.9	User interface for the column generalisation pattern.	51
5.10	Column/Table Equivalence.	52

5.11	User interface for the column/table equivalence	53
5.12	Table Generalisation Equivalence.	53
5.13	User interface for the table generalisation equivalence	54
5.14	Addition of Subset Transformation Pattern	54
5.15	User interface for the addition of subset transformation.	55
5.16	Addition of Intersection Transformation Pattern	56
5.17	User interface for the addition of intersection transformation.	57
5.18	Addition of Foreign Key Pattern	57
5.19	User interface for the addition of foreign key transformation.	58
5.20	Addition of Many-To-Many Table Pattern	58
5.21	User interface for the addition of many-to-many table transformation.	59
5.22	Redundant column removal transformation	59
5.23	User interface for the redundant column removal transformation	60
5.24	Redundant Foreign Key Removal (a)	61
5.25	Redundant Foreign Key Removal (b)	61
5.26	User interface for the redundant foreign key removal transformation.	62
6.1	Dependency graph of the transaction in Example 6.1.	65
6.2	Grouped dependency graph of the transaction in Example 6.1.	65
6.3	Collapsed dependency graph of the transaction in Example 6.1.	66
6.4	Evaluation of the SD algorithm.	69
6.5	Evaluation of the VF2 algorithm.	70
6.6	Evaluation of the graph hash algorithm.	72
6.7	Evaluation of the graph hash algorithm.	72
6.8	Performance evaluation without graph hashing.	73
6.9	Performance evaluation with graph hashing.	73
6.10	Identical transactions returned by the tool.	74
6.11	Transaction viewer window for the transaction in Example 6.2.	75

Chapter 1

Introduction

1.1 Motivation

The volume of information that companies hold has increased significantly in the last decades. According to [Haa07], about 79% of the companies have more than two data stores, while 25% have more than fifteen, making the process of managing and accessing this information cumbersome.

Researchers have studied methods of handling such large amounts of data and several industrial tools have been released, such as CLIO [HHH⁺05], Altova MissionKit [Alt] and Microsoft InfoPath [Inf] (see Section 2.1). They assist database administrators and developers in various scenarios, such as when moving data between different applications. The structure of the data stored in these data stores can be described by **schemas** and **views**.

Definition 1.1 Schema

A *schema* defines the set of possible instances, *i.e.* database states [BM07]. It describes a set of elements, also called **schema objects** (or simply **objects**), connected by some structure [RB01]. The actual values stored in a database that conform to a schema are called **instances** of the schema. □

Definition 1.2 View

A *view* is a subset of a database, from the application perspective. □

Of particular interest in database theory is the task of combining several existing or proposed schemas into a global, unified schema, a process called **schema integration** [BLN86], which can be of two types, depending on the context in which it occurs:

- **View integration** (or *logical database design*) is a process that produces a global conceptual schema from a set of user views. In the end, the users' and the application's requirements are satisfied in the best possible manner.
- **Database integration** (or *global schema design*) is a process that produces a global virtual view from several (possibly heterogeneous) schemas, as shown in Figure 1.1. An example of data integration is shown in Example 1.1. The rest of this paper deals only with this type of integration.

Example 1.1. Consider two web sites selling audio CDs, one called *Music Store* and the other *Mp3 Shop*. The schema of *Music Store* includes the table *audio*, shown in Table 2.1, which has attributes for the identifier of the CD, identifier of the author, title of the CD, identifier of the

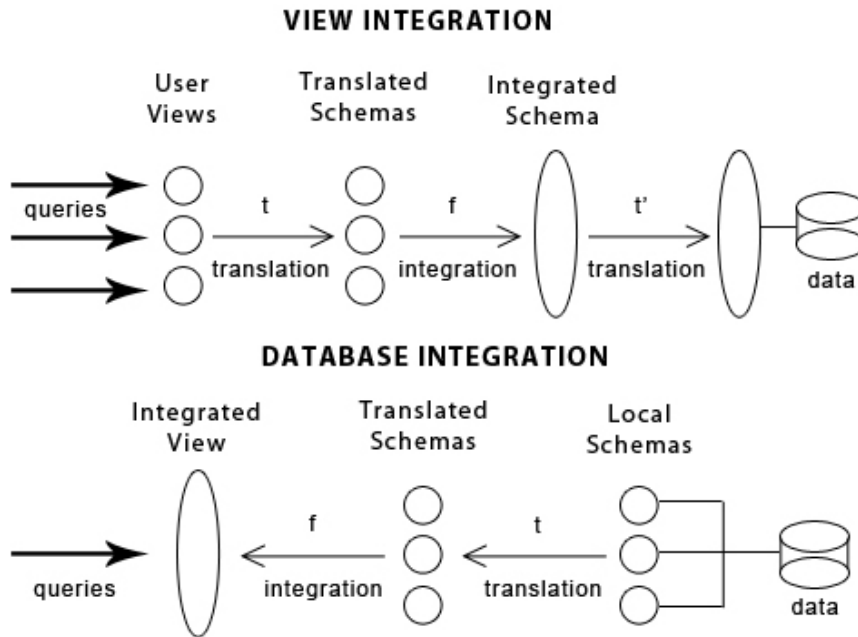


Figure 1.1: View and database integration, as described in [MIR93].

format, release date and price. The schema for *Mp3 Shop* includes the table *cds*, shown in Table 2.1, and contains the identifier of the CD, the identifier of the artist, the name of the album, the price, the average mark given by the persons that purchased the CD and the identifier of the label that released the album.

If we wanted to allow the user browse CDs from both stores, we would have to provide him with a virtual view over the two schemas, which can be done using data integration techniques presented in this paper.

In this case we would have to rename table *cds* to *audio* and columns *cdid* and *albumname* from the *cds* table to *id* and *title*, respectively. By merging the two schemas, a virtual view over the two schemas is produced that can be queried to retrieve audio items from both stores.

audio
id
authorid
title
formatid
releasedate
price

Table 1.1: Table *audio* in the first schema

cds
cdid
albumname
price
avgmark
labelid

Table 1.2: Table *cds* in the second schema

□

While the goal of data integration seems simple, it hides many challenges. One of them is that there are many different formats that the data can be expressed in, such as SQL, XML or object-oriented, which leads to *heterogeneity*. Some models contain constructs that cannot be expressed in other models. For instance, some variations of the Entity-Relationship (ER) metamodel model generalisation of entities, a feature that is not present in the SQL metamodel. Other challenges include choosing the best data sources to use, optimising queries or execution plans and dealing

with uncertainty.

Although individual successes in building tools for data integration have been reported, some of which are presented in Chapter 2, there is still no unified understanding of this process. There have been attempts to formalise schema integration, some of which are explained in Sections 2.1 and 2.2. This thesis aims to bridge the gap between theory and practice, by describing the implementation of a database integration tool.

1.2 Objectives

The main objective of this project is the implementation of a tool used to interactively guide the user through the integration process of several schemas expressed in the SQL metamodel. Throughout this paper, we will call this tool the *Interactive Database Integration Tool*. The tool should be built on top of the AutoMed [BKL⁺04, SRM], a framework for performing database integration using Both As View (BAV) rules and the Hypergraph Data Model (HDM) as the Common Data Model (CDM).

The integration of several schemas should respect the structure presented in [BLN86] and should be done by applying well-known transformation patterns, such as the ones listed in [MP97, MP98]. In the end, a global virtual schema is obtained, which can be queried using the Intermediate Query Language (IQL).

The architecture of the system should be flexible, allowing the introduction of new transformation patterns without having to modify the existing code, and modular, allowing the model (*i.e.* data and behaviour) of the application to be reused independent of the user interface.

In addition, we aim to propose and implement a component to support the identification of patterns in sequences of BAV transformations, which in this thesis we call BAV transactions. This is done by analysing the history of the transactions and looking for identical transactions.

1.3 Contributions

The outcome of this project is a tool written in Java and used to interactively guide the user through the integration of several schemas expressed in the SQL metamodel. The following list describes the main contributions of this project:

- A formal definition of the SQL Metamodel, as an instance of the framework presented in [MP98].
- A tool that interactively guides the user through the integration process described in [BLN86], by the application of well-known transformation patterns. A total of 18 such patterns have been implemented in the tool.
- A flexible, modular and extensible architecture, that promotes the reuse of individual components.
- A powerful transformation patterns framework, such that the introduction of a new pattern is done without modifying any of the existing code.
- A component for the dynamic extraction of patterns from BAV transactions using dependency graphs, something that we are not aware to have been attempted before in the context of the Both As View mapping approach.

1.4 Report Structure

This report is structured as follows:

- Chapter 2 briefly describes the work that has been done in the field of schema integration and lists several industrial tools that have been developed for this purpose.
- Chapter 3 describes the main features of the *Interactive Database Integration Tool*.
- Chapter 4 outlines the architecture of the system and the transformation patterns framework.
- Chapter 5 presents the transformation patterns implemented in the application and demonstrates their applicability and their integration in the tool.
- Chapter 6 introduces our method for the dynamic extraction transformation patterns from a history of BAV transactions and the implementation of this component in the tool.
- Chapter 7 summarises the work discussed in this report and offers suggestions of how the work can be continued in the future.

Chapter 2

Background

The notion of **schema integration** was first introduced in the mid 1980s in [BLN86] to describe both view integration and database integration, which were considered disjoint concepts before. Ever since, researchers have tried to formalise integration and describe generic solutions for it. This chapter presents some of the research done in this area in the last decades.

The structure of this chapter is as follows. Section 2.1 describes model management, a generic approach for the data programmability problem, Section 2.2 presents the data integration problem from two perspectives, one theoretical and another one practical. Section 2.3 introduces a formal description of the SQL metamodel. Section 2.4 focusses on the Intermediate Query Language. Section 2.5 lists some of the tools available for schema integration.

2.1 Model Management

One of the oldest database research topics is how to allow users to access large databases, a problem that is called the *data programmability problem*. Model management [BM07] is a generic approach to solving this issue. It represents a set of abstract operations that are applied to **Match** schemas, **Merge** schemas, **Difference** schemas, **Compose** schemas, **Translate** schemas into other data models and **Generate** data transformations from mappings.

Definition 2.1 Model management system

A *model management system* is a component that supports the creation, compilation, reuse, evolution and execution of mappings between schemas represented in a wide range of metamodels [BM07]. □

Definition 2.2 Mapping

A *mapping* represents a relationship between the instances of two schemas [BM07]. Formally, if D_1 and D_2 are the sets of possible instances of schemas S_1 and S_2 , a mapping between S_1 and S_2 is a subset of $D_1 \times D_2$, with \times representing the Cartesian product. A mapping is expressed in a mapping language as a set of mapping constraints, which define the subset of $D_1 \times D_2$. Intermediate Query Language (IQL) [JPZ03], described in Section 2.4, is the choice of query language used throughout this paper to specify mappings. Examples of mappings expressed in IQL can be found in Section 2.2. □

Definition 2.3 Metamodel

A *metamodel* is a language for expressing schemas [BM07]. Examples of metamodels include

relational, SQL, XML Schema (XSD), Entity-Relationship, UML, object-oriented (OO), Service Modelling Language (SML) and Web Ontology Language (OWL). Examples of schemas expressed in two different metamodels are shown in Example 2.1 and Example 2.2. \square

Example 2.1. Consider schema S_1 for the *Music Store* web site described in Example 1.1, expressed in the relational metamodel and containing the *audio* table. The relational metamodel contains the following constructs: relations (*i.e.* sets of tuples), attributes, primary keys and foreign keys. Figure 2.1 shows an instance of the schema.

- The *audio* relation contains the mandatory attributes *id*, *authorid*, *formatid*, the optional attribute *releasedate* and a primary key constraint with only one attribute, *id*:
audio(id, authorid, title, formatid, releasedate?, price)
- The schema contains two foreign key constraints from *audio* to the *author* and *format* relations:
audio.authorid \rightarrow author.id
audio.formatid \rightarrow format.id
- The *author* relation contains the mandatory attributes *id*, *firstname*, *lastname* and a primary key constraint with only one attribute, *id*:
author(id, firstname, lastname)
- The *format* relation contains the mandatory attributes *id*, *name*, the optional attribute *bitrate* and a primary key constraint with only one attribute, *id*:
format(id, name, bitrate?)

audio					
<u>id</u>	authorid	title	formatid	releasedate	price
1	1	Beethoven: Complete Symphonies	1	5 Mar 2007	19.99
2	1	The Very Best of Beethoven	2	3 Oct 2005	7.50
3	2	Mozart: Great Piano Concertos	1	NULL	7.00
4	2	Mozart: Complete Violin Concertos	3	10 May 1993	8.00
5	2	Very Best of Mozart	2	6 Feb 2006	7.83
6	3	Bach - Clavierbung Books 1	1	NULL	15.00
7	3	Essential Bach	1	13 Mar 2000	6.97

author			format		
<u>id</u>	firstname	lastname	<u>id</u>	name	bitrate
1	Ludwig	van Beethoven	1	mp3-320	320kbps
2	Wolfgang Amadeus	Mozart	2	wav	NULL
3	Johann Sebastian	Bach	3	flac	NULL

Figure 2.1: Sample data for the relational schema S_1

\square

Example 2.2. Consider schema S_2 in Figure 2.2, for the *Mp3 Shop* web site described in Example 1.1, expressed in the binary Entity-Relationship (ER) metamodel, as described in [MP97], and containing the *cds* table.

\square

The need for model management is motivated by the the fact that sources may be heterogeneous or can have logical schemas that are different from the physical schemas. Both of these cases can

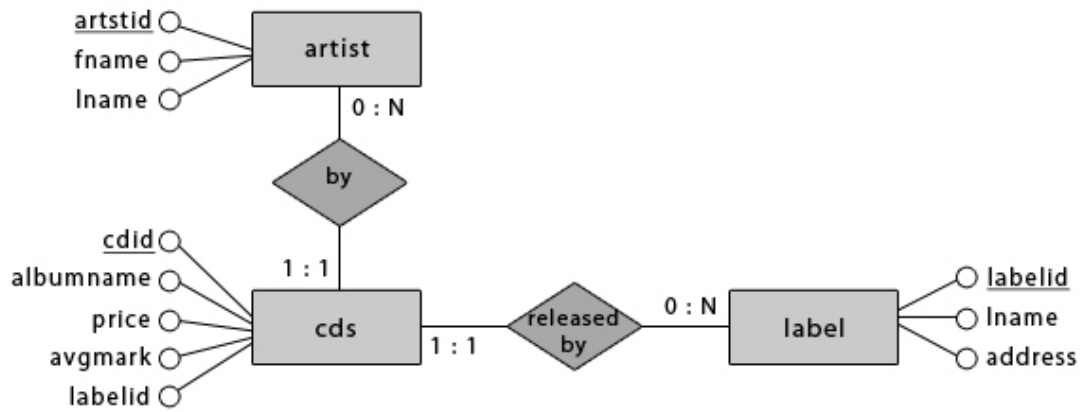


Figure 2.2: The *Mp3 Shop* schema, S_2 , expressed in the Entity Relationship metamodel

be solved by using mappings between the different representations, e.g. map constructs from a metamodel to constructs in another metamodel.

Two types of mappings have been defined in the literature: *engineered mappings* and *approximate mappings* [BM07]. Engineered mappings are precisely specified and tested, while in approximate mappings there is no notion of a correct answer and it is up to the user to analyse the data and make a decision about it, usually on the fly.

Engineered mappings are used in a wide range of scenarios [BM07], which includes but is not limited to:

- *Message mappings*, used to translate between different data formats. Industrial tools that support this functionality are Altova [Alt], Stylus Studio [Sty], Microsoft BizTalk [Biz] and Oracle WebCenter Interaction [Ora].
- *Query mediators*, used in data integration to access heterogeneous databases. Tools for this have been developed in areas such as bioinformatics and medical informatics, an example being the Kleisli system [DBH⁺99]. A more detailed presentation of this scenario is given in Section 2.2.
- *Wrapper generation*, used for instance to produce an object-oriented wrapper for a relational database. This issue has been addressed in tools such as Oracle Toplink [Top] and Hibernate [Hib].
- *Report writers*, which are used to produce reports from structured data sources. Examples of tools used to generate reports include Microsoft SQL Server Reporting Services [Sql] and Crystal Reports [Cry].
- *Data translation*, used to move data between different applications. Microsoft InfoPath [Inf] can be used for this.

Mappings are useful in two situations:

- **Given two schemas, generate a mapping**

The process of creating a mapping between a source schema and a target schema is done in three steps. First the correspondences between the two schemas are identified. Next, they are converted into mapping constraints, which can be optionally translated into transformations.

Definition 2.4 Correspondences

Correspondences are pairs of elements from two schemas that are related [BM07]. □

Example 2.3. The element *author* in Example 2.1 is related to the element *artist* in Example 2.2, so a correspondence exists between them. □

Definition 2.5 Transformation

A *transformation* is a functional mapping constraint, such as a query or view definition [BM07]. □

Finding the correspondences between the source and target schemas is also called *schema matching* and in the model management system is described by the abstract operation *Match*. Several solutions for automatic schema matching have been suggested in the literature [Riz04, MRBM05], most of them making use of the schema definition (e.g. the data types and element names). Researchers have tried to semi-automate or fully automate this process [RB01], by relying for instance on ontologies and lexical databases.

Schema matching presents a great challenge, as there is no unique or universal solution to finding the correspondences between two schemas [Gal07]. This is why most tools guide the user through the matching process and allow him to choose from several different mappings the correct one.

- **Map a schema to another schema**

P. Berstein defined in the model management an abstract operation, *ModelGen*, used to translate a source schema from a metamodel into an equivalent target schema defined in another metamodel, a process that is also called data exchange.

Definition 2.6 Data exchange

Data exchange is the task of restructuring data from a source schema to a target schema [HHH⁺05]. It can be implemented as a set of rules that map every construct from one metamodel to a universal metamodel or from the universal metamodel to the target metamodel. □

Example 2.4. Consider the relational schema defined in Example 2.1. If we wanted to convert this to the XML metamodel, we could use Entity-Relationship (ER) as the universal metamodel and first map from relational to ER and after that from ER to XML. □

While the output of the mapping in data integration or wrapper generation is usually a query or a view, there are other scenarios in which the mapping runtime must take into account several issues, such as update propagation, error checking, debugging, access control, integrity constraints, indexing or notification.

The model management system contains other abstract operations used for schema evolution:

- *Diff* for finding the information present in a schema that is absent in another schema.

- *Merge* for taking two schemas and a mapping between them and finding out where the two schemas overlap.
- *Invert* for reversing a transformation.

The abstract operations listed above can theoretically be used to solve the data programmability problem, although, in practice, we do not know if they are complete and we also lack precise semantics for them [Haa07]. A possible solution to overcome this problem is to think about database management in an even more abstract way and find a method to represent all information needed for this task [Haa07].

2.2 Data Integration

As presented in Section 2.1, engineered mappings can be used to build query mediators to access heterogeneous databases, a process that is called *data integration*. Heterogeneity is given by the fact that data can be expressed in different metamodels and can be defined by different people.

Researchers have looked at data integration from two different perspectives, one theoretical and one practical. Recent work in this area attempts to bridge the gap between the two.

2.2.1 Data Integration in Theory

M. Lenzerini analysed the theory behind data integration. He introduced in [Len02] data integration systems, which are systems that can be used to combine data from different sources. This way, the user is presented a global view over the data and has to interact only with it, rather than having to write queries for each of the sources.

Definition 2.7 Data integration system

Formally, a *data integration system* is defined as $I = \langle G, S, M \rangle$, where G is a global schema (also called a mediated schema) expressed in a language L_G over an alphabet A_G , S is the source schema expressed in a language L_S over an alphabet A_S , M is the mapping between G and S . \square

Example 2.5. One possible global schema, S_g , obtained by integrating the source schemas from Examples 2.1 and 2.2 is shown in Figure 2.3. \square

The source schema defines the real database states (*i.e.* where the actual data is stored), whereas the global schema provides an integrated view over the source schema. The relationship between them is given by mappings, as presented in Definition 2.2. Formally, a mapping is a set of assertions of the form $q_s \rightsquigarrow q_G$ and $q_G \rightsquigarrow q_s$, where q_s and q_G are queries over the source schema and over the global schema, respectively.

If we consider a source database D that conforms to the schema S , we can say that a database B for the integration system I is legal with respect to D if B conforms to schema G and B satisfies the mapping M with respect to D .

Several approaches for defining mappings in data integration systems have been proposed: Local as View (LAV), Global as View (GAV), Both as View (BAV) and Global-Local as View (GLAV). The first three are briefly described in the next sections.

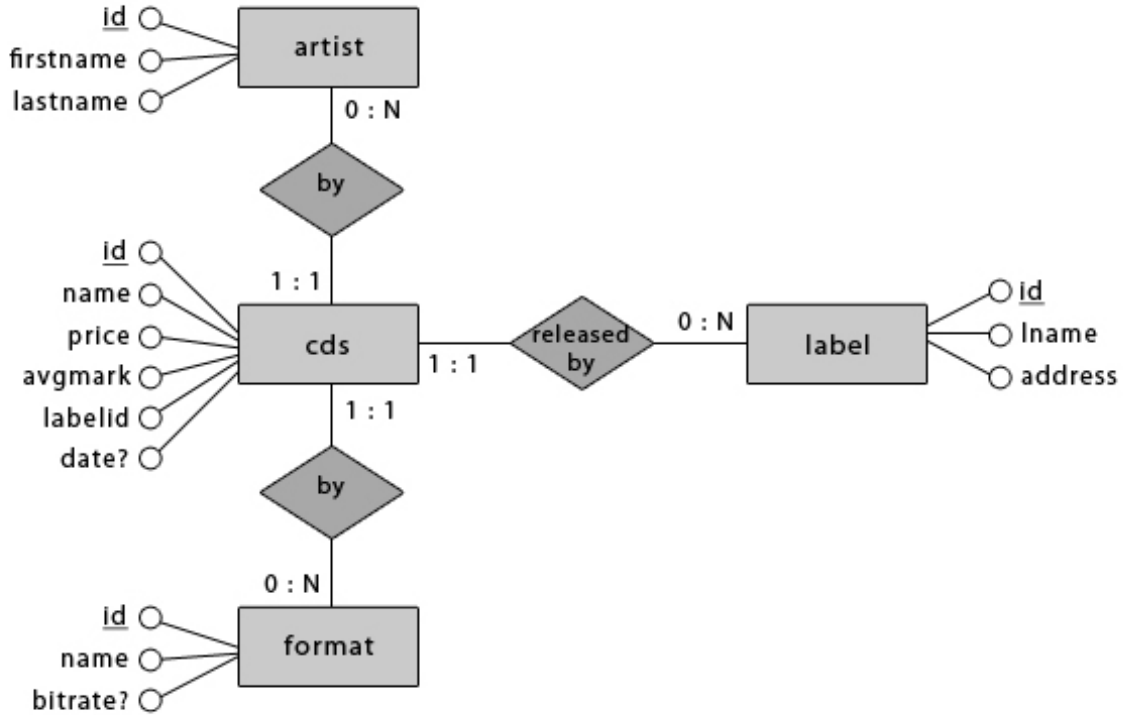


Figure 2.3: Mediated schema, S_g , expressed in the binary Entity-Relationship metamodel.

Local as View

In the Local as View (LAV) approach, the source database S is mapped as a view over the global database G . Formally, the assertions in the mapping M are of the form $s \rightsquigarrow q_G$, where s is an element of S .

Example 2.6. In LAV, the *artist* entity in schema S_2 in Example 2.2 can be expressed by the following mappings, as views over the global schema S_g .

$S_2:\text{entity}:\langle\langle\text{artist}\rangle\rangle \text{ :- } S_g:\text{entity}:\langle\langle\text{artist}\rangle\rangle$
 $S_2:\text{attribute}:\langle\langle\text{artist},\text{id}\rangle\rangle \text{ :- } S_g:\text{attribute}:\langle\langle\text{artist},\text{artistid}\rangle\rangle$
 $S_2:\text{attribute}:\langle\langle\text{artist},\text{fname}\rangle\rangle \text{ :- } S_g:\text{attribute}:\langle\langle\text{artist},\text{firstname}\rangle\rangle$
 $S_2:\text{attribute}:\langle\langle\text{artist},\text{lname}\rangle\rangle \text{ :- } S_g:\text{attribute}:\langle\langle\text{artist},\text{lastname}\rangle\rangle$

□

In order to further characterise an element s of S , researchers have introduced in [Len02] a new notation $as(s)$, used to represent how accurate is the source with respect to the associated view. It can take one of the following three values:

- **Sound views.** The source s is said to be *sound* if, given a source database D and a global database B , any tuple in D that satisfies s also appears in q_G^B , the set of tuples in B that satisfy q . If a tuple that satisfies s does not appear in D , we cannot decide that it does not satisfy the global view. Informally, this implies that the mapping produces a subset of the correct answers.

- **Complete views.** The source s is said to be *complete* if, given a source database D and a global database B , any tuple in D that satisfies s *might* appear in the q_G^B , but this does not necessarily have to happen. If a tuple that satisfies s does not appear in D , we know that it cannot appear in the global view. Informally, this implies that the mapping produces all correct answers and possibly some incorrect ones.
- **Exact views.** The source s is said to be *exact* if, given a source database D and a global database B , all tuples in D that satisfies s and only those tuples appear in the q_G^B . Informally, this implies that the mapping produces all correct answers and no incorrect ones.

The most important advantage of LAV is that adding a new source is done by simply inserting an assertion into the mapping. The drawback is that it works on global schemas that are unlikely to suffer any changes.

Global as View

In the Global as View (GAV) approach, the global schema G is modelled as a set of views over the source schema S . Formally, the assertions in the mapping M are of the form $g \sim q_S$, where g is an element of G . Intuitively, this means that there exists a mediator that defines exactly how elements from the source database will be retrieved. Similarly to LAV, each element g of G can be characterised by $as(g)$, which can be sound, complete or exact.

Example 2.7. In GAV, the *artist* element in schema S_g in Example 2.5 can be expressed by the following mappings, where S_2 is defined in Example 2.2.

```

 $S_g$ :entity:⟨⟨artist⟩⟩ :-  $S_2$ :entity:⟨⟨artist⟩⟩
 $S_g$ :attribute:⟨⟨artist,id⟩⟩ :-  $S_2$ :attribute:⟨⟨artist, artistid⟩⟩
 $S_g$ :attribute:⟨⟨artist,firstname⟩⟩ :-  $S_2$ :attribute:⟨⟨artist, fname⟩⟩
 $S_g$ :attribute:⟨⟨artist,lastname⟩⟩ :-  $S_2$ :attribute:⟨⟨artist, lname⟩⟩

```

□

The advantage of GAV over LAV is that it favours sources that are stable and query processing reduces to a simple unfolding process [Len02].

Both as View

The Both as View (BAV) mapping language was introduced in [MP03] and it subsumes both GAV and LAV. In this approach, there is a bidirectional mapping between the source schema S and global schema G . A mapping operates on a single object (e.g. XML element, SQL column, etc.) and at any time a new schema is defined based on the previous schema by modifying only one construct.

The following primitive transformations have been defined in [MP04] and can be applied to a schema S_{before} , resulting in a different schema S_{after} :

- $add(C:\langle s \rangle, q)$ - S_{after} contains a new construct s of type C . q is a query over S_{before} , specifying the extent of c in terms of the existing constructs of S_{before} .
- $delete(C:\langle s \rangle, q)$ - S_{after} does not contain the construct s of type C . The extent of s may be recovered by executing q on schema S_{after} .

- $rename(C:\langle s \rangle, \langle s' \rangle)$ - the construct s of type C from S_{before} is renamed to s' and has the same type in S_{after} .
- $extend(C:\langle s \rangle, ql, qu)$ - S_{after} contains a new construct s of type C . ql specifies the minimum extent of s and may take the special value $Void$. qu specifies the maximum extent of s and may take the special value Any .
- $contract(C:\langle s \rangle, ql, qu)$ - S_{after} does not contain the construct s of type C . ql and qu have the same meaning as in $extend$.

The BAV transformations add and $delete$ specify exact mappings, while their counterparts $extend$ and $contract$ are used to define non-exact mappings. It can also be noticed that add and $delete$ are special cases of $extend$ and $contract$, when $ql = qu$.

Example 2.8. Consider schema S_1 defined in Example 2.1 and expressed in the relational meta-model. If we want to add a new table $audio_released$, containing only those audio items that are known to have been released, *i.e.* have a non-null $releasedate$, the following BAV transformations would have to be applied to S_1 , resulting in a new schema. The extents of the transformations are specified in the Intermediate Query Language, presented in Section 2.4.

- ① Add the new relation $audio_released$:
`add (relation:⟨audio_released⟩, [{x} | {x,y} ← ⟨audio,releasedate⟩; y <> Null])`
- ② Add the primary key attribute id to the new relation:
`add (attribute:⟨audio_released, id,notnull⟩, [{x, x} | {x} ← ⟨audio_released⟩])`
- ③ Add the mandatory attribute $releasedate$ to the new relation:
`add (attribute:⟨audio_released,releasedate,notnull⟩,
[{x, y} | {x} ← ⟨audio_released⟩; {x,y} ← ⟨audio,released⟩])`
- ④ Create the primary key constraint for the $audio_released$ relationship:
`add (primarykey:⟨audio_released_pk,audio_released,⟨audio_released,id⟩⟩)`
- ⑤ Create a foreign key constraint from the id attribute of $audio_released$ to $audio$:
`add (foreignkey:⟨audio_released_audio_fk,
audio_released,⟨audio_released,id⟩,audio,⟨audio,id⟩⟩)`
- ⑥ Delete the optional attribute $releasedate$ from the original relation:
`delete (attribute:⟨audio,releasedate,null⟩,
⟨audio_released,releasedate⟩ ++ [{x,Null} | {x} ← ⟨audio⟩ - ⟨audio_released⟩])`

□

Sequences of transformations, such as the one defined in Example 2.8, are called **pathways** and are useful in schema evolution. For instance, if there exists a pathway T_1 between a local schema S_L and a global schema S_{G1} and S_{G1} evolves to S_{G2} by some pathway T_2 , then a mapping between S_L and S_{G2} is obtained by concatenating the two individual pathways, *i.e.* $T_1;T_2$.

