ON RESOLVING SEMANTIC HETEROGENEITIES AND DERIVING CONSTRAINTS IN SCHEMA INTEGRATION

QI HE

(B.Sc., Fudan University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Abstract

A challenge in schema integration is schematic discrepancy, i.e., meta information in one database correspond to data values in another. The purposes of this work were to resolve schematic discrepancies in the integration of relational, ER and XML schemas, and to derive constraints in schema transformation in the context of schematic discrepancies.

In the integration of relational schemas with schematic discrepancies, a theory of schema transformation was developed. The theory was on the properties (i.e., reconstructibility and commutativity) of schema-restructuring operators and the properties (i.e., information preservation and non-redundancy) of schema transformation.

Qualified functional dependencies which are functional dependencies holding over a set of relations or a set of horizontal partitions of relations were proposed to represent constraints in heterogeneous databases with schematic discrepancies. We proposed algorithms to derive qualified functional dependencies in schema transformation in the context of schematic discrepancies. The algorithms are sound, complete and efficient to derive some qualified functional dependencies. The theory of qualified functional dependency derivation is useful in data integration/mediation systems and multidatabase interoperation.

In the integration of ER schemas which are more complex than relational schemas, we resolved schematic discrepancies by transforming the meta information of schema constructs into attribute values of entity types. The schema transformation was proven to be both information preserving and constraint preserving.

The resolution of schematic discrepancies for the relational and ER models can be extended to XML. However, the hierarchical structure of XML brings new challenges in the integration of XML schemas, which was the focus of our work. We represented XML schemas in the Object-Relationship-Attribute model for Semi-Structured data (or ORASS). We gave an efficient method to reorder objects in a hierarchical path, and proposed a semantic approach to integrate XML schemas, resolving the inconsistencies of hierarchical structures. The algorithms were proven to be information preserving.

We believe this research has richly extended the theories of schema transformation and the derivation of constraints in schema integration. It may effectively improve the interoperability of heterogeneous databases, and be useful in building multidatabases, data warehouses and information integration systems based on XML.

Acknowledgement

First of all, I would like to thank my supervisor Prof Ling Tok Wang. He taught me the way of research and presentation, and the spirit of continuous improvement. As a researcher, he is a man of insight and experience. His comments are always suggestive and pertinent. As a supervisor, he is patient and strict. It's lucky but not easy to be his student. He leads me along the way here. Without his help, the thesis would never have been come into being.

Thank Dr. Stéphane Bressan and Dr. Chan Chee Yong for the effort and time to read the thesis and the valuable comments based on which I improved the thesis much.

Thank Prof Zhou Aoying and Prof Ooi Beng Chin. They provided me with the opportunity to pursue the PhD degree in Singapore.

I am also thankful to my colleagues in SoC and all my friends in Singapore: Chen Ding, Chen Ting, Chen Yabin, Chen Yiqun, Chen Yueguo, Chen Zhuo, Cheng Weiwei, Dai Jing, Ding Haoning, Fa Yuan, Fu Haifeng, Hu Jing, Huang Yang, Huang Yicheng, Jiao Enhua, Li Changqing, Li Xiaolan, Li Yingguang, Liu Chengliang, Liu Shanshan, Liu Xuan, Lu Jiaheng, Ni Yuan, Pan Yu, Sun Peng, Wang Shiyuan, Wang Yan, Xia Chenyi, Xia Tian, Xiang Shili, Xie Tao, Xu Linhao, Yang Rui, Yang Xia, Yang Xiaoyan, Yang Tian, Yao Zhen, Yu Tian, Yu Xiaoyan, Zhang Han,

Zhang Wei, Zhang Xiaofeng, Zhang Zhengjie, Zheng Wei, Zheng Wenjie, Zhou Xuan, and Zhou Yongluan. Thank them not only for the help and encouragement, but also for the dispute. The friendship among us will be a treasure in my life.

Special thanks go to my friend Ni Wei for his warm heart and wisdom. He pushed me when I hesitated, guided me when I was lost and accompanied me when I was hurt. With self discipline, he can be something one day. I have no doubt about that.

Finally, thank my parents. They are always at my back no matter what I do.

Contents

List of Figures

Chapter 1

Introduction

Traditionally, database application uses software, called a database management system managing a multitude of data located in one site. Modern applications require easy and consistent access to multiple databases. A *multidatabase system* (i.e., MDBS) addresses this issue. A MDBS is a collection of cooperating but autonomous database systems (called *component database systems*). Such a system provides controlled and coordinated manipulation of the component databases. In building a MDBS, schema integration plays an important role. Schema integration is the activity to integrate the schemas of existing or proposed databases into a global, unified schema. Users can access the data of those component databases through the integrated schema. The differences and inconsistencies of data models, schemas and data among those databases are transparent to users.

A data warehouse is a "subject-oriented, integrated, time varying, non-volatile collection of data that is used primarily in making decisions in organizations [28]." Unlike a MDBS, a data warehouse contains consolidated data from several operational databases and other sources. However, similar information may be stored in different schemas in source databases, schema integration is therefore a necessary stage before data integration in which duplicate and inconsistency of data are

removed.

Another application of schema integration is view integration in database design. View integration is a process of producing a schema of a proposed database by integrating different user views. There are two reasons for view integration in database design: (1) the structure of database is too complex to be modelled in a single view, and (2) user groups have their own requirements and expectations of data. View integration is on schema level and usually processed during conceptual database design.

As XML becomes more and more a de facto standard to represent and exchange data in e-business, information mediation/integration based on XML provides a competitive advantage to businesses [48]. XML schema integration is a necessary stage in building an integration system for either transaction or analytical processing purpose.

Correspondingly, schema integration can be divided into to 2 classes according to the data models, one on flat models such as relational, ER or object-oriented model, and the other one on hierarchical models such as XML. In general, in schema integration, people usually need to resolve different kinds of semantic heterogeneities:

- *Naming conflict* - Homonyms and synonyms are the two sources of naming conflicts. Renaming is a frequently chosen solution in existing work.

- *Key conflict* - Different keys may be assigned as the identifier of the same concept in different schemas. For example, attributes SSNO and EMPNO may be identifiers for the entity types of EMPLOYEE in two schemas.

- *Structural conflict* - The same real world concept may be represented in two schemas using different schema constructs [4, 39]. For example, the same

concept publisher may be modelled as an entity type in one schema, but an attribute in another schema.

- *Domain mismatch* - Domain mismatch occurs when we have conflict between the domains of equivalent attributes. E.g., the value set for an attribute EXAM_SCORE may be in grades (A, B, C etc) in one database and in marks in another database. Given the corresponding rules between the grades and marks, we can resolve this kind of conflicts.

- *Constraint conflict* - Two schemas may represent different constraints on the same concept [38]. For example, the conflict occurs on the cardinality constraints. For instance, PHONE_NO may be a single valued attribute in one schema, but multi-valued in another schema. Another example involves different constraints on a relationship type such as TEACH. Assuming that instructors can teach more than one course, one schema may represent TEACH as 1:n (a course has an instructor) and another schema may represent it as m:n (some courses may have more than one instructors).

- *Classification inconsistency* - hyponyms or hypernyms, i.e., an object class is less or more general than another object class [10, 52].

- *Schematic discrepancy* - Schema construct names in one schema correspond to attribute values in another. We will explain this kind of semantic inconsistency by an example in Section 1.1 below.

Furthermore, in the integration of XML schemas, we should also resolve the inconsistency of hierarchical structures. For example, the same binary relationship type between INSTRUCTOR and COURSE is represented as a path INSTRUCTOR/COURSE in one schema tree, i.e., listing the courses taught by each instruc-

tor, but COURSE/INSTRUCTOR in another, i.e., listing the instructors of each course.

To integrate the schemas of sources in different models (e.g., the relational, object-relational, network or hierarchical model), we should first translate them to the same data model, e.g., the ER model, and then transform the ER schemas to consistent ones in which semantic heterogeneities are resolved. At last, we integrate the transformed schemas by merging the equivalent structures.

In schema transformation, we usually require that the original and transformed schemas represent exactly the same real world facts, although with different modelling constructs. A *semantic preserving* schema transformation is both *information preserving* and *constraint preserving*. Informally, a transformation is information preserving if any instance of the original schema can be losslessly converted into an instance of the transformed schema, and vice versa. A transformation is constraint preserving if the constraints expressed in the original schema can also be expressed in the transformed schema.

In this work, we studied the resolution of schematic discrepancies in the integration of relational or ER schemas, i.e., transforming schematically discrepant schemas into consistent ones. We also studied the derivation of constraints (in particular, an extension to functional dependencies) in schema transformation. This is significant because: (1) a schema transformation should be constraint preserving, and (2) constraints are very useful in multidatabase systems. One of the interesting points is that constraints (i.e., functional dependencies) can be used to verify information preserving schema transformations. Note some semantic rich models (e.g., ER) themselves support (cardinality) constraints. Then the derivation of constraints is involved in schema transformation rather than a separate process.

In the integration of XML schemas, the new challenges come from the hierar-

chical structures of XML. The resolution of some semantic heterogeneities such as naming conflicts and domain mismatches for the flat models (e.g., the relational or ER model) can be adapted to the hierarchical model of XML directly. For some other heterogeneities, e.g., structural conflicts and schematic discrepancies, we should consider the hierarchical structures of XML in the resolution. Furthermore, besides all these heterogeneities, the inconsistency of hierarchical structures may occur alone among XML schemas. Our solution is to separate the resolutions of structural conflicts and schematic discrepancies from the handling of hierarchical structures in the integration of XML schemas. That is, we first resolve the structural conflicts and schematic discrepancies using the resolutions similar to those for the flat models in schema transformations, ignoring the hierarchical characteristics of XML, and then resolve the inconsistencies of hierarchical structures in the integration of the transformed schemas. We will focus on the second stage, i.e., the resolution of the inconsistency of hierarchical structures, in the integration of XML schemas.

In the rest of this section, we first introduce the semantic heterogeneity of schematic discrepancy by an example in relational databases. Then we introduce an extension of functional dependencies in multidatabases. Finally, we present the objectives and organizations of this thesis.

1.1 Schematic discrepancies by examples

In relational databases, schematic discrepancy occurs when the same information is modelled differently as attribute values, relation names or attribute names in different databases, as shown in the example below. For ease of presentation, we assume naming conflicts have been resolved if any. Furthermore, we assume that

the same information is represented in the same form when it is the attribute values, the relation names or the attribute names in databases.

Example 1.1. *In Figure 1.1, we give four databases $DB1$ to $DB4$ recording the same information: supplying prices of products (identified by p#) by suppliers (identified by s#) in different months. In $DB1$, all the information, i.e., product numbers, supplier numbers, months and prices are modelled as attribute values. In $DB2$, the months Jan, ..., Dec are attribute names whose values are prices in those months; in $DB3$, each relation with a month as its name records the supplying information in that month; in $DB4$, each relation with a supplier number as its name records products' prices in each month by that supplier.*

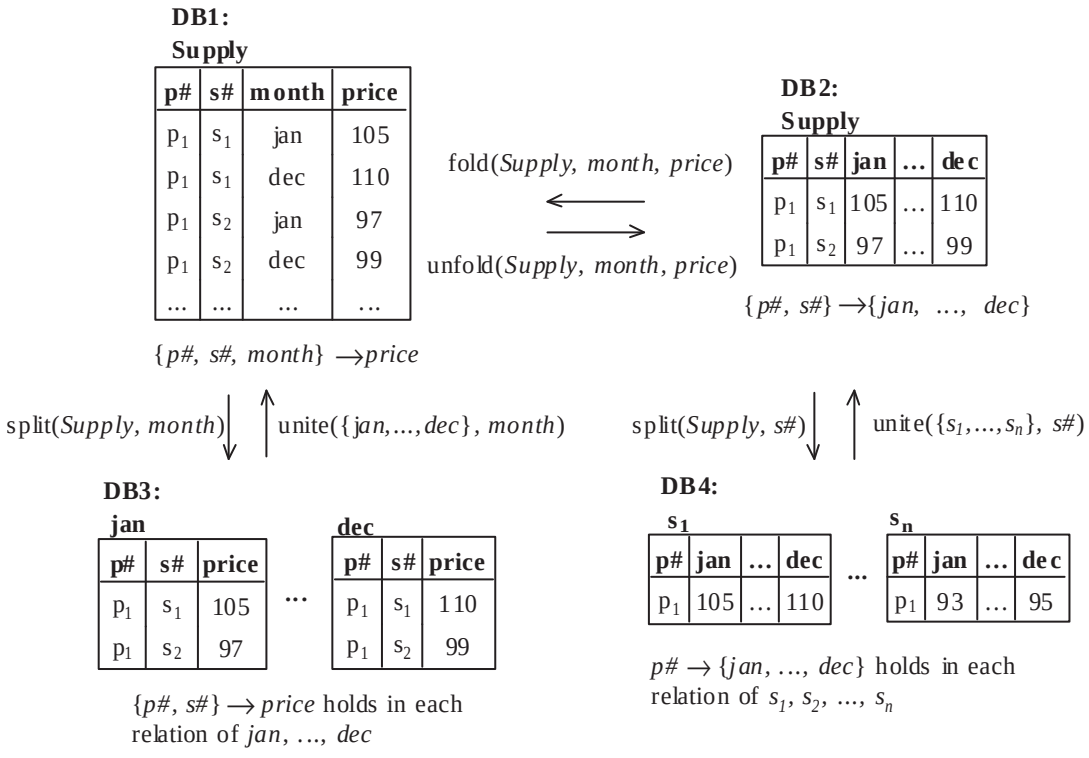


Figure 1.1: Schematic discrepancy: months and supplier numbers are modelled differently in these databases

The schemas of Figure 1.1 are schematically discrepant from each other: the

values of the attribute month in $DB1$ correspond to attribute names of $DB2$ and $DB4$, or relation names of $DB3$, and the values of the attribute s# in $DB1$ correspond to the relation names in $DB4$.

In each database, we assume a product's price is functionally dependent on the product number, supplier number and month. This constraint is expressed as different functional dependencies in these databases: in $DB1$, the constraint is expressed as a functional dependency $\{p\#, s\#, month\} \rightarrow price$; in $DB2$, it is expressed as $\{p\#, s\#\} \rightarrow \{jan, \ldots, dec\}$, i.e., the product numbers and supplier numbers determine the prices of each month; in $DB3$, it is expressed as $\{p\#, s\#\} \rightarrow price$ in each relation, i.e., in each month, the product numbers and supplier numbers determine the prices; in $DB4$, it is expressed as $p\# \rightarrow \{jan, \ldots, dec\}$ in each relation of s_i. □

Schematic discrepancy arises frequently since the names of schema constructs often capture some intuitive semantic information. Some researchers argue that even within the relational model it is common to find data represented in schema constructs. Real examples of such disparity abound [32, 34, 54]. Originally raised as a conflict to be resolved in schema integration, schematically discrepant structures have been used to solve some interesting problems:

- In [54], Miller identified three scenarios in which schematic discrepancies may occur, i.e., database integration, data publication on the web and physical data independence.

- In e-commerce, data are conventionally stored as "horizontal row presentation", i.e., $(Oid, A_1, \ldots, A_n)$ where Oid is the IDs of objects and $A_1, \ldots, A_n$ are the attributes of objects. Agrawal et al. [3] argued that the new generation of e-commerce applications require the data schemas that are constantly

evolving and sparsely populated. The conventional horizontal row representation fails to meet these requirements. They represented objects in a vertical format ($Oid, AttributeName, AttributeValue$) storing an object as a set of tuples. Each tuple consists of an object identifier and attribute name-value pair. They found that a vertical representation of objects is much better on storage and querying performance than the conventional horizontal row representation. On the other hand, to facilitate writing queries, they need to create a logical horizontal view of the vertical representation, and transform queries on this view to the vertical table.

- In data warehousing, users usually require generating report tables (e.g., $DB2$, $DB3$ or $DB4$ of Figure 1.1) which are schematically discrepant from fact data (e.g., $DB1$ of Figure 1.1).

Lakshmanan et al. [34] developed four restructuring operators, *fold*, *unfold*, *unite* and *split* (introduced in Section 3.1 below), to implement transformations between schematically discrepant databases. However, the properties of these operators have not been well studied. Are these operators information preserving and constraint preserving? How to implement a transformation with the minimum number of operators? We will study these problems in this thesis.

Existing work [32, 33, 35] focused on the development of languages with which users can query over schematically discrepant databases. Their work is based on the relational model, and considered a special kind of schematic discrepancy, i.e., relation names or attribute names in one database correspond to data values in another database. A general case may be: a relation name (or attribute name) corresponds to the values of several attributes. For example 1.1, suppose we have another database consisting of a set of relations, such that each relation stores the prices of products supplied by one supplier in one month. That is, each relation

name contains the information of a supplier number and a month. This cannot be handled by previous approaches. We study the issue from the schema-integration point of view. In particular, we will resolve a general issue of schematic discrepancy in the integration of schemas in the ER model that is more complex than the relational model.

1.2 Functional dependencies in multidatabases

Integrity constraints play important roles in not only individual databases, but also multidatabases. The following example shows an application of functional dependency, i.e., a special kind of integrity constraint, in schema and data integration.

Example 1.2. *Suppose we want to integrate two relations of two bookstores* $BS1(isbn, title, price)$ *and* $BS2(isbn, title, price)$. *Suppose in each bookstore, the books with same isbn number have the same title and price, i.e., isbn is the keys of the relations. Can we just integrate them into a schema as* $BS1$ *or* $BS2$*? The answer would be negative if we have the constraint:* a book with an isbn number has the same title but not necessary the same price in the two bookstores. *As value inconsistency would occur on the price attribute for the same book. Actually, the functional dependency* $isbn \rightarrow title$ *is a "global" functional dependency that holds over the union of the two relations* $BS1$ *and* $BS2$*, while the functional dependency* $isbn \rightarrow price$ *is a* $isbn \rightarrow price$ *is a "local" functional dependency holding in individual relations.*

According to these dependencies, it would be better to distinguish a book's prices of the two bookstores in an integrated schema, e.g., $Book(isbn, title, BS1\_price, BS2\_price)$ *with the key* $isbn$*, or* $Book(isbn, title, store, price)$ *with the 2 functional dependencies* $isbn \rightarrow title$ *and* $\{isbn, store\} \rightarrow price$ *(the derivation of functional*

dependencies will be discussed in Chapter 6). We note that the second integrated schema is not in second normal form. It can be normalized into two relations: $Book(isbn, title)$ *and* $BookPrice(isbn, store, price)$.