A popular data integration system that uses BAV transformations is AutoMed, presented in Section 2.5.3. An example of data integration in AutoMed using the BAV primitive transformations described above can be found in [BAV]. The rest of this paper deals only with BAV transformations.

2.2.2 Data Integration in Practice

L. Hass analysed in [Haa07] the challenges that businesses face when performing information integration. She described this as a four step process: understanding the data, standardisation, specification and execution.

Understanding is concerned with discovering the structure of the data, such as keys, constraints or data types. *Standardisation* involves deciding a common representation for the data, for instance choosing a common format for phone numbers. During the *specification* step, the actual mappings used to translate the sources into the target are defined. Finally, in the *execution* phase the integration is performed either by copying the integrated data or by providing a virtual view over it.

The industry still lacks powerful tools for information integration and there is still much room for research in this area.

2.3 The SQL Metamodel

In [MP98], a general formal framework for schema transformation is described, in which schemas are represented as directed, labelled, nested hypergraphs in the **Hypergraph Data Model** (HDM). In Definition 2.8, we remind the reader what a hypergraph is.

Definition 2.8 Hypergraph

A *hypergraph* is a generalisation of a graph, where an edge can connect any number of vertices. \square

A model M expressed in the HDM is a triple $\langle S, I, Ext_{S,I} \rangle$, where S is a schema, I is an instance of S and $Ext_{S,I}$ represents an extension mapping from S to I . Eight primitive transformations operate over models and result either in a model, if the transformation is successful, or in the special value "undefined", denoted ϕ , otherwise: *renameNode*, *renameEdge*, *addConstraint*, *delConstraint*, *addNode*, *delNode*, *addEdge* and *delEdge*.

When performing database integration, schemas may be expressed in different metamodels and have to be translated in a common language, called the **Common Data Model (CDM)**. HDM is the choice of CDM used throughout this paper and has been implemented in AutoMed (see Section 2.5.3).

The SQL metamodel can be represented as an instance of this framework and contains four types of constructs: tables, column, primary keys and foreign keys.

Definition 2.9 Primary Keys

A *primary key* is a subset of the columns of a table that uniquely identifies every row. A primary key constraint is the rule that no two different rows may have the same values stored in the primary key column(s). \square

Definition 2.10 Foreign Keys

A *foreign key* is a set of columns in one table (the source table) whose values must match the values of the primary key column(s) of one row in another table (the target table).

A foreign key constraint is a type of referential constraint and is the rule that the values of the foreign key column(s) in the source table are valid if they appear in the primary key column(s) of the target table (with the exception of *NULL*). \square

The following conventions are used to formally specify the metamodel as an instance of the

framework in [MP98]:

- *Names* represents the names of vertices and edges in the underlying hypergraph, as defined in [MP98]
- *Types* represents the set of distinct data types that are supported
- $Seq(X)$ is the set of finite sequences of elements in the set X
- $Seq_n(X)$ is the set of sequences of n elements in the set X

Definition 2.11 SQL Metamodel

A schema expressed in the *SQL metamodel*, S , is a quintuple $\langle Tables, Cols, Assoc, PK, FK \rangle$, where:

- $Tables \subseteq Names$ is the set of tables.
- $Cols \subseteq Names$ is the set of columns.
- $Assoc \subseteq Tables \times Cols$ for the association of columns to tables.
- $PK \subseteq Names \times Tables \times Seq(Cols)$ is the set of primary key constraints.
- $FK \subseteq Names \times Tables \times Seq_n(Cols) \times Tables \times Seq_n(Cols)$ is the set of foreign key constraints.

□

In Definition 2.11, $Tables \cup Cols$ represents the vertices in the hypergraph, $Assoc$ the edges and $PK \cup FK$ the constraints.

In the SQL metamodel, both primary key and foreign key constraints have a name associated with them. In order to be able to rename a constraint, the framework presented in [MP98] must be augmented with the following primitive transformations:

- $addConstraint\langle Names \times Constraints \rangle$ extends the $addConstraint\langle Constraints \rangle$ transformation to associate a name with a constraint.
- $renameConstraint\langle Names \times Names \rangle$ renames a constraint. It is successful provided that the new name is not the name of an existing constraint.

The following primitive transformations of the SQL metamodel are defined in terms of the primitive transformations of the extended framework:

- $rename_X\langle from, to \rangle$, where $X \in \{T, C\}$, representing tables and columns, respectively:
 $renameNode\langle from, to \rangle$
- $rename_X\langle from, to \rangle$, where $X \in \{PK, FK\}$, representing primary key and foreign key constraints, respectively:
 $renameConstraint\langle from, to \rangle$
- $add_T\langle table, q \rangle$ adds a table $table$ with the extent q :
 $addNode\langle table \rangle$
- $del_T\langle table \rangle$ removes a table $table$:
 $delNode\langle table \rangle$
- $add_C\langle table, col, q \rangle$ adds a column col to the table $table$ with the extent q :
 $addNode\langle col, q \rangle$
 $addEdge\langle table, col \rangle$

- $del_C\langle table, col \rangle$ removes a column col from the table $table$:
 $delEdge\langle table, col \rangle$
 $delNode\langle col \rangle$
- $add_{PK}\langle name, table, col_1, \dots, col_n \rangle$ adds a primary key constraint:
 $addConstraint\langle createPK\langle name, table, col_1, \dots, col_n \rangle \rangle$
- $del_{PK}\langle name, table, col_1, \dots, col_n \rangle$ removes a primary key constraint:
 $delConstraint\langle createPK\langle name, table, col_1, \dots, col_n \rangle \rangle$
- $add_{FK}\langle name, t_1, col_1, \dots, col_n, t_2, col_1, \dots, col_n \rangle$ adds a foreign key constraint:
 $addConstraint\langle createFK\langle name, t_1, col_1, \dots, col_n, t_2, col_1, \dots, col_n \rangle \rangle$
- $del_{FK}\langle name, t_1, col_1, \dots, col_n, t_2, col_1, \dots, col_n \rangle$ removes a foreign key constraint:
 $delConstraint\langle createFK\langle name, t_1, col_1, \dots, col_n, t_2, col_1, \dots, col_n \rangle \rangle$

In the above transformations, two shorthand notations have been used:

- $createPK\langle name, table, col_1, \dots, col_n \rangle$ denotes a primary key constraint on table $table$ containing the columns col_1, \dots, col_n .
- $createFK\langle name, t_1, col_1, \dots, col_n, t_2, col_1, \dots, col_n \rangle$ denotes a foreign key constraint sourced at table t_1 , columns col_1, \dots, col_n , and targeted at table t_2 , columns col_1, \dots, col_n .

The primitive transformations of the SQL metamodel are semantically sound, *i.e.* they always return another model. They are used to map SQL schemas to other SQL schemas and can be generalised into *composite transformations*, *i.e.* sequences of primitive transformations.

Interactive Database Integration Tool uses schemas expressed in the SQL metamodel, as there are many popular Database Management Systems (DBMS) that support SQL: PostgreSQL, MySQL, Oracle, SQL Server, SQLite, etc.

2.4 Intermediate Query Language (IQL)

Intermediate Query Language (IQL) [JPZ03] is a functional language, which has been developed for the AutoMed data integration system, described in Section 2.5.3. It is a common query language used to express queries written in high-level query languages [JPZ03], such as SQL.

IQL includes constants (which can be strings, booleans, integers and real numbers), variables, identifiers, tuples (e.g. $\{1, 2, 3\}$) and lists (e.g. $\{[1, 2, 3]\}$). They are used to define list comprehensions, which take the form:

$$[vars \mid gen_1; gen_2 \dots ; pred_1; pred_2; \dots]$$

where gen_i represent generators, $pred_i$ represent predicates and $vars$ contains variables that appear in at least one of the generators.

Definition 2.12 Generators

Generators "iterate a pattern over a list-valued expression, where a pattern is either a variable or a tuple of patterns" [JPZ03]. □

Definition 2.13 Filters

Filters are boolean-valued expressions that filter the information generated by the generators of the comprehension [JPZ03]. \square

In IQL, schema objects are denoted by $\text{constructName} : \langle\langle \text{definition} \rangle\rangle$. For instance, the IQL definition of the table construct from the SQL metamodel is $\text{table} : \langle\langle \text{tableName} \rangle\rangle$, where tableName is the name of the table. The definition of the column construct takes the form $\text{column} : \langle\langle \text{tableName}, \text{columnName}, \text{optionality}, \text{type} \rangle\rangle$, where tableName is the name of the parent table holding the column, columnName is the name of the column, optionality specifies the optionality of the column and can take one of the values $\{\text{null}, \text{nonnull}\}$ and type mentions the type of the values stored in the column.

In IQL queries, schema objects can be referenced using their shorthand notation. For instance, the short notation for tables is $\langle\langle \text{tableName} \rangle\rangle$ and for columns is $\langle\langle \text{tableName}, \text{columnName} \rangle\rangle$.

Example 2.9. A direct representation of the S_1 schema in Example 2.1 expressed in IQL is:

```
relation:⟨⟨audio⟩⟩ = [(1, 1, 'Beethoven : Complete Symphonies', 1, '5 Mar 2007', 19.99),
                    (2, 1, 'The Very Best of Beethoven', 2, '3 Oct 2005', 7.50),
                    (3, 2, 'Mozart : Great Piano Concertos', 1, NULL, 7.00),
                    (4, 2, 'Mozart : Complete Violin Concertos', 3, '10 May 1993', 8.00),
                    (5, 2, 'Very Best of Mozart', 2, '6 Feb 2006', 7.83),
                    (6, 3, 'Bach – Clavierbung Books1', 1, NULL, 15.00),
                    (7, 3, 'Essential Bach', 1, '13 Mar 2000', 6.97)]
```

```
relation:⟨⟨author⟩⟩ = [(1, 'Ludwig', 'van Beethoven'), (2, 'Wolfgang Amadeus', 'Mozart'),
                    (3, 'Johann Sebastian', 'Bach')]
```

```
relation:⟨⟨format⟩⟩ = [(1, 'mp3 – 320', '320kbps'), (2, 'wav', NULL), (3, 'flac', NULL)]
```

 \square

Example 2.10. An alternative representation of the schema S_1 in Example 2.1 is shown below. This type of representation is used throughout the rest of this paper.

```
relation:⟨⟨audio⟩⟩ = [(1), (2), (3), (4), (5), (6), (7)]
```

```
attribute:⟨⟨audio, id⟩⟩ = [(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7)]
```

```
attribute:⟨⟨audio, authorid⟩⟩ = [(1, 1), (2, 1), (3, 2), (4, 2), (5, 2), (6, 3), (7, 3)]
```

```
attribute:⟨⟨audio, title⟩⟩ = [(1, 'Beethoven : Complete Symphonies',
                    (2, 'The Very Best of Beethoven',
                    (3, 'Mozart : Great Piano Concertos',
                    (4, 'Mozart : Complete Violin Concertos',
                    (5, 'Very Best of Mozart',
                    (6, 'Bach – Clavierbung Books1',
                    (7, 'Essential Bach')]
```

```
attribute:⟨⟨audio, formatid⟩⟩ = [(1, 1), (2, 2), (3, 1), (4, 3), (5, 2), (6, 1), (7, 1)]
```

```
attribute:⟨⟨audio, releasedate⟩⟩ = [(1, '5 Mar 2007', (2, '3 Oct 2005', (3, NULL), (4, '10 May 1993',
                    (5, '6 Feb 2006', (6, NULL), (7, '13 Mar 2000')]
```

```
attribute:⟨⟨audio, price⟩⟩ = [(1, 19.99), (2, 7.50), (3, 7.00), (4, 8.00), (5, 7.83), (6, 15.00), (7, 6.97)]
```

```
relation:⟨⟨author⟩⟩ = [(1), (2), (3)]
```

```
attribute:⟨⟨author, id⟩⟩ = [(1, 1), (2, 2), (3, 3)]
```

attribute:⟨⟨author, firstname⟩⟩ = [⟨1, 'Ludwig'⟩, ⟨2, 'WolfgangAmadeus'⟩, ⟨3, 'JohannSebastian'⟩]
 attribute:⟨⟨author, lastname⟩⟩ = [⟨1, 'vanBeethoven'⟩, ⟨2, 'Mozart'⟩, ⟨3, 'Bach'⟩]
 relation:⟨⟨format⟩⟩ = [⟨1⟩, ⟨2⟩, ⟨3⟩]
 attribute:⟨⟨format, id⟩⟩ = [⟨1, 1⟩, ⟨2, 2⟩, ⟨3, 3⟩]
 attribute:⟨⟨format, name⟩⟩ = [⟨1, 'mp3 - 320'⟩, ⟨2, 'wav'⟩, ⟨3, 'flac'⟩]
 attribute:⟨⟨format, bitrate⟩⟩ = [⟨1, '320kbps'⟩, ⟨2, NULL⟩, ⟨3, NULL⟩]

□

A join can be defined in IQL by combining several generators with filters or by using the same variable name in different generators, as shown in Example 2.11.

$$[\{a,b,c\} \mid \{a,b\} \leftarrow \langle\langle table_1, col_1 \rangle\rangle; \{a,c\} \leftarrow \langle\langle table_2, col_2 \rangle\rangle]$$

Example 2.11. Consider the following SQL query used to select all CDs which have Mozart as composer and cost more than 5\$ from the database presented in Example 2.1:

```

SELECT author.firstname, author.lastname, audio.title
FROM audio
JOIN author ON audio.authorid = author.id
WHERE author.lastname = 'Mozart' AND audio.price >5.00;

```

Using the alternative representation demonstrated in Example 2.10, this can be expressed in IQL as:

$$[\{fn,ln,t\} \mid \{id,aid\} \leftarrow \langle\langle audio,authorid \rangle\rangle; \{id,t\} \leftarrow \langle\langle audio,title \rangle\rangle; \{id,p\} \leftarrow \langle\langle audio,price \rangle\rangle; \{aid,fn\} \leftarrow \langle\langle author,firstname \rangle\rangle; \{aid,ln\} \leftarrow \langle\langle author,lastname \rangle\rangle; name = 'Mozart'; price > 5.00]$$

□

List comprehensions can be prefixed with the *distinct* keyword in order to remove duplicates from the resulting list, as shown in Example 2.12.

Example 2.12. An IQL query used to select the first names of all authors from schema S_1 in Example 2.1, with duplicates removed, can be written as:

$$\text{distinct}[\{fname\} \mid \{id, fname\} \leftarrow \langle\langle audio,firstname \rangle\rangle]$$

□

Union between two lists is achieved using the ++ operator, while difference is achieved using the -- operator, as shown in Example 2.13.

Example 2.13. In order to express the union of the identifiers of authors with the identifiers of CDs in schema S_1 in Example 2.1, the following IQL query could be used:

$$[\{id,id\} \mid \{id, id\} \leftarrow \langle\langle author,id \rangle\rangle] ++ [\{id,id\} \mid \{id,id\} \leftarrow \langle\langle audio,id \rangle\rangle]$$

The difference operator is used in a similar manner.

□

IQL also contains anonymous functions, defined using lambda abstractions as in Example 2.14, and let expressions, similar to those found in the Haskell programming language.

Example 2.14. A simple function used to multiply two numbers:

$$\text{lambda } \{n, m\} (* n m).$$

□

Intermediate Query Language is a rich language and contains other built-in functions for aggregation, string and date processing, dealing with collections and type conversion and projection over tuples. A full description of the language can be found in [JPZ03].

2.5 Data Integration Systems

At the present time, research is still being done to bridge the gap between the theoretical study of schema integration and the tools used in industry. Several such tools have been developed, some of which work in a wide range of scenarios, and this section presents some of them: Clio, Tukwila, AutoMed and DB-Main.

2.5.1 Clio

Clio [HHH⁺05] is a tool developed by IBM to express schema mappings between two metamodels: XML and relational schemas. It is the first system that attempted to semi-automate the generation of schema mappings.

The general architecture of the system is shown in Figure 2.4. The most important components are the *mapping generation* and the *query generation*. The mapping generation module is used to analyse correspondences between a source and a target and produces a set of logical mappings, which represent abstractions of the physical transformations (e.g. from SQL to XSLT). The query generator is used to translate from logical mappings to executable queries.

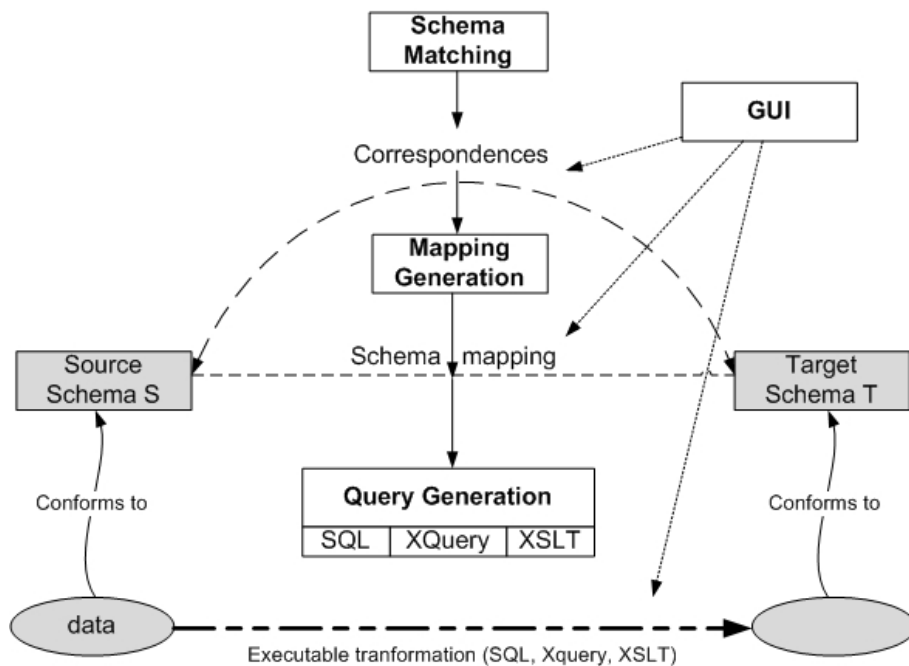


Figure 2.4: Architecture of the CLIO system

Correspondences can either be generated by the *schema matching* module or can be manually inserted by the user in the *GUI*. The user can also inspect generated transformations, an idea that has also been discussed by Bernstein in [BM07]. He suggested that a good tool should offer to the user candidate solutions to consider.

When representing logical mappings, Clio uses an internal notation based on generators, which bind variables to elements in sets, and conditions, which filter the information. This notation is similar to the Intermediate Query Language, covered in Section 2.4.

Each logical mapping generated is converted into a query graph that encodes how the target elements are populated from the data in the source. This graph is then used in the query generator to group and join information, in order to produce a final query.

Clio proved to be one of the most powerful tools for semi-automatic schema mapping. It has also been incorporated into IBM Rational Data Architect (RAD), a system used to explore logical schemas, discover relationships and produce physical schemas [Haa07].

2.5.2 Tukwila

Tukwila [IFF⁺99] is a popular data integration system that focused on optimising query processing over several heterogeneous sources when the data transferred is of moderate or large size.

This is a major challenge, because in a distributed environment there are no statistics about the data, which are vital to a good query execution planner, the data arrival rates cannot be approximated and the information stored in different sources can overlap.

Tukwila introduced adaptivity in two places: between the optimiser and the execution engine and inside the execution engine. This is done by evaluating queries only partially when there is not enough metadata and by using adaptive operators. The adaptive operators defined are the double pipelined hash join (“a join implementation which executes in a symmetric, data-driven manner” [IFF⁺99]) and techniques for adapting execution when there is not enough memory.

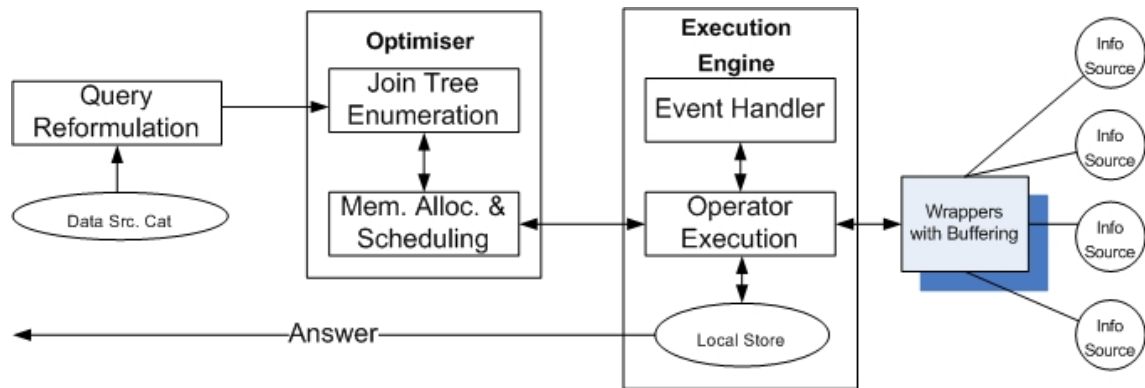


Figure 2.5: Architecture of the Tukwila system

The architecture of Tukwila is shown in Figure 2.5. The user sends queries to the mediated relational schema. The data source catalogue contains metadata about the different sources involved in the transaction, such as the information stored by each source and how data overlaps between sources. Based on the data source catalogue, the query is reformulated as a set of queries over the sources. The query optimiser takes the result of the previous step and creates partial execution plans, which are fed into the query execution engine. The execution engine analyses the query plans and supports incremental re-optimisation. The communication between the engine and the sources is done using wrappers, which are needed for instance when the sources are expressed in different metamodels.

The most important components of this architecture are the query optimiser and the query execution engine. Unlike the most database management systems (DBMS), the query optimiser does not necessarily create complex execution plans, being able to produce partial execution plans as well in an incremental fashion by constantly communicating with the execution engine. This is the case when there are no statistics about the data.

The query execution engine is responsible with executing the query plans received from the optimiser and of gathering statistics after each operation, which are used at later stages.

Tukwila represented a major success in query optimisation and execution for data integration.

More details about the current state of the development of Tukwila can be found in [Tuk].

2.5.3 AutoMed

AutoMed [BKL⁺04] is the first implementation of the Both As View (BAV) approach (described in Section 2.2.1) for heterogeneous data integration.

Whenever a data source is imported, its schema is added to a repository, which is situated at the center of AutoMed’s architecture, shown in Figure 2.6. The repository is split into the **model definitions repository** (MDR) and the **schema transformation repository** (STR).

MDR stores information about various metamodels. Each metamodel is described as a combination of nodes, edges and constraints in the **hypergraph data model** (HDM) [BKL⁺04], which is the Common Data Model (CDM) used by the tool. This module allows users to define new data modelling languages in the MDR and share them with other users. Translating between a source and target schema expressed in different language models is done by first converting constructs to HDM and after that analysing the constraint information.

STR stores schemas, in terms of the data modelling concepts in the MDR, and BAV transformations between schemas [BKL⁺04]. In STR, all data modelling languages have a common representation, described in more detail in [BKL⁺04].

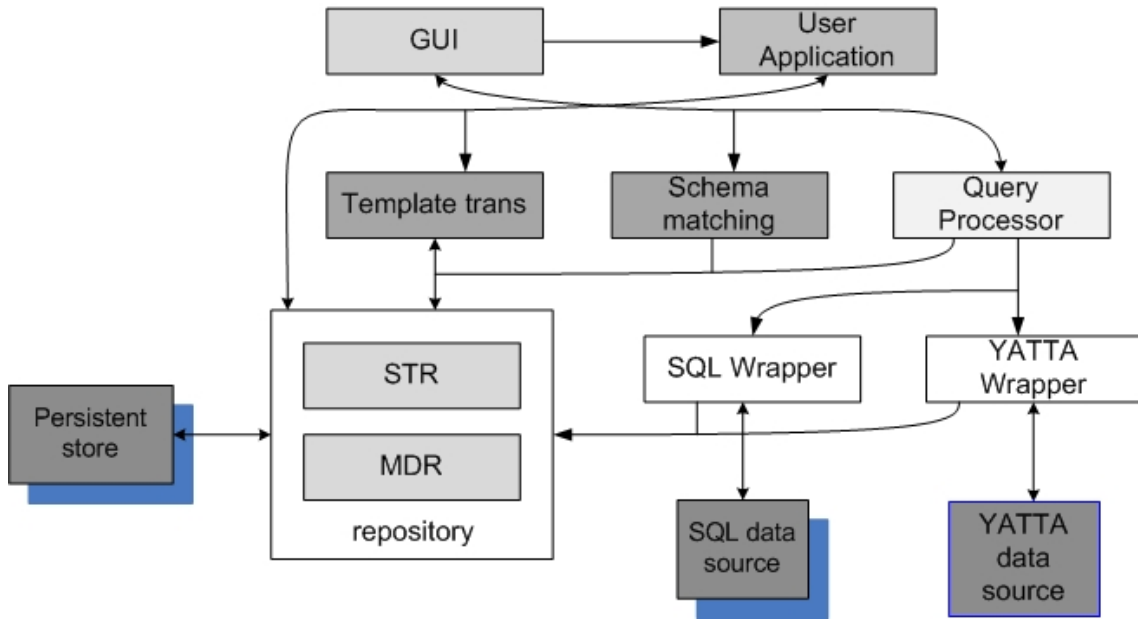


Figure 2.6: Architecture of the AutoMed Software

AutoMed contains a **schema matching module**, used to automatically identify correspondences (as defined in Definition 2.4) between elements in different sources. Unlike other approaches which only identify the compatibility relationship between elements, in [Riz04] five types of relationships between two elements are identified:

- *equivalence* iff $Dom_{int}(A) = Dom_{int}(B)$
- *subsumption* iff $Dom_{int}(B) \subset Dom_{int}(A)$
- *intersection* iff $Dom_{int}(A) \cap Dom_{int}(B) \neq \emptyset, \exists C : Dom_{int}(A) \cap Dom_{int}(B) = Dom_{int}(C)$
- *disjointness* iff $Dom_{int}(A) \cap Dom_{int}(B) = \emptyset, \exists C : Dom_{int}(A) \cup Dom_{int}(B) \subseteq Dom_{int}(C)$

- *incompatibility* iff $Dom_{int}(A) \cap Dom_{int}(B) = \emptyset, \neg \exists C : Dom_{int}(A) \cup Dom_{int}(B) \subseteq Dom_{int}(C)$

The notation $Dom_{int}(x)$ represents the intended domain of x , *i.e.* the real-world entities that the object represents, while $\exists C : condition$ represents that there is a real-world concept that can be represented by an existing or non-existing schema element C that satisfies the *condition* [Riz04].

The **template transformation module** is used to generate transformations (as defined in Definition 2.5) between different sources. AutoMed supports composite transformations (also called template transformations) constructed from primitive transformations.

It has been shown in [SRM] that AutoMed can be extended in practice to create a model management system, as presented in Section 2.1, with all operators, including *Match* and *ModelGen*.

2.5.4 DB-Main

DB-Main [DBM] is a data-modelling and data-architecture tool. It includes functionality for designing, maintaining, evolving, transforming and reverse engineering databases.

It also contains two toolboxes, one for schema integration and one for defining transformations, which, together, form the **integration toolkit** [dDNmP⁺00]. It offers three forms of integration: manual, semi-automatic and fully automatic. It presents several integration strategies to the analyst and it is up to him to solve conflicts, as shown in Figure 2.7.

DB-Main can be extended with a **mapping assistant**, used to define and maintain a history of all transformations between objects belonging to different schemas.

DB-Main was first released in 1991, and, due to its success, it is still being used in education, research and industry.

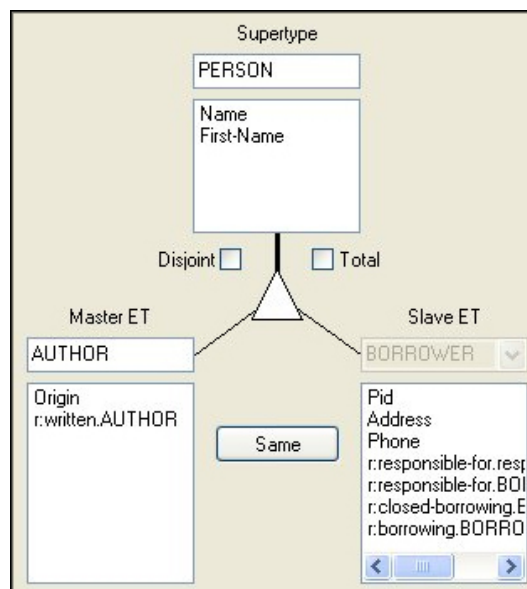


Figure 2.7: The DB-Main integration toolkit. Attributes belonging to two different entities, but having the same semantics, can be merged.

Chapter 3

The Interactive Database Integration Tool

Interactive Database Integration Tool is a software application developed in Java on top of the AutoMed public library. The goal of the application is to guide the user through the integration process of several schemas expressed in the SQL metamodel (described in Section 2.3) using well-known transformation patterns.