In conclusion, functional dependencies can be used to detect value inconsistencies and design good integrated schemas, and to normalize integrated schemas. □

Classical functional dependencies are proposed to represent constraints on individual relations, which may be inadequate in multiple, distributed and heterogeneous databases. In this work, we will propose qualified functional dependencies, i.e., the functional dependencies holding over a set of relations or a set of the horizontal partitions of relations, to represent useful constraints in multidatabases. In the following two examples, the constraints cannot be expressed by conventional functional dependencies. However, they can be expressed by qualified functional dependencies.

Example 1.3. *For Example 1.2, the dependency* $isbn \rightarrow title$ *holds over the union of the two relations* $BS1$ *and* $BS2$. *This constraint can be represented as a functional dependency:*

$$\{BS1, BS2\}(isbn \rightarrow title)$$

in which $\{BS1, BS2\}$ *indicates the set of relations over which the dependency holds.* □

Example 1.4. *Given a relation* $Emp(emp\#, name, isMgr, phone\#)$ *that is obtained by integrating a relation of ordinary staff and a relation of managers, such that* $isMgr$ *is a boolean attribute indicating whether an employee is a manager or an ordinary employee, we know that each ordinary employee has one phone, and a manager may have a few. We can the constraint as a qualified functional*

dependency:

$$Emp(emp\#, isMgr_{\sigma=\{`false'\}} \rightarrow phone\#)$$

in which σ means "selection", and $isMgr_{\sigma=\{`false'\}}$ indicates that the dependency only holds over the tuples with $isMgr$ taking the $false$ value. □

In database integration, source databases are usually distributed (i.e., data may be divided and stored in several databases) and heterogeneous (i.e., similar data may be represented in different forms in the source databases). In particular, with schematic discrepancy, schema and data transformations/integrations are usually implemented by not only the relational algebra, but also the restructuring operators (i.e., fold, unfold, unite and split).

The derivation of constraints usually accompanies with schema transformation/integration, i.e., deriving the constraints on the transformed/integrated schemas from the constraints on the source schemas. The inference of view dependencies (i.e., inferring the functional dependencies for view relations from the functional dependencies on original relations) has been studied in [2, 22]. However, in the presence of schematic discrepancy, to derive qualified functional dependencies in schema transformations, the existing inference rules of functional dependencies for the relational algebra are not enough. We need to find rules of qualified functional dependencies for the restructuring operators.

1.3 Objectives and organization

Our objective is to resolve schematic discrepancies in the integration of relational, ER or XML schemas, and to derive/preserve qualified functional dependencies in the transformation and integration of the schemas. For the relational model, we studied the properties of the 4 restructuring operators fold, unfold, unite and

split and the properties of the transformations between schematically discrepant schemas. We also studied the representation, derivation and uses of qualified functional dependencies in schema transformation in multidatabases.

Then we extend the theory of schema transformation and qualified functional dependency in the relational model to the ER model. The new challenges come from the rich semantics of the ER model. In the integration of ER schemas, we should resolve more complex and general schematic discrepancies than the issue in the relational model. Qualified functional dependencies are represented as cardinality constraints in the ER model, and the propagation of cardinality constraints is involved in schema transformation rather than a separate process.

We also extend the resolution of schematic discrepancies in the integration of XML schemas. The new challenges come from the hierarchical structure of XML which is the focus of our study.

In Chapter 2, we introduce two semantic models, i.e., the ER approach for flat data and ORASS approach for XML data. In Chapter 3, we review related work. In Chapter 4, we analyze the knowledge gap of existing work, and state the issues studied in this thesis. The main contribution of this work constitutes of 4 parts (chapters):

1. *The theory of schema transformation in relational databases.* In Chapter 5, we develop a theoretical framework for schema transformation in relational databases by defining formally the properties of restructuring operations and discrepant schema transformations. In particular, we present the reconstructibility and commutativity of the restructuring operators and the lossless-ness and non-redundancy of transformations between schematically discrepant schemas.

2. *Representation, derivation and application of constraints in multidatabases.*

In Chapter 6, we introduce the notion of qualified functional dependency to represent some constraints in multidatabases, and study the inference of qualified functional dependencies in schema transformation. Soundness, completeness and time complexity are proven for the inference rules and algorithms. We also introduce some applications of the derivation of qualified functional dependencies in data integration systems and in a multidatabase language SchemaSQL [35].

3. *Integration of relational databases with schematic discrepancies using the ER model.* In Chapter 7, we propose an approach to the resolution of schematic discrepancy in the integration of ER schemas.

4. *Integration of XML schemas.* In Chapter 8, we propose a semantic approach to the integration of XML schemas, resolving the inconsistencies of the hierarchical structures of source schemas.

Finally, Chapter 9 concludes the whole thesis.

Several portions of this work have been published in some international conferences [24, 25] and journals [26].

This thesis should provide a theoretical work for schema transformation and the inference of constraints in schema transformation. It may help researchers and engineers improve solutions to the interoperability of heterogeneous databases, and be useful in building multidatabases, data warehouses and information integration systems based on XML.

Chapter 2

Preliminaries

Schema integration is usually performed on semantic rich models, e.g., the ER model for relational or other flat data or the Object-Relationship-Attribute model for Semi-Structured data (or ORASS) [43]. The reasons are:

1. A semantic model provides adequate schema constructs (e.g., entity types, relationship types, attributes of entity types and attributes of relationship types in the ER model) to model an enterprise. These schema constructs correspond to real world concepts well. This facilitates the task of schema matching [63].

2. A semantic model supports integrity constraints (e.g., cardinality constraints in in the ER model imply functional dependencies and multivalued dependencies) integration, as we will show later.

In this work, we will study some un-resolved semantic inconsistencies in the integration of ER schemas (i.e., for flat data) and in the integration of ORASS schemas (i.e., for hierarchical data such as XML). We first introduce these two models below.

2.1 ER approach

In the ER model, an entity is an object in the real world and can be distinctly identified. An *entity type* is a collection of similar entities that have the same set of predefined common attributes. An attribute of an entity type can be *single-valued*, i.e., 1:1 (there is a one-to-one mapping from the entities to the attribute values) or m:1 (many-to-one), or *multivalued*, i.e., 1:m (one-to-many) or m:m (many-to-many). A minimal set of attributes of an entity type E whose values uniquely identifies the entities of E is called a *key* of E. An entity type may have more than one key and we designate one of them as the *identifier* of the entity type. A relationship is an association among two or more entities. A *relationship type* is a collection of similar relationships that satisfy a set of predefined common attributes (a relationship type may not have any attributes). A minimal set of the identifiers of some participating entity types in a relationship type R that uniquely identifies the relationships of R is called a *key* of R. A relationship type may have more than one key and we designate one of them as the *identifier* of the relationship type.

The cardinality constraints of ER schemas incorporate functional dependencies and multivalued dependencies. For example, in the ER schema of Figure 2.1, K1, K2 and K3 are the identifiers of entity types E1, E2 and E3, A1 is a one-to-one attribute of E1, A2 is a many-to-one attribute of E2, A3 is a many-to-many attribute of E3, and B is a many-to-one attribute of R. These cardinality constraints are represented as different arrows in the figure. Furthermore, the cardinalities of E1, E2 and E3 in R are m, m and 1 respectively, represented on the edges between the relationship type and the entity types. The cardinality constraints imply the following functional dependencies and multivalued dependencies:

K1 $\rightarrow$ A1 and A1 $\rightarrow$ K1, as A1 is a 1:1 attribute of E1;

K2 → A2, as A2 is a m:1 attribute of E2;

K3 ↠ A3, as A3 is a m:m attribute of E3;

K1, K2 → K3, as {K1, K2} is the identifier of the relationship type R, and the cardinality of E3 is 1 in R;

K1, K2 → B, as B is a m:1 attribute of R.

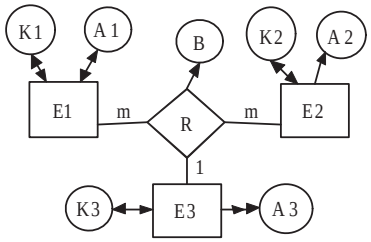


Figure 2.1: Dependencies in ER schema

2.2 ORASS approach

In this thesis, we adopt the semantic model, ORASS, to represent XML schemas. ORASS has four kinds of schema constructs:

1. *object class*, i.e., a set of entities in the real world, like an entity type in an ER diagram, a class in an object-oriented diagram, or an element in a semi-structured data model. An object class is characterized by a name.

2. *relationship type*, i.e., a set of relationships among the objects of some classes. A relationship type in the ORASS data model represents a nesting relationship. Each relationship type has a degree and participation constraints. The *degree* of a relationship type is the number of the object classes involved in the relationship type.

3. *attribute of object class*, i.e., a property of an object class. One of the features that distinguishes semi-structured data from structured data is that not all object classes are expected to have the same set of attributes, and because of this the attributes of objects are heterogeneous.

4. *attribute of relationship type*, i.e., a property of a relationship type.

With ORASS, an XML schema is represented as a tree structure with object classes as rectangles and attributes as circles (filled circles denote the *identifiers* of the owning object classes). A relationship type among object classes is specified on the last edge in the path linking those object classes. The XML data instance can be modeled using an ORASS instance diagram. The ORASS instance diagram has labeled rectangles for object instances, labeled circles for attribute and their associated data, and the edges represent relationship instances.

In the following example, we explain an ORASS schema diagram and its instance diagram.

Example 2.1. *The schema of Figure 2.2 models the supply information of products supplied by some suppliers in some months.*

In Figure 2.2, the three rectangles SUPPLIER, PROD and MONTH represent three object classes. The label "SPM, 3" on the edge from PROD to MONTH means that the 3 object classes SUPPLIER, PROD and MONTH constitute a ternary relationship type SPM. Attributes under an object class may belong to the object class or a relationship type, e.g., the attribute M# is an identifier of the object class MONTH, while PRICE is an attribute of the relationship type SPM (this is indicated by the label "SPM" on the edge from the object class MONTH to the attribute PRICE).

Figure 2.3 shows an instance (consisting of 3 relationships of SPM) of the

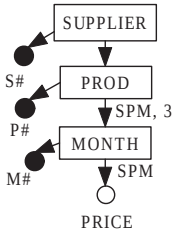


Figure 2.2: ORASS schema diagram

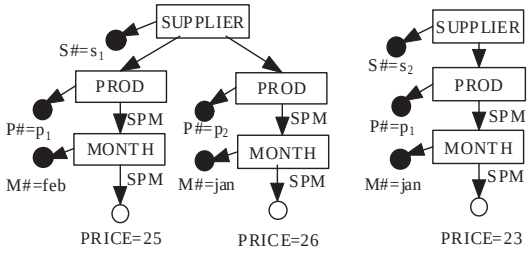


Figure 2.3: ORASS instance diagram

```
<!ELEMENT SUPPLIER (PROD*)>
<!ELEMENT PROD (MONTH*)>
<!ELEMENT MONTH EMPTY>

<!ATTLIST SUPPLIER S# ID #REQUIRED>
<!ATTLIST PROD P# CDATA #REQUIRED>
<!ATTLIST MONTH M# CDATA #REQUIRED>
<!ATTLIST MONTH PRICE CDATA #REQUIRED>
```

```
<SUPPLIER S# = "s1">
   <PROD P# = "p1">
      <MONTH M# = "feb" PRICE = "25" />
   </PROD>
   <PROD P# = "p2" />
      <MONTH M# = "jan" PRICE = "26" />
   </PROD>
</SUPPLIER>
<SUPPLIER S# = "s2">
   <PROD P# = "p1">
      <MONTH M# = "jan" PRICE = "23" />
   </PROD>
</SUPPLIER>
```

Figure 2.4: Corresponding DTD and XML document sections

schema of Figure 2.2. Figure 2.4 gives the corresponding DTD and XML document sections of Figure 2.2 and 2.3. □

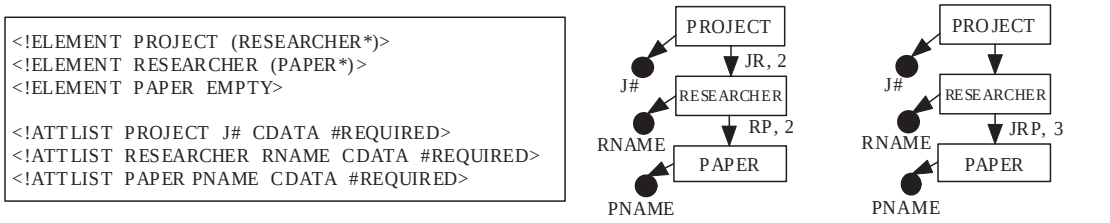
The *participation constraints* of the object classes in a relationship type and the *quantifiers* of attributes (i.e., the symbol ? represents an optional attribute, + represents the number of an attribute can be one to many, and * represents the number of an attribute can be zero to many) can be specified in ORASS. However, we omit them here, as the resolution of constraint conflicts can be adapted from the resolution of constraint conflicts for ER schemas, and therefore is not the focus of our work. In ORASS, a *reference* (represented as a dashed arrow in a diagram) links 2 object classes, representing a foreign key constraint.

Comparing with ORASS, DTD and XML Schema [1] do not provide much semantics for effective schema integration, i.e.,

1. DTD/XML Schema can only express generic binary relationships between elements and child-elements, while ORASS can express specific relationships with any degree.

   In practice, XML data may contain high degree relationships among the elements in a path, such as the ternary relationship type SPM of Figure 2.2. Note in general, a high degree relationship type could not be losslessly decomposed into a set of binary relationship types, unless it satisfies the condition (i.e., some multivalued dependencies) of "lossless join decomposition". For example, in Figure 2.3, the ternary relationships of SPM cannot be losslessly decomposed into the binary relationships of SP (between SUPPLIER and PROD) and PM (between PROD and MONTH).

2. DTD/XML Schema does not explicitly represent relationship types. This may cause some ambiguity.



(a) DTD (b) two binary relationship types (c) a ternary relationship type

Figure 2.5: an ambiguous DTD corresponding to two ORASS schemas

For example, the DTD of Figure 2.5 (a) can be interpreted in two ways: (1) for each project, list all the project members; for each project member, list all his papers; (2) for each project and each member of the project, list all the papers of the project written by the project member.

This is not a problem in ORASS, as we explicitly represent relationship types in an ORASS schema. For example, the two interpretations of the DTD of Figure 2.5 (a) would be represented as two different schemas of Figure 2.5 (b) and (c) in ORASS. One has two binary relationship types JR and RP, and the other one has a ternary relationship type JRP.

3. DTD/XML Schema does not distinguish attributes of relationship types from attributes of object classes, although this kind of information is necessary in schema transformation.

   For example, in Figure 2.2, PRICE is an attribute of the ternary relationship type SPM, i.e., the values of PRICE are determined by the supplier numbers, product numbers and months. In schema transformation, when swapping PROD and MONTH in Figure 2.2, if we do not know that PRICE is an attribute of the relationship type SPM (note that DTD/XML Schema cannot express PRICE as an attribute of a relationship type), we may attach it to the object class MONTH during the swap. Then in the trans-

formed schema (path) SUPPLIER/MONTH/PROD, PRICE becomes an attribute of MONTH or of the binary relationship type between SUPPLIER and MONTH, which is wrong. ORASS explicitly indicates the attributes of object classes and the attributes of relationship types.

4. The ID attribute of DTD assigns a unique identifier to an element, which is unique in a document. The key element of XML Schema is an extension of the ID in DTD, such that it must have a unique value, and must be present.

   The ID of DTD (or the key of XML Schema) cannot be used to identify entities (or objects) in the real world. For example, in Figure 2.3, part p_1 is supplied by two suppliers s_1 and s_2, and there are two PROD elements with the same P# value p_1, so P# is not unique within the selected PROD elements. Therefore we cannot define P# as an ID attribute in the DTD in Figure 2.4 (or as a key in the XML Schema).

   In order to integrate data in schema integration, we need to know some "semantic identifiers" of object classes, e.g., social security numbers of people, which identify entities in the real world. The identifiers of object classes in ORASS are such semantic identifiers.

5. In DTD, the type of an element is defined by the element name and the types of the sub-elements. The nesting definition of element types makes it costly to identify equivalent elements which should have the same name and sub-elements. Similarly, in XML Schema, complex types are defined in a nesting way, which are decoupled from element names. However, it would not be a problem for ORASS in which the description of an object class is self-content, independent of the descendent object classes. The underlying reason of this difference is that DTD/XML Schema only support generic (composite,

binary) relationships among an element and its sub-elements, while ORASS can express specific relationships among object classes.

Actually, ORASS and DTD/XML Schema model information at different levels for different purposes. ORASS is a conceptual model (like the ER approach) for the design of semi-structured database [43], the integration of XML schemas [74], XML view support [11, 12, 13, 46], XML graphical language [56, 57], and the design of functional dependencies for XML [40]. On the other hand, DTD/XML Schema is a formal, structural definition for the validation of XML data.

Some concepts of ORASS, e.g., relationship types and attributes of relationship types, are adapted from the ER approach. However, ORASS is different from the ER approach:

1. In ORASS, the object classes of a relationship type are ordered in a hierarchical path. The hierarchical structure of ORASS brings some challenges in schema integration. ER is a flat model in which the entity types of a relationship type are at the same level, and cannot represent the hierarchical structure of XML data.

2. In ORASS, relationship types are represented on the edges of hierarchical structures instead of as particular constructs, and the attributes of a relationship type are attached to the lowest object class in the relationship type (as we do not have particular constructs for relationship types). It becomes tricky to preserve the information of relationship types and the attributes of relationship types in the transformation of ORASS schemas. However, this is not a problem in the transformation of ER schemas.

The difference between ORASS and the nested relational model lies in the semi-structuredness of ORASS. In an ORASS diagram, not all objects of the same class

are expected to have the same set of child objects and attributes.

Chapter 3

Literature review

We review some related work in this chapter, and analyze the knowledge gaps and state our research problems in the next chapter.

3.1 Restructuring operators and discrepant schema transformation

For the integration of schematically discrepant databases and the other applications of schematically discrepant structures introduced in Section 1.1, people need transformations in the context of schematic discrepancy. Lakshmanan et al. [34] developed four restructuring operators, fold, unfold, unite and split (originally introduced in the context of the tabular algebra [23]), to transform relations.

For example, in Figure 1.1, these restructuring operations [1] are used to transform between the 4 databases $DB1$ to $DB4$. Intuitively, unfold makes attribute values become attribute names; fold is the reverse of unfold. Split horizontally partitions a relation on the values of an attribute; unite is the reverse of split. The formal definitions of the four operators are given below, as adapted from [34]:

[1] an operation is an operator with necessary parameters.

- *unfold(R, B, C).* Let R be a relation with the schema $R(A_1, \ldots, A_n, B, C)$, and $A_1, \ldots, A_n$, B and C be the attributes of R. The operation unfold(R, B, C) transforms R to a relation $S(A_1, \ldots, A_n, b_1, \ldots, b_m)$, where $b_1, \ldots, b_m$ are the distinct values appearing in the column B of R. The content of S is defined as:

$$S = \{(a_1, \ldots, a_n, c_1, \ldots, c_m) | (a_1, \ldots, a_n, b_i, c_i) \in R, 1 \leq i \leq m\}.$$

- *fold(R, B, C).* Let R be a relation with the schema $R(A_1, \ldots, A_n, b_1, \ldots, b_m)$. Suppose the attribute names $b_1, \ldots, b_m$ are the values in $dom(B)$, i.e., the domain of attribute B, and all the entries appearing in the columns $b_1, \ldots, b_m$ of R are from $dom(C)$, for some attribute names $B, C \notin \{A_1, \ldots, A_n\}$. The operation fold($R$, B, C) transforms R to a relation $S(A_1, \ldots, A_n, B, C)$, defined as:

$$S = \{(a_1, \ldots, a_n, b_i, c_i) | \exists t \in R : t[A_1, \ldots, A_n] = (a_1, \ldots, a_n) \ \& \ t[b_i] = c_i\}.$$

- *split(R, B).* Let R be a relation with the schema $R(A_1, \ldots, A_n, B)$. The operation split(R, B) transforms R to a set of relations $b_i(A_1, \ldots, A_n)$, for each b_i appearing in the column B of R. The content of b_i is defined as:

$$b_i = \{t[A_1, \ldots, A_n] | t \in R \ \& \ t[B] = b_i\}.$$

- *unite(ℝ, B).* Let $\mathbb{R} = \{b_1, b_2, \ldots, b_m\}$ be a set of relations in a given database, such that each relation name b_i $(i = 1, 2, \ldots, m)$ is an element of the domain of some fixed attribute B, and each relation has the schema $b_i(A_1, \ldots, A_n)$. The operation unite($\mathbb{R}$, B) transforms the set of the relations

$\{b_1, \ldots, b_m\}$ into a relation $S(B, A_1, \ldots, A_n)$, defined as:

$$S = \{t | \exists t' \in b_i : t[A_1, \ldots, A_n] = t'[A_1, \ldots, A_n] \ \& \ t[B] = b_i\}.$$

For example, in Figure 1.1, we can transform $DB1$ to $DB4$ in two steps: first transform $DB1$ to $DB2$ with an operation unfold$(Supply, month, price)$, then transform $DB2$ to $DB4$ with an operation split$(Supply, s\#)$. In general, we have:

Definition 3.1. *A* discrepant schema transformation *is a transformation consisting of a sequence of restructuring operations.* □

A discrepant schema transformation transforms a relation (or a set of relations) R to one (or a set of relations) S, such that R and S are schematically discrepant from each other. Note that each step of a discrepant schema transformation may comprise one restructuring operation or a set of (fold or unfold) operations.

For example, in Figure 3.1, we may transform $DB4$ (in Figure 1.1) to $DB5$ with a set of operations $\{\text{fold}(s_i, month, price) | \ i = 1, 2, \ldots, n\}$, such that each fold operation transforms one relation s_i of $DB4$ to the corresponding relation of $DB5$.

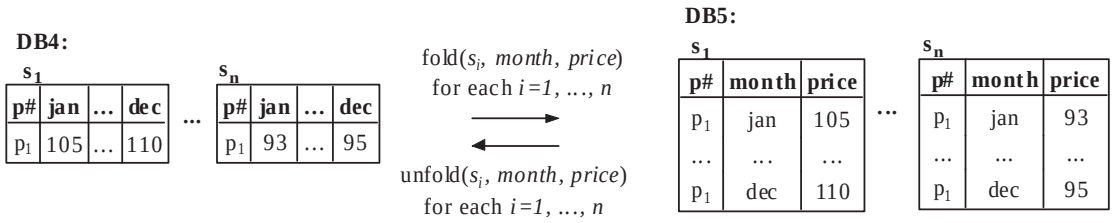


Figure 3.1: Transforming $DB4$ to $DB5$ with a set of fold operations, and the converse with a set of unfold operations

In general, schema and data transformations in relational databases can be implemented by the restructuring operators and the relational algebra (i.e., selection, projection, join and union) [34].

3.2 Data dependencies and the derivation of constraints in schema transformation

An extension to functional dependencies in the database design world are the functional dependencies that partially hold in a relation, in the sense that only some tuples, called exceptions, break the dependencies. These dependencies include "weak functional dependencies" [42], "afunctional dependencies" [9, 8] and "partial functional dependencies" [20]. A horizontal decomposition through a functional dependency is accomplished using the concept of exception. The usual way to do this is relaxing the functional dependency in order to obtain a sub-relation verifying the dependency, and isolating the exceptions to that dependency in a different relation.

In individual relations, the previous work is similar to ours in the sense that either a weak functional dependency (or some other similar dependencies) or a qualified functional dependency may hold over a a sub-relation. Based on qualified functional dependencies, we can also develop a theory of horizontal decomposition (which would be similar to split operations).

However, qualified functional dependency is more precise and general than the previous work:

1. Qualified functional dependencies are quantitative while the dependencies of the previous work are qualitative. That is, a weak functional dependency (or some other similar dependencies) predicates that some tuples (but do not know which tuples) in the relation would violate the functional dependency, while a qualified functional dependency indicates exactly what kind of tuples in a relation (or in a set of relations) satisfy a functional dependency.

2. Qualified functional dependency is more general than the previous work. A weak functional dependency (or some other similar dependencies) holds over

a sub-relation, while a qualified functional dependency may hold over a set of sub-relations. This is because the previous dependencies were proposed for database design purpose, not for the representation of constraints in multi-databases.

Further more, the schema transformations (i.e., split, unite, unfold and fold) based on qualified functional dependencies are more extensive than the schema transformations (i.e., horizontal decomposition which is similar to split) based on weak functional dependencies, partial functional dependencies etc.

We give the sound and complete sets of inference rules and propagation rules of qualified functional dependencies. We are not aware of any complete axiomatizations for the dependencies of the previous work [20, 9, 8, 42].

Most of the existing relational dependencies, such as functional dependencies, multivalued dependencies, embedded multivalued dependencies or join dependencies, were defined on individual relations. Researchers have proposed some unifying frameworks which provide general perspectives on those dependencies. One of the most powerful methods is to use "tableaux" (a table form representation) to present dependencies, and use "chase" (a procedure based on the successive application of constraints to tableaux) to analyze implication and construct axiomatization [2].

Example 3.1. *Given a relational schema (A, B, C), let $A \rightarrow B$ and $B \rightarrow C$ be two functional dependencies on it, we want to know whether a functional dependency $A \rightarrow C$ holds on the relational schema. In Figure 3.2, we apply the two given functional dependencies on the tabular representation of the relation in sequence, and get Figure 3.2 (c) in which the two tuples with the same A value also have the same C value. It means that the functional dependency $A \rightarrow C$ holds on the relational schema. The application of functional dependencies to a tabular representation is actually a procedure of implication of functional dependencies.*

A	B	C
x	y	z
x	y'	z'

(a) The tabular representation

A	B	C
x	y	z
x	y	z'

(b) Applying the functional dependency $A \rightarrow B$

A	B	C
x	y	z
x	y	z

(c) Applying the functional dependency $B \rightarrow C$

Figure 3.2: Illustration of the chase

□

The current framework of tableaux and chase unify the representation and implication of a range of dependencies in individual databases. They cannot represent and imply the constraints in multidatabases such as Examples 1.3 and 1.4 which may hold over a set of relations and have restrictions on attribute values.

The issue of inferring view dependencies (i.e., inferring the functional dependencies or multivalued dependencies for view relations from the dependencies on original relations) was introduced in [2, 22]. In [2], Abiteboul et al. proved that if the dependencies are functional dependencies and multivalued dependencies and the view is defined by an SPCU expression (i.e., using the operations of selection, projection, cross-product and union), testing the implication of a view dependency can be done in polynomial time. In [22], Gottlob proposed an efficient algorithm to compute covers for the functional dependencies embedded in a subset of a given relational schema (i.e., the functional dependencies for a projection of the original relation).

In multidatabases, the classical functional dependencies are not enough to represent the constraints, and we need to extend them to qualified functional dependencies, as shown in Section 1.2. Schema transformations in multidatabases contain not only the relational algebra, but also the restructuring operators of fold, unfold, unite and split. Therefore, the inference of view dependencies in individual

databases should be extended to multidatabases.

Some work has been done in the derivation of the constraints for an integrated schema from the constraints on component schemas in schema integration. Their work was based on semantic rich models, e.g., the object-oriented approach [64, 72] or the ER approach [38].

In [64], Reddy et al. integrated object schemas by merging equivalent object classes. Domain mismatches and naming conflicts among equivalent object classes were resolved. They derived global constraints for the integrated schema during schema integration. The constraints are represented as "production rules", which subsume two kinds of constraints:

Intra-object-constraints describe the relationships among the properties of a single object class. This kind of constraints include single-property constraints (i.e., domain constraints) and multi-property constraints, e.g., a teacher with the designation 'professor' must receive a salary exceeding '90k'.

Inter-object-constraints describe the relationships among two or more different object classes. In particular, given two object classes $O1$ and $O2$, one of four relationships exists between the instance sets of them: equivalence, subclass-superclass, disjoint and overlapping. For example, a dean must be a member of the teaching faculty, i.e., the object class DEAN is a subclass of another class FACULTY.

In [71], Vermeer and Apers proposed an instance-based database integration paradigm, where objects rather than classes are the units of integration. They resolved naming conflicts, domain mismatches and object class-attribute conflicts, and built class hierarchies in the integration.

Based on the instance-based approach to database integration, Vermeer and

Apers defined some necessary conditions under which the derivation of global constraints is possible [72]. They use some object-oriented specification language to express constraints on an object-oriented database. The constraints include the intra-object-constraints mentioned above, key constraints and inclusion dependencies. Furthermore, they identified two roles of ICs in database integration. First, a set of ICs describing the valid states of an integrated view can be derived from the constraints defined on underlying databases. Second, ICs can be used as a semantic check on the validity of the specification of an integrated view.

The work of [64, 72] did not consider schematic discrepancy in schema integration and constraints such as qualified functional dependencies in constraint derivation.

In the integration of ER schemas, Lee and Ling [38] resolved many constraint conflicts in merging equivalent schema constructs, i.e., domain mismatch, value inconsistency (i.e., conflict in attribute values) and cardinality conflict. Cardinality conflict could occur on attributes or relationship types: attribute conflict occurs when two semantically equivalent attributes do not have the same cardinalities; relationship conflict occurs when the same participating entity types of a relationship type have different cardinalities in different databases.

This work assumed that naming conflicts, key conflicts, structural conflicts and schematic discrepancies among component schemas have been resolved. They resolved constraint conflicts etc in the last step of schema integration, i.e., merging equivalent schema constructs. However, they did not study how to obtain the constraints on the transformed schemas from the constraints on the original schemas in the transformation of component schemas to resolve schematic discrepancy, structural conflicts, etc.

3.3 Resolution of structural conflicts in the integration of ER schemas

Previous approaches [4, 36, 67] enumerated the following types of structural conflicts in the integration of ER schemas: (1) an entity type in one schema is modelled as an attribute of an entity type or a relationship type in another schema; (2) an entity type in one schema is modelled as a relationship type in another schema; (3) a relationship type in one schema is modelled as an attribute of an entity type or a relationship type in another schema; (4) an attribute of a relationship type is modelled as an attribute of an entity type.

Moreover, Lee and Ling [39] found that if the individual schemas have been designed properly and the semantic equivalences among the schemas identified correctly, then the main structural conflict is that between an entity type and an attribute, i.e., the first type of conflicts mentioned above. They gave an algorithm to transform an attribute in one schema into an entity type in another schema without any loss of semantics. The rest of the conflicts are automatically resolved after they had resolved the first type of conflicts.

3.4 XML schema integration and data integration

Some work in XML schema integration was based on DTDs. In [29], Jeong and Hsu applied a tree grammar inference technique to generate a set of tree grammar rules (i.e., corresponding to an integrated schema) from source DTDs. In [65], Rodriguez-Gianolli and Mylopoulos devised a semantic approach to integrate a set of DTDs into a common conceptual schema. They merged the common elements and built

ISA and inter-schema relationships among the elements of different source schemas. Because of the inadequacy of DTD in XML schema integration (see Section 2.2), the integration approaches based on DTD may cause some problems. For example, they did not consider the relationship types involving 3 or more elements, and did not distinguish attributes of relationship types from those of elements.

As to the integration of ORASS schemas, Yang et al. [74, 75] resolved attribute-object conflicts (i.e., attributes in one schema correspond to object classes in another), classification inconsistencies (i.e., an object class in one schema is the union of several object classes in another) and ancestor-descendant conflicts (i.e., an object class is a parent of another class in one schema, but the converse in another schema) in the integration of ORASS schemas. They assigned each source schema a weight of importance, and tried to keep the characteristics of source schemas with larger weights in the integrated schema. This work considered more semantics (e.g., relationship types with high degrees and attributes of relationship types) than the work on DTD. However, as they treated object classes rather than relationship types as the semantic units of integration, they cannot ensure that the information of relationship types can be preserved in the integration. The criterion of "weight" is too vague sometimes, taking some objective and specific criteria may help the automatization of the integration process.

In XML schema integration, an integrated schema would be semi-structured because source schemas would be not only semi-structured by themselves, but also heterogeneous from each other. In particular, in source schemas, equivalent object classes may have either different attributes and sub object classes, or the same attributes and sub object classes but with different participation constraints. In [44], Liu and Ling presented a data model to represent partial and inconsistent data, and defined the operators of union, intersection and difference on the semi-structured

data. The union operation combines sets of semi-structured data and records inconsistency in the meantime, the intersection operation finds common information in sets of semi-structured data and indicates inconsistency in the meantime, while the difference operation finds the information in the first set of semi-structured data but not in the second set.

In some XML data integration systems, e.g., Xyleme [17], Nimble [18], LoPiX [49] and YAT [14, 15], the developers either provide an XML query language for users to write integrated schemas by hand, or assume that an integrated schema and the mapping from source schemas to the integrated schema have been given already. They focused on query processing through the integrated schema.

Some work focused on the translation/integration of relational data into XML. For Clio [61], a schema mapping tool, Popa et al. presented a framework for mapping between any combination of XML and relational schemas, in which a high-level, user-specified mapping is translated into semantically meaningful queries that transform source relational data into the target XML data. The transformed data satisfy the constraints and structure of the target schema, and preserve the semantics of the source. In [5], Benedikt et al. proposed a framework for integrating data from multiple relational sources into an XML document that both conforms to a given DTD and satisfies predefined XML constraints. They proposed a formalism, called "Attribute Integration Grammar (AIG)" with which users can specify data transformation rules to compute XML data from relational databases.

In the work of [5, 61], users specify a transformed/integrated XML schema and mappings from source relational schema(s) to the target XML schema, and then the systems automatically transform/integrate the data according to the mappings. In other words, they focused on data transformation/integration instead of schema transformation/integration.

3.5 Ontology merging

On the Semantic Web, users annotate data using ontology languages, e.g., RDF [37] and OWL [66]. Ontology was proposed to standardize the specifications of data and knowledge. However, in fact, it is very hard to create "standard" ontologies shared by all individuals and organizations on the Semantic Web, and it can be expected that many different ontologies will appear [70, 73]. Then in order to enable inter-operation, mediation is required between these ontologies. In [16], Bruijn et al. presented three use cases which capture the functionality required for ontology mediation on the Semantic Web, i.e., creating ontology mapping, ontology merging, and instance mediation.

In ontology, data is classified into classes and related in triplets consisting of *subjects*, *predicates* and *objects*. Under this way, people can specify attributes of classes, or binary relationships (e.g., *ISA*) between classes. In general, ontology can be regarded as a conceptual model (such as the ER approach) to explicitly represent the semantics (such as classifications, relationships and attributes of classes) of data for the understandability of both machines and humans. As in schema integration, we need to solve some semantic heterogeneities in ontology merging and instance mediation, such as naming conflicts, domain mismatch and class-hierarchy inconsistencies. The resolutions to these issues in schema integration can be adapted to solve the similar issues in ontology merging.

Existing work in ontology matching and merging focused on the identification and merging of similar classes, similar attributes and similar (binary) relationships between classes and between classes and attributes. Such methods and tools include GLUE [19], a system which employs machine learning technologies to semi-automatically create mappings between heterogeneous ontologies, RDFT [59], an approach to the integration of product information over the Web by exploiting

the data model of RDF, and PROMPT [58], an interactive ontology merging tool. Some other tools also consider more complex matching information. For example, MAFRA [47] handles the correspondences between classes and attributes, or between attributes and relationships. MOMIS [6, 7, 10] is a typical mediation system to integrate heterogeneous data sources. Given a set of disparate data sources, MOMIS first wraps the disparate schemas in an object-oriented language, then creates intra- and inter-schema relationships (i.e., synonyms and hypo/hypernyms). Then a global ontology is obtained by uniting similar classes of the local ontologies. A query to the global ontology can be translated to the local representation because of the mapping rules between the local and global ontologies.

Ontology is used to represent semantics, and does not support hierarchies of XML.

3.6 Model management

In [51], Melnik et al. proposed a programming platform "Rondo" for generic model-management system, in which high-level operators are used to manipulate models and mappings between models. Their goal was to reduce the amount of programming required for the development of metadata-intensive applications, e.g., database design, data integration, data translation, model-driven web site management and data warehousing. In that work, they represented relational or XML schemas as directed labelled graphs, and defined the operator Merge to combine two schemas into one. This is not as simple as set union because the schemas have structures, so the semantics of duplicates and duplicate removal may be complex. The implementation of Merge consists of three steps:

1. node renaming, i.e., taking the same name for the equivalent schema con-

structs.

2. graph union, i.e., merging equivalent nodes and edges in the graphs of different schemas. This step may produce some conflicts, e.g., an attribute belonging to two relations.