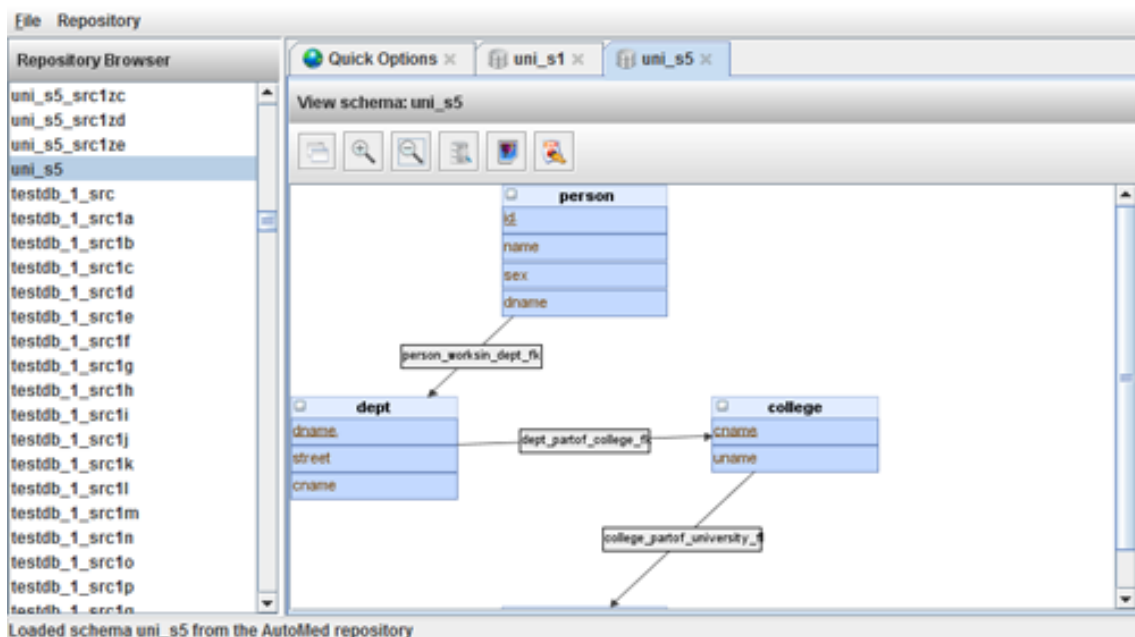


Figure 3.1: The *Interactive Database Integration Tool*.

Before proceeding to a detailed discussion about the architecture of the application, in Chapter 4, and a theoretical description of the transformation patterns implemented in the tool, in Chapter 5, the main features of the tool are presented in this chapter.

In this chapter, Section 3.1 describes the main window of the tool, Section 3.2 presents the SQL schema diagrams generated in the application, Section 3.3 introduces the notion of integration project and how it is implemented in the tool, Section 3.4 discusses about how to add a new schema to the AutoMed repository and Section 3.5 explains how a schema can be queried in the tool.

3.1 The Main Window

In order to run the tool, a working version of AutoMed and Java Runtime Environment (JRE) must be installed on the computer. The main window is split into four parts: the main menu bar, located at the top, the repository browser on the left, the tabbed area on the right and the status bar at the bottom, as illustrated in Figure 3.1.

From the main menu bar, the user can create/save/load an integration project, hide the repository browser or perform operations on the AutoMed repository, such as adding a new schema to the repository.

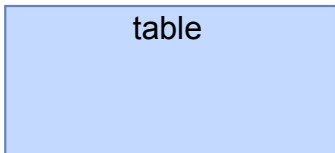


By right-clicking any schema in the repository browser, the user has the option to view the diagram of the schema, rename the schema, remove it or retract it from the repository. The difference between removal and retraction is that, in addition to removing the schema from the AutoMed repository, retraction removes all the schemas derived from the schema being retracted.

The tabbed area on the right contains three types of tabs:

- **Quick Options.** Displayed when first running the application and contains buttons for some of the most important features of the project.
- **SQL Schema Diagram.** Opened when choosing to view a diagram from the repository browser. Explained in detail in Section 3.2.
- **Integration Project.** Opened when performing integration of several schemas.

3.2 SQL Schema Diagrams

Schemas expressed in the SQL metamodel are visually represented as diagrams. Table 3.2 contains the four SQL constructs, together with their visual representation and the scheme definition associated with the construct.

Name	Visual representation	Definition
table		$table : \langle\langle table \rangle\rangle$
column		$column : \langle\langle table, col, opt, type \rangle\rangle$, where $opt \in \{null, notnull\}$ and $type$ represents the data type of the column. If the column is part of the primary key, its name is underlined. If the column is optional, its name is suffixed with a question mark.
primary key		$primarykey : \langle\langle name, t, \langle\langle t, c_1 \rangle\rangle, \dots, \langle\langle t, c_n \rangle\rangle \rangle\rangle$

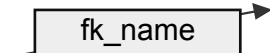
foreign key		$foreignkey : \langle \langle name, t_1, \langle \langle t_1, c_1 \rangle \rangle, \dots, \langle \langle t_1, c_n \rangle \rangle \rangle, t_2, \langle \langle t_2, c_1 \rangle \rangle, \dots, \langle \langle t_2, c_n \rangle \rangle \rangle$
-------------	---	---

Table 3.2: Visual representation and description of the SQL constructs

Figure 3.2 shows a SQL schema diagram generated using the tool. The tool bar displayed on top allows the user to auto-arrange the diagram, zoom in/out, execute an IQL query and retrieve the results and export the diagram to the PNG and Extended PostScript (EPS) formats.

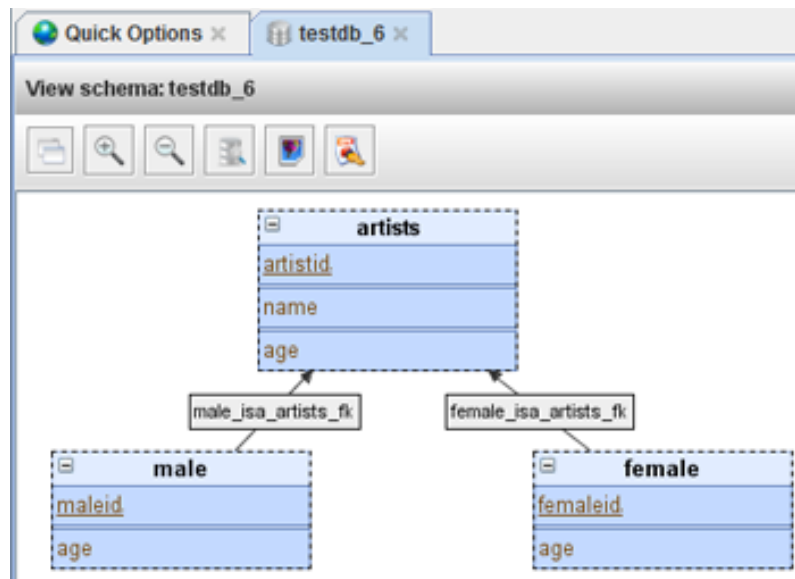


Figure 3.2: The SQL Schema Diagram viewer.

3.3 Integration Projects

The goal of the application is to guide the user through the integration of several schemas expressed in the SQL metamodel. In the tool, this process is captured as an **integration project**, and is based on the steps of the integration process described in [BLN86]: *pre-integration* and *schema transformation*. Different issues and approaches for this process have been identified, some of which are presented in [PS98].

The main menu bar contains items that allow the user to start a new integration project, save or open an existing integration project. Projects are stored on the disk as XML files.

3.3.1 Pre-integration

In general, pre-integration is done in order to establish a common understanding of the databases being integrated. During this step, schemas are translated in a common data model (CDM), which in AutoMed is the hypergraph data model (HDM).

In the tool, the user performs an analysis and chooses from the AutoMed repository the schemas to be integrated and defines the integration strategy, which can be either binary or n-ary, as shown in Figure 3.3. The integration strategy is a tree, which governs the order in which the schemas are integrated. Integration proceeds by traversing the tree in reverse Breadth First Search order, *i.e.* starting from the last level and going up to the root.

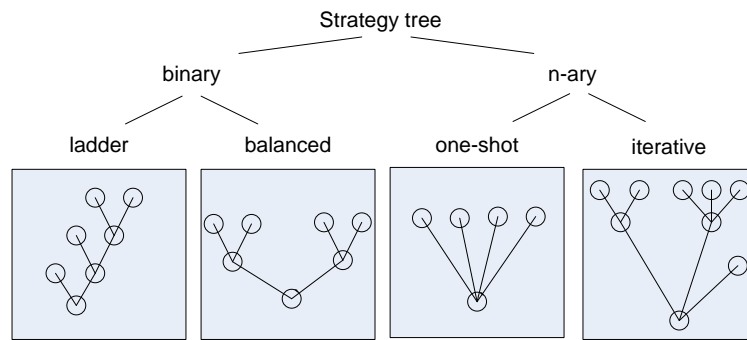


Figure 3.3: Different types of integration strategy trees

In the tool, the strategy tree is constructed by adding schemas to the strategy diagram, selecting the schemas to be integrated in the same step, right-clicking the diagram and combining the selected schemas in an intermediate schema. Once the integration strategy is completed, the user can proceed to the schema transformation phase.

3.3.2 Schema Transformation

The actual integration is performed by traversing the strategy tree and, for each step in the tree, performing schema transformation, a process that is divided into three steps: schema conforming, schema merging and schema improvement.

Schema Conforming

Before transforming the schemas, correspondences between them have to be identified. This is done by searching for schema objects that model the same real-world facts. A correspondence is usually expressed intensionally, as the relationship between the types of the schema objects and not extensionally, between the instances of the databases. At the present time, it is up to the user performing the integration to manually identify the correspondences, although in future this will be done automatically, by employing techniques from *schema matching*.

Once the correspondences have been discovered, the user can proceed to the actual transformation of the schemas. Schema conforming is concerned with solving *conflicts*, *i.e.* cases when the same real-world concept is represented in different ways. A detailed presentation of the different types of conflicts is presented in [kSH99], although, in practice, the following types of conflicts are encountered frequently:

- **Naming conflicts** may arise as a result of different people using their own terminology and naming when designing a database schema. Two types of naming conflicts exist:
 - *Homonyms* when the same name is used for two different concepts. This type of conflict is solved by renaming one of the objects or by prefixing the names of both objects.
 - *Synonyms* when different names are used for the same concept. This type of conflict is solved by assigning the same name to the two objects.
- **Structural conflicts** arise when the same concept is modelled in different ways. They are solved by transforming the schemas, such that the concept is expressed in the same manner. For instance, this type of conflict occurs between a schema containing a table *student* with an optional column *username*, while in another schema the *username* column appears as mandatory in a table *members*, which is a child of *student*. In this case, the same real-world

concept, that of students that have an username associated to their account, is represented in two different ways.

- **Data/metadata conflicts** arise when values appearing in the instance of one database correspond to metadata (type names) in the schema of another database. For instance, this type of conflict occurs when one schema contains a table *student* with a column *college* taking one of the values $\{ICL, Berkley\}$, *i.e.* students from either Imperial College London or Berkeley University, and another schema that contains the table *student* and two child tables called *icl_student* and *berkeley_student*, that model the same information as in the first schema.
- **Data conflicts** arise at instance level when corresponding schema objects store values in different ways. For instance, consider one schema containing a table *student* with a *birthday* column, stored in the DD-MM-YYYY format, while in another schema the *birthday* column is expressed in the MM-DD-YYYY format. This type of conflict might occur as a result of different persons defining the two schemas.

Schema Merging

During the schema merging phase, the component schemas being integrated in the current step are superimposed, such that concepts appearing in all schemas are represented by the same constructs. The final schema is improved by identifying related concepts and adding constructs to represent these relationships.

Schema Improvement

During the schema improvement phase, transformations are applied over the merged schema in order to improve it. This is done until the resulting schema satisfies the three qualities presented in [BLN86]: completeness, minimality and understandability.

In all three steps described above, transformations are applied over schemas, some of which are described in Chapter 5. In the tool, when the user selects several objects belonging to one schema, by holding the Control key pressed and clicking objects in the diagram, and right-clicks the diagram, a menu pops up showing the transformations that can be applied to the selected objects. By selecting one of the transformations, a dialog box appears and the user is guided through the application of the transformation.

Transformations can also be applied in textual mode, by clicking the "Execute Transformations" button displayed on top of SQL schema diagrams. In the integration view, the user can also execute BAV transactions, which are sequences of primitive BAV transformations that are either executed together or are not executed at all, by clicking the "Execute Transaction" button. Transformations can be reverted by clicking the *Undo* button displayed in the menu bar of the schema.

3.4 Adding a Schema to the Repository

The application currently supports SQL schemas of databases running on the PostgreSQL database management system. Schemas can be added to the AutoMed repository by clicking the *Add New Source* menu item in the *Repository* menu and by providing the details of the database. After adding them to the repository, schemas appear in the repository browser on the left and can be used in integration projects.

3.5 Querying a Schema Using IQL

A schema can be queried using IQL at any time by clicking the "Execute an IQL Query" button situated in the tool bar of a SQL diagram. The contents of a table or a column can be quickly viewed by simply double-clicking the element in the SQL diagram.

AutoMed supports the incremental processing of a query, so only a small number of results are fetched and displayed at once. The user is allowed to move to the next page of results by clicking the "Next Page" button. Since the execution of certain queries takes a long time, retrieval of the results is done in a separate thread, in order to avoid blocking the user interface of the application.

Unfortunately, at the present time IQL does not support the retrieval of just a portion of the tuples in a list, which is the equivalent of the *LIMIT* < number > *OFFSET* < number > clause in SQL. This makes it hard to move back and forth in the result set, so in our implementation the user is only allowed to move forward.

3.6 Summary

Interactive Database Integration Tool is a software application built on top of the AutoMed library that guides the user through the integration of several schemas. The user can start or continue an integration project, which is divided into two phases: pre-integration and schema transformation.

Schema transformation is a three step process: schema conforming, schema merging and schema improvement. This phase is performed by applying well-known transformation patterns, until the final virtual schema is obtained.

The tool also supports the addition of new schemas in the AutoMed repository, querying of the schemas using IQL and several tools used to guide the user through the integration process.

Chapter 4

Architecture of the Application

The architecture of a software product is one of the most critical aspects. It dictates how extensible and maintainable an application is and how well it will cope to changes in future.

This chapter presents the architecture of *Interactive Database Integration Tool* and is structured as follows. Section 4.1 gives an overview of the architecture of the application, Section 4.2 outlines the transformation patterns framework, Section 4.3 discusses about the model and Section 4.4 presents the view. In Section 4.5, a short tutorial is given about how a new transformation pattern can be introduced in the application. This also serves as proof about the strength of the architecture employed.

4.1 Overview of the Architecture

The goal of the architecture presented in this paper is to clearly separate the model (*i.e.* data and behaviour) from the view (*i.e.* user interface), whilst creating a powerful and extensible transformation patterns framework. In the end, adding a new transformation pattern into the application should be as simple as extending one class, without the modification of the existing code.

The diagram in Figure 4.1 contains the main packages of the application. The packages at the top of the hierarchy depend on the packages below them. In other words, in order to reuse a package in another product, all packages below it must be extracted as well.

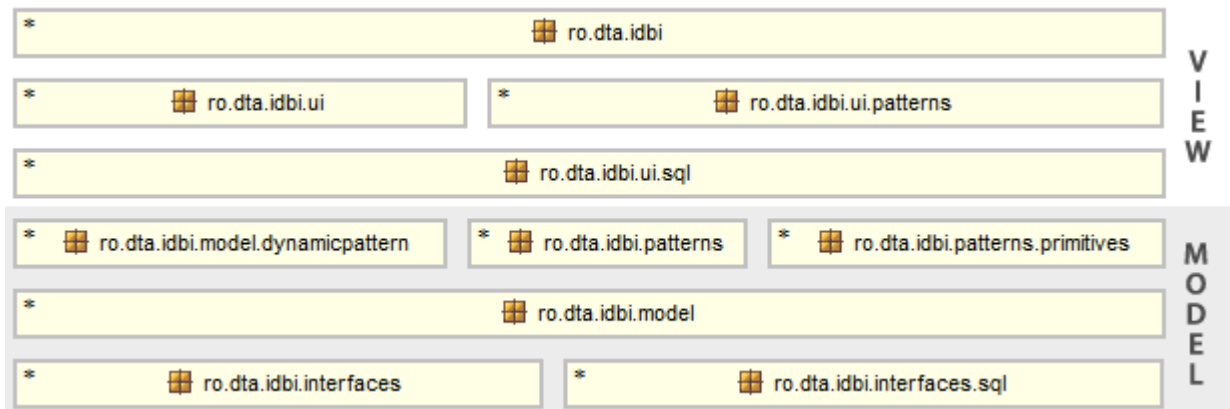


Figure 4.1: Architecture of the Application. Diagram generated using Structure101 [Str].

The two interface packages at the bottom of the hierarchy contain interfaces used to break cyclic dependencies, *i.e.* modules that depend on each other. In this way, for instance, the *ro.dta.idbi.patterns* package can be reused in another product without having to reuse the user

interface of the application, which is above it in this hierarchy.

The **model** of the application is contained in four packages situated at the bottom of the hierarchy: *ro.dta.idbi.model*, *ro.dta.idbi.patterns* and *ro.dta.idbi.patterns.primitives* and *ro.dta.model.dynamicpattern*. Package *ro.dta.idbi.model* contains classes used to define new transformation patterns (*i.e. AbstractPattern*), new primitive BAV transformations (*i.e. AbstractPrimitivePattern*), handle integration projects (*i.e. IntegrationProject*) and perform operations in the AutoMed repository (*i.e. AutoMedUtils*). This package is presented in more detail in Section 4.3.

Package *ro.dta.idbi.patterns* contains the implementation of the transformation patterns described in Chapter 5, while package *ro.dta.idbi.patterns.primitives* contains the implementation of the five primitive BAV transformations: *add*, *delete*, *extend*, *contract* and *rename*. Package *ro.dta.idbi.dynamicpattern* contains the implementation of the pattern discovery method described in Chapter 6.

The **view** of the application is contained in three packages: *ro.dta.idbi.ui*, *ro.dta.idbi.ui.sql* and *ro.dta.idbi.ui.patterns* and is presented in detail in Section 4.4. *ro.dta.idbi.ui* contains the user interface elements of the application, while *ro.dta.idbi.ui.sql* contains the visual elements for the constructs in the SQL metamodel and for SQL diagrams. *ro.dta.idbi.ui.patterns* contains classes for the user interface of the transformation patterns defined in *ro.dta.idbi.patterns* and a class used to define the UI for new transformations (*i.e. AbstractPatternDialog*).

Since the model of the application is situated below the view in the diagram in Figure 4.1, it is obvious that it can be independently reused in another application.

4.2 The Transformation Patterns Framework

The goal of the transformation patterns framework is to allow the introduction of new transformation patterns in the application without having to modify the existing code. This goal can be satisfied by respecting the following conventions:

Convention 4.1. All transformation patterns must extend *AbstractPattern* (described in Section 4.3) and are stored and loaded from the *ro.dta.idbi.patterns* and *ro.dta.idbi.patterns.primitives* packages.

Convention 4.2. All transformation pattern must contain the following static fields:

- String NAME: the short name of the pattern
- String DESCRIPTION: a description of what the pattern does
- String COMMAND: the command associated with the pattern
- EnumSet<IntegrationOperation> OPERATION: the schema transformation steps that the pattern can be applied in. This can be CONFORMING, MERGING, IMPROVING or any combination of these.

Convention 4.3. The user interface associated with transformation patterns must extend *AbstractPatternDialog*, be stored in the *ro.dta.idbi.ui.patterns* package and their name must be the name of the transformation pattern class suffixed with the word *Dialog*.

Convention 4.4. If a transformation pattern has a user interface class associated with it, it must contain a static method with the signature `boolean isValid(Schema, Object[])`, returning true if the transformation can be applied to the selected objects in the context of the schema passed as parameter.

In the user interface, described in Section 4.4, if the user selects several items and right-clicks the SQL diagram, the packages listed in Convention 4.1 are scanned using the Java Reflection API and only those patterns that can be applied in the current integration step are considered. This is done by comparing the static field `OPERATION` with the current integration step.

Next, the `isValid` method of all patterns that have a user interface associated with them is called and only those patterns that can be applied to the currently selected items are shown.

The execution of a command is done by comparing the `COMMAND` static field to the name of the command being executed. If a pattern is found, it is instantiated and the `execute` method, described in Section 4.3, is fired.

This strategy ensures that no code modification is necessary when creating a new transformation pattern, if the conventions listed above are respected.

4.3 The Model of the Application

The model of the application holds the data and behaviour and is composed of four packages: *ro.dta.idbi.model*, *ro.dta.idbi.patterns*, *ro.dta.idbi.patterns.primitives* and *ro.dta.idbi.dynamic-pattern*. This section presents some of the most important features and classes in the model.

The *AbstractPattern* Class

The *AbstractPattern* abstract class, shown in Figure 4.3, lies at the foundation of the model of the application. By subclassing this class new transformation patterns can easily be introduced in the application. The execution of any pattern follows the same six steps, as illustrated in Figure 4.2:

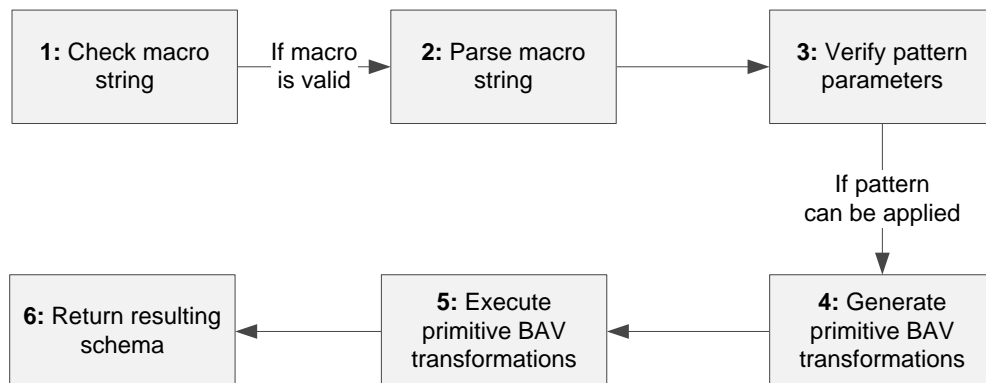


Figure 4.2: Steps in the execution of a transformation pattern.

1. **Check macro string.** Every transformation pattern has a macro command associated with it, which is expanded into one or several primitive BAV transformations. The macro command passed to the transformation is validated against a regular expression. The `pattern` field must be initialised in the constructor of the subclass with the appropriate regular expression associated with the command.
2. **Parse macro string.** The parameters of the pattern are extracted from the textual representation of the macro command. This is done by implementing the abstract method `void parse (String trans)`.
3. **Verify pattern parameters.** Some transformation patterns, called **knowledge-based patterns** [MP98], must satisfy certain conditions in order for them to be successful. The

- boolean `verify()` method returns true if the parameters of the transformation pattern, extracted from Step 2, satisfy these conditions.
4. **Generate primitive BAV transformations.** The macro command is expanded into primitive BAV transformations, expressed in textual form. This is accomplished by implementing the abstract method `List<String> getPrimitiveCommands()`, which returns a list of the primitive BAV transformations.
 5. **Execute primitive BAV transformations.** The primitive BAV transformations are executed in sequence.
 6. **Return resulting schema.** The final schema is returned as a result of the execution of the transformation pattern.

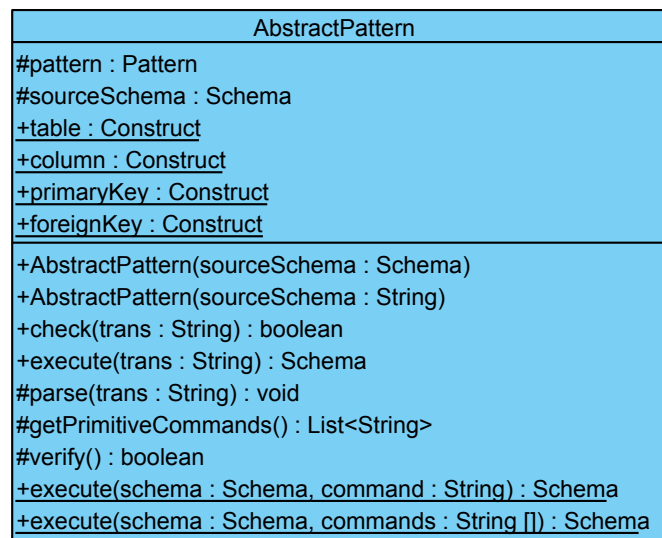


Figure 4.3: The *AbstractPattern* class diagram.

These steps are implemented in the source code of the application in the `execute` method of the *AbstractPattern* class, as shown in Listing 4.1.

Listing 4.1: Execution of a transformation pattern

```
public Schema execute(final String trans) {
    // Step 1
    if (!check(trans)) {
        throw new IllegalArgumentException("Command is invalid.");
    }

    // Step 2
    parse(trans);

    // Step 3
    if (!verify()) {
        throw new IllegalArgumentException("Can't apply this transformation.");
    }

    // Step 4
    List<String> commands = getPrimitiveCommands();

    // Step 5
    Schema crtSchema = sourceSchema;
```

```

for (String cmd : commands) {
    try {
        crtSchema = AbstractPattern.execute(crtSchema, cmd);
    } catch (Exception e) {
        throw new TransformationException(e);
    }
}

// Step 6
return crtSchema;
}

```

The class contains two static methods named `execute`, illustrated in Figure 4.3. One of them is for the execution of only one macro command using the strategy described in Section 4.2, while the other method is for the execution of a sequence of macro commands.

The *AbstractPattern* class also contains static fields for the basic SQL constructs used by the AutoMed library (*i.e.* table, column, primary key and foreign key), and several pre-defined regular expressions, listed in Table 4.2, which are commonly used in practice.

Name of the static field	Description
SPACE_REGEX	Any number of spaces
COMMA_REGEX	One comma surrounded by any number of spaces
NAME_REGEX	An alphanumerical sequence of characters
TABLE_REGEX	Name of a table, <i>i.e.</i> $\langle\langle table \rangle\rangle$
COL_REGEX	Name of a column, <i>i.e.</i> $\langle\langle table, col \rangle\rangle$
VAL_SEQ_REGEX	Sequence of values, <i>i.e.</i> $\{v_1, v_2, \dots, v_n\}$
COL_SEQ_REGEX	Sequence of COL_REGEX, separated by commas
TABLE_SEQ_REGEX	Sequence of TABLE_REGEX, separated by commas
TABLE_AND_COLS_REGEX	One TABLE_REGEX, followed by a sequence COL_REGEX
TABLE_DEF_REGEX	Definition of a table, <i>i.e.</i> $table : \langle\langle table \rangle\rangle$
COL_DEF_REGEX	Definition of a column, <i>i.e.</i> $column : \langle\langle table, col, opt, type \rangle\rangle$
PK_DEF_REGEX	Definition of a primary key, <i>i.e.</i> $primarykey : \langle\langle name, table, \langle\langle table, col_1 \rangle\rangle, \dots, \langle\langle table, col_n \rangle\rangle \rangle\rangle$
FK_DEF_REGEX	Definition of a foreign key <i>i.e.</i> $foreignkey : \langle\langle name, table_1, \langle\langle table_1, col_1 \rangle\rangle, \dots, \langle\langle table_1, col_n \rangle\rangle, table_2, \langle\langle table_2, col_1 \rangle\rangle, \dots, \langle\langle table_2, col_n \rangle\rangle \rangle\rangle$

Table 4.2: Regular expressions in the *AbstractPattern* class

By combining the constants shown in Table 4.2, regular expressions for new transformation patterns can be computed.

Example 4.1. Consider the following macro command for a transformation pattern:

```
myCommand (<<table1,col1>>, <<table1>>, <<table2>>, {x,y,z})
```

A regular expression for this macro could be defined as:

```
"myCommand" + SPACE_REGEX + "(" + COL_REGEX + COMMA_REGEX + TABLE_SEQ_REGEX
+ COMMA_REGEX + VAL_SEQ_REGEX + "\)"
```

□

When defining a new transformation pattern and subclassing *AbstractPattern*, the developer is only concerned with initialising the `pattern` field, implementing the abstract methods and adding the four static fields listed in Convention 4.2.

All 18 patterns presented in Chapter 5 extend *AbstractPattern*. An example of how to implement a transformation pattern is given in Section 4.5.

The *AbstractPrimitivePattern* Class

The primitive BAV transformations are implemented as transformation patterns in the tool and extend *AbstractPrimitivePattern*, which in turn extends *AbstractPattern*. The reason for doing this is to allow the user to execute both transformation patterns and primitive transformations in textual form, whereas the primitive BAV transformations are executed in an object-oriented manner in the AutoMed library, by calling methods.

Another advantage of this approach is that the reverse transformations associated with patterns can be automatically derived, as shown in Section 5.1, something that is cumbersome using the object-oriented manner.

AbstractPrimitivePattern is extended by five classes, representing the five primitive BAV transformations: *AddPrimitive*, *DeletePrimitive*, *ExtendPrimitive*, *ContractPrimitive* and *RenamePrimitive*.

The *Pattern* Class

The *Pattern* class holds metadata about transformation patterns, such as the name, a description, the command associated with the pattern, the schema transformation steps (*i.e.* conforming, merging or improvement) that the pattern can be applied in, the class associated with the model of the pattern and the class associated with the UI of the pattern, if one exists.

The *IntegrationProject* Class

The *IntegrationProject* class is used to model a schema integration project. It holds information about the integration strategy and the current progress. It offers the possibility to advance through an integration project and save/load a project to/from an XML file. The parsing of XML files is done using the Xerces [Xer] public library.

The class also holds a history of the transformations applied to the schemas. This is an implementation of the memento design pattern [GHJV95], which makes it possible to restore one of the schemas being integrated at the current step to a previous state.

The *AutoMedUtils* Utility Class

The *AutoMedUtils* utility class holds static methods used to perform various tasks in the AutoMed repository, such as:

- Wrap a schema
- Run an IQL query over a schema and get the results

- Run an IQL query incrementally over a schema and get the results
- Merge a list of schemas
- Get the primary key column(s) of a table
- Get the parent tables of a table
- Get the foreign key constraint between a parent and a child table
- Get the foreign key constraint between two tables, given the source and target columns

The *Utils* Utility Class

The *Utils* class contains static utility methods for retrieving a list of patterns which can be applied to a given schema transformation step and for generating the scheme definition of the four constructs in the SQL metamodel. The most significant methods in the class are:

- `List<Pattern> getPatterns(IntegrationOperation operation)`.
Get the transformation patterns that can be applied in the schema transformation step passed as a parameter.
- `String genTableRepresentation(String tblName)`.
Generate the scheme definition of a table, given its name.
- `String genColRepresentation(Object tblName, Object colName, Object nullable, Object type)`.
Generate the scheme definition of a column, given its description.
- `String genFKRepresentation(String fkName, String sourceTable, String targetTable, List<String> sourceCols, List<String> targetCols)`.
Generate the scheme definition of a foreign key, given its name, the source and target tables, and the column associated to the source and target tables.
- `String genPKRepresentation(Object[] pkArr)`.
Convert an object array describing a primary key to the scheme definition of the primary key.