3. conflict resolution, e.g., deleting the relationship (edge) between a relation and an attribute if the attribute belongs to two relations. This step requires human feedback. Heuristic was developed to select the edges to be deleted when conflicts occur.

When applying this work to XML schema integration, they treated XML elements as semantic units, and only considered binary relationships between elements and between elements and attributes. When a conflict occurs, they resolved it using the heuristic. However, it cannot be ensured that the Merge operation is information preserving. The Merge operation integrates two schemas at once.

More specifically, Pottinger and Bernstein [62] examined the problem of merging two schemas given correspondences between them. They resolved structural conflicts, data model conflicts, and the conflicts of domain constraints and cardinality constraints. In application of their algorithm in XML schema integration, they first merged equivalent elements and equivalent attributes of different schemas, then created (binary, directed, kinded) relationships between the elements and between the elements and attributes. Finally, they removed the redundant relationships that can be implied by others. Again, this work only handled binary relationships, and merged two schemas at once.

Chapter 4

Knowledge gaps and research problems

4.1 Theory of discrepant schema transformation

In data integration, source databases are usually distributed (i.e., data may be divided and stored in several databases) and heterogeneous (i.e., similar data may be represented in different forms in the source databases). In particular, schematic discrepancy is a common kind of heterogeneity among databases. Schema and data transformations/integrations are usually implemented by not only the relational algebra, but also the 4 restructuring operators (introduced in Section 3.1). The theory of the relational algebra (such as algebra law and lossless schema transformation) has been well studied. However, the theory of the restructuring operators was less studied, although they were used to solve many interesting problems in database integration, data publication on the web and physical data independence [34, 54].

Our purpose is:

1. to study the algebraic laws of the restructuring operators.

2. to study lossless and non-redundant discrepant schema transformations.

Our work focused on pure discrepant schema transformations. It can be extended to general transformations (consisting of the relational algebra and restructuring operations) using the existing results. The restructuring operators for the relational model can also be extended to the ER model and the hierarchical model of XML. This is not trivial as the schema constructs of the ER model are more complex, and the hierarchical structures of XML bring new challenges in the resolution, as discussed in Sections 4.3 and 4.4 later.

4.2 Representing, deriving and using dependencies in schema transformation

functional dependencies are designed to represent constraints in a single database, which are inadequate in a multidatabase environment. To represent and infer data dependencies in schema transformation, we extend functional dependencies to qualified functional dependencies, i.e., the functional dependencies holding over a set of relations or a set of the horizontal partitions of relations, to represent useful constraints in multidatabases, as shown in Example 1.3 and 1.4.

Weak functional dependencies (or partial functional dependencies, or some other similar dependencies) are proposed for the purpose of database design, and incapable to represent the constraints such as qualified functional dependencies in multidatabases. The reason is that weak functional dependencies are specified on individual relations, and do not exactly state which tuples satisfy the dependencies. Qualified functional dependencies cannot be represented and implied in the framework of tableaux and chase [2] either, as a qualified functional dependency may hold over a set of relations and have restrictions on attribute values.

As mentioned, to integrate multiple, distributed and heterogeneous databases,

we may use schema transformations implemented by not only the relational algebra, but also the restructuring operators to resolve schematic discrepancy. Consequently, to derive qualified functional dependencies in such transformations, the inference rules for functional dependencies and the relational algebra [2, 22] are not enough. In particular, we need rules to infer qualified functional dependencies for the restructuring operators.

Some work [64, 72] has been done on the derivation of the constraints for an integrated schema from the constraints on component schemas in schema integration. However, they did not consider schematic discrepancy among component schemas in schema integration. Their representations of constraints do not subsume qualified functional dependencies. They did not prove the completeness and complexity of their methods.

In the resolution of constraint conflicts in the integration of ER schemas, Lee et at [38] assume that all the semantic heterogeneities except constraint conflicts have been resolved and the component schemas are consistent. Our work complements theirs when we apply our theory of qualified functional dependency derivation in the integration of ER schemas. That is, in the presence of schematic discrepancy among source schemas, we first use our theory to derive the dependencies on the intermediate transformed schemas which are transformed from the source schemas with restructuring operations, and then resolve the constraint conflicts in merging the transformed schemas using the approaches of [38].

In summary, our purpose is:

1. to define qualified functional dependencies to represent interesting constraints in a multidatabase environment, and to find a sound and complete set of inference rules to infer unknown qualified functional dependencies from known ones in fixed relational schemas.

2. to study the propagation of qualified functional dependencies in discrepant schema transformations. That is, in a discrepant schema transformation, given a set of qualified functional dependencies on the original relations, we should derive a cover of the qualified functional dependencies on the transformed relations. The derivation rules and algorithms should be sound, complete and efficient. Note although we focus on discrepant schema transformations, the result can be extended to general schema transformations using the existing results of [2, 22], i.e., the inference rules of functional dependencies for the relational algebra.

3. to apply our theory to improve the multidatabase interoperability. In particular, we will verify the uniqueness of SchemaSQL views by deriving qualified functional dependencies, and explore some usages of qualified functional dependency derivation in data integration systems. The usages include the verification of lossless transformation, the normalization of integrated schemas, the detection of duplicates and inconsistencies, etc.

4.3 Resolving schematic discrepancies in the integration of ER schemas

Previous work in the integration of ER schemas [4, 36, 67, 38] has concentrated mostly on the resolution of naming conflicts, key conflicts, structural conflicts and constraint conflicts. A special kind of schematic discrepancy has been studied in the interoperation of relational databases [30, 32, 35]. They dealt with the discrepancy of relation names or attribute names in one database corresponding to attribute values in another. A general issue is yet to be solved. The ER model contains the rich schema constructs of entity types, relationship types, attributes of entity types

and attributes of relationship types, compared to the simple relational model. This causes a diversity of schematic discrepancies on ER schemas. To resolve schematic discrepancies in the ER model, we need to extend the resolution for the relational model.

Furthermore, the work on the relational model is at the "structure level", i.e., they only transformed the structures of schemas and data, but did not consider the constraint issue in the resolution of schematic discrepancies. However, the importance of constraints can never be overestimated in both individual and multidatabase systems. As the ER model supports cardinality constraints inherently, we should transform the constraints in the transformation and integration of ER schemas.

In summary, our purpose is:

1. to represent the meta information of schemas in a formal way.

2. to resolve schematic discrepancy in the integration of ER schemas.

3. to preserve the semantics, i.e., information and constraints, in schema transformation.

4. to study the propagation of cardinality constraints in the transformation of ER schemas. This should be an extension of the theory of qualified functional dependency derivation in the ER model.

The resolution of schematic discrepancy for ER schemas can be extended to XML schemas in the ORASS model. However, this is not trivial because of the hierarchical structure of XML. In this work, we propose to separate the resolution of schematic discrepancies and the handling of hierarchical structures in the integration of XML schemas in ORASS. That is, we resolve schematic discrepancy by transforming source XML schemas, using the resolution similar to that for

ER schemas, and then merge the transformed schemas, resolving the hierarchical inconsistencies among them.

4.4 Resolving hierarchical inconsistency in the integration of XML schemas

In some XML data integration systems, e.g., Xyleme [17], Nimble [18], LoPiX [49] and YAT [14, 15], the developers either provided an XML query language for users to write mediated schemas by hand, or assumed that a mediated schema and the mapping from source schemas to the mediated schema have been given already. They focused on query processing through a mediated schema instead of schema integration. However, it is usually not a simple task to integrate disparate schemas by hand, given different kinds of semantic heterogeneities. Schema integration can be regarded as a preceding component of theirs, which generates a mediated schema and the mappings (semi-)automatically. Furthermore, it is especially useful for some applications:

1. Schema integration is the only task, e.g., in XML database design and XML schema repository which provides a centralized management of XML schemas to track the usage of schemas, and avoid proliferating redundant schemas.

2. In some other applications, e.g., the cache of the search engines on the Web, there are so many sources on the Web that it is hard to integrate them by hand.

Some work in schema matching [63] focused on the detection of equivalent elements and structures of source schemas using machine learning or IR techniques.

Schema integration typically follows schema matching to reduce human participation.

In the previous work in the integration of flat schemas, e.g., ER or relational schemas, the researchers have developed approaches to the resolutions of naming conflicts, key conflicts, structural conflicts and constraint conflicts. These approaches can be adapted to resolve the similar conflicts in the integration of XML schemas. Our resolution to schematic discrepancies in the integration of ER schemas can also be extended to XML. However, the hierarchical structure and the semi-structuredness of XML brings some new challenges in the integration of XML schemas. Although the issue of semi-structuredness was studied in [44], existing work paid little attention to the hierarchical characteristic of XML. For example, two paths of two schemas may contain the same set of elements, and represent the equivalent relationship types among these elements, but have different hierarchical structures (i.e., the elements in the two paths have different hierarchical orders). This inconsistency should be resolved in merging the two paths (relationship types).

Most of the existing work in XML schema integration, e.g., [51, 62, 65, 74], treat elements (or object classes) as "first-class citizens". That is, they would first merge equivalent elements, and then handle the relationships among elements (e.g., break the circles of relationships produced in merging elements). This strategy works in the integration of traditional flat schemas, such as ER or relational schemas, but may cause the loss of information in the integration of XML schemas. The reason is that those approaches may destroy the relationships among elements and the hierarchical structures (that imply some context information) of XML schemas.

In this work, we will develop an approach to XML schema integration in which relationships are treated as the semantic units. That is, if we treat objects as atoms and relationships as molecules, our approach is at the molecular level while

the previous work is at the atomic level. The maximum benefit is that we can preserve more information (i.e., relationships and attributes of relationships) in schema integration. Furthermore, we respect the hierarchical structure of XML, and require integrated schemas preserve some important hierarchical characteristics of source schemas.

Previous work on DTD or other models [65, 29, 51, 62] only handle binary relationships between elements. In practice, XML data may contain higher degree relationships among elements, such as the ternary relationship type SPM of Figure 2.2. In general, a higher degree relationship type could not be losslessly decomposed into a set of binary relationship types. That is, a ternary relationship type should be handled as a whole instead of as two binary relationship types.

The work on DTD also does not distinguish the attributes of relationship types (e.g., the attribute PRICE of the relationship type SPM in Figure 2.2) from the attributes of elements. Consequently, their approaches may loss the information of attributes of relationships in schema transformation, and transform source schemas to some schemas with different meaning, as discussed in Section 2.2.

We will use ORASS to represent XML schemas, which support relationship types of any degrees and distinguish attributes of object classes and attributes of relationship types.

Yang et al. [74] used the user-specified weight of importance to assess source schemas and to compute integrated schemas. Sometimes such specification would be not easy for users, but important to the structure of the integrated schema. We propose to find some objective and specific criteria emanating from applications of XML schema and data integration, and use these criteria to guide the integration of XML schemas.

A recent research direction is to annotate information with ontology languages, e.g., RDF or OWL, and tackle semantic heterogeneity issues in ontology merging. However, it is inconvenient to represent hierarchical data such as XML using the current ontology languages such as RDF and OWL. For example, given the ternary relationship type SPM of Figure 2.2, we may represent the relationship type as an abstract class, and relate it to the three participating object classes SUPPLIER, PROD and MONTH using some binary relationship types (predicates) of *involving* in ontology. But then the hierarchical structure of the original relationship type SPM is lost. In other words, some additional hierarchical information needs to be stored to represent XML data in ontology. Actually, existing work in ontology integration did not consider ternary and higher degree relationships. They treated object classes as first-class citizens, and focused on the resolution of different classifications in source ontologies (some work handled binary relationships between classes and between classes and attributes). On the other hand, ORASS is designed for semi-structured data such as XML, and our work should focus on the relationships and hierarchical structures of XML in schema integration.

Previous work in schema transformation/integration used integration assertions [39, 67], schema correspondences [61], morphisms [51], or mappings [62] to relate equivalent and similar constructs in two source schemas. These approaches can only integrate two schemas once. Our purpose is to integrate a set of XML schemas once.

In summary, our purpose is:

1. to find some objective and specific criteria of XML schema integration.

2. to propose an algorithm to integrate a set of XML schemas, treating relationship types as first-class citizens.

3. to consider the rich semantics of real world applications, e.g., the relationships

among several objects, attributes of objects and attributes of relationship types. This can be achieved by use of the ORASS model.

4. to resolve the inconsistencies of hierarchical structures of source data.

5. to preserve the information of relationship types, attributes of object classes and attributes of relationship types, and important hierarchical characteristic of source data in the integration.

Chapter 5

Lossless and non-redundant schema transformation

Schema transformation plays an important role in schema integration. In the integration of relational schemas, we resolve schematic discrepancies among them by transforming them with the restructuring operators. As mentioned in Chapter 1, schema transformation should be information preserving and constraint preserving. Furthermore, a transformation should be efficient, i.e., free of redundant operations. We study the issues of information preservation and non-redundancy in the chapter, and the constraint issue in the next chapter.

In this chapter, we studied the reconstructibilities and commutativities of the restructuring operations, and the lossless-ness and non-redundancy of discrepant schema transformations.

5.1 Algebraic laws of restructuring operators

Like many other algebra, there are some laws that the restructuring operators obey. Although the restructuring operators are useful in the work in schematic

discrepancy (see Section 1.1), the properties of them are less-studied. This caused some problems in the work using them, e.g., a schema transformation including some restructuring operations may loss information. In this section, we study two important properties, i.e., reconstructibilities and commutativities, of restructuring operators.

5.1.1 Reconstructibility

A set of relations can be transformed to another set of relations without any loss of information, and recovered back, hence the name of *reconstructibility*.

Property 1 (Reconstructibility of split). *Given a relation* $R(A_1, \ldots, A_n, B)$ *with* $dom(B) = \{b_1, \ldots, b_m\}$*, we have:*

$$unite(split(R,\ B),\ B) = R.$$

Proof. Let $\mathbb{R} = \text{split}(R,\ B)$, and $R' = \text{unite}(\text{split}(R,\ B),\ B)$. For each tuple $t = (a_1, \ldots, a_n, b_i)$ in R', there must be a relation $b_i \in \mathbb{R}$, and the tuple $(a_1, \ldots, a_n)$ is in the relation b_i. As the relation b_i is produced by the operation $\text{split}(R,\ B)$, $t \in R$. So $R' \subseteq R$.

On the other hand, for each tuple $t = (a_1, \ldots, a_n, b_i)$ in R, the tuple $(a_1, \ldots, a_n)$ is in the relation b_i after the operation $\text{split}(R,\ B)$. Then $t \in R'$ after the following unite operation. So $R \subseteq R'$. □

For example, in Figure 1.1, we transform $DB1$ to $DB3$ with a split operation, and recover $DB1$ with a unite operation.

Property 2 (Reconstructibility of unite). *Given a set of relations* $\{b_1, \ldots, b_m\}$ *such that these relations have the same schema and the relation names are from the domain of an attribute* B*, we have:*

$$split(unite(\{b_1, \ldots, b_m\},\ B),\ B) = \{b_1, \ldots, b_m\}.$$

Proof. The proof is similar to the proof of Property 1. □

For example, in Figure 1.1, we transform $DB3$ to $DB1$ with a unite operation, and recover $DB3$ with a split operation.

An operation is reconstructible if it defines a one-to-one mapping from the original relations onto the transformed relations. However, a fold operation may be a many-to-one mapping, and an unfold operation may be a one-to-many mapping. The following example shows a fold operation that is not reconstructible.

Example 5.1. *In Figure 5.1, given a relation with the schema $R(A, b_1, b_2)$, suppose the attribute names b_1, b_2 are the values of a fixed attribute B, and the values of the attributes b_1, b_2 (i.e., c_1, c_2, c_3 and c_4) are from the domain of another attribute C. By applying the operation fold(R, B, C), we can transform either of the two instances of R (i.e., $R(I1)$ and $R(I2)$) into the same relation instance of S. That is, the mapping from the instances of R onto the instances of S is many-to-one, which makes the recovery impossible. We say the fold operation is "lossy (or non-lossless)".*

R (I1)

A	b_1	b_2
a_1	c_1	c_2
a_1	c_3	c_4

R (I2)

A	b_1	b_2
a_1	c_1	c_4
a_1	c_3	c_2

S

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_1	b_1	c_3
a_1	b_2	c_4

Figure 5.1: A lossy fold transformation: the transformation from R (I1 or I2) to S is un-recoverable.

However, if the functional dependency $A \rightarrow \{b_1, b_2\}$ held in R (which is not true in $R(I1)$ and $R(I2)$), the operation would be reconstructible. That is, functional dependencies can be used to verify reconstructible fold operations. □

The following two propositions give the sufficient conditions (i.e., functional dependencies) for reconstructible fold and unfold operations.

Proposition 5.1 (Condition of reconstructible fold). *Given a relational schema* $R(A_1, \ldots, A_n, b_1, \ldots, b_m)$ *such that the attribute names* $b_1, \ldots, b_m$ *are from the domain of an attribute* B *and the values of these attributes are from the domain of another attribute* C*, if the functional dependency*

$$\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$$

holds, then for any two instances $R(I1)$ *and* $R(I2)$*, if fold(*$R(I1)$*,* B*,* C*) = fold(*$R(I2)$*,* B*,* C*), then* $R(I1) = R(I2)$*.*

Proof. Let S = fold($R(I1)$, B, C). For each tuple $t = (a_1, \ldots, a_n, c_1, \ldots, c_m)$ in $R(I1)$, $(a_1, \ldots, a_n, b_i, c_i)$ is in S, i.e., fold($R(I2)$, B, C), for each $i = 1, \ldots, m$. So there must be a set of tuples $\{t_1, \ldots, t_m\}$ in $R(I2)$, such that $t_i.A_1 = a_1, \ldots, t_i.A_n = a_n, t_i.b_i = c_i$ for each $i = 1, \ldots, m$. As the functional dependency $\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$ holds in any instance of R, $t_1, \ldots, t_m$ are all the same, i.e., t. That is, $t \in R(I2)$. So $R(I1) \subseteq R(I2)$. Similarly, we can prove $R(I2) \subseteq R(I1)$. □

Proposition 5.2 (Condition of reconstructible unfold). *Given a relation* $R(A_1, \ldots, A_n, B, C)$*, if the functional dependency*

$$\{A_1, \ldots, A_n, B\} \rightarrow C$$

holds, then the result of the operation unfold(R*,* B*,* C*) is unique.*

Proof. Suppose the result is not unique. Let $S1$ and $S2$ be two possible results of the operation unfold(R, B, C). $S1$ and $S2$ must have the same set of attributes, i.e., $\{A_1, \ldots, A_n, b_1, \ldots, b_m\}$ for $\{b_1, \ldots, b_m\}$ the set of the distinct values of B in R. Let $t = (a_1, \ldots, a_n, c_1, \ldots, c_m)$ be a tuple in $S1$. Then $(a_1, \ldots, a_n, b_i, c_i)$ is in

R for each $i = 1, \ldots, m$. So there must be a set of tuples $\{t_1, \ldots, t_m\}$ in $S2$, such that $t_i.A_1 = a_1, \ldots, t_i.A_n = a_n, t_i.b_i = c_i$ for each $i = 1, \ldots, m$. As the functional dependency $\{A_1, \ldots, A_n, B\} \rightarrow C$ holds in R, two tuples of $S2$ with the same values $a_1, \ldots, a_n$ of $A_1, \ldots, A_n$ have the same value of the attribute b_i for $i = 1, \ldots, m$. So $t_1, \ldots, t_m$ are all the same, i.e., t. That is, $t \in S2$. So $S1 \subseteq S2$. Similarly, we can prove $S2 \subseteq S1$. So $S1 = S2$. This contradicts with the assumption that $S1 \neq S2$. □

For practical case, a fold (unfold) operation makes sense only if the functional dependency holds. We will regard it as a necessary condition for the fold (unfold) operations in the rest of the paper. Knowing the conditions of the reconstructibilities of fold and unfold operations, we can present the reconstructibilities below.

Property 3 (Reconstructibility of fold). *Given a relation R(A_1, ..., A_n, b_1, ..., b_m) such that the attribute names $b_1, \ldots, b_m$ are from the domain of an attribute B and the values of these attributes are from the domain of another attribute C, we have:*

If the functional dependency $\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$ holds, then

unfold(fold(R, B, C), B, C) = R.

Proof. Let $S = \text{fold}(R, B, C)$ and $R' = \text{unfold}(S, B, C)$. For each tuple $t = (a_1, \ldots, a_n, c_1, \ldots, c_m) \in R$, $(a_1, \ldots, a_n, b_i, c_i) \in S$ for each $i = 1, \ldots, m$. Then there must be a set of tuples $\{t_1, \ldots, t_m\}$ in R', such that $t_i.A_1 = a_1, \ldots, t_i.A_n = a_n, t_i.b_i = c_i$ for each $i = 1, \ldots, m$. As the functional dependency $\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$ holds in R, c_i is unique for a given $(a_1, \ldots, a_n, b_i)$ in S. Then in R', given the values $a_1, \ldots, a_n$ of $A_1, \ldots, A_n$, the values of the attributes $b_1, \ldots, b_m$ are unique. So $t_1, \ldots, t_m$ are all the same, i.e., t. That is, $t \in R'$. So $R \subseteq R'$.

On the contrary, for each tuple $t' = (a_1, \ldots, a_n, c_1, \ldots, c_m) \in R'$, there must be a set of tuples $(a_1, \ldots, a_n, b_i, c_i) \in S$ for each $i = 1, \ldots, m$. Then there must be a set of tuples $\{t_1, \ldots, t_m\}$ in R, such that $t_i.A_1 = a_1, \ldots, t_i.A_n = a_n, t_i.b_i = c_i$ for each $i = 1, \ldots, m$. As the functional dependency $\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$ holds in R, $t_1, \ldots, t_m$ are all the same, i.e., t'. That is, $t' \in R$, and $R' \subseteq R$. Consequently, $R = R'$. □

For example, in Figure 1.1, we transform $DB2$ to $DB1$ with a fold operation, and recover $DB2$ with an unfold operation.

Property 4 (Reconstructibility of unfold). *Given a relation $R(A_1, \ldots, A_n, B, C)$, we have:*

If the functional dependency $\{A_1, \ldots, A_n, B\} \rightarrow C$ holds, then fold(unfold(R, B, C), B, C) = R.

Proof. The proof is similar to the proof of Property 3. □

5.1.2 Commutativity

Now we introduce another kind of property, *commutativity* of the restructuring operators. In general, given a transformation T on a set (possibly a singleton set) of relations $\mathbb{R}$ consisting of two (sets of) restructuring operations $t1$ and $t2$, let T' be a transformation obtained by changing the order of $t1$ and $t2$ in T, if $T(\mathbb{R}) = T'(\mathbb{R})$, then we say $t1$ and $t2$ are commutative.

The following example shows the commutativity between a set of fold operations and a unite operation.

Example 5.2. *Suppose in Figure 1.1, we want to transform $DB4$ into $DB1$. The transformation can be implemented with either of the following two sequences of*

operations:

$$unite(\{fold(s_i, month, price)|i = 1, \ldots, n\}, s\#), \ \ or$$

$$fold(unite(\{s_1, \ldots, s_n\}, s\#), month, price).$$

In the first transformation, we first perform a fold operation on each relation of DB4, then perform a unite on the folded relations. In the second transformation, we first perform a unite operation on the relations of DB4, then perform a fold on the united relation. In other words, the (set of) fold and the unite operations can commute in the transformation. □

Commutativity also applies to other pairs of operations, e.g., fold and split, two fold operations etc. This property is presented in Appendix A.1. In general, all pairs of restructuring operations except the pair of unite and split operations, the pair of two unite operations and the pair of two split operations are commutative, if the pair of operations have no common parameters of attributes.

We have studied the reconstructibilities and commutativities of the restructuring operators. Then we can study the theory of a discrepant schema transformation consisting of a sequence of sets of restructuring operations.

5.2 Lossless and non-redundant transformations

In practice, one is mostly interested in information-preserving transformations [55], i.e. transformations such that both original and transformed relations represent exactly the same real world facts, though with different syntaxes. A formal definition is given below.

Definition 5.1 (Lossless transformation). *Given a discrepant schema transformation T on a set (possibly a singleton set) of relations $\mathbb{R}$. If there exist an*

inverse transformation T' of T, such that $T'(T(\mathbb{R})) = \mathbb{R}$, then T is called a lossless transformation. □

Any restructuring operation satisfying the reconstructibility property is a lossless transformation. In general, we have the following result:

Theorem 5.1. *Given a discrepant schema transformation T, if the fold and unfold operations in T satisfy the reconstructibilities (i.e., Properties 3 and 4), then T is a lossless transformation.*

Proof. This can be concluded from the reconstructibilities of the restructuring operations. □

As the reconstructibilities of fold and unfold require certain functional dependencies to hold over relational schemas (Properties 3 and 4), the theorem indicates that to verify the lossless-ness of a transformation, we need to derive functional dependencies during the transformation, which will be discussed in the next chapter.

A lossless transformation may contain redundant operations. Using the reconstructibility and commutativity of restructuring operations, we can simplify a transformation by removing the redundant operations from it.

Example 5.3. *In Figure 1.1, we have two ways to transform $DB4$ into $DB2$:*

$T1$: unfold (unite ($\{$fold $(s_i, month, price) | i = 1, \ldots, n\}, s\#), month, price)$

$T2$: unite ($\{s_1, \ldots, s_n\}, s\#$)

That is, $T1(DB4) = T2(DB4) = DB2$. In $T1$, we first perform a set of fold operations on the relations of $DB4$, and transform them into $s_i(p\#, month, price)$ for each $i = 1, \ldots, n$. Then we unite these folded relations into the relation of $DB1$. Finally we unfold the united relation and get the relation of $DB2$ as the result. The transformation can also be implemented by performing a unite operation on the

relations of $DB4$ directly, i.e., $T2$. Thus $T1$ is a redundant transformation with the unnecessary fold and unfold operations. □

A non-redundant transformation is one only consisting of the necessary restructuring operations. Before giving its formal definition, we first define a *sub-transformation relationship* "$\preceq$" between two transformations.

Definition 5.2. *Given two discrepant schema transformations T and T' on the same set of relations $\mathbb{R}$, we define a partial order $\preceq$ as follows: $T \preceq T'$ iff there is a one-to-one mapping h from the set of all the operations of T to a set of some operations of T', such that (1) h preserves the operations of T, i.e., $\forall t \in T\ \exists h(t) \in T'$, such that t and $h(t)$ are of the same kind of operator, and have the same parameters of attributes, and (2) h preserves the important operation orders of T, i.e., for any $t1, t2 \in T$, such that $t1$ precedes $t2$ in T, and $t1$ and $t2$ have some common parameters of attributes, then $h(t1)$ precedes $h(t2)$ in T'.* □

Definition 5.3 (Non-redundant transformation). *A discrepant schema transformation T on a set of relations $\mathbb{R}$ is* non-redundant *iff for any other transformation T' on $\mathbb{R}$,*

$$\textit{if } T'(\mathbb{R}) = T(\mathbb{R}) \textit{ and } T' \preceq T, \textit{ then } T \preceq T'. \quad \square$$

For Example 5.3, $T2$ is a non-redundant transformation. However, $T1$ is redundant, as we have $T2$ such that $T2(\mathbb{R}) = T1(\mathbb{R})$ and $T2 \preceq T1$, but not the converse. We call the set of fold operations and the unfold operation in $T1$ a pair of *reverse operations*.

Definition 5.4 (Reverse operations). *Given a discrepant schema transformation T on a set of relations $\mathbb{R}$, let $t1$ and $t2$ be two sets of operations of T, we call $t1$ and $t2$ a pair of* reverse operations *iff there exists another transformation*

T' on $\mathbb{R}$, such that $T(\mathbb{R}) = T'(\mathbb{R})$, and there is a one-to-one mapping h from the set of some operations of T except $t1$ and $t2$ to the set of all the operations of T', such that h preserves the operations and the important operation orders of T (see Definition 5.2). □

In a discrepant schema transformation, fold and unfold (or unite and split) operations are reverse restructuring operations, if they have the same parameters of attributes.

Lemma 5.1. *In a discrepant schema transformation T, we have two kinds of reverse restructuring operations:*

1. *two operations unite$(\mathbb{R}, B)$ and split(R, B), where $\mathbb{R}$ is a set of relations and R is a relation produced during the transformation T.*

2. *two sets of operations $\{fold(R, B, C)|R \in \mathbb{R}\}$ and $\{unfold(S, B, C)|S \in \mathbb{S}\}$, where $\mathbb{R}$ and $\mathbb{S}$ are two sets (possibly singleton sets) of relations produced during the transformation T.*

Proof. The proof is given in Appendix A.2. □

In general, we can remove all the pairs of reverse operations from a lossless transformation T in three steps:

1. For any pair of reverse operations T_i and T_j $(i < j)$ of T, such that the operations of T_{i+1}, T_{i+2}, ..., T_{j-1} do not have the same parameters of attributes as T_i and T_j, we swap T_i and T_{i+1}, T_{i+2}, ..., T_{j-1} one by one according to the commutativity of restructuring operations;

2. Cancel the pair of reverse operations T_{j-1} and T_j according to the reconstructibility of restructuring operations;

3. Repeat the two steps till no more pairs of reverse operations exist in the transformation.

Finally, we get a transformation T' that is equivalent to T, but does no contain any pair of reverse operations.

For Example 5.3, we can simplify $T1$ into $T2$ by first swapping the fold and unite operations, then cancelling the fold and unfold operations together.

Intuitively, in a non-redundant transformation, we never unite a set of relations, and split the united relation later on, or fold a relation, and unfold the folded relation later on. In general, we have the following result:

Lemma 5.2. *A lossless transformation is non-redundant iff it does not contain any pair of reverse operations.*

Proof. The proof is given in Appendix A.3. □

As mentioned, all the pairs of reverse operations in a lossless transformation can be removed in three steps. From Lemma 5.2, we can get the following result:

Theorem 5.2. *A lossless transformation can be simplified to a non-redundant lossless transformation.* □

5.3 Summary

In relational databases, schematic discrepancy occurs when the attribute names or relation names in one schema correspond to the attribute values in another schema. Researchers [34] have developed the 4 restructuring operators fold, unfold, unite and split to transform discrepant schemas into consistent ones. However, the properties of these operators have not been well studied. We studied the reconstructibilities and commutativities of these operators. We found unite and split operations are

always reconstructible, but fold and unfold operations are not reconstructible unless some functional dependencies hold on the original relations (i.e., Properties 3 and 4). In a discrepant schema transformation that is a sequence of sets of restructuring operations, two adjacent sets of operations can commute if they have different parameters of attributes (i.e., the properties in Appendix A.1). Using the reconstructibilities and commutativities of restructuring operators, we can simplify a lossless discrepant schema transformation to a non-redundant lossless transformation by removing reverse operations.

Without loss of generality, in the rest of the paper, we make the assumption that a discrepant schema transformation is lossless and non-redundant. But remember that certain functional dependencies should be satisfied to ensure lossless fold and unfold operations.

Chapter 6

Deriving and using qualified functional dependencies in multidatabases

Conventional functional dependencies are inadequate to represent constraints in multidatabases, as shown in Section 1.2. We introduce qualified functional dependencies, i.e., an extension to functional dependencies, to represent constraints in multidatabases. We give a sound and complete set of inference rules to imply qualified functional dependencies in fixed relations, and an algorithm to compute attribute closures with respect to a set of qualified functional dependencies. We also study the derivation of qualified functional dependencies in a discrepant schema transformation. In particular, we give propagation rules to derive qualified functional dependencies for transformed schemas from qualified functional dependencies on original schemas. The rules are sound, complete and efficient to derive a broad class of qualified functional dependencies in schema transformation. Finally, we introduce some applications of qualified functional dependencies in data integration/mediation systems and in multidatabase interoperation.

6.1 Qualified functional dependencies

6.1.1 Definition of qualified functional dependency

Qualified functional dependencies are the functional dependencies holding over a set of relations or a set of horizontal partitions of relations. In other words, they qualify the relations and the tuples of the relations over which the constraints hold.

Definition 6.1 (Qualified functional dependency). *In general, given a set of relational schemas* $\mathbb{S}$ *with the same set of attributes* U*, we can represent a qualified functional dependency as:*

$$\mathbb{R}(A_{1\ \sigma=S_1}, A_{2\ \sigma=S_2}, \ldots, A_{n\ \sigma=S_n}, X \rightarrow Y)$$

Syntax of the qualified functional dependency:

- $\mathbb{R} \subseteq \mathbb{S}$ *represents the set of relational schemas over which the qualified functional dependency holds.*
- $A_{i\ \sigma=S_i}$ *for each* $i = 1, \ldots, n$*, satisfies* $A_i \in U$ *and* $S_i \subseteq dom(A_i)$*, indicating the qualification of the attribute values within which the qualified functional dependency holds. For ease of reference, we call each* $A_{i\ \sigma=S_i}$ *a* qualification attribute *from* U*.*
- $X \subseteq U$ *and* $Y \subseteq U$ *are two sets of* regular attributes.

Semantics of the qualified functional dependency:

We say the given qualified functional dependency holds over $\mathbb{R}$*, if the following holds for any two tuples* $t1$*,* $t2$ *from any instance of* $\mathbb{R}$ *(*$t1$*,* $t2$ *may come from one or two relation instances): If* $t1.A_i \in S_i$ [1] *and* $t2.A_i \in S_i$ *for each* $i = 1, \ldots, n$*,*

[1] $t1.A_i$ represents the value of the attribute A_i in $t1$.

and $t1.X_j = t2.X_j$ *for each attribute* $X_j \in X$*, then* $t1.Y_k = t2.Y_k$ *for each attribute* $Y_k \in Y$*. This completes the definition of a qualified functional dependency.* □

Note that in this definition, the attribute values of $A_i, i = 1, \ldots, n$ are not necessary to be the same. They are only required in the value set S_i. In Section 1.2, Examples 1.3 and 1.4 show two real examples of qualified functional dependencies.

In general, given a qualified functional dependency

$$\mathbb{R}(A_1\ {}_{\sigma=S_1}, A_2\ {}_{\sigma=S_2}, \ldots, A_n\ {}_{\sigma=S_n}, X \rightarrow Y),$$

let $R = \cup_{R_i \in \mathbb{R}}(\sigma_{A_1 \in S_1, \ldots, A_n \in S_n} R_i)$ ($\cup$ is the union and σ is the selection operator of the relational algebra), then the functional dependency $X \rightarrow Y$ holds over R. If a qualified functional dependency only contains regular attributes and holds over a relational schema, then it is just a conventional functional dependency.

6.1.2 Inference rules of qualified functional dependencies in fixed schemas

In general, let F be a set of qualified functional dependencies for a set of relational schemas $\mathbb{R}$, and let f be a qualified functional dependency also for $\mathbb{R}$. We say F *(logically) implies* f, if every instance of $\mathbb{R}$ that satisfies the dependencies in F also satisfies f. We define F^+, the *closure of* F for $\mathbb{R}$, to be the set of qualified functional dependencies that are logically implied by F. To understand logical implications among qualified functional dependencies in fixed schemas, we provide a complete set of inference rules, meaning that from a given set of qualified functional dependencies F for $\mathbb{R}$, the rules allow us to deduce all the true qualified functional dependencies for $\mathbb{R}$, i.e., those in F^+. Without causing confusion, in Section 6.2 below, we will give another kind of rules (called *propagation rules*) which allow us to infer the

qualified functional dependencies of the transformed relations from the qualified functional dependencies of the original relations in a schema transformation.

The inference rules are given below. We assume for each qualification attribute $A_{\sigma=S}$ of a qualified functional dependency, the domain of A is a finite and fixed set.

Inference rules of qualified functional dependencies. Given a set of relational schemas $\mathbb{S}$ with the same set of attributes U, and a set of qualified functional dependencies F for $\mathbb{S}$, let $\overline{X}$ be a mixed set of regular and qualification attributes from U ($\overline{X}$ may comprise only regular or qualification attributes), and let $Y \subseteq U$ and $Z \subseteq U$ be two sets of regular attributes. Let $A \in U$ be an attribute not occurring in $\overline{X}$. $\mathbb{R}1 \subseteq \mathbb{R} \subseteq \mathbb{S}$, $S1 \subseteq S \subseteq dom(A)$. We have the following inference rules:

(A1) *Partition on relation set.* If $\mathbb{R}(\overline{X} \rightarrow Y)$ holds, then $\mathbb{R}1(\overline{X} \rightarrow Y)$ holds.

(A2) *Composition on relation set.* If $\{R_i, R_j\}(\overline{X} \rightarrow Y)$ holds for any $R_i, R_j \in \mathbb{R}$, then $\mathbb{R}(\overline{X} \rightarrow Y)$ holds.