4.4 The View of the Application

The view of the application contains the user interface (UI) elements and is built on top of the model, by referencing it. This section presents some of the most important classes in the three view packages: *ro.dta.idbi.ui*, *ro.dta.idbi.ui.sql* and *ro.dta.idbi.ui.patterns*.

The *MainFrame* Class

The *MainFrame* class represents the main frame of the application. It contains references to the *RepositoryBrowserPanel* displayed on the left, where schemas from the AutoMed repository are shown, and the tabbed pane on the right, which holds tabs for integration projects or schema diagrams.

The *AbstractPatternDialog* Class

The *AbstractPatternDialog* abstract class is used to define the user interface associated with a transformation pattern, in the spirit of the figures presented in Chapter 5.

When subclassing it, a static method `boolean isValid(Schema schema, Object[] objects)` must be implemented, according to Convention 4.4. The method returns true if the transformation can be applied in the context of the schema and selected objects passed as parameters. This method is

used to filter and display to the user only the transformations that can be applied to the selected constructs.

Two abstract methods must be implemented, `mxGraphComponent createOriginalDiagram()` and `mxGraphComponent createTransformedDiagram()`, showing before and after snapshots. The transformed diagram should also allow the user to customise the parameters of the pattern.

The field `macroField` is used to display the equivalent macro command and should be updated whenever the transformed diagram is changed.

The user interface for all 18 transformation patterns presented in Chapter 5 are implemented by extending this class.

The *AbstractDiagram* Class

The abstract class *AbstractDiagram* is used to define new types of diagrams. In the application two types of diagrams exist: *StrategyDiagram* and *SqlSchemaDiagram*. This class could be extended to introduce diagrams for other metamodels, such as Entity-Relationship or relational.

Every diagram contains an instance of a *mxGraphComponent* from the JGraph [JGr] public library, holding the elements of the diagram, and the functionality to save the diagram to various formats.

The `save` method is an implementation of the Strategy Design Pattern [FFBS04]. The family of algorithms for saving a *mxGraphComponent* to disk is represented by the interface *ISaveStrategy*, illustrated in the UML class diagram in Figure 4.4. The strategy used to save the diagram is chosen at runtime and is passed as parameter to the `save` method of the *AbstractDiagram* class.

Three such strategies have been implemented in the application, used to save a diagram in XML, PNG or EPS formats. Possible extensions include exporting to SVG, JPEG or PDF.

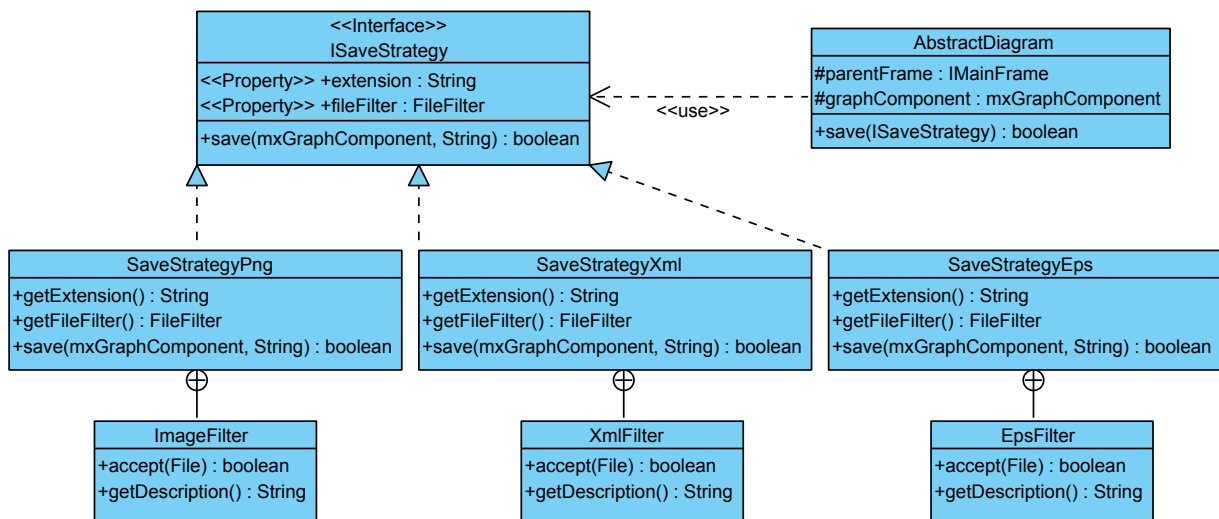


Figure 4.4: Save diagram strategies.

The *StrategyDiagram* Class

The *StrategyDiagram* class extends *AbstractDiagram* and is used when defining the integration strategy tree. It exposes the functionality to save and load from the XML format and to check if the strategy is valid. A strategy is said to be valid if it does not contain duplicate schemas and all intermediate nodes (*i.e.* intermediate schemas) have unique names assigned to them.

StrategyDiagram contains an instance of the *StrategyGraph* class, which extends *mxGraph*

and is a type of graph that allows the user to edit the values of vertices marked as intermediate, *i.e.* intermediate schemas in the integration strategy.

The *SqlSchemaDiagram* Class

Schemas expressed in the SQL metamodel are visually displayed as diagrams, which essentially are graphs. The *SqlSchemaDiagram* class extends *AbstractDiagram* and decodes a schema from the AutoMed repository and shows it to the user.

Instances of the *SqlSchemaDiagram* class are usually contained in instances of *SqlSchemaPanel*, which displays a tool bar, allowing the user to perform various operations on the SQL schema.

The *SqlGraph* Class

The *SqlGraph* class holds information about a SQL schema. It extends the *mxGraph* class of the JGraph [JGr] public library. In the graph $G = (V, E)$ describing a SQL schema, V is the set of tables and columns and E is the set of foreign key constraints. In JGraph, the elements in both V and E are called **cells**.

The following classes extend *AbstractSqlCell*, model the user interface of constructs from the SQL metamodel and can be added to a *SqlGraph*:

- **SqlTableCell**: visual representation a table. It contains functionality to add/remove/get columns, get the foreign keys sourced at this table and get the primary key column(s) of the table.
- **SqlColumnCell**: visual representation of a column. Columns can only be added to tables and are described by their name, the parent table, optionality and type.
- **SqlForeignKeyCell**: visual representation of a foreign key. It holds information about the source and target tables and the source and target columns that are part of the foreign key.

This class also contains style definitions, which are passed in the constructor of the above types of cells:

Name of style	Description
TABLE	Default style of a table
SEL_TABLE	Table with an orange background
COLUMN	Default style of a column
SEL_COL	Column with an orange background
PRIMARYKEY	Primary key column
SEL_PK	Primary key column with an orange background
FK_EDGE	Default style of a foreign key

Table 4.3: Cell styles defined in the *SqlGraph* class.

This class can be extended to create new types of SQL diagrams with different behaviour, such as the *EditableGraph* class, which allows the user to edit the contents of cells that have the style property set to SEL_TABLE or SEL_COLUMN, or the *NormaliseGraph* class used in the *NormalisationPattern* (described in Section 5.1.1), which allows the user to drag and drop columns in the graph.

4.5 Adding a New Transformation Pattern

Consider the example of defining the forward transformation of the *Child/Table Equivalence*, presented in Section 5.1.5, which can be applied during the schema conforming step. This is done by creating a class named *ColToTable* in the package *ro.dta.idbi.patterns* (according to Convention 4.1), which subclasses *AbstractPattern*, and performing the following steps six steps. Other patterns may be implemented in the same manner.

Step 1: Defining the properties of the pattern

The four static fields describing the transformation pattern, listed in Convention 4.2, have to be added, as shown in Listing 4.2.

Listing 4.2: Description of the Pattern

```
/**
 * Name of the pattern
 */
public static final String NAME = "Column to Table";

/**
 * Description of the pattern
 */
public static final String DESCRIPTION = "Move column to a new table";

/**
 * Name of the command associated with the pattern
 */
public static final String COMMAND = "col_to_table";

/**
 * Integration phase during which the command is applied
 */
public static final EnumSet<IntegrationOperation> OPERATION = EnumSet
    .of(IntegrationOperation.CONFORMING);
```

Step 2: Initialising the pattern field

The macro associated with this transformation pattern is:

```
col_to_table (<<table,col>>,<<new_table>>)
```

The pattern field is initialised in the constructor of *ColToTable* with the regular expression associated with the pattern, as shown in Listing 4.3.

Listing 4.3: Initialising the pattern field

```
/**
 * Column to table pattern class constructor
 *
 * @param sourceSchema
 *         Source schema over which the transformation is applied
 */
```

```

public ColToTable(final Schema sourceSchema) throws NotFoundException,
    IntegrityException {
    super(sourceSchema);

    pattern = Pattern.compile("^" + COMMAND + SPACE_REGEX + "\\((" + COL_REGEX + ")
        " + COMMA_REGEX + "(" + TABLE_REGEX + ")\\)");
}

```

Step 3: Implement the parse method

The abstract method `void parse(String trans)` is implemented and the elements defining the pattern are extracted from the textual representation of the command. In this case, the elements are the column to be moved to a new table and the name of the new table.

Listing 4.4: The parse method

```

@Override
protected void parse(final String trans) {
    final Matcher matcher = pattern.matcher(trans);
    matcher.find();

    origColStr = matcher.group(1);
    origCol = parseColName(origColStr);
    newTableStr = parseTableName(matcher.group(2));
}

```

Step 4: Implement the verify method

The abstract method `boolean verify()` is used to verify if the command satisfies all constraints and can be applied. In this case, there are no constraints to be satisfied, so the method simply returns `true`.

Listing 4.5: The verify method

```

@Override
protected boolean verify() {
    return true;
}

```

Step 5: Implement the getPrimitiveCommands method

The abstract method `List<String> getPrimitiveCommands()` returns the extension of the macro command as a sequence of primitive BAV transformations, expressed in textual form.

In Listing 4.6, the class *ExtentGenerator* is used to generate the extents of the BAV transformations for this transformation pattern. The static methods belonging to the *Utils* class are explained in Section 4.3.

Listing 4.6: The getPrimitiveCommands method

```

@Override
protected List<String> getPrimitiveCommands() throws NotFoundException,
    IntegrityException {

```



```

final List<String> result = new ArrayList<String>();
final SchemaObject colObj = sourceSchema.getSchemaObject(origColStr);
final IExtentGenerator gen = new ExtentGenerator(origColStr, origCol);
StringBuilder cmd = new StringBuilder();

// Step 1: add new table
String iql = gen.getExtentsForStep(1).get(0);
cmd.append("add ");
cmd.append(Utils.genTableRepresentation(newTableStr));
cmd.append(", ").append(iql).append(" ");
result.add(cmd.toString());

// Step 2: add column to the new table
iql = gen.getExtentsForStep(2).get(0);
cmd.setLength(0);
cmd.append("add ");
Object[] colScheme = colObj.getSchemeDefinition();
cmd.append(Utils.genColRepresentation(newTableStr, origCol.getValue(),
    colScheme[2], colScheme[3]));
cmd.append(", ").append(iql).append(" ");
result.add(cmd.toString());

// Step 3: make new column PK in new table
if (sqlModel.isFeatureInUse(SQLModelDef.PRIMARY_KEY)) {
    Object[] pkArr = new Object[3];
    pkArr[0] = newTableStr + "_pk";
    pkArr[1] = newTableStr;
    pkArr[2] = "<<" + newTableStr + ", " + origCol.getValue() + ">>";
    cmd.setLength(0);
    cmd.append("add ");
    cmd.append(Utils.genPKRepresentation(pkArr));
    cmd.append(" ");
    result.add(cmd.toString());
}

// Step 4: Create FK column in original table
if (sqlModel.isFeatureInUse(SQLModelDef.FOREIGN_KEY)) {
    List<String> col = new ArrayList<String>();
    col.add(origCol.getValue());
    cmd.setLength(0);
    cmd.append("add ");
    cmd.append(Utils.genFKRepresentation(origCol.getKey(), newTableStr, col));
    cmd.append(" ");
    result.add(cmd.toString());
}

return result;
}

```

Step 6 (Optional): Create the UI class

This step is only performed if the transformation pattern should have an user interface associated with it. According to Convention 4.3, the *AbstractPatternDialog* class must be extended and the

new class must be named *ColToTableDialog* and stored in the package *ro.dta.idbi.ui.patterns*.

Convention 4.4 states that a static method named *isValid* must be created. The transformation pattern can only be applied to mandatory columns, as shown in Listing 4.7.

Listing 4.7: The *isValid* method

```
public static boolean isValid(final Schema schema, final Object[] objects) {
    if (objects.length != 1 || !(objects[0] instanceof ISqlColumn)) {
        return false;
    }

    final ISqlColumn col = (ISqlColumn) objects[0];
    return !col.isNullable();
}
```

The two abstract methods presented in Section 4.4 must be implemented in order to display the before and after views, and *macroField* must be updated to display the command used to perform the transformation. The final user interface is illustrated in Figure 5.11.

Step 7 (Optional): Implement the reverse transformation

Since we are dealing with a bi-directional transformation, it makes sense to implement the reverse transformation. This is done by creating a class *ColToTableRev* in the package *ro.dta.idbi.patterns* by following the same steps as above.

As explained in Section 5.1, the primitive commands of the reverse transformation are derived automatically by reversing the order of the reverse transformations. This can be easily done, as shown in Listing 4.8, because the primitive commands are expressed in textual form.

Listing 4.8: The *getPrimitiveCommands* method in the reverse transformation

```
@Override
protected List<String> getPrimitiveCommands() throws NotFoundException,
    IntegrityException {
    ColToTable forward = new ColToTable(sourceSchema, origColStr, newTableStr);
    List<String> commands = forward.getPrimitiveCommands();
    List<String> result = new ArrayList<String>();
    for (String cmd : commands) {
        result.add(0, AbstractPrimitivePattern.getReverseTrans(cmd));
    }

    return result;
}
```

At this point, because of the way the transformation patterns framework, described in Section 4.2, is implemented, no other modifications are necessary and the pattern is available in the user interface.

4.6 Unit Testing

Unit testing is concerned with testing individual components of the application. The tests serve as proof that any modification of the code will not introduce bugs in the application and are also a good example of how the code can be used.

The transformation patterns module is the most critical module of the application. For each of the 18 transformation patterns implemented in the tool, a test class has been implemented (using the JUnit [JUn] public library), which contains four test methods: one to check that the `check` method returns `false` for invalid macro commands, one to check that the `check` method returns `true` for legal commands, one to check that no transformations are executed if an invalid macro command is passed to the `execute` method and one to check that executing a valid command produces the expected results.

This way, whenever the transformation patterns are modified, the whole test suite can be executed to check that all patterns produce the expected result and that no bugs have been introduced in the code.

4.7 Implementation Statistics

The following table shows some statistics about the implementation of the tool. In the table, the McCabe cyclomatic complexity [McC76] is defined as a software metric that indicates the complexity of a program. It is measured by analysing the control flow graph, *i.e.* a graph representation of all paths that might be traversed in a program. In general, a low complexity is preferred, as it limits the number of possible paths in the code. From the total lines of code that the source code contains, approximately 99% have been implemented from scratch.

Number of packages	12
Number of classes	191
Number of interfaces	25
Total lines of code	Approx. 16.500
McCabe cyclomatic complexity	2.1
Number of test methods	109

The project was initiated with the idea in mind that at some point it might be extended by other persons, so producing a clean, fully-commented code was a considered important. The comments associated to the code are publicly available at [IDB].

4.8 Summary

This chapter presented the architecture of the application, something that has received considerable attention throughout the development of this project. By ensuring that the model and view are efficiently divided and that no circular dependencies exist, individual modules can be reused in other products.

Using the transformation patterns framework proposed, new patterns can easily be introduced by performing as little as six steps, without having to modify any of the existing code. This makes the architecture highly extensible and maintainable. A short demonstration of how a new pattern can be added to the tool has been included in this chapter.

In order to ensure that by modifying the code no bugs are introduced in the application, unit tests have been implemented.

Chapter 5

Transformation Patterns

Transformation patterns represent sequences of primitive BAV transformations, which are used to map an input schema to an output schema and that occur often in practice. They are applied during the *schema transformation* step of the database schema integration process.

The schema illustrated in Figure 5.1 is used to exemplify the transformation patterns presented in this chapter. It holds information about *artists*, which can be *male* or *female*, and their work, which can either be *audio* or *video*. It also contains

The structure of this chapter is as follows. Section 5.1 presents some of the most common patterns applied during the **schema conforming** phase, Section 5.2 presents transformations for the **schema merging** phase and Section 5.3 presents transformations for the **schema improvement** phase. Every transformation pattern is given with a short description, the macro(s) used to apply the pattern and an example of how the pattern is implemented in the tool.

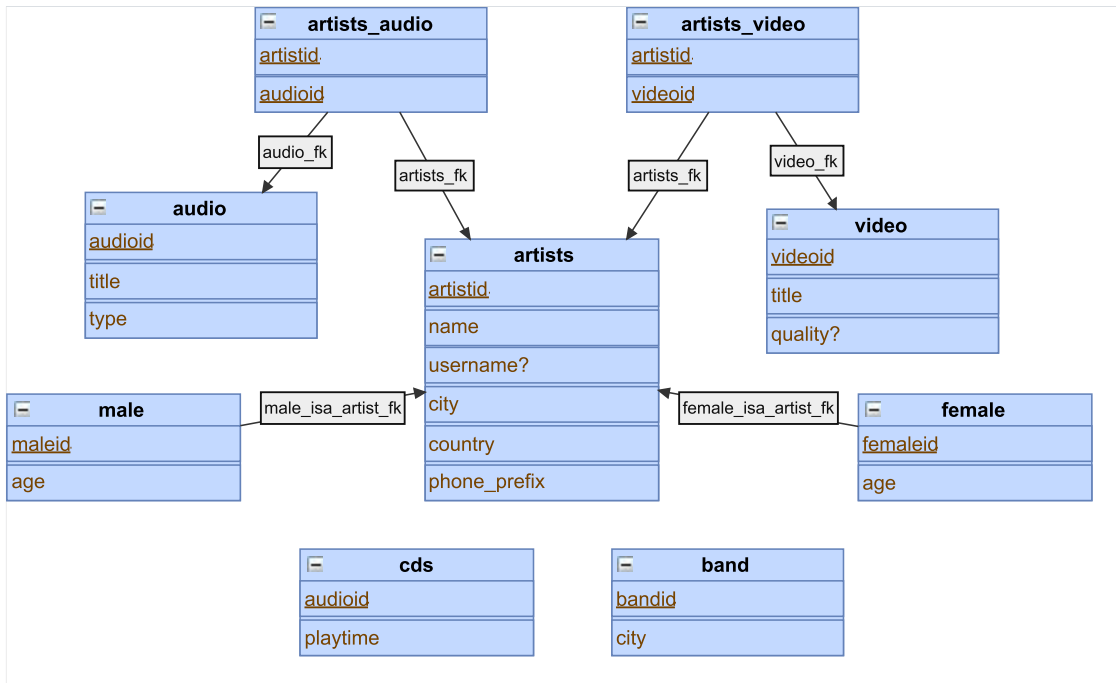


Figure 5.1: Schema used throughout this chapter. Diagram generated using the tool.

5.1 Schema Conforming

Unlike the schema merging and schema improvement transformations, schema conforming transformations are applied bi-directionally and are called **equivalences**. Given a sequence of primitive

BAV transformations, the reverse transformation sequence can be automatically derived by applying the reverse transformations in reverse order.

Consider the following primitive BAV transformations for the SQL metamodel, where *iq1* represents the extent of the construct:

- ① add (table:⟨t⟩, iql)
- ② add (column:⟨t,c,m,d⟩, iql)
- ③ add (primarykey:⟨pk,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩)
- ④ add (foreignkey:⟨fk,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩)
- ⑤ rename (table:⟨t1⟩, table:⟨t2⟩)
- ⑥ rename (column:⟨t,c1,m,d⟩, column:⟨t,c2,m,d⟩)
- ⑦ rename (primarykey:⟨pk1,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩,
primarykey:⟨pk2,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩)
- ⑧ rename (foreignkey:⟨fk1,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩,
foreignkey:⟨fk2,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩)

The associated reverse BAV transformations are:

- ① delete (table:⟨t⟩, iql)
- ② delete (column:⟨t,c,m,d⟩, iql)
- ③ delete (primarykey:⟨pk,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩)
- ④ delete (foreignkey:⟨fk,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩)
- ⑤ rename (table:⟨t2⟩, table:⟨t1⟩)
- ⑥ rename (column:⟨t,c2,m,d⟩, column:⟨t,c1,m,d⟩)
- ⑦ rename (primarykey:⟨pk2,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩,
primarykey:⟨pk1,t,⟨t,c1⟩, ..., ⟨t,cn⟩⟩)
- ⑧ rename (foreignkey:⟨fk2,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩,
foreignkey:⟨fk1,t1,⟨t1,c1⟩, ..., ⟨t1,cn⟩,t2,⟨t2,c1⟩, ..., ⟨t2,cn⟩⟩)

Example 5.1. The following sequence of BAV transformations is used to move the *age* column from the *male* and *female* tables from the schema in Figure 5.1 to their parent table, *artists*:

- ① add (column:⟨artists,age,null,int4⟩, ⟨female,age⟩ ++ ⟨male,age⟩)
- ② delete (column:⟨female,age,notnull,int4⟩,
[{x,age} | {x} ← ⟨female⟩; {x,age} ← ⟨artists,age⟩])
- ③ delete (column:⟨male,age,notnull,int4⟩,
[{x,age} | {x} ← ⟨male⟩; {x,age} ← ⟨artists,age⟩])

The reverse transformation sequence can be derived automatically:

- ① add (column:⟨male,age,notnull,int4⟩,
[{x,age} | {x} ← ⟨male⟩; {x,age} ← ⟨artists,age⟩])
- ② add (column:⟨female,age,notnull,int4⟩,
[{x,age} | {x} ← ⟨female⟩; {x,age} ← ⟨artists,age⟩])
- ③ delete (column:⟨artists,age,null,int4⟩,⟨female,age⟩ ++ ⟨male,age⟩)

□

In this section more emphasis is put on the forward transformations, as the reverse transformations are implemented in a similar way.

5.1.1 Table Normalisation Equivalence

Description of the Transformation Pattern

In database theory, normalisation is the process of restructuring a database schema in order to minimise redundancy. Redundancy can be expressed formally in terms of the patterns of dependencies in a table. *Dependencies* represent logical constraints on databases.

Definition 5.1 Functional dependency

Let $\vec{A} = \{a_1, a_2 \dots a_n\}$ be a set of columns belonging to the same table. We say that \vec{A} *functionally determines* a column B, denoted $\vec{A} \rightarrow B$, if each \vec{A} value is associated with exactly one B value. This is also called a *functional dependency* (FD). Formally, the dependency holds if in a table T :

$$\forall u, v \in T, u[a_i] = v[a_i], 1 \leq i \leq n \Rightarrow u[B] = v[B]$$

Other types of dependencies are multi-valued dependency (MVD), join dependency (JD) and transitive dependency (TD). \square

Definition 5.2 Extended functional dependency

Definition 5.1 can be extended, such that \vec{A} functionally determines a set of columns $\vec{B} = \{b_1, b_2 \dots b_m\}$, denoted $\vec{A} \rightarrow \vec{B}$, if \vec{A} functionally determines each column in \vec{B} . \square

artists					
artistid	name	username?	country	city	phone_prefix
1	Tudor Dobrila	NULL	Romania	Iasi	+40
2	John Doe	JohnDoe	Romania	Iasi	+40
3	Mary Jones	MaryJones	Romania	Iasi	+40
4	Andrew King	NULL	UK	London	+44
5	Sandy Queen	Sandy	UK	London	+44
6	Monica Green	NULL	US	New York	+35

Table 5.1: The *artists* table, containing redundancy.

Normal forms [Cod70] represent guidelines for database design. They enforce restrictions on the schema and help minimise the redundancy. Several normal forms have been introduced in [Cod70] and [Fag79], such as the first normal form (1NF), second normal form (2NF), third normal form (3NF), Boyce-Codd Normal Form (BCNF), etc.

Example 5.2. We can notice that the example in Table 5.1 is in 2NF, but not in 3NF, because $\{city\} \rightarrow \{country, phone_prefix\}$. \square

Macro Command of the Transformation Pattern

Most of the normal forms involve performing a lossless decomposition of a table *orig_table* in order to remove a dependency. This can be accomplished in the database integration tool using the following macro:

```
normalise_table (⟨⟨orig_table⟩⟩, ⟨⟨new_table⟩⟩, {a1, a2...an}, {b1, b2...bm}),
```

where $\{a_1, a_2 \dots a_n\}$ represents the determinant set, $\{b_1, b_2 \dots b_m\}$ is the dependent set and *new_table* is the name of the table being introduced.

The macro is expanded into the following primitive BAV transformations:

- ① Add the new table *new_table*:

```
add (table:⟨⟨new_table⟩⟩, distinct [ {a1, a2...an} | {x, a1} ← ⟨⟨orig_table, a1⟩⟩; {x, a2} ← ⟨⟨orig_table, a2⟩⟩;
... ; {x, an} ← ⟨⟨orig_table, an⟩⟩ ] )
```
- ② Add the columns in the determinant set to the new table:

```
add (column:⟨⟨new_table, ci, o, t⟩⟩,
      distinct [ { {a1, a2...an}, ai } | {a1, a2...an} ← ⟨⟨new_table⟩⟩ ], 1 ≤ i ≤ n)
```
- ③ Add the Primary Key constraint with the determinant set columns:

```
add (primarykey:⟨⟨new_table_pk,
      new_table, ⟨⟨new_table, c1⟩⟩, ⟨⟨new_table, c1⟩⟩, ..., ⟨⟨new_table, c1⟩⟩⟩⟩)
```
- ④ Add the Foreign Key constraint from the columns in the original table to the newly created columns:

```
add (foreignkey:⟨⟨orig_table_isa_new_table_fk,
      orig_table, ⟨⟨orig_table, a1⟩⟩, ⟨⟨orig_table, a2⟩⟩, ..., ⟨⟨orig_table, an⟩⟩,
      new_table, ⟨⟨new_table, c1⟩⟩, ⟨⟨new_table, c2⟩⟩, ..., ⟨⟨new_table, cn⟩⟩⟩⟩)
```
- ⑤ Add the columns in the dependent set to the new table:

```
add (column:⟨⟨new_table, di, o, t⟩⟩, distinct [ { {a1, a2...an}, bi } | {x, a1} ← ⟨⟨orig_table, a1⟩⟩; {x, a2}
← ⟨⟨orig_table, a2⟩⟩; ... ; {x, an} ← ⟨⟨orig_table, an⟩⟩; {x, bi} ← ⟨⟨orig_table, bi⟩⟩ ] ), 1 ≤ i ≤ m)
```
- ⑥ Delete the columns in the dependent set from the original table:

```
delete (column:⟨⟨orig_table, bi, o, t⟩⟩, [ {x, bi} | {x, a1} ← ⟨⟨orig_table, a1⟩⟩; {x, a2} ← ⟨⟨orig_table, a2⟩⟩;
... ; {x, an} ← ⟨⟨orig_table, an⟩⟩; { {a1, ..., an}, bi } ← ⟨⟨new_table, di⟩⟩ ] ), 1 ≤ i ≤ m)
```

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the macro:

```
reverse_normalise_table (⟨⟨orig_table⟩⟩, ⟨⟨new_table⟩⟩, {a1, a2...an}, {b1, b2...bm})
```

Implementation of the Transformation Pattern

The forward transformation can only be applied to tables. The user is prompted with a window allowing him to define the name of the new table and the determinant and dependent sets by dragging and dropping columns from the original table to the newly created table, as illustrated in Figure 5.2.

After calling `normalise_table (⟨⟨artists⟩⟩, ⟨⟨cities⟩⟩, city, {country, phone_prefix})` on the schema in Figure 5.1, a functional dependency $\{country\} \rightarrow \{phone_prefix\}$ still exists, so the schema is not in 3NF. By calling `normalise_table (⟨⟨cities⟩⟩, ⟨⟨countries⟩⟩, country, phone_prefix)`, this functional dependency is removed. The result of applying these two transformations is shown in Figure 5.3. This is a lossless decomposition, because by joining the three tables using the following IQL query, we get the original table shown in Table 5.1:

```
[ {a, ni, u, ci, co, p} | {a, n} ← ⟨⟨artists, name⟩⟩; {a, u} ← ⟨⟨artists, username⟩⟩; {a, ci} ← ⟨⟨artists, city⟩⟩; {ci, co}
← ⟨⟨cities, country⟩⟩; {co, p} ← ⟨⟨countries, phone_prefix⟩⟩ ]
```

The reverse transformation can be applied to two tables, if there exists a foreign key constraint from one the tables to the other's primary key column(s), and has a similar user interface.

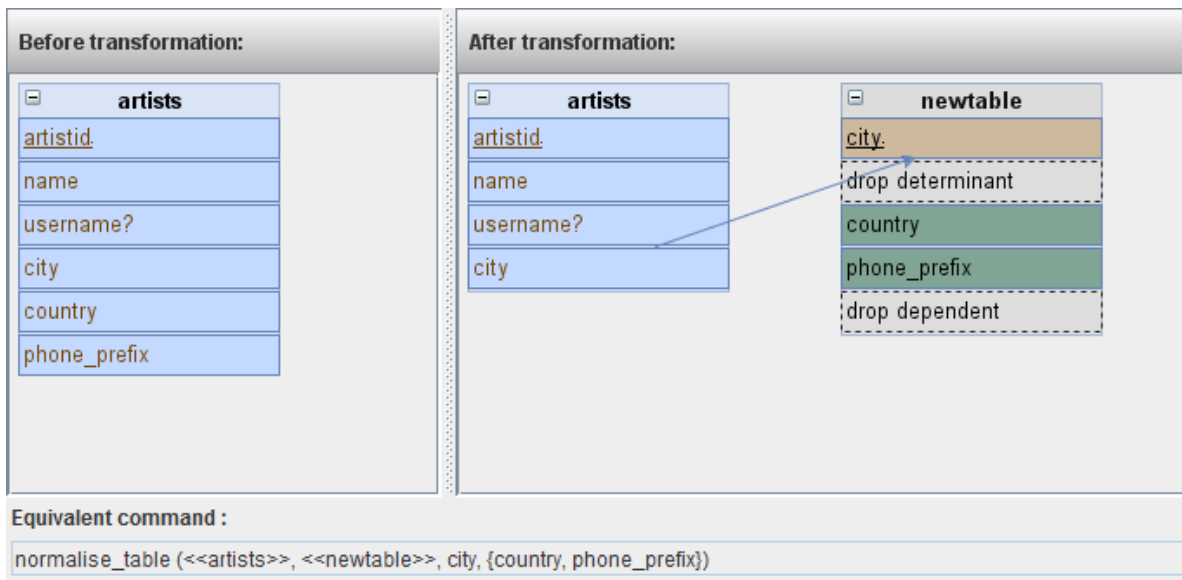


Figure 5.2: User interface for the normalisation pattern.



Figure 5.3: Schema in 3NF after the application of the normalisation pattern.

The Functional Dependency Discovery Tool

In order to assist the user with discovering functional dependencies, a functional dependency discovery tool has been added to the application, which can be accessed from the main menu bar of the application in the "Tools" menu, or by right-clicking any table in SQL diagrams. It uses the approach called TANE presented in [HKPT99] and the source code available at [TAN]. The tool runs on computers that have Perl installed on them. In addition to the usual functional dependencies (see Definition 5.1), TANE also discovers *approximate functional dependencies*.

Definition 5.3 Approximate functional dependency

An *approximate functional dependency* is a functional dependency that almost holds, *i.e.* by removing some rows from the table the functional dependency holds. For instance, the position in a company can be approximately determined by the salary that an employee earns. The number of rows that must be removed from a relation r in order to obtain a functional dependency has been formalised in [HKPT99] as:

$$error(X \rightarrow A) = \min\{|s| \mid s \subseteq r \text{ and } X \rightarrow A \text{ holds in } r \setminus s\} / |r|$$

Given a threshold $0 \leq \epsilon \leq 1$, we say that $X \rightarrow A$ is an approximate functional dependency iff $error(X \rightarrow A)$ is at most ϵ . \square

Example 5.3. Consider the table shown in Table 5.1, where another row has been inserted, in which the *phone_prefix* for Romania is set to +35, instead of +40, as illustrated in Table 5.2.

artists					
<u>artistid</u>	name	username?	country	city	phone_prefix
7	Alexander James	NULL	Romania	Bucharest	+35

Table 5.2: The *artists* table, containing redundancy.

In this case the functional dependency $\{country\} \rightarrow \{phone_prefix\}$ does not hold. But, if we take $\epsilon = 0.2$ and compute that $error(\{country\} \rightarrow \{phone_prefix\}) = 1/7 = 0.14 < \epsilon$, so we get that $\{country\} \rightarrow \{phone_prefix\}$ is an approximate functional dependency. This can be used to determine that the phone prefix of Romania in the last row is incorrect and the situation can be rectified. \square

The worst case time complexity of the algorithm is exponential in the number of the attributes, something that cannot be avoided, as all combinations of attributes must be considered, and linear in the number of tuples contained by the relation. This implies that the tool performs very well for tables with a small number of attributes and a very large number of rows.

5.1.2 Mandatory Column and Total Generalisation Equivalence

Description of the Transformation Pattern

This equivalence exists between two schemas, where in one schema there exists a table t with a mandatory column c , taking one of the values $\{v_1, v_2, \dots, v_n\}$, and in the other schema the column c is replaced by n tables t_1, t_2, \dots, t_n , all of which are children of t , as shown in Figure 5.4.

Definition 5.4 Parent-child relationship

A table t_p is called a *parent* of another table t_c if they have the same primary key column(s) and, in addition, there exists a foreign key constraint from the primary key column(s) of t_c to the primary key column(s) of t_p . In this case t_c is called a *child* of t_p . \square

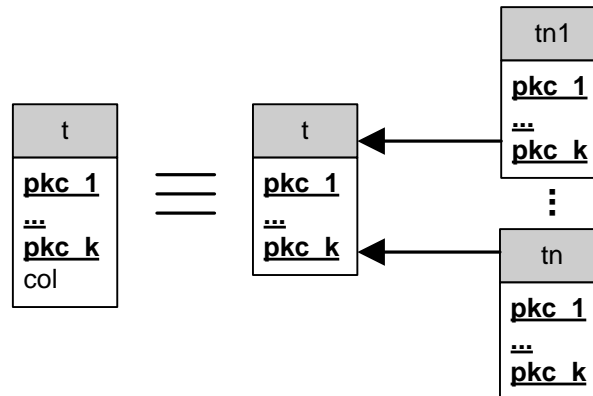


Figure 5.4: Mandatory Column and Total Generalisation Equivalence.

Macro Command of the Transformation Pattern

The forward transformation can be applied using the following macro, with the column col in table $table$ taking the distinct values $\{v_1, v_2, \dots, v_n\}$. This results in the tables $new_table_1, new_table_2, \dots, new_table_n$ being introduced as children of $table$.

```
col_total_generalisation(⟨⟨table, col⟩⟩, ⟨⟨new_table_1⟩⟩, ..., ⟨⟨new_table_n⟩⟩, {val_1, ..., val_n})
```

The macro is expanded into the following BAV transformations:

- ① Add the new child tables:


```
add (table:⟨⟨new_table_i⟩⟩,
      [ {x} | {x, 'v_i'} ← ⟨⟨table, col⟩⟩ ]), 1 ≤ i ≤ n
```

- ② Add the primary key column(s) to the child tables:
 add (column:⟨⟨*new_table_i*, *pkc_j*, *o*, *t*⟩⟩,
 [{ {*pkc₁*, *pkc₂*, ..., *pkc_k*}, *pkc_j* } | {*pkc₁*, *pkc₂*, ..., *pkc_k* } ← ⟨⟨*new_table_i*⟩⟩ }], $1 \leq i \leq n$, $1 \leq j \leq k$
- ③ Add the primary key constraints to the child tables:
 add (primarykey:⟨⟨*new_table_i*, *pk*, *new_table_i*, ⟨⟨*new_table_i*, *pkc₁*⟩⟩, ⟨⟨*new_table_i*, *pkc₂*⟩⟩,
 ..., ⟨⟨*new_table_i*, *pkc_k*⟩⟩⟩⟩), $1 \leq i \leq n$
- ④ Add the foreign key constraints from the child tables to the parent table:
 add (foreignkey:⟨⟨*new_table_i*, *isa_table_fk*,
new_table_i, ⟨⟨*new_table_i*, *pkc₁*⟩⟩, ⟨⟨*new_table_i*, *pkc₂*⟩⟩, ..., ⟨⟨*new_table_i*, *pkc_k*⟩⟩,
table, ⟨⟨*table*, *pkc₁*⟩⟩, ⟨⟨*table*, *pkc₂*⟩⟩, ..., ⟨⟨*table*, *pkc_k*⟩⟩⟩⟩), $1 \leq i \leq n$
- ⑤ Remove the column *col* from the parent table:
 delete (column:⟨⟨*table*, *col*, *o*, *t*⟩⟩,
 [{ {*pkc₁*, *pkc₂*, ..., *pkc_k*}, '*v₁*' } | {*pkc₁*, *pkc₂*, ..., *pkc_k* } ← ⟨⟨*new_table₁*⟩⟩ } ++
 ... ++ [{ {*pkc₁*, *pkc₂*, ..., *pkc_k*}, '*v_n*' } | {*pkc₁*, *pkc₂*, ..., *pkc_k* } ← ⟨⟨*new_table_n*⟩⟩ }])

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the following macro:

```
remove_col_generalisation(⟨⟨table, col⟩⟩, ⟨⟨new_table1⟩⟩, ..., ⟨⟨new_tablen⟩⟩, {val1, ..., valn})
```

Implementation of the Transformation Pattern

The forward transformation can be applied to mandatory columns that take a small number of distinct values. The user is shown a dialog box allowing him to customise the names of the newly created tables.

If we consider the *type* column from the *audio* table shown in Figure 5.1 to take one of the values {*'mp3'*, *'wav'*, *'flac'*} and call `col_total_generalisation` (⟨⟨*audio*, *type*⟩⟩, ⟨⟨*flac*⟩⟩, ⟨⟨*mp3*⟩⟩, ⟨⟨*wav*⟩⟩, {*'flac'*, *'mp3'*, *'wav'*}), three new tables are introduced (*i.e.* *mp3*, *flac* and *wav*), all being children of *audio*, as illustrated in Figure 5.5.

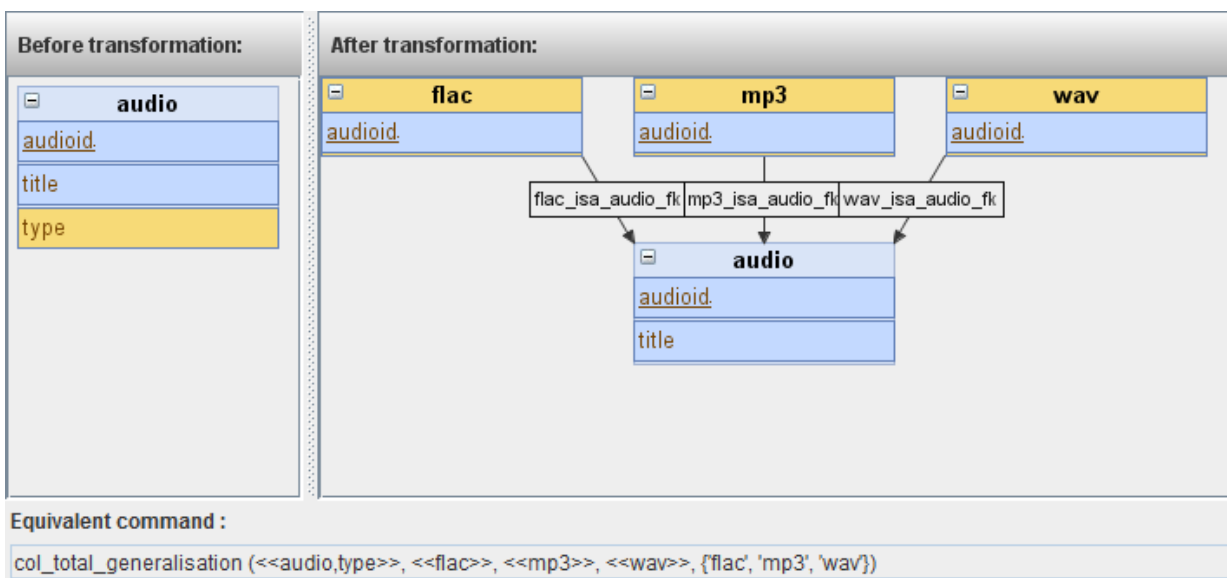


Figure 5.5: User interface for the mandatory column and total generalisation pattern.

The reverse transformation can be applied by selecting n tables and a common parent of the n tables.

5.1.3 Optional Column/Child Table Equivalence

Description of the Transformation Pattern

This equivalence exists between two schemas, where an optional column c belonging to a table t_p is moved to a child table t_c and is marked as mandatory, as shown in Figure 5.6.

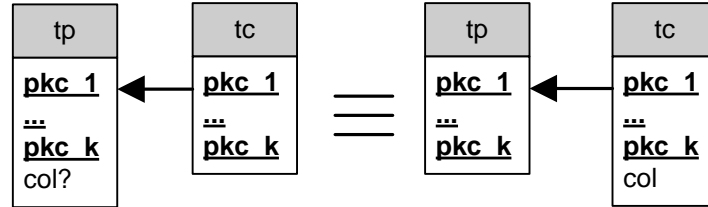


Figure 5.6: Optional Column/Child Table Equivalence.

Macro Command of the Transformation Pattern

The forward transformation can be applied using the following macro, with the optional column col in table $table$ being moved to $child_table$ and marked as mandatory:

```
col_to_subtype ((table, col), (child_table))
```

The macro is expanded into the following BAV transformations. Steps 1-4 are only executed if the child table does not exist.

- ① Add the child table:


```
add (table:(child_table),
      [ {pkc1, pkc2, ..., pkck} | {{pkc1, pkc2, ..., pkck}, col} ← (table, col)])
```
- ② Add the primary key column(s) to the child table:


```
add (column:(child_table, pkci, o, t), (table, pkci))
```
- ③ Add the primary key constraint to the child table:


```
add (primarykey:(child_table_pk,
      child_table, (child_table, pkc1), (child_table, pkc2), ..., (child_table, pkck)))
```
- ④ Add the foreign key constraint from the child table to the parent table:


```
add (foreignkey:(child_table_isa_table_fk,
      child_table, (child_table, pkc1), (child_table, pkc2), ..., (child_table, pkck),
      table, (table, pkc1), (table, pkc2), ..., (table, pkck)))
```
- ⑤ Add the column col to the child table and mark it as notnull:


```
add (column:(child_table, col, notnull, t), (table, col))
```
- ⑥ Remove the optional column col from the parent table:


```
delete (column:(table, col, null, t), (child_table, col))
```

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the macro:

```
subtype_to_col ((child_table, col), (table))
```

Implementation of the Transformation Pattern

The forward transformation can be applied to optional columns. Figure 5.7 shows an application of the pattern for moving the *username* column from the *artists* table to a child table called *members*.

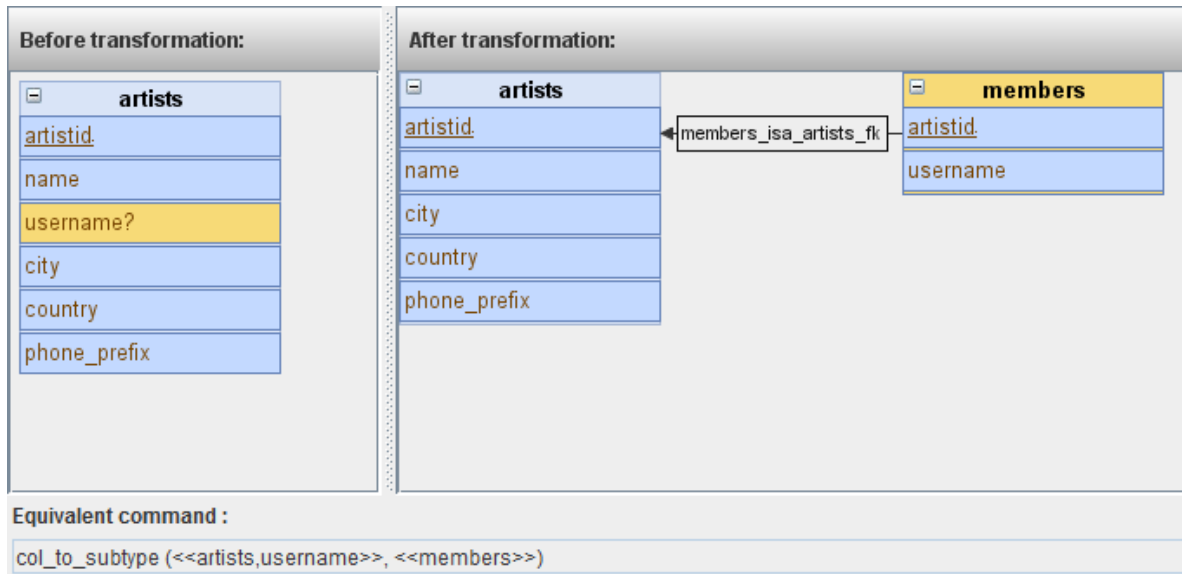


Figure 5.7: User interface for the optional column/child table transformation pattern.

The reverse transformation can be applied to two tables, where one is the parent of the other and, in addition, the child contains a mandatory column that is not part of the primary key constraint.

5.1.4 Column Generalisation Equivalence

Description of the Transformation Pattern

This equivalence exists between two schemas, where one of the schemas contains n tables, all of which are children of another table t and, in addition, the n tables have a common column col . The second schema contains the same tables with the col column existing only in the parent table t . This is illustrated in Figure 5.8.

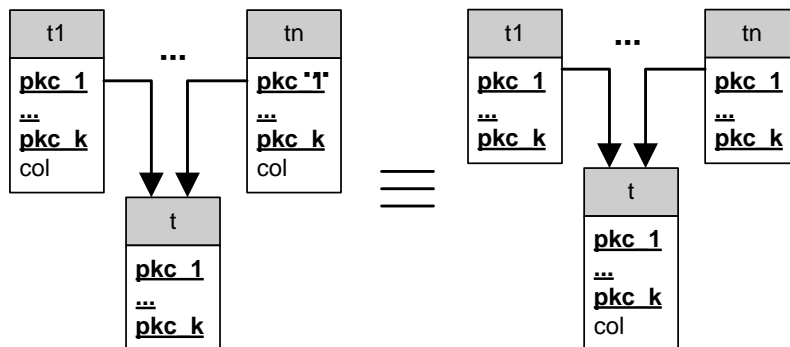


Figure 5.8: Column Generalisation Equivalence.

Macro Command of the Transformation Pattern

The forward transformation can be applied using the following macro, which causes column col to be moved from the child tables $table_1, table_2, \dots, table_n$ to their common parent, $parent_table$:

```
cols_to_supertype(<<parent_table>>, <<table1, col>>, <<table2, col>>, ..., <<tablen, col>>)
```

The macro is expanded into the following primitive BAV transformations:

- ① Add the column to the parent table:

```
add (column:⟨⟨parent_table, col, o, t⟩⟩, ⟨⟨table1, col⟩⟩ ++ ⟨⟨table2, col⟩⟩ ++ ... ++ ⟨⟨tablen, col⟩⟩)
```

- ② Delete the column from the child tables:

```
delete (column:⟨⟨tablei, col, o, t⟩⟩, [ {x,col} | {x} ← ⟨⟨tablei⟩⟩; {x,col} ← ⟨⟨parent_table, col⟩⟩ ]),  
1 ≤ i ≤ n
```

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the macro:

```
col_to_subtypes(⟨⟨parent_table, col⟩⟩, ⟨⟨table1⟩⟩, ⟨⟨table2⟩⟩, ..., ⟨⟨tablen⟩⟩)
```

Implementation of the Transformation Pattern

The forward transformation can be applied by selecting n columns with the same name, col , from n different tables and a common parent of the n tables, where col will be moved.

For instance, in the example schema from Figure 5.1, the *age* column from the *male* and *female* table might be moved to the *artists* table, which is a parent of both *male* and *female*. This is achieved by calling `cols_to_supertype (⟨⟨person⟩⟩, ⟨⟨male,age⟩⟩, ⟨⟨female,age⟩⟩)` and the result is shown in Figure 5.9.

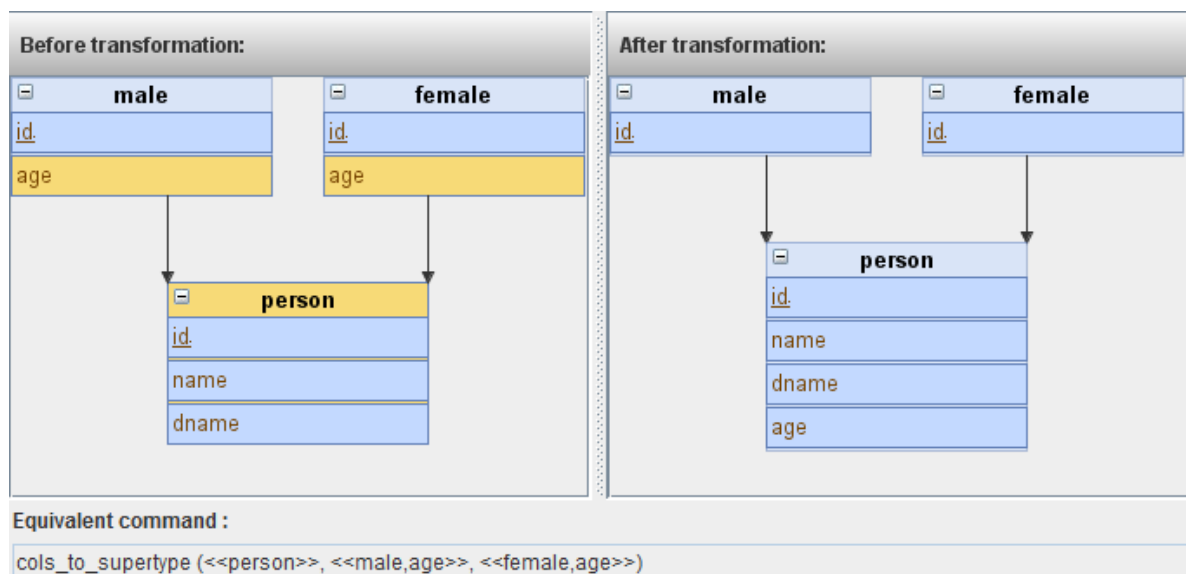


Figure 5.9: User interface for the column generalisation pattern.

The reverse transformation can be applied by selecting n tables, out of which $n - 1$ tables have the other table t_p as their common parent, and a column belonging to t_p to distribute to the n children. This can only be done if the relationship between the $n - 1$ child tables and t_p is of total generalisation, otherwise information is lost as a result of applying the pattern.

5.1.5 Column/Table Equivalence

Description of the Transformation Pattern

This equivalence exists between two schemas, where one of the schemas contains a table t with a column col , while in the other schema column col is represented as a table. In other words, the second schema contains a table t_1 with col as the primary key column and a foreign key constraint from t to t_1 , as shown in Figure 5.10.

The reason for introducing such a pattern is that a column from one schema might have more significance in another schema and be represented as a table.

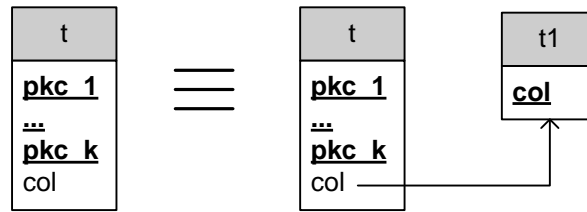


Figure 5.10: Column/Table Equivalence.

Macro Command of the Transformation Pattern

The forward transformation can be applied using the following macro, which causes a table *new_table* to be introduced, with the primary key column *col*, and a foreign key from *table* to *new_table*:

```
col_to_table (⟨⟨table, col⟩⟩,⟨⟨new_table⟩⟩)
```

The macro is expanded into the following BAV transformations:

- ① Create the new table:

```
add (table:⟨⟨new_table⟩⟩, distinct [ {col} | {y, col} ← ⟨⟨table, col⟩⟩])
```
- ② Add the primary key column to the new table:

```
add (column:⟨⟨new_table, col, o, t⟩⟩, distinct [ {col, col} | {y, col} ← ⟨⟨table, col⟩⟩])
```
- ③ Add the primary key constraint to the new table:

```
add (primarykey:⟨⟨new_table_pk, new_table, ⟨⟨new_table, col⟩⟩⟩)
```
- ④ Add the foreign key from the original table to the new table:

```
add (foreignkey:⟨⟨table_to_new_table_fk, table, ⟨⟨table, col⟩⟩, new_table, ⟨⟨new_table, col⟩⟩⟩⟩)
```

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the macro:

```
table_to_col (⟨⟨new_table⟩⟩,⟨⟨table, col⟩⟩)
```

Implementation of the Transformation Pattern

The forward transformation can be applied to any column. For instance, in the schema presented in Figure 5.1, the *country* column from the *artists* table might have more significance in another schema and be represented as a table.

By calling `col_to_table (⟨⟨artists, country⟩⟩, ⟨⟨countries⟩⟩)`, a new table *countries* is introduced. The result is shown in Figure 5.11.

The reverse transformation can be applied to two tables, where one of the tables contains a foreign key to the primary key column of the other table.

5.1.6 Introduction of Total Generalisation Equivalence

Description of the Transformation Pattern

This equivalence exists between two schemas, where one of the schemas contains n tables t_1, t_2, \dots, t_n which have the same primary key column(s) and the other schema contains a table tp that is a parent of the n tables, as illustrated in Figure 5.12.

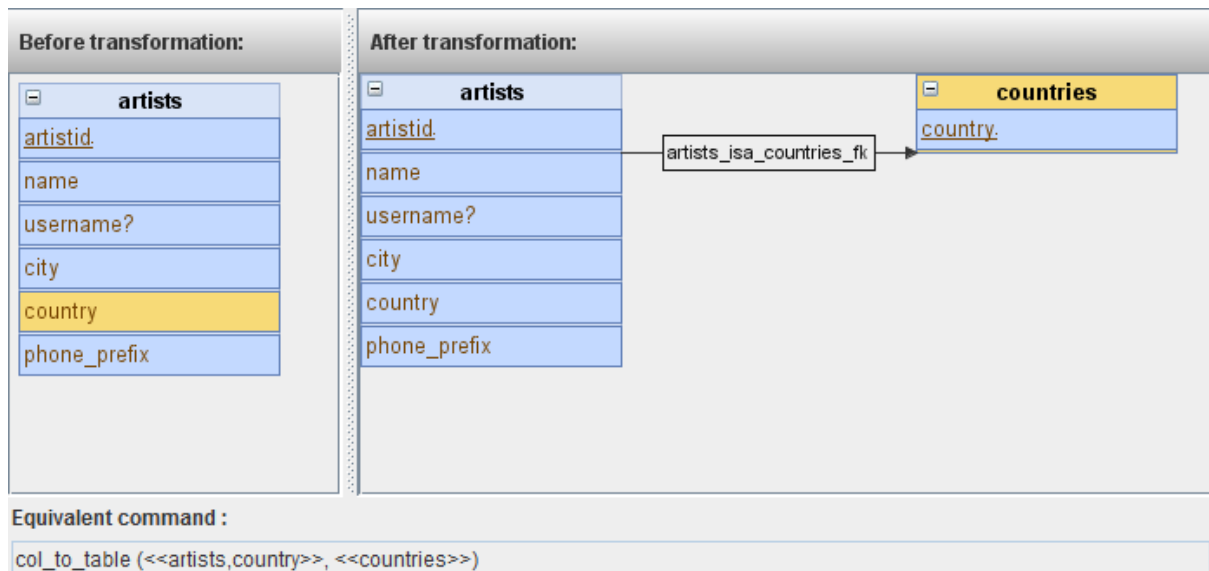


Figure 5.11: User interface for the column/table equivalence

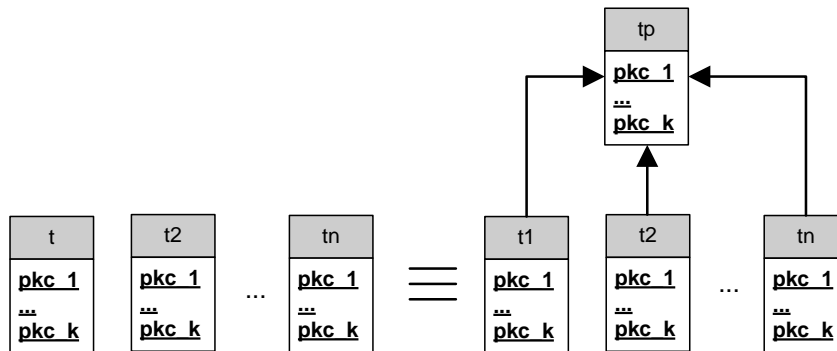


Figure 5.12: Table Generalisation Equivalence.

Macro Command of the Transformation Pattern

The forward transformation can be applied using the following macro, which causes *parent* to be introduced as the parent of t_1, t_2, \dots, t_n .

```
table_total_generalisation (<<parent>>, <<t1>>, <<t2>>, ..., <<tn>>)
```

The macro is expanded into the following BAV transformations:

- ① Create the parent table:

```
add (table:<<parent>>, <<t1>> ++ <<t2>> ++ ... ++ <<tn>>)
```
- ② Add the primary key column(s) to the parent table:

```
add (column:<<parent, pkci, o, t>>,
  [ {x, pkci} | {x, pkci} ← <<t1, pkc1>> ] ++
  [ {x, pkci} | {x, pkci} ← <<t2, pkci>> ] ++
  ... ++ [ {x, pkci} | {x, pkci} ← <<tn, pkci>> ] ), 1 ≤ i ≤ k
```
- ③ Add the primary key constraint to the parent table:

```
add (primarykey:<<parentpk,
  parent, <<parent, pkc1>>, <<parent, pkc2>>, ..., <<parent, pkcn>>))
```
- ④ Add the foreign key constraints from the child tables to the parent table:

```
add (foreignkey:<<ti-isa-parent-fk,
```

$$t_i, \langle \langle t_i, pk_{c_1} \rangle \rangle, \langle \langle t_i, pk_{c_2} \rangle \rangle, \dots, \langle \langle t_i, pk_{c_n} \rangle \rangle, \\ parent, \langle \langle parent, pk_{c_1} \rangle \rangle, \langle \langle parent, pk_{c_2} \rangle \rangle, \dots, \langle \langle parent, pk_{c_n} \rangle \rangle \rangle, 1 \leq i \leq n$$

The associated reverse transformation pattern is derived automatically as shown at the beginning of Section 5.1 and is given by the macro:

```
remove_table_generalisation (⟨⟨parent⟩⟩, ⟨⟨t1⟩⟩, ⟨⟨t2⟩⟩, ..., ⟨⟨tn⟩⟩)
```

Implementation of the Transformation Pattern

The forward transformation can be applied to any n tables that have the same primary key column(s). For instance, if we rename both the *maleid* column of the *male* table and the *femaleid* column of the *female* table to *id* and call `table_total_generalisation (⟨⟨human⟩⟩, ⟨⟨male⟩⟩, ⟨⟨female⟩⟩)`, the table *human* is introduced as a parent of *male* and *female*, as shown in Figure 5.13.