(A3) *Partition on qualification.* If $\mathbb{R}(\overline{X}, A_{\sigma=S} \rightarrow Y)$ holds, then $\mathbb{R}(\overline{X}, A_{\sigma=S1} \rightarrow Y)$ holds.

(A4) *Composition on qualification.* If $\mathbb{R}(\overline{X}, A_{\sigma=\{a_i,a_j\}} \rightarrow Y)$ holds for any $a_i, a_j \in S$, then $\mathbb{R}(\overline{X}, A_{\sigma=S} \rightarrow Y)$ holds.

(A5) *Single-valued qualification.* If $a \in dom(A)$, then $\mathbb{R}(A_{\sigma=\{a\}} \rightarrow A)$ holds.

(A6) *Assembly.* If $\mathbb{R}(\overline{X}, A_{\sigma=\{a\}} \rightarrow Y)$ holds for each $a \in S$, then $\mathbb{R}(\overline{X}, A, A_{\sigma=S} \rightarrow Y)$ holds.

(A7) *Reflexivity.* If $Y \subseteq \overline{X}$, then $\mathbb{R}(\overline{X} \rightarrow Y)$.

(A8) *Augmentation.* If $\mathbb{R}(\overline{X} \rightarrow Y)$ holds, then $\mathbb{R}(\overline{X}, Z \rightarrow Y, Z)$ holds.

(A9) *Transitivity.* If $\mathbb{R}(\overline{X} \rightarrow Y)$ and $\mathbb{R}(\overline{X1}, Y \rightarrow Z)$ hold for $\overline{X1}$ a set (possibly an empty set) of some qualification attributes in $\overline{X}$, then $\mathbb{R}(\overline{X} \rightarrow Z)$ holds.

(A10) *Dummy qualification.* $\mathbb{R}(\overline{X} \rightarrow Y)$ iff $\mathbb{R}(\overline{X}, A_{\sigma=dom(A)} \rightarrow Y)$ holds. □

Rules A5 and A7 give the *trivial dependencies*. Rules A7, A8 and A9 extend Armstrong's Axioms, the inference rules of functional dependencies. Note that in Rule A9, the inferred qualified functional dependency inherits all the qualification attributes of the given qualified functional dependencies.

The following rule is derived from the above rules and useful in the rest of the paper.

(A11) *Disassembly.* $\mathbb{R}(\overline{X}, A \rightarrow Y)$ iff $\mathbb{R}(\overline{X}, A_{\sigma=\{a\}} \rightarrow Y)$ holds for each $a \in dom(A)$.

For example, in $DB1$ of Figure 1.1, the functional dependency $\{p\#, s\#, month\} \rightarrow price$ is equivalent to a set of qualified functional dependencies $Supply(p\#, s\#, month_{\sigma=\{mi\}} \rightarrow price)$ for each $mi = jan, \ldots, dec$. That is, in each month, product numbers and supplier numbers uniquely determine prices.

Theorem 6.1. *The inference rules A1 to A10 are sound, complete and irreducible.*

Proof. The proof is given in Appendix A.4. Note that the proof should use attribute closures defined below. □

We define attribute closures with respect to qualified functional dependencies as follows:

Definition 6.2 (Attribute closure). *Given a set of qualified functional dependencies F, let $\mathbb{R}$ be a set of relational schemas, W be a set of qualification attributes, and X be a set of regular attributes, we define $X^{+}_{\mathbb{R},W}$, the* attribute closure of X

under the qualification of W in $\mathbb{R}$ with respect to F, *as the set of attributes* A *such that* $\mathbb{R}(W, X \rightarrow A)$ *can be deduced from* F *by the inference rules A1 to A10.* $\square$

Given a set of qualified functional dependencies F, often we want to know whether a particular qualified functional dependency $\mathbb{R}(W, X \rightarrow Y)$ follows from F, where W is a set of qualification attributes, and X and Y are two sets of regular attributes. The solution is to compute $X^{+}_{\mathbb{R},W}$, the attribute closure of X under the qualification of W in $\mathbb{R}$ with respect to F, and then to check whether Y is a subset of the closure.

The algorithm to compute an attribute closure w.r.t a set of qualified functional dependencies will be given in Section 6.1.3.

We define the equivalence between two sets of qualified functional dependencies: Let F and G be 2 sets of qualified functional dependencies on the same set of relational schemas. We say F and G are *equivalent* if $F^{+} = G^{+}$. That is, each qualified functional dependency in F is also in G^{+}, and each qualified functional dependency in G is also in F^{+}.

6.1.3 Compute attribute closures with respect to qualified functional dependencies

In this sub-section, we will give an algorithm to compute attribute closure w.r.t a set of qualified functional dependencies. Qualified functional dependency is a powerful expression that can represent a broad range of constraints. However, we restrict our study to a set of qualified functional dependencies F satisfying 3 conditions:

1. *single-valued qualification.* the qualification attributes of any qualified functional dependency of F are restricted to take single values, i.e., any qualifi-

cation attribute has a form of $A_{\sigma=\{a\}}$.

2. *disjoint regular and qualification attributes.* let $Z1$ be the set of all the regular attributes, and $Z2$ be the set of all the attributes occurring in the qualification attributes of the qualified functional dependencies of F. Then $Z1 \cap Z2 = \emptyset$.

3. *individual or all relations.* Each qualified functional dependency of F holds either in a single relation, or in a set of relational schemas $\mathbb{R} = dom(B)$ for some attribute B whose values are modelled as relation names.

The three conditions restrict the expressiveness of qualified functional dependencies. With the three restrictions, we focus our study on common qualified functional dependencies in practice - the qualified functional dependencies that may be transformed into (or derived from) some functional dependencies in a discrepant schema transformation. There're two reasons why we introduce the three restrictions: (1) functional dependencies are the most common and useful dependencies in databases, we are interested in those qualified functional dependencies that may be transformed into (or derived from) some functional dependencies in a schema transformation. (2) The computation of an attribute closure with respect to a set of general qualified functional dependencies would be too slow. By restricting our study in a set of common and useful qualified functional dependencies, we can give efficient methods to compute attribute closures.

We hereby present an algorithm CLOSURE below to compute an attribute closure w.r.t. a set of qualified functional dependencies satisfying the above 3 conditions.

We explain the algorithm by an example below.

Example 6.1. *In DB1 of Figure 1.1, given a set of qualified functional dependencies*

Algorithm:CLOSURE

Input: A set of relational schemas $\mathbb{S}$ with the same set of attributes U, a set of relational schemas $\mathbb{R} \subseteq \mathbb{S}$, a set of qualification attributes W from U, a set of regular attributes X from U, a set of qualified functional dependencies F satisfying the 3 conditions.

Output: $X^{+}_{\mathbb{R},W}$ w.r.t. F.

1: Let $\{X_1, \ldots, X_n\}$ be the maximum subset of X, such that each X_i, $i = 1, \ldots, n$, occurs in some qualification attributes of the qualified functional dependencies of F;
2: $closure := U$;
3: **for** any $x_1 \in dom(X_1), \ldots, x_n \in dom(X_n)$, such that $\{X_1{}_{\sigma=\{x_1\}}, \ldots, X_n{}_{\sigma=\{x_n\}}\} \sqsupseteq W[X_1, \ldots, X_n]$ */* The satisfaction relation $\sqsupseteq$ and projection $W[X_1, \ldots, X_n]$ are defined in Definitions A.1 and A.2 in Appendix A.4. */* **do**
4: $closure1 := X \cup \{A | A_{\sigma=\{a\}} \in W\}$;
5: $W1 := W \cup \{X_1{}_{\sigma=\{x_1\}}, \ldots, X_n{}_{\sigma=\{x_n\}}\}$;
6: **repeat**
7: **if** there is a qualified functional dependency $\mathbb{R}1(\overline{Y} \rightarrow Z)$ in F such that $\mathbb{R} \subseteq \mathbb{R}1$ and $W1 \cup closure1 \sqsupseteq \overline{Y}$ **then**
8: $closure1 := closure1 \cup Z$;
9: **end if**
10: **until** no change on $closure1$
11: $closure := closure \cap closure1$;
12: **end for**
13: **return** $closure$.

$$Supply(p\#, s\#, month_{\sigma=\{mi\}} \rightarrow price)$$

for each $mi \in \{jan, \ldots, dec\}$*, let* $X = \{p\#, s\#, month\}$ *be a set of attributes, we compute the attribute closure of* X *in the Supply relation, i.e.,* $X^+_{Supply,\emptyset}$*, using the CLOSURE algorithm below.*

For each $mi \in \{jan, \ldots, dec\}$*, let* $W1 = \{month_{\sigma=\{mi\}}\}$*, then after each iteration of the inner loop (Line 6 to 10),* $closure1 = X^+_{Supply,W1} = \{p\#, s\#, month, price\}$*. Consequently, the algorithm returns the set* $closure = X^+_{Supply,\emptyset} = \{p\#, s\#, month, price\}$*.* □

Theorem 6.2. *Algorithm CLOSURE correctly computes* $X^+_{\mathbb{R},W}$ *w.r.t.* F*.*

Proof. The proof is given in Appendix A.5. □

Finally, as to the complexity of Algorithm CLOSURE, we have the following lemma.

Lemma 6.1. *In Algorithm CLOSURE, let* Z *be the set of all the attributes occurring in the qualification attributes of the qualified functional dependencies of* F*. Let* m *be the cardinality of* Z*, and* d *be an upper bound for the cardinalities of the domains of the attributes of* Z*. Then the algorithm takes time* $O(d^m|U||F|^2)$*.*

Proof. The outer loop of the CLOSURE algorithm (Line 3 to 12) will be iterated for d^m times at most. For each iteration of the outer loop, the inner loop (Line 6 to 10) would be repeated for $O(|F|^2)$ times, and each iteration of the inner loop takes $O(|U|)$ time. Consequently, the whole algorithm takes $O(d^m|U||F|^2)$ time. □

From Lemma 6.1, we know that the performance of Algorithm CLOSURE depends much on the structure of the set Z. When the parameters m and d are constants, the algorithm runs in polynomial time. In Section 6.2.3, we will study the complexity of the CLOSURE algorithm further in the context of discrepant schema transformations.

6.2 Deriving qualified functional dependencies in schema transformations

In this section, the implication of qualified functional dependencies extends to transforming relations. In general, given a transformation T, let $\mathbb{R}$ and $\mathbb{S}$ be, respectively, the sets of the original and transformed relations [2] of T; let F be a set of qualified functional dependencies for $\mathbb{R}$, and f be a qualified functional dependency for $\mathbb{S}$; let r be the instance of $\mathbb{R}$ satisfying the dependencies of F, and s be the instance of $\mathbb{S}$ transformed from r by T. We say F *(logically) implies* f, if s satisfies f. Note that unlike the implication of qualified functional dependencies in fixed schemas, now the given set of dependencies F and the implied dependency f hold in the original relations and transformed relations respectively. To understand logical implications among qualified functional dependencies in transforming relations, we provide a set of propagation rules, meaning that from a given set of qualified functional dependencies F for the set of the original relations $\mathbb{R}$, the rules allow us to deduce the qualified functional dependencies for the set of the transformed relations $\mathbb{S}$.

6.2.1 Propagation rules

We first give the propagation rules for split/unite operations then for unfold/fold operations in a pairwise way.

(1) *Propagation of qualified functional dependencies in application of a split/unite operation.* Let $R(A_1, \ldots, A_n, B)$ be an original relation with $dom(B) = \{b_1, \ldots, b_m\}$, and $b_i(A_1, \ldots, A_n)$ for $i = 1, \ldots, m$ be the transformed relations using split(R,

[2] This study is based on relations (consisting of the schemas and instances) instead of relational schemas, as a discrepant schema transformation may change data to attribute names or relation names, or converse.

B), i.e., the distinct values of B in R become the relation names of the transformed relations. Let $\overline{X}$ be a mixed set of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $Y \subseteq \{A_1, \ldots, A_n\}$ be a set of regular attributes; let $\mathbb{R} \subseteq \{b_1, \ldots, b_m\}$ be a set of relations. We have the following rule:

(P1) $R(B_{\sigma=\mathbb{R}}, \overline{X} \rightarrow Y)$ holds iff $\mathbb{R}(\overline{X} \rightarrow Y)$ holds.

The same rule holds for the unite operation, when $\{b_1, \ldots, b_m\}$ is the set of the original relations, and R is the transformed relation using unite($\{b_1, \ldots, b_m\}$, B). □

Rule P1 means that in application of a split operation, the restriction on the values of the attribute B in the given qualified functional dependency becomes the restriction on the relation set over which the inferred qualified functional dependency holds, as the values of B become the names of the transformed relations. We hereby give an example to apply this rule.

Example 6.2. *Suppose in Figure 1.1, we transform $DB1$ into $DB3$ by a split operation. Given the functional dependency in the relation of $DB1$: $\{p\#, s\#, month\} \rightarrow price$ which is equivalent to a set of qualified functional dependencies in the same relation (by the disassembly rule A11):*

$$Supply(month_{\sigma=\{mi\}}, p\#, s\# \rightarrow price)$$

for each attribute value $mi = jan, \ldots, dec$, we can derive a functional dependency for each relation of $DB3$ by applying the propagation rule P1 to each of the qualified functional dependencies in $DB1$, i.e.,

$$mi(p\#, s\# \rightarrow price)$$

for each relation name $mi = jan, \ldots, dec$ in $DB3$. That is, the qualification on the

attribute value of month in the original qualified functional dependency becomes the qualification on the relation of the derived qualified functional dependency. □

A unite operation is a qualified functional dependency preserving transformation, as described below:

Lemma 6.2. *In application of unite, Rule P1 changes any qualified functional dependency on the original relations into an equivalent one on the transformed relation.* □

Although unite is a qualified functional dependency preserving transformation, split is not. Given the same conditions as those in Rule P1, in application of split, a qualified functional dependency $R(\overline{X} \rightarrow B)$ will not be transformed to any dependency on the transformed relations, as the values of B become the names of the transformed relations.

We then give the propagation rules of qualified functional dependencies in application of a set of unfold/fold operations. We study based on a set of unfold/fold operations instead of individual ones because some qualified functional dependencies would hold over a set of relations which are transformed together by a set of unfold/fold operations.

(2) *Propagation of qualified functional dependencies in application of a set of unfold/fold operations.* Let $R_i(A_1, \ldots, A_n, B, C)$ for each $i = 1, \ldots, l$ be a set of original relations, and $S_i(A_1, \ldots, A_n, b_1, \ldots, b_m)$, $i = 1, \ldots, l$, be the transformed relations by performing unfold(R_i, B, C) on each relation of R_i. That is, the values of B in R_i, $\{b_1, \ldots, b_m\}$, become attribute names in S_i, and the values of C in R_i become the values of the attributes $b_1, \ldots, b_m$ in S_i. Let $\overline{X}$ be a mixed set of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $Y \subseteq \{A_1, \ldots, A_n\}$ be a set of regular attributes. Let $\mathbb{R} = \{R_{i1}, \ldots, R_{ij}\}$ be a subset of $\{R_1, \ldots, R_l\}$, and

$\mathbb{S} = \{S_{i1}, \ldots, S_{ij}\}$, a subset of $\{S_1, \ldots, S_l\}$, be the transformed relations of those in $\mathbb{R}$. We have the following rules:

(P2) $\mathbb{R}(B_{\sigma=\{b_i\}}, \overline{X} \rightarrow C)$ holds iff $\mathbb{S}(\overline{X} \rightarrow b_i)$ holds.

(P3) $\mathbb{R}(B_{\sigma=\{b_i\}}, \overline{X}, C \rightarrow Y)$ holds iff $\mathbb{S}(\overline{X}, b_i \rightarrow Y)$ holds.

(P4) $\mathbb{R}(\overline{X} \rightarrow Y)$ holds iff $\mathbb{S}(\overline{X} \rightarrow Y)$ holds.

The three rules also hold for fold operations, when $S_i, i = 1, \ldots, l$, are the original relations, and $R_i, i = 1, \ldots, l$ are the transformed relations by performing fold(S_i, B, C) on each relation of S_i. $\square$

In application of unfold operations, Rules P2 and P3 mean that the restriction on the value of the attribute B in the given qualified functional dependency becomes the restriction on the attribute name in the inferred qualified functional dependency. Rule P4 is trivial as no change happens to the attributes involved in the given qualified functional dependency during the transformation. We hereby give an example to apply Rule P2:

Example 6.3. *Suppose in Figure 1.1, we transform $DB1$ into $DB2$ by an unfold operation. Given the functional dependency in the relation of $DB1$:*
$\{p\#, s\#, month\} \rightarrow price$ which is equivalent to a set of qualified functional dependencies in the same relation:

$$Supply(month_{\sigma=\{mi\}}, p\#, s\# \rightarrow price)$$

for each attribute value $mi = jan, \ldots, dec$, we can derive a set of functional dependencies in $DB2$ by applying Rule P2 to each of the qualified functional dependencies in $DB1$, i.e.,

$$Supply(p\#, s\# \rightarrow mi)$$

for each attribute name $mi = jan, \ldots, dec$ in the relation of $DB2$. That is, the

functional dependency $\{p\#, s\#\} \rightarrow \{jan, \ldots, dec\}$ *holds in the relation of* $DB2$. □

Both fold and unfold operations are not qualified functional dependency preserving transformations. However, fold operations preserve qualified functional dependencies with a certain form, as stated below:

Lemma 6.3. *Let* $S_i(A_1, \ldots, A_n, b_1, \ldots, b_m)$ *for each* $i = 1, \ldots, l$ *be a set of original relations, and* $R_i(A_1, \ldots, A_n, B, C)$, $i = 1, \ldots, l$, *be the transformed relations by performing fold*(S_i, B, C) *on each relation of* S_i. *Given any qualified functional dependency* f *on the set of the original relations, such that* f *contains at most one* b_i $(1 \leq i \leq m)$ *as a regular attribute, Rule P2, P3 or P4 changes it into a qualified functional dependency* g *on the set of the transformed relations, such that* f *is equivalent to* g, *i.e.,* f *implies* g, *and vice versa.* □

6.2.2 Deriving qualified functional dependencies in discrepant schema transformations

Using the inference rules A1 to A10 and the propagation rules P1 to P4, we can derive qualified functional dependencies in discrepant schema transformations. Qualified functional dependencies are powerful to express a broad class of dependencies, but the inference and propagation of general qualified functional dependencies would take much time. To reduce the time complexity and focus on the dependencies which are popular in practice, we study a special class of qualified functional dependencies, called *simple qualified functional dependencies*, defined below. Intuitively, we are interested in those dependencies which are represented as functional dependencies in a canonical schema (a canonical schema is one in which all interesting information is modelled as attribute values, e.g., $DB1$ in Figure 1.1).