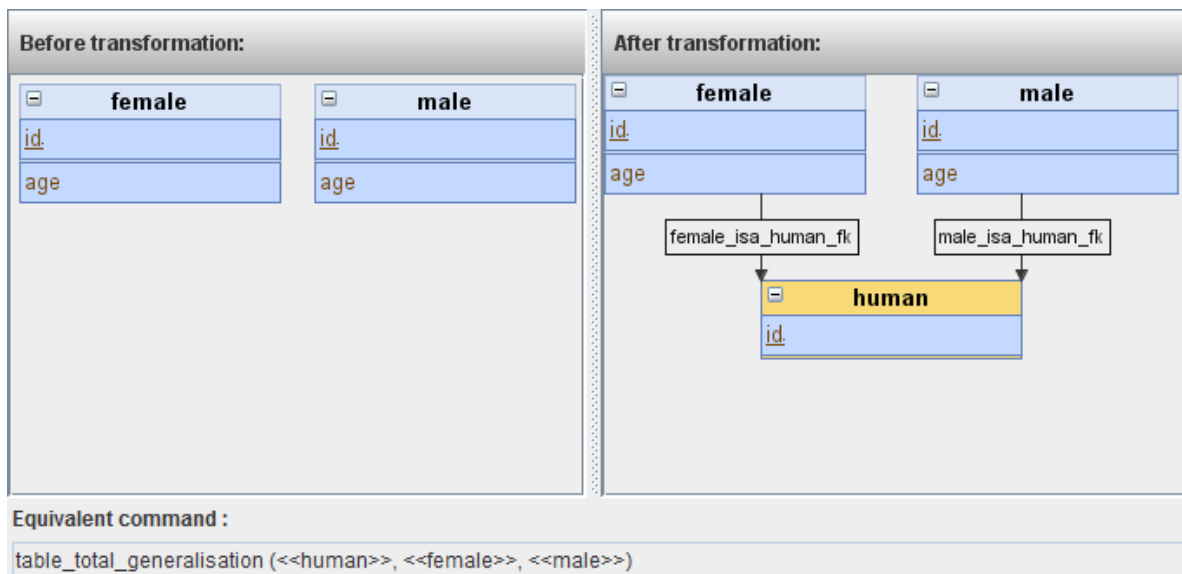


Figure 5.13: User interface for the table generalisation equivalence

The reverse transformation can be applied by selecting n tables and a common parent of the n tables.

5.2 Schema Merging

5.2.1 Addition of Subset

Description of the Transformation Pattern

This transformation pattern maps an input schema containing two unrelated tables t_p and t_c into another schema where t_p is the parent of t_c . This is done by creating a foreign key constraint from the primary key column(s) of t_c to the primary key column(s) of t_p , as illustrated in Figure 5.14. This transformation can be applied only if $t_c \subset t_p$.

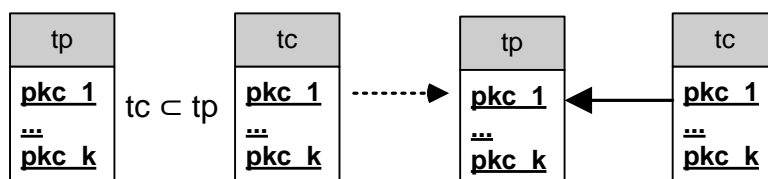


Figure 5.14: Addition of Subset Transformation Pattern

Macro Command of the Transformation Pattern

The following macro can be used to apply this transformation:

```
add_subset (<<parent>>, <<child>>)
```

The expansion of the macro contains only one primitive BAV transformation:

- ① add (foreignkey:<<child_isa_parent_fk>>,

child,<<child,pkc₁>>,<<child,pkc₂>>,...,<<child,pkc_n>>,<>

parent,<<parent,pkc₁>>,<<parent,pkc₂>>,...,<<parent,pkc_n>>))

Implementation of the Transformation Pattern

The transformation pattern can be applied to two tables that have the same primary key column(s). In order to define the parent-child relationship, the user has to drag an arrow from the child table to the parent table.

In Figure 5.1, if we rename the column *bandid* from the table *band* to *artistid* by executing the command `rename (column:<<band,bandid>>, column:<<band,artistid>>)`, we can introduce a parent-child relationship by dragging an arrow from *band* to *artists*, as shown in Figure 5.15. This can also be achieved using the command `add_subset(<<artists>>, <<band>>)`.

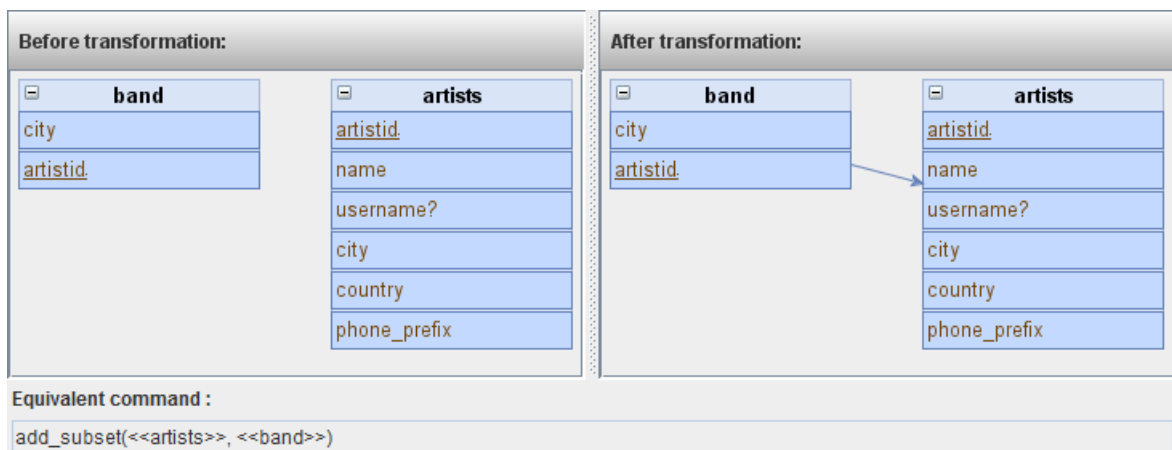


Figure 5.15: User interface for the addition of subset transformation.

5.2.2 Addition of Union

This transformation pattern maps a schema containing n unrelated tables that have the same primary key column(s) to an output schema containing a new table that is a parent (also called the union) of the original tables. This pattern is identical to the forward transformation of the *Introduction of total generalisation equivalence* presented in Section 5.1.6.

5.2.3 Addition of Intersection

Description of the Transformation Pattern

This transformation pattern maps a schema containing n tables that have the same primary key column(s) to an output schema containing a new table that is a child (also called an intersection) of the original tables, as illustrated in Figure 5.16.

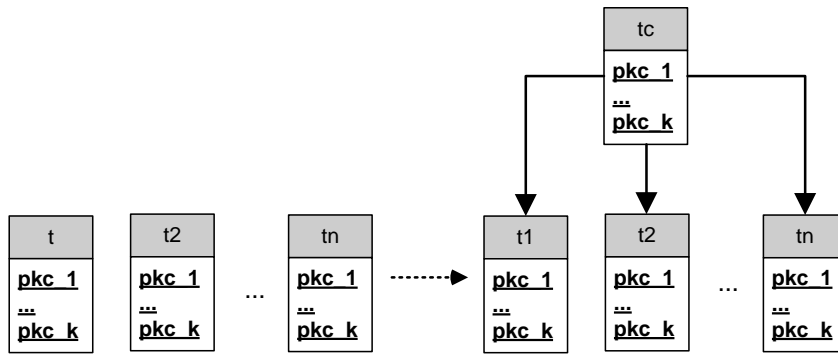


Figure 5.16: Addition of Intersection Transformation Pattern

Macro Command of the Transformation Pattern

The following macro can be used to apply this transformation, where *new_table* is introduced as a child of tables $table_1, table_2, \dots, table_n$:

```
add.intersection (⟨⟨new_table⟩⟩, ⟨⟨table1⟩⟩, ⟨⟨table2⟩⟩, ..., ⟨⟨tablen⟩⟩)
```

The macro is expanded into the following BAV transformations:

- ① Create the child table:


```
add (table:⟨⟨new_table⟩⟩,
      ⟨⟨table1⟩⟩ intersect ⟨⟨table2⟩⟩ intersect ... intersect ⟨⟨tablen⟩⟩ )
```
- ② Add the primary key column(s) to the child table:


```
add (column:⟨⟨new_table, pkci, o, t⟩⟩,
      ⟨⟨table1, pkci⟩⟩ intersect ⟨⟨table2, pkci⟩⟩ intersect ... ⟨⟨tablen, pkci⟩⟩), 1 ≤ i ≤ k
```
- ③ Add the primary key constraint to the child table:


```
add (primarykey:⟨⟨new_table_pk,
      new_table, ⟨⟨new_table, pkc1⟩⟩, ⟨⟨new_table, pkc2⟩⟩, ..., ⟨⟨new_table, pkck⟩⟩⟩)
```
- ④ Add the foreign key constraint from the child table to each of the parent tables:


```
add (foreignkey:⟨⟨new_table_isa_tablei,
      new_table, ⟨⟨new_table, pkc1⟩⟩, ⟨⟨new_table, pkc2⟩⟩, ..., ⟨⟨new_table, pkck⟩⟩,
      tablei, ⟨⟨tablei, pkc1⟩⟩, ⟨⟨tablei, pkc2⟩⟩, ..., ⟨⟨tablei, pkck⟩⟩⟩), 1 ≤ i ≤ n
```

Implementation of the Transformation Pattern

The transformation can be applied to n tables that have the same primary key column(s). For instance, in the example schema from Figure 5.1, we could execute `add_intersection (⟨⟨audio_cds⟩⟩, ⟨⟨cds⟩⟩, ⟨⟨audio⟩⟩)` in order to create a new table *audio_cds*, holding information about audio CDs, as shown in Figure 5.17.

5.2.4 Addition of Foreign Key

Description of the Transformation Pattern

This transformation maps an input schema to an output schema that contains an additional foreign key constraint between two tables, which in the ER metamodel is called a one-to-many relationship.

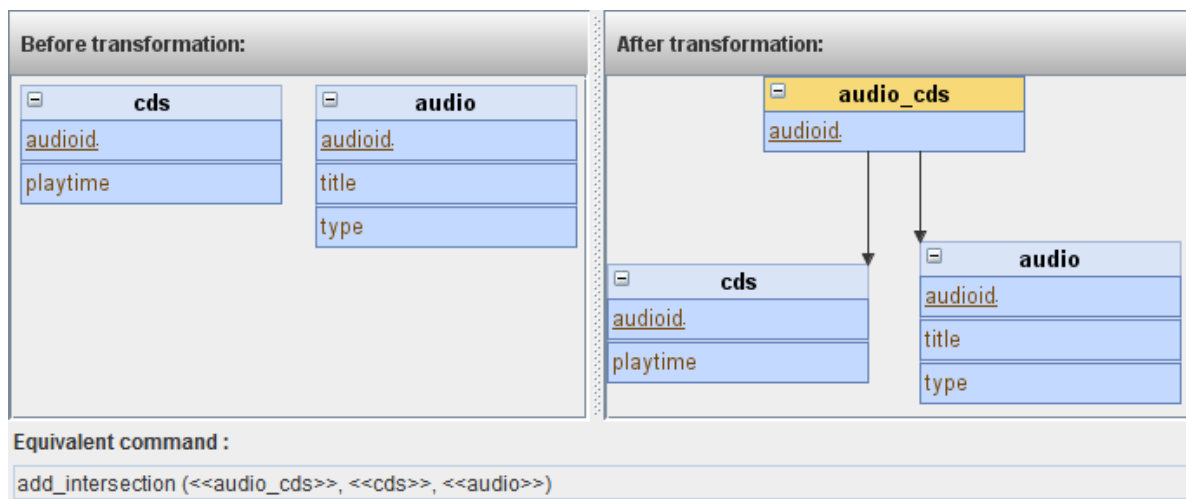


Figure 5.17: User interface for the addition of intersection transformation.

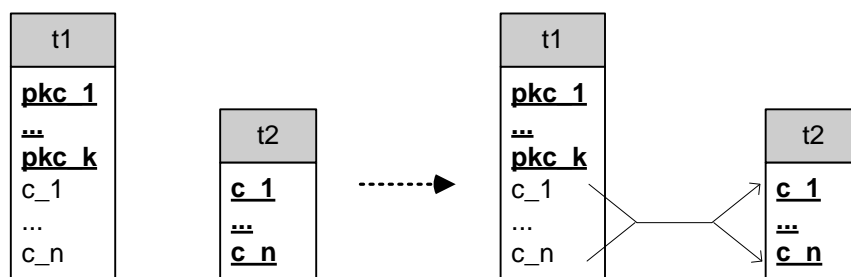


Figure 5.18: Addition of Foreign Key Pattern

Macro Command of the Transformation Pattern

The following macro can be used to apply this transformation, where fk_name is the name of the newly introduced foreign key constraint, t_1-t_2-fk . $\langle\langle t_1, col_1 \rangle\rangle, \dots, \langle\langle t_1, col_n \rangle\rangle$ represent the columns in the source table t_1 belonging to the foreign key and $\langle\langle t_2, col_1 \rangle\rangle, \dots, \langle\langle t_2, col_n \rangle\rangle$ represent the columns in the target table belonging to the foreign key.

```
add_otm_rel (fk_name, << t1, col1 >>, ..., << t1, coln >>, << t2, col1 >>, ..., << t2, coln >>)
```

The macro is expanded into the following BAV transformations:

- ① Create the columns in the source table, only if they did not previously exist:


```
extend (column:<< t1, coli, o, t >>, Range Void Any), 1 ≤ i ≤ n
```
- ② Add the foreign key constraint from t_1 to t_2 :


```
add (foreignkey:<< fk_name, t1, << t1, col1 >>, ..., << t1, coln >>, t2, << t2, col1 >>, ..., << t2, coln >> >>>)
```

Implementation of the Transformation Pattern

The transformation can be applied to any two tables. The user defines the foreign key relationship by dragging arrows from the columns in the source table to the primary key column(s) in the target table.

For instance, consider the example from Figure 5.3, after the application of the normalisation pattern. A foreign key from the *city* column of the *band* table to the *city* column from the *cities* table can be introduced by executing the command `add_otm_rel (<<band,city>>, <<cities,city>>)`, as shown in Figure 5.19.



Figure 5.19: User interface for the addition of foreign key transformation.

5.2.5 Addition of Many-To-Many Table

Description of the Transformation Pattern

This transformation maps an input schema to an output schema where a new table is introduced to represent the many-to-many relationship between two tables. The new table contains the primary key columns of both tables and two foreign key constraints from the new table to each of the two tables are added, as shown in Figure 5.20.

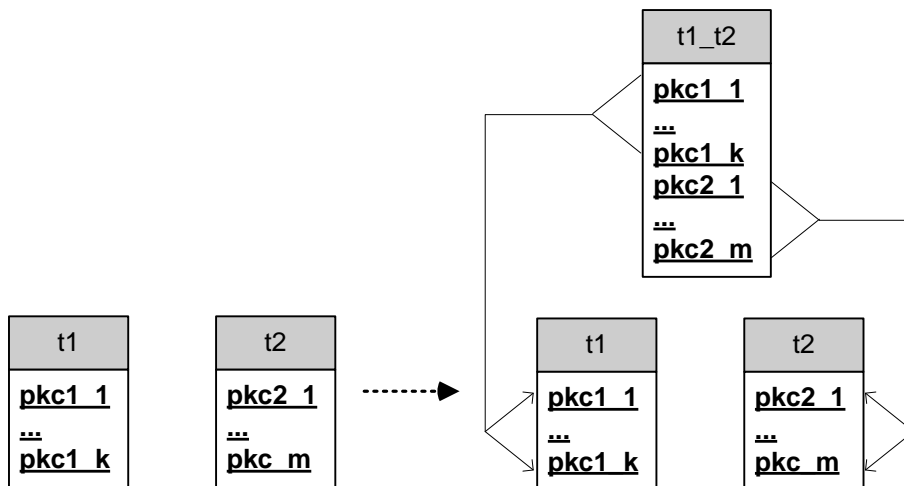


Figure 5.20: Addition of Many-To-Many Table Pattern

Macro Command of the Transformation Pattern

The following macro can be used to apply this pattern, causing a new table *new_table* to be introduced:

```
add_mtm_rel (<<t1>>, <<t2>>, <<new_table>>)
```

The macro is expanded into the following BAV transformations:

- ① Create the new table:


```
extend (table:<<new_table>>, Range Void Any)
```
- ② Add the primary key column(s) from t_1 to the new table:


```
extend (column:<<new_table, pkc1i, o, t>>, Range Void Any), 1 ≤ i ≤ k
```
- ③ Add the primary key column(s) from t_2 to the new table:


```
extend (column:<<new_table, pkc2i, o, t>>, Range Void Any), 1 ≤ i ≤ m
```
- ④ Add the primary key constraint to the new table:


```
add (primarykey:<<new_table_pk, new_table, <<new_table, pkc11>>, ..., <<new_table, pkc1k>>, <<new_table, pkc21>>, ..., <<new_table, pkc2m>>>>)
```

- ⑤ Add the foreign key constraint from the new table to the primary key column(s) of t_1 :
`add (foreignkey:⟨⟨new_table_t1_fk,new_table,⟨⟨new_table,pkc1_1⟩⟩, ..., ⟨⟨new_table,pkc1_k⟩⟩, t1, ⟨⟨t1,pkc1_1⟩⟩, ..., ⟨⟨t1,pkc1_k⟩⟩⟩⟩)`
- ⑥ Add the foreign key constraint from the new table to the primary key column(s) of t_2 :
`add (foreignkey:⟨⟨new_table_t2_fk,new_table,⟨⟨new_table,pkc2_1⟩⟩, ..., ⟨⟨new_table,pkc2_m⟩⟩, t2, ⟨⟨t2,pkc2_1⟩⟩, ..., ⟨⟨t2,pkc1_m⟩⟩⟩⟩)`

If there are two primary key columns with the same name, one belonging to t_1 and one to t_2 , their name must be changed to something unique before adding them to the new table. This can be done by prefixing the name of the column with the name of the parent table.

Implementation of the Transformation Pattern

The transformation can be applied to any two tables. In order to introduce a many-to-many table between the *audio* and *band* tables in Figure 5.1, such that every audio item belongs to one or more bands and every band can publish one or more audio items, the command `add_mtm_rel (⟨⟨band⟩⟩, ⟨⟨audio⟩⟩, ⟨⟨band_audio⟩⟩)` has to be executed, as illustrated in Figure 5.21.

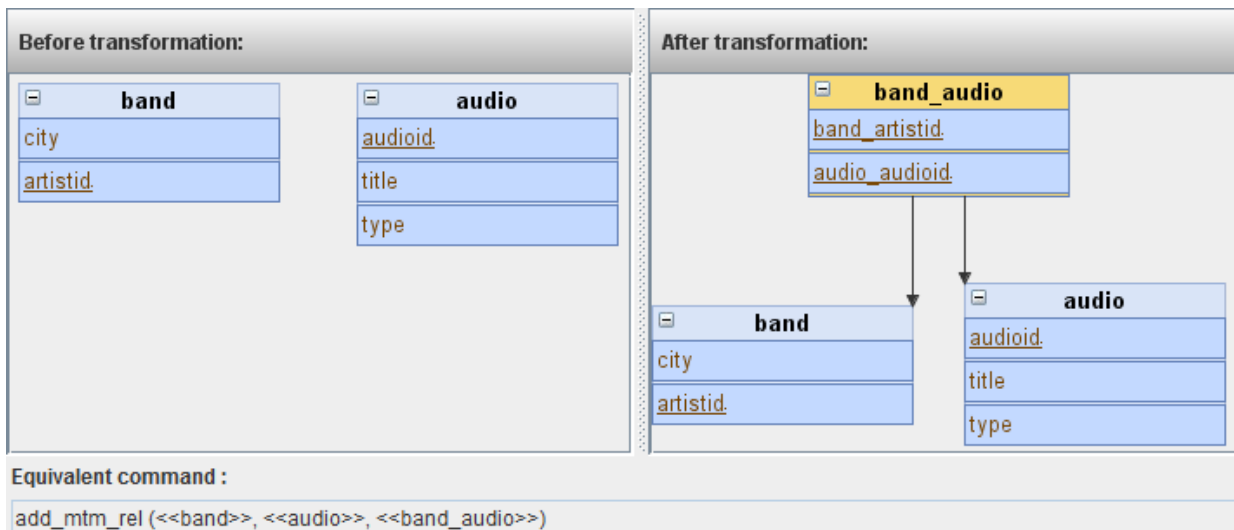


Figure 5.21: User interface for the addition of many-to-many table transformation.

5.3 Schema Improvement

5.3.1 Redundant Column Removal

Description of the Transformation Pattern

The transformation shown in Figure 5.22 maps an input schema containing two tables t_p and t_c , with t_p a parent of t_c , where both t_p and t_c contain a column named col , to an output schema, where the redundant column col is removed from t_c .

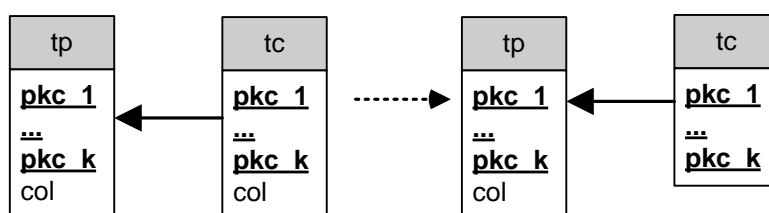


Figure 5.22: Redundant column removal transformation

Macro Command of the Transformation Pattern

The macro associated with this transformations is:

```
redundant_col_removal (<< $t_p$ , col>>, << $t_c$ , col>>)
```

The expansion of the macro contains only one primitive BAV transformation:

- ① Remove the column col from the child table:
`delete(column:<< t_c , col, o, t>>, [{ x , y } | { x } ← << t_c >>; { x , y } ← << t_p , col>>])`

Implementation of the Transformation Pattern

The transformation can be applied by selecting the columns with the same name from the parent and child tables.

If we consider the result of the transformation presented in Figure 5.15, we can notice that the *city* column from the *band* table is redundant, as it exists in one of its parents, the *artists* table. It can be removed by calling `redundant_col_removal (<<artists,city>>, <<band,city>>)`. The result of this transformation is shown in Figure 5.23.

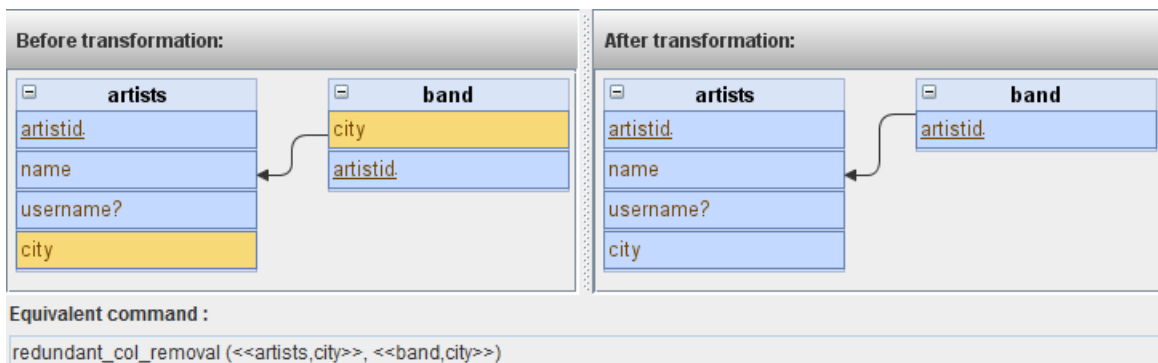


Figure 5.23: User interface for the redundant column removal transformation

5.3.2 Optional Column To Child Table

This transformation is identical to the forward transformation of the *Optional Column/Child Table Equivalence*, presented in Section 5.1.3.

5.3.3 Column Generalisation

This transformation is identical to the forward transformation of the *Column Generalisation Equivalence*, presented in Section 5.1.4.

5.3.4 Redundant Foreign Key Removal

Description of the Transformation Pattern

This transformation maps an input schema containing a foreign key from a child table t_c , which also appears in a parent of t_c , to an output schema in which the redundant foreign key constraint is removed. Two cases exist for this transformation, as shown in Figures 5.24 and 5.25.

Macro Command of the Transformation Pattern

The following macro can be used to apply this transformation, where p_1 is the parent of t_1 and p_2 the parent of t_2 and a foreign key exists between p_1 and p_2 . In case (a) $t_2 \neq p_2$ and in case (b) $t_2 = p_2$.

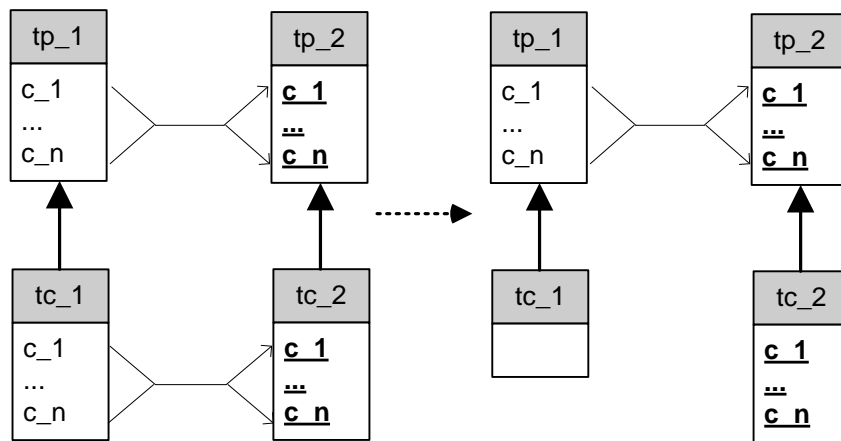


Figure 5.24: Redundant Foreign Key Removal (a)

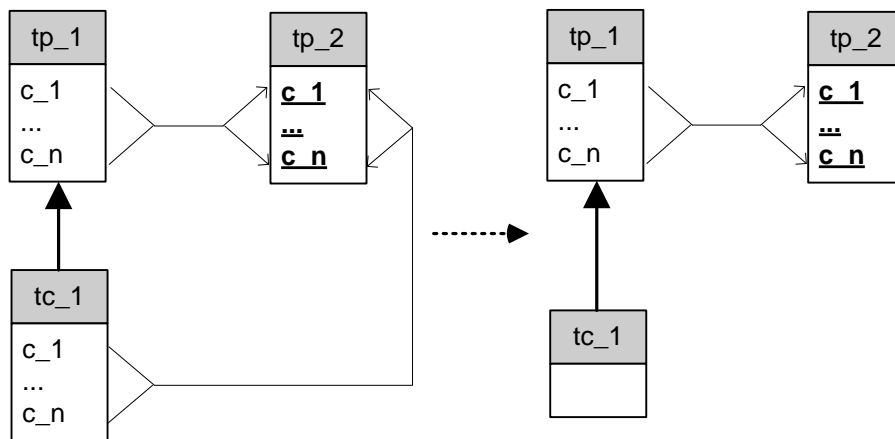


Figure 5.25: Redundant Foreign Key Removal (b)

redundant_fk_removal (

$\langle\langle fk_name1, t_1, \langle\langle t_1, c_1 \rangle\rangle, \dots, \langle\langle t_1, c_n \rangle\rangle, t_2, \langle\langle t_2, c_1 \rangle\rangle, \dots, \langle\langle t_2, c_n \rangle\rangle \rangle\rangle,$
 $\langle\langle fk_name2, p_1, \langle\langle p_1, c_1 \rangle\rangle, \dots, \langle\langle p_1, c_n \rangle\rangle, p_2, \langle\langle p_2, c_1 \rangle\rangle, \dots, \langle\langle p_2, c_n \rangle\rangle \rangle\rangle)$

The macro is expanded into the following BAV transformations:

- ① Delete the redundant foreign key constraint:

delete (foreignkey: $\langle\langle fk_name1, t_1, \langle\langle t_1, c_1 \rangle\rangle, \dots, \langle\langle t_1, c_n \rangle\rangle, t_2, \langle\langle t_2, c_1 \rangle\rangle, \dots, \langle\langle t_2, c_n \rangle\rangle \rangle\rangle)$)

- ② Delete the redundant columns from t_2 , that were part of the redundant foreign key constraint:

delete (column: $\langle\langle t_1, c_i, o, t \rangle\rangle,$
 $[\{x, y\} \mid \{x\} \leftarrow \langle\langle t_1 \rangle\rangle; \{x, y\} \leftarrow \langle\langle p_1, c_i \rangle\rangle; \{z, y\} \leftarrow \langle\langle t_2, c_i \rangle\rangle], 1 \leq i \leq n$)

Implementation of the Transformation Pattern

The transformation can be applied by selecting a redundant foreign key constraint. If we apply the transformations shown in Figures 5.15 and 5.19, we notice that the foreign key from *band* to *cities* is redundant, as another foreign key exists from one of *band*'s parents, *artists*, to *cities*. This transformation is shown in Figure 5.26 and is equivalent to calling the command:

redundant_fk_removal($\langle\langle band_cities_fk, band, \langle\langle band, city \rangle\rangle, cities, \langle\langle cities, city \rangle\rangle \rangle\rangle,$
 $\langle\langle artists_isa_cities_fk, artists, \langle\langle artists, city \rangle\rangle, cities, \langle\langle cities, city \rangle\rangle \rangle\rangle).$

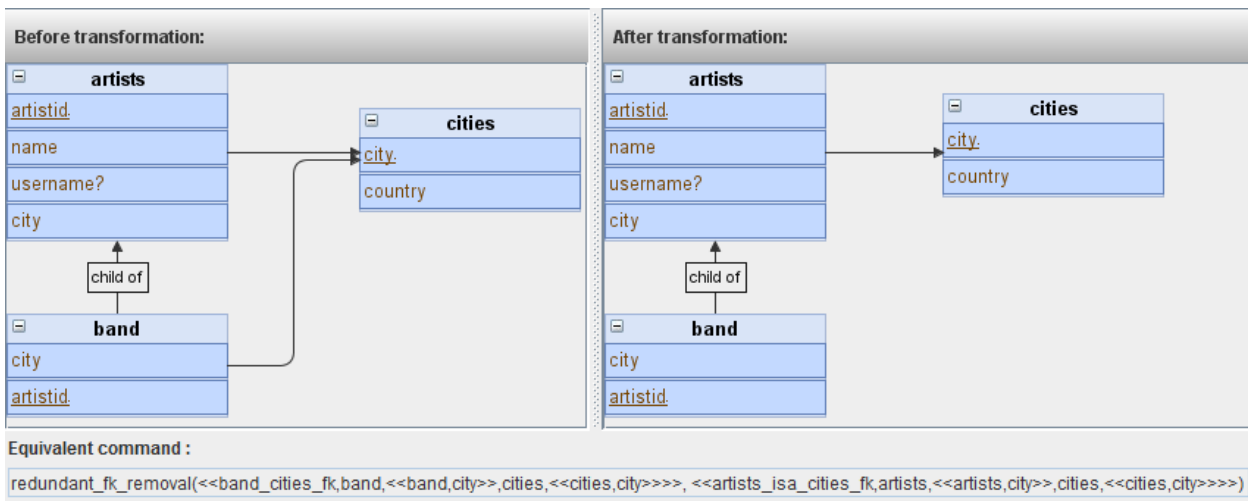


Figure 5.26: User interface for the redundant foreign key removal transformation.

5.4 Summary

A total of 18 transformation patterns have been implemented in the *Interactive Database Integration Tool*: 12 for schema conforming (*i.e.* 6 equivalences), 4 for schema merging and 2 for schema improvement. Some of these patterns can be used in more than one step, such as the *Addition of union* forward transformation, presented in Section 5.1.6, which can be applied during both the conforming and merging steps.

Each of these transformation has an equivalent macro associated with it, which can be called from the tool, and a user interface which guides the user in the application of the pattern.

Chapter 6

Pattern Discovery in BAV Transactions

Chapter 5 presented some of the most common transformation patterns used in database integration. While they are useful in a large number of situations, they do not represent by any means the complete set of possible transformations.

To assist the user in the integration process, a method for the dynamic discovery of transformation patterns from BAV transactions is presented here, something that we are not aware to have been attempted before. In this chapter, BAV transactions are denoted by T and represent sequences of primitive BAV transformations that are either all executed together or they are not executed at all, *i.e.* transactions are *atomic*.

Example 6.1. The following BAV transaction is obtained by expanding the macro `normalise_table` (`<<artists>>`, `<<cities>>`, `city`, `{country, phone_prefix}`) introduced in Section 5.1.1:

- ① `add (table:<<cities>>, distinct [{city} | {x,city} ← <<artists,city>>])`
- ② `add (column:<<cities,city,notnull,text>>,distinct [{city, city} | {city} ← <<artists>>])`
- ③ `add (primarykey:<<cities_pk,cities,<<cities,city>>>>)`
- ④ `add (foreignkey:<<artists_isa_cities_fk,artists,<<artists,city>>,cities,<<cities,city>>>>)`
- ⑤ `add (column:<<cities,country,notnull,text>>,distinct [{city,country} | {x,city} ← <<artists,city>>; {x,country} ← <<artists,country>>])`
- ⑥ `add (column:<<cities,phone_prefix,notnull,text>>,distinct [{city,phone_prefix} | {x,city} ← <<artists,city>>; {x,phone_prefix} ← <<artists,phone_prefix>>])`
- ⑦ `delete (column:<<artists,country,notnull,text>>,[{x,country} | {x,city} ← <<artists,city>>; {city,country} ← <<cities,country>>])`
- ⑧ `delete (column:<<artists,phone_prefix,notnull,text>>,[{x,phone_prefix} | {x,city} ← <<artists,city>>; {city,phone_prefix} ← <<cities,phone_prefix>>])`

□

The method that we propose here is independent on the metamodel used and is related to the field of *pattern discovery*, something that has been studied intensively in the last decades with significant results in the fields of bioinformatics [Rea98] and web mining [CMS97].

This chapter is structured as follows. Section 6.1 outlines the pattern discovery algorithm, Section 6.2 presents two algorithms for graph isomorphism that have been implemented and evaluated

in the context of our method, Section 6.3 outlines a graph hashing algorithm, Section 6.4 focusses on the performance evaluation of the method and Section 6.5 specifies how the method has been implemented and integrated in the *Interactive Database Integration Tool*.

6.1 Overview of the Method

The goal of the method that we propose is to extract patterns from a history of BAV transactions, which we will denote as H , using techniques employed in parallel task scheduling. For simplicity, we are dealing with transactions that only contain primitive BAV transformations. Extending the method to handle transformation macros as well is trivial, by replacing each macro in a transaction with its expansion.

In order to be able to extract new patterns, a unique representation of a BAV transaction needs to be introduced. From Example 6.1, it is obvious that certain operations could be executed in parallel, *i.e.* the order of their execution does not affect the final result. For instance, operations 5 could be executed before operation 4. This is because the execution of operation 5 *does not depend* on the execution of operation 4.

In Chapter 3 of [MSM04], a generic process for designing a concurrent system is introduced. Each step of the process is represented by a pattern, but only the following two patterns present interest in our method:

- *The task decomposition pattern* describes how the problem is decomposed into tasks that can run in parallel.
- *The group tasks pattern* is used to group similar tasks.

The task decomposition pattern involves finding the proper representation for a BAV transaction. The "depends on" relationship presented above can be best described using dependency graphs, which can be used to identify all transformations can run in parallel.

Definition 6.1 Dependency relationship

A *dependency relationship* $R = V \times V$ is a transitive relationship, which specifies that if $(a, b) \in R$, then a depends on b . In other words, a cannot be processed until b has been processed. The transitivity property states that if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$. \square

Definition 6.2 Dependency graph

A *dependency graph* is a directed graph used to model the dependencies between several objects. Formally, it is denoted as $G = (V, E)$, with V representing the set of objects analysed (*i.e.* the vertices) and $E \subseteq V \times V$ the set of edges, with $E \subseteq E^+$, where E^+ denotes the transitive closure of E .

The transitive closure E^+ of a binary relationship E is the transitive relationship that contains E and E^+ is minimal. \square

Dependency graphs have been widely used in compiler technology, fault management and program analysis. They map well to BAV transactions, where each individual instruction modifies only one construct of a schema. They can be used as good descriptors of such a sequence of primitive instructions.

Given a BAV transaction, the dependency graph is constructed by first parsing each primitive BAV transformation. For each transformation, the type, the schema object that is modified and

the set of dependent schema objects are extracted. For instance, instruction 4 in Example 6.1 is used to *add a foreign key*, it modifies the schema object named *artists_isa_cities_fk* and depends on the schema objects $\langle\langle artists \rangle\rangle$, $\langle\langle artists, city \rangle\rangle$, $\langle\langle cities \rangle\rangle$ and $\langle\langle cities, city \rangle\rangle$.

After the parsing phase, a vertex is added to the graph for each transformation. For each dependent schema object of the current transformation, an edge is added from the current vertex to the vertex associated with the last transformation that modified the dependent schema object, if one exists. For instance, an edge will be added from the vertex that represents transformation 4 in Example 6.1 to the node that represents transformation 2, where schema object $\langle\langle cities, city \rangle\rangle$ has last been modified. This is done to represent the fact that transformation 4 can only be executed after transformation 2.

A primitive BAV transformation can only depend on one of the previous transformations, so the final dependency graph will be *acyclic*, *i.e.* there will be no two transformations that depend on each other.

At this stage, the dependency graph in Figure 6.1 is obtained for the transaction in Example 6.1, where each vertex is labelled with the index of the transaction and the action performed.

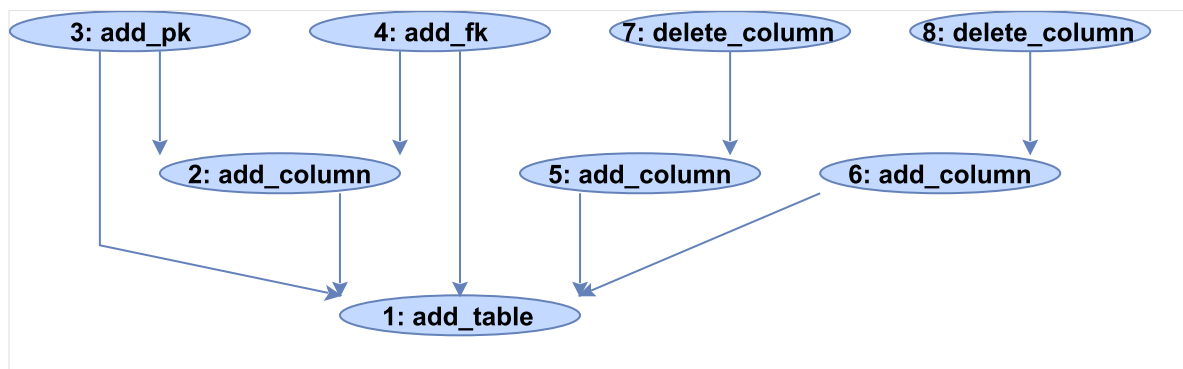


Figure 6.1: Dependency graph of the transaction in Example 6.1.

The next step is the application of the group tasks pattern. Generally speaking, this is done by performing an analysis on the dependency graph and grouping tasks that share the same *constraints*. In our method, only one type of constraint exists, *viz* the set of schema objects that a transformation depends on. For instance, we can see in Figure 6.1 that transformations 3 and 4 share the same dependency set $\{1, 2\}$, so they can be grouped. Grouping is an iterative process, which stops when there are no more vertices that can be grouped. After this step, the graph in Figure 6.2 is obtained.

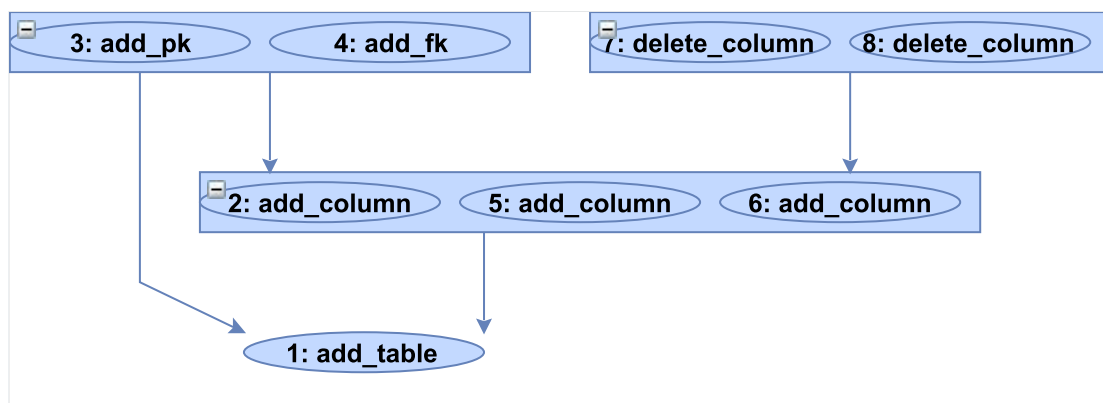


Figure 6.2: Grouped dependency graph of the transaction in Example 6.1.

It can be noticed from Figure 6.2 that the dependency graphs can contain two types of nodes: *atomic*, used to represent only one primitive BAV transformation, and *group*, used to represent

several independent transformations.

In general, we are not interested in the number of times a transformation (e.g. *add_column*) appears in a group vertex, as this may vary from one execution of the pattern to another. To represent the situation when a group contains a transformation that is applied two or more times, similar atomic vertices in groups are collapsed into a single vertex, denoted *action**, where *action* is the name of the action performed. For instance, the three *add_column* atomic vertices in Figure 6.2 will be combined in a single vertex labelled *add_column**, as shown in Figure 6.3.

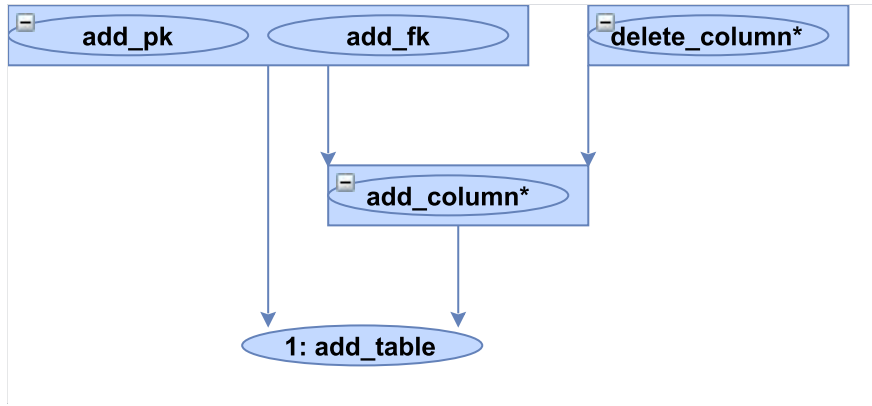


Figure 6.3: Collapsed dependency graph of the transaction in Example 6.1.

The collapsed dependency graph obtained is called the **signature** of the BAV transaction, denoted $Sign(T)$.

Example 6.2. Consider an *account* tab with the columns $\{no, type, cname, rate, sortcode, manager\}$ and the primary key columns $\{no, cname, manager\}$. If we knew that a functional dependency $\{no\} \rightarrow \{type, rate, sortcode\}$ existed, we could execute the following transaction to remove it. This is identical to executing *normalise_table* ($\langle\langle account \rangle\rangle, \langle\langle account2 \rangle\rangle, no, \{type, rate, sortcode\}$).

- ① `add (table:⟨⟨account2⟩⟩,distinct [{no} | {x,no} ← ⟨⟨account,no⟩⟩])`
- ② `add (column:⟨⟨account2,no,notnull,int4⟩⟩,distinct [{no,no} | {no} ← ⟨⟨account⟩⟩])`
- ③ `add (primarykey:⟨⟨account2_pk,account2,⟨⟨account2,no⟩⟩⟩)`
- ④ `add (foreignkey:⟨⟨account_isa_account2_fk,account,⟨⟨account,no⟩⟩,account2,⟨⟨account2,no⟩⟩⟩)`
- ⑤ `add (column:⟨⟨account2,type,notnull,text⟩⟩,distinct [{no,type} | {x,no} ← ⟨⟨account,no⟩⟩; {x,type} ← ⟨⟨account,type⟩⟩])`
- ⑥ `add (column:⟨⟨account2,rate,null,float4⟩⟩,distinct [{no,rate} | {x,no} ← ⟨⟨account,no⟩⟩; {x,rate} ← ⟨⟨account,rate⟩⟩])`
- ⑦ `add (column:⟨⟨account2,sortcode,notnull,int4⟩⟩,distinct [{no,sortcode} | {x,no} ← ⟨⟨account,no⟩⟩; {x,sortcode} ← ⟨⟨account,sortcode⟩⟩])`
- ⑧ `delete (column:⟨⟨account,type,notnull,text⟩⟩,[{x,type} | {x,no} ← ⟨⟨account,no⟩⟩; {no,type} ← ⟨⟨account2,type⟩⟩])`
- ⑨ `delete (column:⟨⟨account,rate,null,float4⟩⟩,[{x,rate} | {x,no} ← ⟨⟨account,no⟩⟩; {no,rate} ← ⟨⟨account2,rate⟩⟩])`
- ⑩ `delete (column:⟨⟨account,sortcode,notnull,int4⟩⟩,[{x,sortcode} | {x,no} ← ⟨⟨account,no⟩⟩; {no,sortcode} ← ⟨⟨account2,sortcode⟩⟩])`

The signature of this transaction is identical to the one in Figure 6.3, which indicates that this transaction and the transaction in Example 6.1 are instances of the same pattern, namely the *table normalisation pattern*. \square

By identifying other transactions with the same signature, patterns can be extracted from a history of transactions. This can be done by checking for isomorphism between two such signatures, which is explained in more detail in Section 6.2, and verifying if the labelling of the two graphs is consistent.

Definition 6.3 Graph isomorphism

An *isomorphism of graphs* $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijection between the vertex sets of G_1 and G_2 , denoted $f: V_1 \rightarrow V_2$ such that if two vertices x and y are adjacent in G_1 , *i.e.* $(x, y) \in E_1$, then $f(x)$ and $f(y)$ are adjacent in G_2 , *i.e.* $(f(x), f(y)) \in E_2$.

In other words, two graphs are isomorphic if there exists a permutation of the rows and columns in the adjacency matrix of one of the graphs that leads to the adjacency matrix of the other graph. \square

Graph isomorphism is known to be expensive to compute, as no known polynomial algorithm has been proposed yet. The worst case time complexity of the brute force method, *i.e.* computing all possible mappings between the vertices of two graphs, is upper bounded by $O(n!)$, where n is the number of vertices in each of the graphs. Several optimisations exist, some of which are presented in 6.2, but that only improve the execution time in practice and not the time complexity.

If we have a history H of BAV transactions and perform a new transaction T , then the time complexity of checking if $Sign(T)$ is in H by checking for isomorphism between $Sign(T)$ and the signature of all transactions in H is upper bounded by $O(n! * X)$, where $X = |H|$. Since the number of elements in H could grow unbounded, this would lead to very poor results over time.

This method can be improved by computing the hash of a signature, as demonstrated in Section 6.3, and storing the history as a hash table. Hash tables present the advantage of constant amortised time complexity for retrieval. Thus, if a new transaction T is performed, its signature $Sign(T)$ is computed, the hash of the signature $Hash(Sign(T))$ is calculated. We denote by Y the time required to compute the hash. Next, only those transactions in H that have the same hash are tested for isomorphism and label consistency. If the history contains N transactions whose hash is equal to the hash of the new transaction, then the amortised time complexity is $O(Y + N * n!)$, with N in practice significantly smaller than X , *i.e.* $N \ll X$.

If the number of matches of a signature in the history C is greater than a given threshold M , then the user is prompted with a message, suggesting him to define a new transformation pattern. In our implementation, $M = 3$. The outline of the algorithm is given in Listing 6.1.

Listing 6.1: Outline of the pattern discovery algorithm

```

PROCEDURE execute(String [] transaction)
    // Compute the signature of the transaction
    DepGraph dg = Sign(transaction);
    // Hash the signature of the transaction
    String dgHash = Hash(DG);
    // Search in the history for transactions with the same hash
    DepGraph [] matches = H(DGHash);
    // Number of matches
    Integer C = 0;

    // Go through signatures with the same hash
    FOR DepGraph crtMatch : matches DO
        // Check for isomorphism and label consistency

```

```

    IF areIsomorphic(dg, crtMatch) and areLabelConsistent(dg, crtMatch)
        C = C + 1;
    END IF
END FOR

IF C > M THEN
    prompt user to define new transformation pattern
END IF

// Add the transaction to the history
addToHistory(H, transaction, dg, dgHash);
END PROCEDURE

```

The method only uses the information about what constructs in a schema are modified and does not handle the extents of the individual transformations. This is because there is no way to check if two queries are equivalent. In the end, when defining a new transformation pattern, it is up to the user to define the extents of the primitive BAV transformations.

6.2 Graph Isomorphism

The graph isomorphism problem presents a great challenge in the field of algorithms. This is because it is known that the problem is in the class of NP, *i.e.* problems verifiable in polynomial time by a deterministic Turing machine, but no relation has been discovered between the problem and two well known subsets of NP [Kar72]: P, the class of problems solvable in polynomial time, and NP-complete, the class of problems for which no polynomial time is known.

On the other hand, the generalisation of this problem, *subgraph isomorphism*, where given two graphs G_1 and G_2 the question is whether a subgraph G_1 is isomorphic to G_2 , is known to be NP-complete [Coo71].

The brute force method can solve this problem in $O(n!)$ time complexity, with n being the number of vertices in each of the graphs, making the computation of a mapping between the vertices of the two graphs very hard.

Several techniques have been proposed in the last decades and, despite the fact that some of them increase the worst case time complexity, they work very well in practice. Two such algorithms have been implemented in the tool and evaluated: the Schmidt-Druffel (SD) algorithm [SD76] and the Vento-Foggia (VF2) [CFSV01] algorithm. A more complete comparison of different algorithms for graph isomorphism is given in [FSV01].

In our implementation, the performance of these algorithms has been evaluated on an Intel i5 430M processor by randomly generating pairs of isomorphic graphs with an increasing number of vertices and the probability of an edge existing between two vertices of p . For each case, a number of 50 runs have been executed and the average execution time, expressed in milliseconds, has been recorded.

6.2.1 The Schmidt-Druffel Algorithm

The Schmidt-Druffel (SD) algorithm [SD76] was published in 1976. Unlike their predecessors, who relied mostly on degrees of the nodes to characterise a graph, the authors of this algorithm used the *distance matrix* for this, denoted D , as it contains more information. In a distance matrix, the element at (i, j) specifies the length of the shortest path between vertices i and j . It can be

computed using the Floyd-Warshall algorithm [Flo62]. The distance matrices of the two graphs being checked for isomorphism are denoted D^1 and D^2 .

The SD method is structured into two steps: the computation of an initial partitioning and the backtracking step.

The computation of an initial partitioning of the vertices is done by computing the *characteristic matrix* of the graphs, denoted X^1 and X^2 . The characteristic matrix is obtained by composing two matrices: the *row characteristic matrix* and *column characteristic matrix*. In the row characteristic matrix, the element at (i, m) specifies the number of vertices at distance m from vertex i . In the column characteristic matrix, the element at (i, m) specifies the number of vertices that vertex i is at distance m from.

A node i from G_1 can map to a node j from G_2 only if $X_{i,k}^1 = X_{j,k}^2, \forall k$, so vertices with the same rows in the characteristic matrices are placed in the same initial classes, thus obtaining an initial partition.

The backtracking algorithm refines the initial partition until there is a 1:1 mapping between the vertices of the two graphs, if such a mapping exists. This is done by choosing two vertices that belong to the same class and analysing if the mapping between the two vertices is *consistent*. A mapping between i and j is consistent if: (a) the elements in the i -th row and column of D^1 have correspondents in the j -th row and column of D^2 , for all previously mapped elements, and (b) the remaining elements in those rows and columns do not rule out mappings between vertices that have not been mapped.

The worst case time complexity of the first step is $O(n^3)$, which is the upper bound of the Floyd-Warshall algorithm. In the original paper of the algorithm, the authors demonstrate that the upper bound of the second step is $O(n * n!)$, so the overall worst time complexity is $O(n * n!)$, which is worse than the brute force method.

This algorithm has been evaluated on pairs of isomorphic graphs with up to 200 vertices and the results are shown in Figure 6.4. We can notice a rapid increase in the average execution time. Since the time complexity of the algorithm is independent on the number of edges, only the case when the edge probability is $p = 20\%$ has been considered.

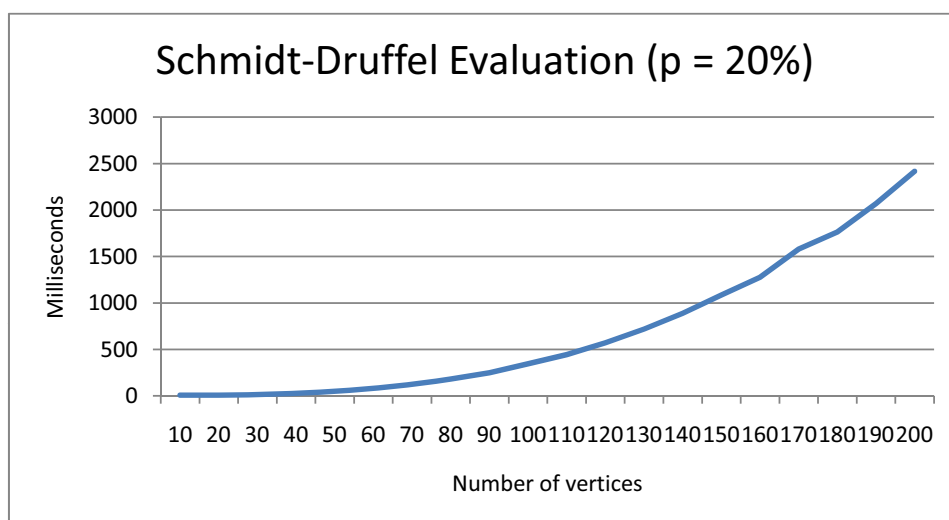


Figure 6.4: Evaluation of the SD algorithm.

6.2.2 The Vento-Foggia Algorithm

The Vento-Foggia algorithm (VF2) [CFSV01] is a more recent algorithm, published in 2001. It is an improved version of the VF algorithm presented in [CFSV99]. The strategy employed in VF2 is a depth-first search.

For each state s of the search tree, the authors define four sets: the *out-terminal set* $T_1^{out}(s)$ contains the nodes in G_1 that have not been previously mapped, but are successors of a node that has been mapped, $T_1^{in}(s)$ contains the nodes in G_1 that have not been previously mapped, but are predecessors of a node that has been mapped. Similarly, $T_2^{out}(s)$ and $T_2^{in}(s)$ are defined for G_2 .

In the above definition, a node i is called a *predecessor* of a node j if there exists an edge from i to j , and a node i is called a *successor* of a node j if there exists an edge from j to i .

The information contained in the four sets described above is used to prune the search tree at every step. This is done by restricting the pair of candidates for inclusion in the mapping to be from $T_1^{out}(s) \times \{minT_2^{out}(s)\}$ or from $T_1^{in}(s) \times \{minT_2^{in}(s)\}$, where $\{minT\}$ denotes the element with the smallest label in the set T . In case all four sets are empty, the pairs of candidates are simply chosen from the nodes that have not been previously mapped.

Once a pair of candidates (i, j) is generated, its feasibility is analysed. This is done by verifying that if i has an edge to or from a previously mapped vertex, then there exists a correspondent edge to or from j . Additionally, the following must hold after the introduction of the mapping (i, j) : $|T_1^{out}(s)| = |T_2^{out}(s)|$ and $|T_1^{in}(s)| = |T_2^{in}(s)|$. If these conditions are satisfied, then the pair (i, j) is added to the partial mapping and the search can proceed one level down, until there is a 1:1 mapping between the vertices of the two graphs, if such a mapping exists.

This algorithm brought a significant improvement of the memory used, which is bounded by $O(n)$. Previous graph isomorphism algorithms all had a memory requirement of $O(n^3)$.

In order to evaluate the implementation of this algorithm, pairs of isomorphic graphs have been generated with up to 1500 vertices. It can be seen from Figure 6.5 that the performance of the algorithm is significantly better than the performance of Schmidt-Druffel, so it will be the preferred algorithm used in the tool to check for isomorphism.

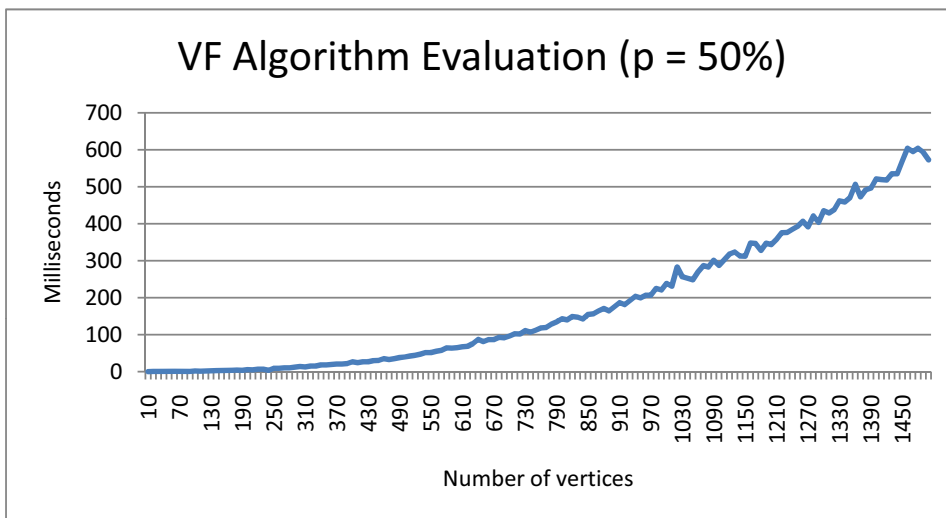


Figure 6.5: Evaluation of the VF2 algorithm.

6.2.3 Towards a Hybrid Graph Isomorphism Component

In certain circumstances, the complexity of checking for isomorphism can be reduced, when certain things are known about the topology of the two graphs, such as when they are planar. In the context of our method, this will often be the case in practice.

Definition 6.4 Planar graph

A *planar graph* is a type of graph that can be embedded in a plane, such that no two edges cross each other. \square

The planarity testing problem is concerned with checking if a graph is planar or not. Numerous methods have been proposed over the years that solve this problem in linear time, which is asymptotically optimal, *i.e.* the worst the performance of the algorithm differs by at most a constant factor from the performance of the best possible algorithm. The first algorithm was published in 1974 in [HT74]. Since then, other methods [ET77, BL76, kSH99] have tried to improve the constant factor of the algorithm and produce better results.

Testing for isomorphism between two planar graphs has been shown in [HW74] to be possible in linear time, which is significantly faster than any of the algorithms presented in the previous sections.

The pattern discovery method could be extended to include a hybrid graph isomorphism component, that first checks if the two graphs being compared are planar and applies the algorithm in [HW74] if this is the case. Otherwise, it applies the SD or VF2 algorithm.