Definition 6.3. *We say a qualified functional dependency* f*:* $\mathbb{R}(\overline{X} \rightarrow Y)$ *is* simple *if it satisfies 3 conditions:*

1. *The set of relations* $\mathbb{R}$ *satisfies either* $\mathbb{R} = dom(A)$ *for some attribute* A *whose values are modelled as relation names, or* $\mathbb{R}$ *contains only one relation.*

2. *The qualified functional dependency only contain regular attributes.*

3. *For each attribute set* $Z = \{b_i |$ *the attribute name* b_i *is a value of an attribute* B*, and the values of the attribute* b_i *are from the domain of another attribute* $C\}$*, either* $\overline{X} \cap Z = \emptyset$*, or* $|\overline{X} \cap Z| = 1$ *and* $Y \cap Z = \emptyset$*.* $\square$

Condition 3 means that either the left hand side of f has no attributes of b_i's, or the left hand side of f has one attribute of b_i's and the right hand side of f has no b_i's.

For Example 1.2, given the two schemas $BS1(isbn, title, price)$ and $BS2(isbn, title, price)$, the dependencies $\{BS1, BS2\}(isbn \rightarrow title)$, $BS1(isbn \rightarrow price)$ and $BS2(isbn \rightarrow price)$ are simple qualified functional dependencies as they hold over either all the relations whose names are from $dom(store)$ or individual relations. These dependencies can be changed to functional dependencies $isbn \rightarrow title$ and $\{isbn, store\} \rightarrow price$ in the integrated schema $Book(isbn, store, title, price)$.

For another example, in $DB4$ of Figure 1.1, given the qualified functional dependencies $s_i(p\# \rightarrow jan, \ldots, dec)$ (for each $i = 1, \ldots, n$) such that the attribute names $jan, \ldots, dec$ are from the domain of another attribute $month$, as $jan, \ldots, dec$ only occur on the right hand sides of the qualified functional dependencies (i.e., satisfying Condition 3 of Definition 6.3), these qualified functional dependencies are simple qualified functional dependencies. These simple qualified functional dependencies can be transformed to functional dependencies in the schema of $DB1$ of Figure 1.1.

The assumption of simple qualified functional dependencies restricts the class of qualified functional dependencies we considered in schema transformation, and has no impact on the schemas and transformations. With this assumption, our purpose is: (1) to control the complexity of the derivation of qualified functional dependencies, and (2) to keep the generality of our method, i.e., to derive a class of common and useful qualified functional dependencies in schema transformations.

Now the problem becomes: given a discrepant schema transformation and a set of simple qualified functional dependencies that hold over the original relations of the transformation, compute (a cover of) all the simple qualified functional dependencies that hold over the target transformed relations.

A naive idea would be: for each step of the discrepant schema transformation, we first apply the inference rules to compute the closure of the qualified functional dependencies on the original relations, then apply the propagation rules to get the qualified functional dependencies on the transformed relations. Finally, we get the qualified functional dependencies on the target transformed relations. For ease of reference, we call an algorithm based on the naive idea *NAIVE_PROPAGATE* (the formal algorithm is omitted).

Theorem 6.3. *Algorithm NAIVE_PROPAGATE is sound and complete to infer simple qualified functional dependencies in a lossless discrepant schema transformation.*

Proof. The proof is given in Appendix A.6. □

The computation of the closure of qualified functional dependencies is necessary in NAIVE_PROPAGATE. As we mentioned in Section 6.2.1, a restructuring operation may not preserve all the qualified functional dependencies of the original relations in the transformed relations.

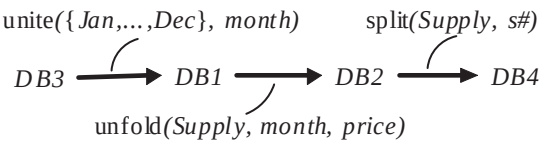
For example, given a relation $R(A, B, C)$, we split (i.e., with an operation split(R, B)) it to a set of relations $b_i(A, C)$ for $i = 1, \ldots, n$ and $b_i \in dom(B)$. Suppose in R, we have two functional dependencies $A \rightarrow B$ and $B \rightarrow C$. These two functional dependencies cannot be changed to any equivalent (qualified) functional dependencies in the transformed schemas. However, they imply a functional dependency $A \rightarrow C$ in R which can be represented as a qualified functional dependency $\{b_1, \ldots, b_n\}(A \rightarrow C)$ in the transformed schemas.

In general, in the derivation of qualified functional dependencies in a discrepant schema transformation, because a restructuring operation may not be qualified functional dependency preserving, we should not only consider the given qualified functional dependencies on the original relations, but also those not given but can be preserved in the schema transformation. In NAIVE_PROPAGATE, we consider all the qualified functional dependencies by computing qualified functional dependency closures.

However, the computation of a qualified functional dependency closure takes exponential time at least, which makes the method impractical. Instead of applying the inference and propagation rules directly, we use some "quick propagation rules" which are derived from the inference rules A1 to A10 and the propagation rules P1 to P4 to infer qualified functional dependencies in discrepant schema transformations, without computing qualified functional dependency closures. The basic idea of the quick propagation rules is: given a set (not necessary a closure) of qualified functional dependencies F on the original relations, we propagate not only the dependencies in F, but also those which are not in but implied by F and can be propagated during a schema transformation. The quick propagation rules, i.e., Algorithms INFER_SPLIT, INFER_UNITE, INFER_UNFOLD and INFER_FOLD, are given in Appendix A.7. We hereby give an example to apply these rules to

infer qualified functional dependencies in a discrepant schema transformation.

Example 6.4. *Suppose in Figure 1.1, we transform $DB3$ into $DB4$ in 3 steps:*



Given a set of qualified functional dependencies in the relations of $DB3$: $mi(p\#, s\# \rightarrow price)$ for each relation name $mi = jan, \ldots, dec$, we compute the qualified functional dependencies in $DB4$ as follows.

After applying the unite operation, we get the dependencies in $DB1$ (by Rule (2) of INFER_UNITE): $Supply(month_{\sigma=\{mi\}}, p\#, s\# \rightarrow price)$ for each attribute value $mi = jan, \ldots, dec$.

After applying the unfold operation, we get the dependencies in $DB2$ (by Rule (1) of INFER_UNFOLD): $Supply(p\#, s\# \rightarrow mi)$ for each attribute name $mi = jan, \ldots, dec$.

After applying the split operation, we get the dependencies in $DB4$ (by Rule (2) of INFER_SPLIT): $s_j(p\# \rightarrow mi)$ for each relation name $s_j = s_1, \ldots, s_n$ and attribute name $mi = jan, \ldots, dec$. That is, in each relation s_j of $DB4$, the functional dependency $p\# \rightarrow \{jan, \ldots, dec\}$ holds. □

Using the quick propagation rules in Algorithms INFER_SPLIT, INFER_UNITE, INFER_UNFOLD and INFER_FOLD, we designed a more efficient algorithm to infer qualified functional dependencies in discrepant schema transformations. That is, in each transformation step, we call one of the four inference algorithms to infer the qualified functional dependencies on the transformed relations, and finally get the qualified functional dependencies on the target transformed relations. For

ease of reference, we call the algorithm using the quick propagation rules *EFFICIENT_PROPAGATE* (the algorithm is given in Appendix A.7), in comparison with the naive algorithm using the inference rules and propagation rules directly.

Theorem 6.4. *Algorithm EFFICIENT_PROPAGATE is sound and complete to infer simple qualified functional dependencies in a lossless discrepant schema transformation.*

Proof. The proof is given in Appendix A.8. □

6.2.3 Complexities of Algorithms EFFICIENT_PROPAGATE and CLOSURE

Given a discrepant schema transformation and a set qualified functional dependencies on the original relations, let G be a minimum cover of the qualified functional dependencies holding over the target transformed relations. Then $|G|$ could be exponential to the number of the transformation steps. That is, the time complexity of Algorithm EFFICIENT_PROPAGATE could be exponential to the number of transformation steps. A precise characterization of large and significant input classes for which the algorithm runs in polynomial time is an interesting problem.

We first derive an upper bound for the algorithm and then present two classes of inputs for which EFFICIENT_PROPAGATE behaves polynomially.

Lemma 6.4. *In Algorithm EFFICIENT_PROPAGATE, let* $T =< T_1, \ldots, T_k >$ *be a discrepant schema transformation,* U *be the set of the attributes of the original relations of* T*,* F_0 *be the set of the qualified functional dependencies holding over the original relations of* T*, and* G *be the set of the qualified functional dependencies produced by an inference algorithm (i.e., INFER_SPLIT, INFER_UNITE, INFER_UNFOLD or INFER_FOLD) during the execution of Algorithm*

EFFICIENT_PROPAGATE. If a is an upper bound of $|G|$, then the algorithm takes time $O(a^4|U|k)$.

Proof. During the i-th ($1 \leq i \leq k$) call of one of the 4 inference algorithms to infer qualified functional dependency for the set of restructuring operations T_i, it would take the most time when T_i is a set of unfold operations, and $O(a^3)$ sequences of triple qualified functional dependency are examined (see Rules 8 and 9 of Algorithm INFER_UNFOLD). For each sequence, say $(f1, f2, f3)$, the following 2 tasks are performed:

1. Checking whether the triple qualified functional dependency ($f1$, $f2$, $f3$) produce a qualified functional dependency according to Rules 8 and 9 of Algorithm INFER_UNFOLD. This task is feasible in $O(|U|)$ time. Note although the set of attributes U would be changed during a transformation, the number of the attributes in an inferred qualified functional dependency would always be $O(|U|)$ during the execution of EFFICIENT_PROPAGATE.

2. Inserting the produced qualified functional dependency into the set G. This requires less than $O(|U|a)$ time.

So for each step T_i of T, it takes $O(a^4|U|)$ time to infer qualified functional dependencies. The total time of EFFICIENT_PROPAGATE is $O(a^4|U|k)$. □

We will present two simple classes of "benign" inputs for EFFICIENT_PROPAGATE. The first class of inputs contains all possible inputs, for which the number of the sets of unfold operations in the discrepant schema transformation is less than a constant c. The second class of inputs is defined through the constraints on the structure of F_0, the set of input qualified functional dependencies holding over the original relations.

Theorem 6.5. *In Algorithm EFFICIENT_PROPAGATE, let* $T =< T_1, \ldots, T_k >$ *be a discrepant schema transformation. If the number of the steps of unfold operations in* T *is less than a constant* c*, then the algorithm runs in polynomial time.*

Proof. In general, there is at most one split operation in T, which generates a set of qualified functional dependencies square in the number of input qualified functional dependencies at most. Unite and fold operations generate sets of qualified functional dependencies with the same sizes as the sets of input qualified functional dependencies. Now let's consider the increase of qualified functional dependencies with an unfold operation.

Let U be the set of the attributes of the original relations of T, F_0 be the set of the qualified functional dependencies holding over the original relations of T, and F_i, $i = 1, \ldots, k$, be the set of input qualified functional dependencies on the transformed relations of T_i. Without loss of generality, suppose that $T_{i+1}, \ldots, T_{i+k1}$ are the unfold operations in T. We have $|F_{i+k1+1}| = O(|F_i|^{3^{k1}})$, which is polynomial in $|F_i|$ under the assumption $k1 < c$. Let a be the upper bound for the cardinalities of the sets of the qualified functional dependencies produced during the execution of Algorithm EFFICIENT_PROPAGATE. Then a is polynomial in $|F_0|$. According to Lemma 6.4, the algorithm takes polynomial time w.r.t the number of input qualified functional dependencies $|F_0|$, the number of attributes $|U|$ and the number of the transformation steps k. □

Then we will give the second class of benign inputs. We first define canonical qualified functional dependencies below.

Definition 6.4. *Let* F *be a set of simple qualified functional dependencies. The set* $canonical(F)$ *contains all the simple qualified functional dependencies* $f = \mathbb{R}(X \rightarrow A)$ *in* F^+ *such that the following properties hold:*

1. Non-trivial. $A \notin X$*;*

2. Left-reduced. *For no proper subset* $Y \subset X$ *it holds* $\mathbb{R}(Y \rightarrow A)$*;*

3. Relation set increased. *For no superset* $\mathbb{S} \supset \mathbb{R}$ *it holds* $\mathbb{S}(Y \rightarrow A)$. □

Given a set of simple qualified functional dependencies F, although $|F^+|$ is always exponential in $|U|$ ($|U|$ is the number of attributes), and often exponential in $|F|$, $|canonical(F)|$ can be very small and is polynomial in $|U|$ or $|F|$ in most cases of practical relevance. For instance, suppose F is a set of conventional functional dependencies. If $|canonical(F)|$ is exponential in $|U|$ or $|F|$, then there must exist an attribute of U which has exponentially many minimal keys. This is not a very common situation. The second class of benign inputs is based on this observation.

Theorem 6.6. *Given a discrepant schema transformation, let* F *be the set of the qualified functional dependencies on the original relations of the transformation. If* $|canonical(F)|$ *is polynomial in* $|F|$*, then EFFICIENT_PROPAGATE runs in polynomial time.* □

Proof. In a discrepant schema transformation T, for each step (i.e., a set of restructuring operations) T_i in T, let F_i be the set of the inferred non-trivial qualified functional dependencies on the transformed relations of T_i. If $|canonical(F)|$ is polynomial in $|F|$, we can prove that $|F_i| = O(|canonical(F)|)$ by induction on i. By applying Lemma 6.4, we know that Algorithm EFFICIENT_PROPAGATE takes polynomial time in this case. □

In Lemma 6.1, we give an upper-bound of Algorithm CLOSURE that is used to compute an attribute closure w.r.t. a set of qualified functional dependencies. The theorem below says that if a set of qualified functional dependencies G is derived from a set of simple qualified functional dependencies in a discrepant schema

transformation with few fold operations, an attribute closure w.r.t. G can be computed efficiently. Note that the qualified functional dependencies of G satisfy the three conditions (i.e., single-valued qualification, disjoint regular and qualification attributes and holding over one or all relations) of the input qualified functional dependencies of Algorithm CLOSURE (see Section 6.1.3), according to Lemma A.6.

Theorem 6.7. *Given a discrepant schema transformation $T =< T_1, \ldots, T_k >$, a set of original relations $\mathbb{R}$ of T, a set of attributes U of $\mathbb{R}$, and a set of qualified functional dependencies F for $\mathbb{R}$. Let G be the set of the qualified functional dependencies derived by Algorithm EFFICIENT_PROPAGATE. If the number of the steps of fold operations in T is less than a constant c, then it takes polynomial time to compute an attribute closure w.r.t G.*

Proof. Let Z be the set of all the attributes occurring in the qualification attributes of the qualified functional dependencies of G. Let m be the cardinality of Z, and d be an upper bound for the cardinalities of the domains of the attributes of Z. Lemma 6.1 says that the computation of an attribute closure w.r.t. G takes $O(d^m|U||G|^2)$ time.

In the transformation T, only unite and fold operations would change a simple qualified functional dependency into a qualified functional dependency with qualification attributes. Note there's at most one unite operation in T if T is a non-redundant transformation. So if the number of the steps of fold operations in T is less than a constant c, then $m < c + 1$. Furthermore, as the attributes of Z are computed by fold and unite operations, the values of those attributes are attribute names or relation names of $\mathbb{R}$. So $d = max(|U|, |\mathbb{R}|)$. Consequently, it takes polynomial time to compute an attribute closure w.r.t. G. □

6.3 Uses of qualified functional dependency derivation

In this section, we first introduce some general uses of our theory of qualified functional dependency derivation in data integration/mediation systems, then discuss a specific application of our theory in a multidatabase language SchemaSQL.

6.3.1 Deriving qualified functional dependencies in data integration/mediation systems

functional dependencies are useful not only in enforcing the integrity of data, but also in different stages of schema and data transformation/integration in data integration/mediation systems, as discussed in this section.

Verifying Lossless Transformations

functional dependencies can be used to verify not only the "lossless join decomposition", but also lossless fold and unfold operations (see the reconstructibilities of fold and unfold, i.e., Properties 3 and 4 in Section 5.1).

Normalizing Integrated Schemas

Consolidating data into a single physical store has been the most effective approach to provide fast, highly available, and integrated access to related information. The applications include coalescing all the required data for a new e-business application for online transactions, and enabling sophisticated data mining of warehoused historical data. In the classical relational theory, functional dependencies are used to detect redundancy and normalize relations. Deriving functional dependencies for integrated schemas becomes important, as schema transformations may

introduce redundancy. For Example 1.2 (in Section 1.2), the integrated schema $Book(isbn, title, store, price)$ is redundant and can be normalized given the functional dependencies on it.

Detecting Duplicates and Inconsistencies in Data Integration

In data integration, the data from different databases may be duplicated or inconsistent. The duplicates and inconsistencies should be detected and resolved.

For Example 1.2, suppose two books from the two bookstores $BS1$ and $BS2$ have the same isbn number but different titles, i.e., an inconsistency occurs on the titles. This can be detected by enforcing the functional dependency $isbn \rightarrow title$ which is global and holds over the union of the two relations $BS1$ and $BS2$. After detecting the inconsistencies, we can use existing techniques, e.g., [45], to resolve them.

On the other hand, suppose the two books have the same isbn number, but different prices. This does not mean there is an inconsistency on the prices, as the functional dependency $isbn \rightarrow price$ is local and holds over each of the two relations. Instead of resolving the "inconsistencies" of the prices, we should distinguish the prices of the books of the two bookstores in an integrated schema.

Verifying Data Integrity in Materialized View Maintenance

In [31], the researchers proposed an incremental view maintenance strategy for schema-restructuring views. The work supported the integration of large yet schematically discrepant data sources into an integrated environment such as a data warehouse, while allowing for incremental propagation of updates. functional dependencies on an integrated schema can be used to verify the integrity of data in the propagation of the operations of insertion and update.

Optimizing Queries

The idea of using integrity constraints to optimize queries is not new [27, 41, 68]. In those works, functional dependencies are usually used to eliminate unnecessary conditions of selection, join and group-by. In [69], functional dependencies are also used to optimize large aggregation queries for OLAP applications.