Due to time constraints, the hybrid component has not been implemented and evaluated and is left as future work.

6.3 Graph Hashing

A method for identifying a graph based on its hash is preferred when needing to extract from a history those graphs that are isomorphic to a given graph. As the number of elements in the history may grow unbounded, applying any of the algorithms presented in Section 6.2 becomes infeasible. Instead, the hash of the graph to be matched is computed and it is used to retrieve graphs that *might be* isomorphic, as collisions might occur in practice. To ensure that the graphs are in fact isomorphic, one of the algorithms presented in Section 6.2 may be applied.

The technique presented in [Por] may be used to compute the hash of a graph. It does this by iteratively expanding every vertex and computing the hash for the graph configuration in the neighbourhood of the vertex. This process is completed when all vertices have distinct hashes, at which point the hashes are sorted, concatenated and the result is hashed again, yielding the hash of the graph.

The implementation of the algorithm was done according to the guidelines in [Por]. The algorithm is highly dependent on the number of edges in the graph, as at each step the neighbourhood of a vertex is hashed. Evaluation has been done by generating pairs of isomorphic graphs for two edge probabilities: 20% and 50%. It can be noticed from Figures 6.6 and 6.7 that the average execution time significantly increases when there are more edges in the graph.

Hashing only becomes useful when the number of transactions in a history exceeds a certain value, something that is realistic in a real-world situation.

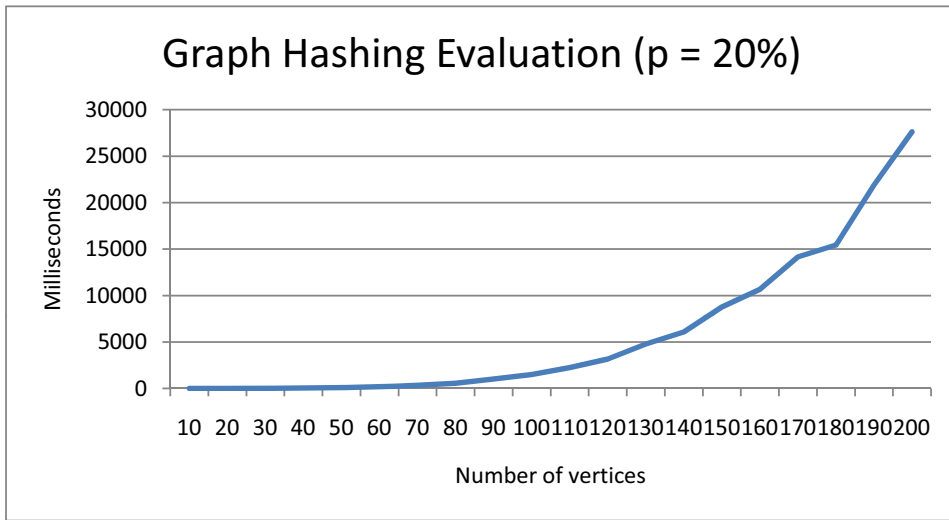


Figure 6.6: Evaluation of the graph hash algorithm.

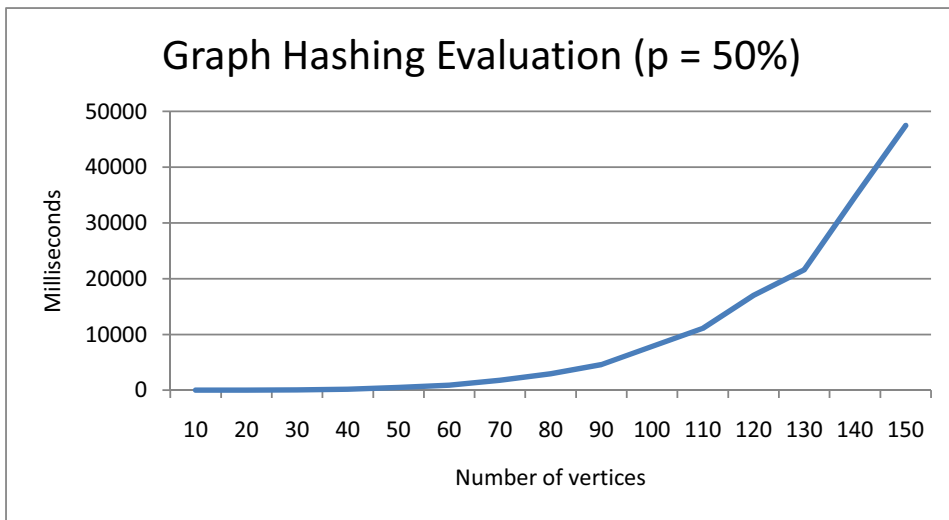


Figure 6.7: Evaluation of the graph hash algorithm.

6.4 Performance Evaluation

The performance of the pattern discovery method has been evaluated on an Intel i5 430M processor, by randomly generating an increasing number of transactions containing both atomic and group nodes. Every transaction contains between 3 and 20 primitive instructions. The object being modified in every individual instruction and the set of dependent objects are chosen from a list of schema objects randomly generated. Every instruction contains between 1 and 5 dependent schema objects.

The performance of the algorithm is expressed as the average execution time necessary when processing a new transaction and checking if the history of transactions contains similar transactions. In our implementation, the execution time is averaged over 50 runs. Two cases have been considered, one when the graph hashing is ignored and another case when hashing is used to prune the search space.

Figure 6.8 presents the performance evaluation in the first case, with the two algorithms presented in Section 6.2 being used. It is clear from the chart that using any of the two graph isomorphism algorithms produces similar results. This is because transaction signatures contain a small number of vertices. The only advantage over the Vento-Foglia algorithm over Schmidt-Druffel is the memory complexity, which is upper bounded by $O(n)$ in VF2, compared to $O(n^3)$ in SD. A

linear increase in the execution time can be noticed, which reaches approximately 12 seconds when the history contains 2500 transactions, something that is not optimal in a real-world situation.

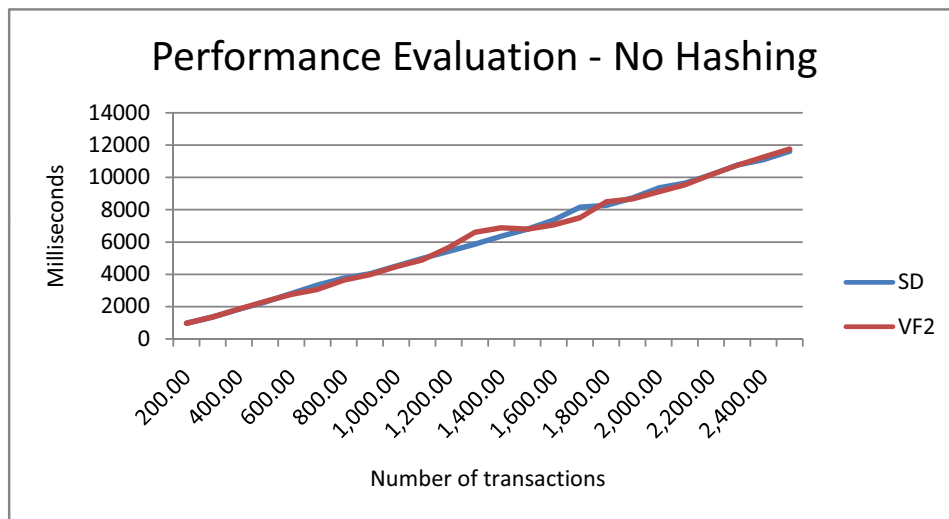


Figure 6.8: Performance evaluation without graph hashing.

Figure 6.9 presents the performance evaluation for the second case, when hashing is used to prune the search space. The number of transactions in the history that have the same signature hash is kept constant at about 2%, something that is realistic in a real-world situation. As before, the two graph isomorphism algorithms produce similar results, but that are significantly better than the ones in the first case. The method was able to extract similar transactions from a history containing a total of 2500 transactions in roughly 0.1 seconds.

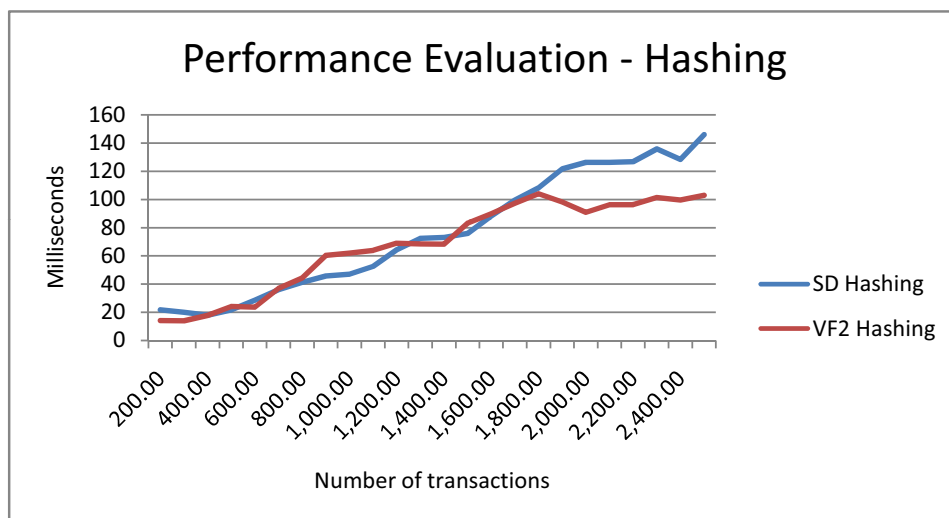


Figure 6.9: Performance evaluation with graph hashing.

The performance evaluation of the pattern discovery method shows that the algorithm supports a very large number of transactions, being capable to match similar transactions in a history of transactions.

6.5 The Method in the Database Integration Tool

The pattern discovery method has been integrated in the *Interactive Database Integration Tool*. When integrating several schemas, the user is allowed to execute BAV transactions over them.

The AutoMed repository has been extended to keep the history of all transactions. This was done by creating a table named *transaction* in the **schema transformation repository** (STR)

(see Section 2.5.3) with the following columns:

- *tid* - the identifier of the transaction
- *instructions* - the primitive BAV transformations in the transaction
- *signature* - the serialisation of the signature
- *hashcode* - hash code of the signature
- *originalschema* - name of the original schema, over which the transaction has been executed
- *finalschema* - name of the final schema, after the execution of the transaction

Whenever a transaction T is executed, its signature and hash code are computed, an identifier for the transaction is generated and a new row is inserted in the *transaction* table. The pattern discovery method is applied and the history is scanned for similar transactions, *i.e.* whose hash codes are equal to the hash of the T . Next, T and the signatures returned from the history are checked for isomorphism and label consistency. If the number of matches is greater than a threshold (in the implementation set to 3), the user is prompted with a dialog box, allowing him to view a list of the matching transactions, such as the one shown in Figure 6.10. It can be noticed from the figure that all returned transactions will have identical hash codes.

TID	Instructions	Hashcode	Signature	Original schema	Resulting schema
1	add (table:<<account...	4fe580305e371423c...	ro.dta.idbi.model.dyn...	testdb_1	testdb_j
102	add (table:<<account...	4fe580305e371423c...	ro.dta.idbi.model.dyn...	accounts	accounts_j
103	add (table:<<cities>>...	4fe580305e371423c...	ro.dta.idbi.model.dyn...	store	store_j

Figure 6.10: Identical transactions returned by the tool.

By right-clicking any of the matching transactions in the table, the user can open a detailed view of the transaction, such as the one presented in Figure 6.11, displaying the instructions that the transaction contains and a visual representation of its signature.

From the main menu bar of the application, the user can open the transaction history browser, allowing him to view all transactions in the history and display only those transactions that have the hash code equal to a value. As before, by right-clicking any of the transactions, the user can open the transaction viewer window for a detailed description of the transaction.

6.6 Summary

In this chapter, a new method for extracting patterns from BAV transformations is presented, which has successfully been integrated in the *Interactive Database Integration Tool*. The method works by computing the signature of the transformation and searching the history for identical signatures.

Similarity is assessed by computing the isomorphism between graphs. Two algorithms for graph isomorphism have been implemented and evaluated: the Schmidt-Druffel (SD) algorithm and the Vento-Foggia (VF2) algorithm. In our method, VF2 is preferred, because it reduces the memory complexity to linear. If two signatures are isomorphic, they are checked for label consistency.

In order to improve the performance of the algorithm, the search space is pruned by computing the hash of the signature and only checking for isomorphism between graphs whose signatures hash to the same value.

A major drawback of the algorithm is that the extraction of new patterns is done in a semi-automatic manner, where the user has to define the extents of the primitive BAV transformations.

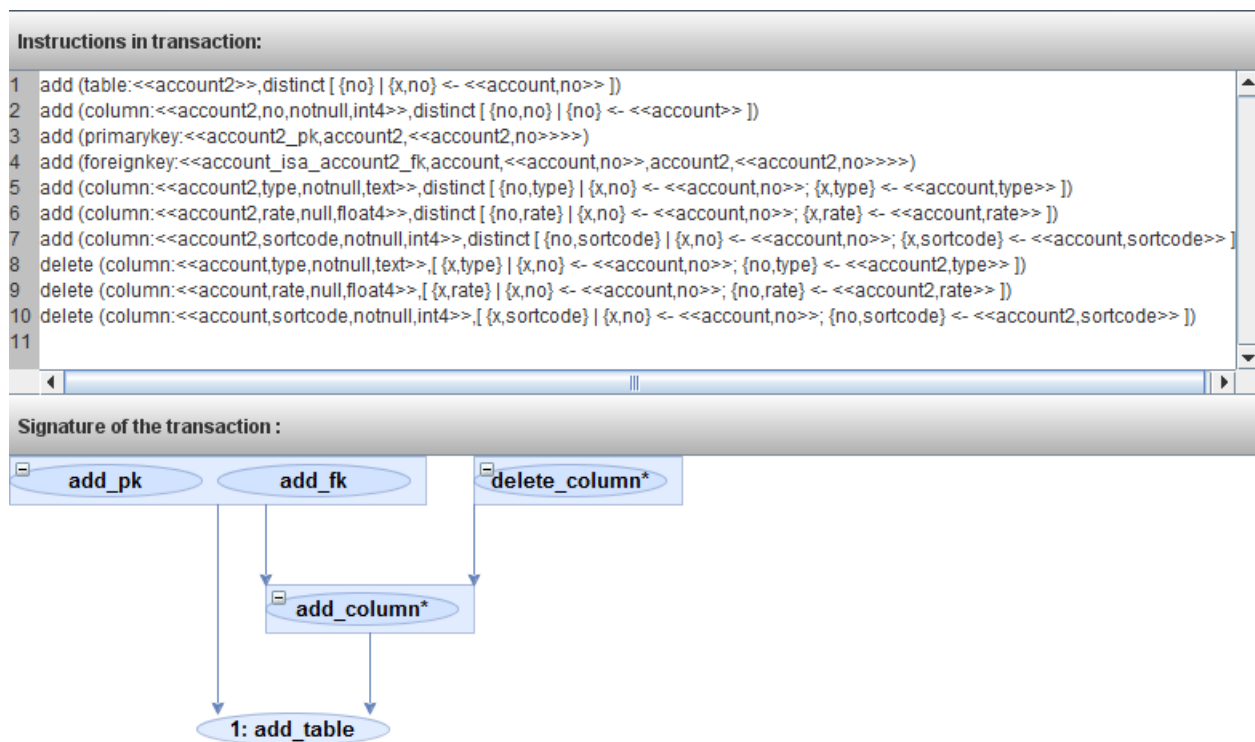


Figure 6.11: Transaction viewer window for the transaction in Example 6.2.

Future work includes implementing the graph isomorphism algorithms and the graph hashing algorithm to run in parallel on multi-core processors, in order to improve their timing, and extending the implementation of the method to handle transformation macros in BAV transactions, not only primitive BAV transformations.

The technique could also be used in view integration. Given a set of source schemas and an integrated schema, the problem is to find the transformations that map the every source schema to the integrated schema. This combinatorial problem could be a NP-complete problem, but further research needs to be done to verify this. In case no polynomial time algorithm can be found to solve it, heuristics can be applied to retrieve the top-K most probable mappings.

Chapter 7

Conclusions and Future Work

This chapter presents the concluding remarks, in Section 7.1, and possible directions for future work, in Section 7.2.

7.1 Conclusions

This thesis presented *Interactive Database Integration Tool*, a software application built on top of the AutoMed framework for guiding the user through the integration process of several schemas expressed in the SQL metamodel. Integration is performed manually by interactively applying well-known transformation patterns, which are expressed in the both as view (BAV) mapping language.

The development of the tool has been both challenging and rewarding. It provided the opportunity to get a better understanding of *database integration* and why this field has received considerable attention in the last three decades. It serves as proof that the gap between theory and practice can be bridged, a problem that researchers have been facing with.

While developing the application, we gained knowledge about the internals of AutoMed. AutoMed uses the hypergraph data model (HDM) as the common data model that schemas are expressed in and the both as view (BAV) mapping language. The advantage of using BAV is that it subsumes two of the previous well-known mapping languages, *i.e.* local as view (LAV) and global as view (GAV).

The implementation has the potential of evolving into a large project. It already measures approximately 16.000 lines of code. This is why, from the engineering point of view, we focused on creating a solid architecture, that can cope with evolution and maintainability. By avoiding cyclic dependencies between modules, we promote the reuse of the model of the application independent of the view.

One of the most important features of our architecture is the transformation patterns framework, which allows developers to introduce new transformation patterns by following six steps and without having to modify any of the existing code. Using this framework we were able to implement 18 transformation patterns, all of which have a user interface associated with them, that guide the user in the application of the pattern.

The final product serves as a solid foundation but, due to time constraints, only a subset of the possible features have been implemented. At the present time, the tool only supports schemas expressed in the SQL metamodel, something that is not realistic in a real-world situation, where schemas can be expressed in the Extensible Markup Language (XML), the Entity-Relationship

(ER) or other metamodels.

Another drawback is that at the present time we assume that the person using the application has a good Intermediate Query Language (IQL) knowledge, something that is not always the case.

Pattern discovery in BAV transactions was also considered in this thesis. A new method for detecting identical transactions in a history of transactions was discussed, which uses techniques from the parallel task scheduling field. It works by computing the signature of a transaction from its dependency graph. This was motivated by the fact that the execution of any of the five primitive BAV transformations results in another schema, where only one schema object is altered.

Similarity between two signatures is assessed by checking if there exists an isomorphism between them, a problem that is known to be expensive to compute. Checking for isomorphism between a large number of pairs of graphs proved to be infeasible in the evaluation of our method, so the technique was improved by computing the hash of a signature, after it is executed, and only scanning the history for signatures with the same hash. This led to a significant decrease in the average execution time, which makes the system support a large number of transactions.

While the pattern discovery method is powerful, it can only be used to discover transactions that are executed often by the user and does not actually introduce on the fly new transformation patterns in the tool. At the present time, we assume that the user has a good understanding of the transformation patterns framework and of the Java programming language and can introduce the pattern himself. In the current version of the database integration tool, the method is only used to present the user with patterns that have been applied many times.

7.2 Future Work

The project can be extended in many directions, by taking advantage of the solid foundation that has been put into place. Most of the ideas listed below present great challenges, but were not researched in great detail, because of the time constraints.

7.2.1 From Manual to Automatic Schema Matching

At the present time, the user is responsible with finding correspondences between objects in different schemas. Research has been done in the field of automatic schema matching in the last ten years, with significant results. In [RB01] and [SE05], two similar classifications of the schema matching approaches are presented. Both papers classify the same types of matchers, such as:

- *String-based matchers* check for name similarity, description similarity between schema objects. Usually, this is measured as a distance function that maps a pair of strings to a real number.
- *Language-based matchers* process the words in the names of schema objects, by applying techniques from Natural Language Processing (NLP).
- *Linguistic resources* make use of thesauri, such as the lexical database WordNet [Mil95], to check for the relationship between words, e.g. they are synonyms.
- *Constraint-based matchers* deal with the constraints that are applied to the definition of schema objects, such as cardinality constraints in the Entity-Relationship metamodel or the primary key and foreign key constraints in the SQL metamodel.
- *Model-based matchers* handle the semantic interpretation of the input of the schema matching. They do this by applying well-known deductive methods, such as the prepositional

satisfiability (SAT) and description logics (DL).

AutoMed contains a schema matching component, outlined in [Riz04, MRBM05] and described in detail in [Riz10]. It only takes into account **semantic mappings**, e.g. "paper represents the same concept as *publication*". At the other end of the spectrum are **data mappings**, e.g. "each value of *birthday* is equal to the concatenation of *day*, *month* and *year*".

The advantage of this method over previous approaches is that it incorporates **uncertainty** in the mappings, *i.e.* how probable a mapping is. They extend previous work by not only considering compatibility mappings, *i.e.* whether two objects are compatible or not, but by considering the five relationships presented in Section 2.5.3.

An uncertain semantic mapping (USM) is obtained by aggregating several USMs from different matchers, which in [Riz10] are also called *experts*. Using the architecture of the schema matching component, new experts can be introduced at runtime. A possible extension of this component would be to apply machine learning techniques to automatically discover the best experts to use in different contexts.

The result of applying the schema matching components to two input schemas are the top-K most probable schema mappings. This could be integrated in the tool, although it does not always return the expected result.

In the context of the *Interactive Database Integration Tool*, the schema matching component could be used to suggest transformation patterns to the user. For instance, if we had a *country_name* column in a table in one schema and a *country* table in another schema and the schema matching tool computed that the two objects are equivalent with 80% certainty, the tool could suggest the user to apply the *column to table* transformation pattern, described in Section 5.1.5.

7.2.2 A Language for Specifying Transformation Patterns

Chapter 4 presented the transformation patterns framework. It can be noticed that most transformation patterns rely on the same constructs, e.g. sequences of schema objects, and simple operations, e.g. iterating over sequences of schema objects. Developing a simple language to handle the definition of transformation patterns could help users with no Java knowledge introduce their own patterns. This could be done by formally specifying the syntax of the language, the operational semantics and the type system. The implementation of this language could be done as a compiler from this language to Java bytecode.

Such a language would be particularly useful in the context of the pattern discovery method presented in this thesis. After the discovery of a frequently used BAV transaction, the tool could ask the user to define the transformation pattern in this language.

7.2.3 From the SQL Metamodel to a Generic Tool

At the moment only schemas expressed in the SQL metamodel are supported by the tool. The project could be extended, by taking a step back and thinking about the integration process in a more generic way. This could be done by introducing other metamodels in the application, such as XML or ER. AutoMed already supports this, by representing any model in the hypergraph data model.

This functionality could also be used to convert between metamodels, although this could lead to the loss of information, as some metamodels contain constructs that cannot be expressed in

other metamodels. For instance, there is no equivalent for the total generalisation constraint from the ER metamodel in the SQL metamodel.

Another issue that could be researched is that of automatically expressing well-known transformation patterns, such as the ones presented in Chapter 5, in different metamodels. The challenge is to identify corresponding constructs in the metamodels and dealing with constructs that are modelled in only one of the metamodels.

Bibliography

- [Alt] Altova missionkit. <http://www.altova.com/>. [Online; accessed 22/3/2011].
- [BAV] The university database integration: An automed example. http://www.doc.ic.ac.uk/automated/techreports/univesity_automated_example.ps. [Online; accessed 02/05/2011].
- [Biz] Microsoft biztalk server. <http://www.microsoft.com/biztalk/en/us/default.aspx>. [Online; accessed 22/3/2011].
- [BKL⁺04] Michael Boyd, Sasivimol Kittivoravikul, Charalambos Lazanitis, Peter M C Brien, and Nikos Rizopoulos. Automed: A bav data integration system for heterogeneous data sources. In *In Proc. CAiSE'04*, pages 82–97. Springer-Verlag, 2004.
- [BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *Journal of Computational Systems Science*, 13:335–379, 1976.
- [BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM COMPUTING SURVEYS*, 18(4):323–364, 1986.
- [BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07*, pages 1–12, New York, NY, USA, 2007. ACM.
- [CFSV99] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing, ICIAP '99*, pages 1172–, Washington, DC, USA, 1999. IEEE Computer Society.
- [CFSV01] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.
- [CMS97] R. Cooley, B. Mobasher, and J. Srivastava. Web mining: Information and pattern discovery on the world wide web. *Tools with Artificial Intelligence, IEEE International Conference on*, 0:0558, 1997.
- [Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing, STOC '71*, pages 151–158, New York, NY, USA, 1971. ACM.
- [Cry] Crystal reports. <http://www.crystalreports.com/>. [Online; accessed 22/3/2011].
- [DBH⁺99] S. B. Davidson, P. Buneman, S. Harker, C. Overton, and V. Tannen. Transforming and integrating biomedical data using kleisli: a perspective. *SIGBIO Newsl.*, 19:8–13, August 1999.
- [DBM] Db-main. <http://www.db-main.eu/>. [Online; accessed 26/4/2011].
- [dDNmP⁺00] Notre dame De, La Paix Namur, Db main Programme, Philippe Thiran, AbdelMajid Chougrani, J l. Hainaut, Jean-Marc Hick, Abdelmajid Chougrani, Jean luc Hainaut, and Jean marc Hick. Case support for the development of federated information systems, 2000.

- [ET77] Shimon Even and Robert Endre Tarjan. Corrigendum: Computing an t -numbering. *tcs* 2(1976):339-344. *Theor. Comput. Sci.*, 4(1):123, 1977.
- [Fag79] Ronald Fagin. Normal forms and relational database operators. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 153–160, New York, NY, USA, 1979. ACM.
- [FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.
- [Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.
- [FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *in Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.
- [Gal07] Avigdor Gal. Why is schema matching tough and what can we do about it. *SIGMOD Record*, pages 2–5, 2007.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. pages 283–293, 1995.
- [Haa07] Laura M. Haas. Beauty and the beast: The theory and practice of information integration. In *ICDT*, pages 28–43, 2007.
- [HHH⁺05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 805–810, New York, NY, USA, 2005. ACM.
- [Hib] Hibernate. <http://www.hibernate.org/>. [Online; accessed 22/3/2011].
- [HKPT99] Yk Huhtala, Juha Krkkinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies, 1999.
- [HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, October 1974.
- [HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 172–184, New York, NY, USA, 1974. ACM.
- [IDB] Interactive database integration tool javadoc. <http://www.airtudor.com/idbi/>. [Online; accessed 2/9/2011].
- [IFF⁺99] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 299–310, New York, NY, USA, 1999. ACM.
- [Inf] Microsoft infopath. <http://office.microsoft.com/en-us/infopath/>. [Online; accessed 22/3/2011].
- [JGr] Junit library. <http://www.jgraph.com/>. [Online; accessed 7/8/2011].
- [JPZ03] Edgar Jasper, Alex Poulovassilis, and Lucas Zamboulis. Processing iql queries and migrating data in the automed toolkit. Technical report, 2003.
- [JUn] Junit library. <http://www.junit.org/>. [Online; accessed 7/8/2011].
- [Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [kSH99] Wei kuan Shih and Wen-Lian Hsu. A new planarity test, 1999.
- [Len02] Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

- [McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [Mil95] George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.
- [MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *In VLDB*, pages 120–133, 1993.
- [MP97] Peter McBrien and Alexandra Poulovassilis. A formal framework for er schema transformation, 1997.
- [MP98] Peter McBrien and Alexandra Poulovassilis. A General Formal Framework for Schema Transformation. *Data and knowledge engineering*, 28(1):47–71, October 1998.
- [MP03] Peter McBrien and Alexandra Poulovassilis. Data integration by bi-directional schema transformation rules. *Data Engineering, International Conference on*, 0:227, 2003.
- [MP04] Peter McBrien and Alexandra Poulovassilis. Defining peer-to-peer data integration using both as view rules, 2004.
- [MRBM05] Matteo Magnani, Nikos Rizopoulos, Peter M C Brien, and Danilo Montesi. Schema integration based on uncertain semantic mappings. In *In International conference of conceptual modeling*, pages 31–46. Springer, 2005.
- [MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
- [Ora] Oracle webcenter interaction. <http://www.oracle.com/technetwork/middleware/webcenter-interaction/overview/index.html>. [Online; accessed 22/3/2011].
- [Por] Thomas E. Portegys. General graph identification with hashing.
- [PS98] Christine Parent and Stefano Spaccapietra. Issues and approaches of database integration. *Commun. ACM*, 41:166–178, May 1998.
- [RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, December 2001.
- [Rea98] Isidore Rigoutsos and et al. Combinatorial pattern discovery in biological sequences: the teiresias algorithm, 1998.
- [Riz04] Nikos Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *In Proc. of 6th ICEIS*, pages 3–8, 2004.
- [Riz10] N. Rizopoulos. Schema matching and schema merging based on uncertain semantic mappings, 2010.
- [SD76] Douglas C. Schmidt and Larry E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23:433–445, July 1976.
- [SE05] Pavel Shvaiko and Jrme Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, pages 146–171, 2005.
- [Sql] Microsoft sql server reporting services. <http://www.microsoft.com/sqlserver/2008/en/us/reporting.aspx>. [Online; accessed 22/3/2011].
- [SRM] Andrew Smith, Nikos Rizopoulos, and Peter Mcbrien. Automated model management.
- [Str] Structure101. <http://www.headwaysoftware.com/products/?code=Structure101>. [Online; accessed 23/8/2011].
- [Sty] Stylus studio. <http://www.stylusstudio.com/>. [Online; accessed 22/3/2011].
- [TAN] Tane: Functional dependency discovery. <http://www.cs.helsinki.fi/research/fdk/datamining/tane/>. [Online; accessed 22/8/2011].

- [Top] Oracle toplink. <http://www.oracle.com/technetwork/middleware/toplink/overview/index.html>. [Online; accessed 22/3/2011].
- [Tuk] Tukwila. <http://tukwila.sourceforge.net/>. [Online; accessed 23/3/2011].
- [Xer] Xerces library. <http://xerces.apache.org/>. [Online; accessed 7/8/2011].