6.3.2 Verifying SchemaSQL views

SchemaSQL is an extension to SQL. It treats data and metadata in a uniform manner, i.e., variables can range over data, attribute names and relation names. Consequently, a SchemaSQL view may define on (and generate) relations with dynamic schemas. Recently, SchemaSQL has been used to solve a broad range of problems [34, 54]. However, a SchemaSQL view definition may generate ambiguous results, as shown in the following example.

Example 6.5. *In Figure 6.1, suppose in the relation Supply, a functional dependency*

$$\{p\#, s\#, month\} \rightarrow price$$

holds. The SchemaSQL statements below define a view SupView that presents the prices of products by months:

```
create view   SupView(p#, T.month)
select        T.p#, T.price
from          Supply T
```

The above statements are similar to a SQL view definition except that a variable T.month is in the "create view" clause. The resulting view schema therefore depends on the instantiation of T.month, i.e., the values of the month attribute in

Supply

$p\#$	$s\#$	$month$	$price$
$p1$	s_1	jan	100
$p1$	s_1	feb	105
$p1$	s_2	jan	95
$p1$	s_2	feb	97

Allocated table

$p\#$	jan	feb
$p1$	100	-
$p1$	-	105
$p1$	95	-
$p1$	-	97

SupView (I1)

$p\#$	jan	feb
$p1$	100	105
$p1$	95	97

SupView (I2)

$p\#$	jan	feb
$p1$	100	97
$p1$	95	105

Figure 6.1: Ambiguous SchemaSQL view: $SupView$ may have one of the two instances I1 and I2

the Supply relation. In this case, the view has a schema of $SupView(p\#, jan, feb)$. *To evaluate this view, a temporarily "allocated table" will be temporarily generated, as shown in Figure 6.1. Each tuple in the allocated table comes from a tuple of Supply with the months modelled as the attribute names. "-" is used to denote the null value. The tuples are then merged in the allocated table, to get the final result. Two tuples are merge-able if for a common attribute, either the attribute values of the 2 tuples are the same, or at least one value is null. In the allocated table, the first tuple can be merged with the second or the 4th tuple. Then the resulting view relation is not unique for the different choices of merged tuples. Two possible results are* $SupView(I1)$ *and* $SupView(I2)$ *as shown in Figure 6.1. That is, the mapping from the original relations onto the view relations is one-to-many.* □

We say a view definition in Example 6.5 is non-unique. In general, we have:

Definition 6.5 (Unique SchemaSQL view). *Let* V *be a view definition in SchemaSQL. Let* $\mathbb{S}1 = \{R \mid R$ *is an original relation (or relation set) on which* V *is defined*$\}$, *and* $\mathbb{S}2 = \{R \mid R$ *is a view relation (or relation set) generated by* $V\}$. *If the view definition* $V : \mathbb{S}1 \mapsto \mathbb{S}2$ *is a many-to-one mapping, we say* V *is* unique. □

Intuitively, for a unique view V, given a query Q against a view relation (or relation set) $S \in \mathbb{S}2$, we have: $Q(S) = Q(V(R)) = Q \circ V(R)$ for some $R \in \mathbb{S}1$.

That is, the query Q against S is mapped onto the unique query $Q \circ V$ against the original relation (or relation set) R, if V is a many-to-one mapping.

The theorem below gives a necessary and sufficient condition to check whether a SchemaSQL view is unique by use of the functional dependencies on the view. To simplify the expression, the theorem only applies to SchemaSQL views generating individual relations without aggregations. The result can be extended to general SchemaSQL views readily.

Theorem 6.8. *A SchemaSQL view is unique iff it satisfies the following condition: if the output schema declaration through the "create view" statement of the view definition has a form of* $R(A_1, \ldots, A_n, B)$*, where* R *is the name of the view relation,* $A_1, \ldots, A_n$ *are attribute names, and* B *is a variable ranging over a set of values* $\{b_1, \ldots, b_m\}$*, then the functional dependency*

$$\{A_1, \ldots, A_n\} \rightarrow \{b_1, \ldots, b_m\}$$

holds in the view schema $R(A_1, \ldots, A_n, b_1, \ldots, b_m)$*.*

Proof. In general, when the declaration of a view schema contains a variable, the mapping from the original relations onto the view relations is many-to-many. However, if the functional dependency holds, there is only one way to merge the tuples of the allocated table in the evaluation of the view. That is, the resulting view relation is unique, and the mapping from the original relations onto the view relations is many-to-one. □

Note that according to the SchemaSQL syntax [35], there's at most one variable in the attribute list of the output schema declaration through a "create view" statement. And the above theorem implies that if a view definition does not contain a variable in the attribute list of the output schema declaration, then the view is

always unique. That is, the theorem can be used to check all the SchemaSQL views which generate individual relations without aggregations.

According to the theorem, in order to check whether a SchemaSQL view is unique, we need to infer the functional dependencies holding on the view relations. SchemaSQL queries/views can be implemented by use of the restructuring operators and the relational algebra (selection, projection, join and union) [34]. Correspondingly, we need to extend our rules and algorithms to the inference of qualified functional dependencies in a transformation including not only restructuring operations, but also the operations of the relational algebra. This would not be hard given the existing results on the inference of functional dependencies for relational algebra views [2, 22]. We hereby give an example to describe this process.

Example 6.6. *The view of Example 6.5 can be implemented in two steps: (1) project out the s# column from the Supply relation, and get an intermediate relation, say Sup*1*(p#, month, price); (2) perform* $unfold(Sup1, month, price)$*, and get the resulting view relation SupView. As Step (1) projects out s#, the given functional dependency* $\{p\#, s\#, month\} \rightarrow price$ *is lost after the projection. Consequently, no functional dependency holds in SupView, which means the view is non-unique.*

On the other hand, if the view schema contains the attribute s#, i.e., $SupView(p\#, s\#, jan, feb)$*, then the view is implemented by performing unfold(Supply, month, price). Using Rule (2) of Algorithm INFER_UNFOLD, we can derive a functional dependency* $\{p\#, s\#\} \rightarrow \{jan, feb\}$ *on SupView. According to Theorem 6.8, the view is unique.* □

6.4 Summary

Conventional functional dependencies are inadequate to represent constraints in multiple and heterogeneous databases. We introduced the new type of constraints called qualified functional dependencies, and presented a concise way to represent them. Qualified functional dependencies are the functional dependencies holding over a set of relations or a set of horizontal partitions of relations. We have presented the inference rules A1 to A10 that allow us to infer new qualified functional dependencies from given ones in fixed relational schemas. The inference rules were proven to be sound, complete and irreducible. Algorithm CLOSURE was proposed to compute attribute closures with respect to a set of qualified functional dependencies. The propagation rules P1 to P4 allow us to derive the qualified functional dependencies on transformed schemas from the qualified functional dependencies on original schemas in application of a (set of) restructuring operator(s). In discrepant schema transformation, we are mostly interested in simple qualified functional dependencies (Definition 6.3) that are the qualified functional dependencies represented as functional dependencies in a canonical schema such as $DB1$ in Figure 1.1. Using the inference rules and propagation rules, we can derive all the simple qualified functional dependencies on the transformed schemas from the simple qualified functional dependencies on the original schemas in a discrepant schema transformation. However, this needs to compute qualified functional dependency closures, and therefore is slow. To avoid the computation of qualified functional dependency closures, we derived the quick propagation rules from the inference rules A1 to A10 and the propagation rules P1 to P4, and Algorithm EFFICIENT_PROPAGATE based on the quick propagation rules to efficiently derive qualified functional dependencies in a discrepant schema transformation. Note that EFFICIENT_PROPAGATE computes a cover of (rather than all) the qualified

functional dependencies on the transformed schemas.

Our theory of qualified functional dependency derivation is useful in data integration/mediation systems and in multidatabase interoperation. It can be used to verify lossless transformations, normalize integrated schemas, detect duplicate and inconsistency and verify data integrity in materialized view maintenance. SchemaSQL is a multidatabase language which was used to solve many problems. However, we found that a SchemaSQL view may be ambiguous. By deriving qualified functional dependencies for a view, we can verify the correctness of it.

Chapter 7

Resolving schematic discrepancies in the integration of ER schemas

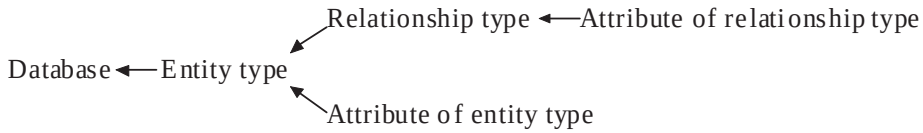
We proposed a framework to represent the meta information (context) of a construct of an ER schema as a set of meta-attributes with metadata. Schematic discrepancy occurs when metadata in one schema correspond to attribute values in the other schema. We resolved the schematic discrepancies of ER schemas by removing the context of schema constructs. The propagation of cardinality constraints is involved in the schema transformation. The resolution algorithms preserve the information and cardinality constraints in schema transformation.

7.1 Meta information of schema constructs

Conceptual modelling is always done within a particular context. In particular, the context of an entity type, relationship type or attribute is the meta-information relating to its source, classification and property.

For example, an entity type JAN_PROD modelling the products supplied in January has a context of the month January (i.e., 'JAN').

In an ER schema, contexts are usually at four levels: *database, entity type, relationship type* and *attribute*. In general, we have the following hierarchy of inheritance relations between contexts at different levels:



That is, an entity type may "inherit" a context from a database (i.e., the context of a database applies to the entities), a relationship type may "inherit" a context from its involving entity types and so on.

The inheritance hierarchy actually reflects the order in which an ER schema is built up. Given an application to be modelled, we first identify entity types, then the attributes of the entity types and the relationship types among the entity types, and finally the attributes of the relationship types. Correspondingly, we first decide the context of a database in which an ER schema is modelled, then the contexts of entity types which may inherit a context from the database, and so on.

We propose to represent the context of schema constructs using ontologies. We treat an ontology as the specification of the representational vocabulary for a shared domain of discourse which includes the definitions of entity types, relationship types, attributes of entity types and attributes of relationship types. We present ontologies at a conceptual level, which could be implemented by ontology languages.

For example, suppose an ontology *SupOnto* describes the concepts in the universe of product supply. It includes entity types *product*, *month*, *supplier*, a ternary relationship type *supply* among *product*, *month* and *supplier*, a binary relationship type *pm* that is a projection of the relationship type *supply* onto the entity types *product* and *month*, attributes *p#* (product number), *pname* (product name), *s#* (supplier number), *m#* (i.e., the identifier of the entity type *month*. The values of

$m\#$ are 'JAN', 'FEB', ..., 'DEC') and *price* that is an attribute of the relationship type *supply*.

We give a formal definition of context that is a set of meta-attributes with values below.

Definition 7.1. *Given an ontology, we represent an entity type (a relationship type, or an attribute* [1]*) E of an ER schema as:*

$$E = T[C_1 = c_1, \ldots, C_m = c_m, inherit\ C_{m+1}, \ldots, C_n]$$

where T is a type of the ontology (T is an entity type or relationship type if E is an entity type, is a relationship type if E is a relationship type, or is an attribute if E is an attribute), $C_1, \ldots, C_n$ are attributes of the ontology, and each c_i is a value of C_i for each $i = 1, \ldots, m$. $C_{m+1}, \ldots, C_n$ respectively have a value of $c_{m+1}, \ldots, c_n$ which are stated in a higher level context (i.e. the context of a database if E is an entity type, the contexts of entity types if E is a relationship type, or the context of an entity type/relationship type if E is an attribute).

This representation means that each instance of E is an instance of T, and satisfies the conditions $C_i = c_i$ for each $i = 1, \ldots, n$. $C_1, \ldots, C_n$ with the values constitute the context *within which E is defined; we call them* meta-attributes*, and their values* metadata *of E. We say E* inherits *the context $\{C_{m+1} = c_{m+1}, \ldots, C_n = c_n\}$. If E inherits all the meta-attributes with the values of a higher level context, we simply represent it as:*

$$E = T[C_1 = c_1, \ldots, C_m = c_m, inherit\ all]$$

[1]Note that as the context of a database would be handled in the entity types which inherit it, we ignore it in the following definition.

For easy reference, we call the set $\{C_1 = c_1, \ldots, C_m = c_m\}$ *the* self context*, and* $\{C_{m+1} = c_{m+1}, \ldots, C_n = c_n\}$ *the* inherited context *of* E. $\square$

Either self or inherited contexts could be empty. In the example below, we represent the entity types, relationship types and attributes in ER schemas using the ontology *SupOnto*.

Example 7.1. *Suppose we are given three ER schemas* $DB1$*,* $DB2$ *and* $DB3$ *of Figure 7.1. They model the similar supply information of products, i.e., product numbers, product names, suppliers and the supplying prices in each month. In* $DB1$*, the supply relationships are modelled as a ternary relationship type SUP. In* $DB2$*, the entity type JAN_PROD models the products supplied in the month of January, and the attributes S1_PRICE, . . . , Sn_PRICE means the prices of the products by the suppliers S1, . . . , Sn. For example, the attribute S1_PRICE of the entity type JAN_PROD means the prices of the products supplied in January by the supplier S1. In* $DB3$*, the relationship type JAN_SUP models the supply relationships between products and suppliers in January. Note that JAN_SUP is a selection of the ternary relationship type SUP of* $DB1$ *(when the value of M# is 'JAN').*

In relational databases, these ER schemas correspond to the following relational schemas (i.e., each entity type having more than one attribute and each relationship type would be transformed into a relation):

DB1: PROD(P#, PNAME), SUP(P#, S#, M#, PRICE)

DB2: JAN_PROD(P#, PNAME, S1_PRICE, . . . , Sn_PRICE),

$\vdots$

DEC_PROD(P#, PNAME, S1_PRICE, . . . , Sn_PRICE)

DB3: PROD(P#, PNAME),

JAN_SUP(P#, S#, PRICE),

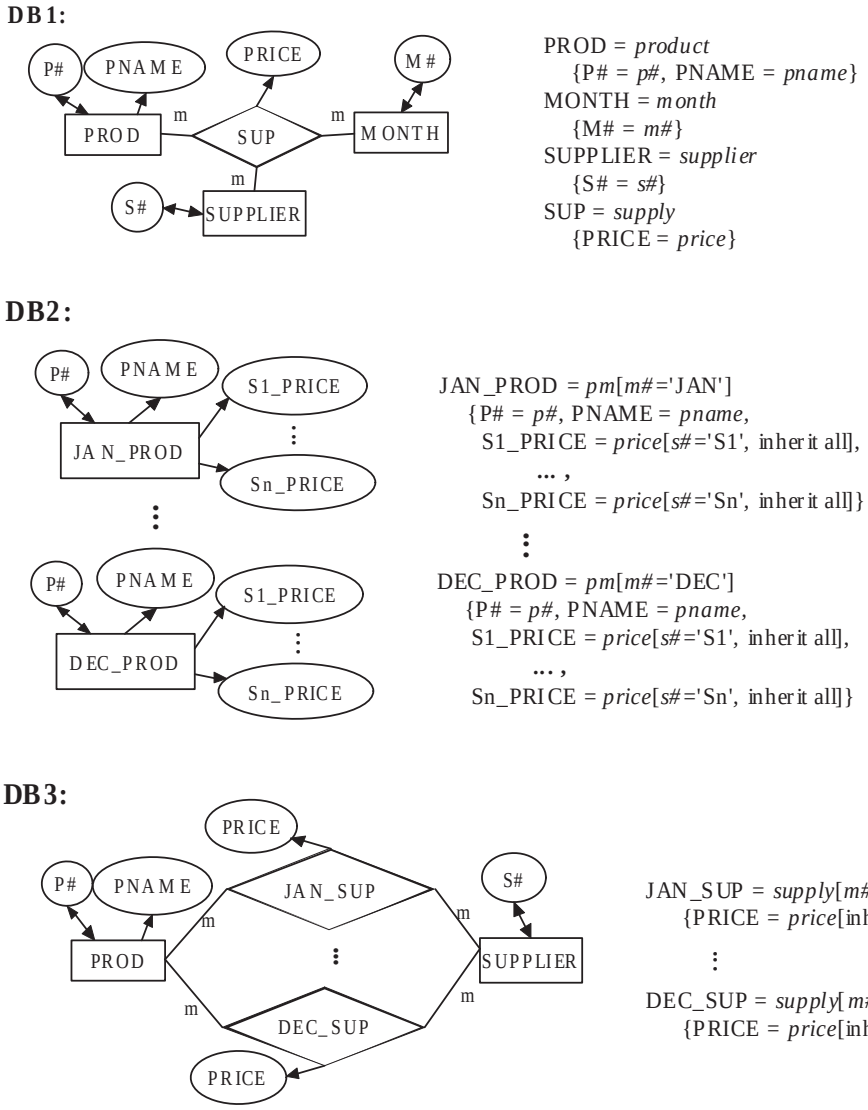


Figure 7.1: ER schemas and their contexts. Schematic discrepancies occur as months and suppliers modelled differently as the attribute values or metadata in $DB1$, $DB2$ and $DB3$

⋮

DEC_SUP(P#, S#, PRICE)

In Figure 7.1, we can represent the contexts of the schema constructs using the ontology SupOnto. For example, the entity type JAN_PROD of DB2 is represented as:

$$JAN\_PROD = pm[m\# = \text{`JAN'}].$$

That is, the context of JAN_PROD is m# =‘JAN’. This means that JAN_PROD corresponds to a relationship type pm when the month is January, i.e., the products supplied in January.

Also in DB2, the attribute S1_PRICE of the entity type JAN_PROD is represented as:

$$S1\_PRICE = price[s\# = \text{`S1'}, \text{ inherit all}].$$

The self context of S1_PRICE is s#=‘S1’, and the inherited context (from the entity type JAN_PROD) is m#=‘JAN’. This means that each value of S1_PRICE of the entity type JAN_PROD is a price of a product supplied by supplier S1 in January.

In DB3, the relationship type JAN_SUP is represented as:

$$JAN\_SUP = supply[m\# = \text{`JAN'}].$$

This means that each relationship of JAN_SUP corresponds to a relationship of supply when the month is January.

Also in DB3, the attribute PRICE of the relationship type JAN_SUP is represented as:

$$PRICE = price[\text{inherit all}].$$

PRICE inherits the context m#=‘JAN’ from its relationship type JAN_SUP. This means that each value of PRICE is a supplying price in January. □

In schema integration, contexts should be declared by the owners of source schemas. Once declared, our integration system can detect the schema matching information from the contexts automatically. For example, two entity types, two relationship types or two attributes are equivalent to each other if they correspond to the same ontology type and have the same context (possibly empty context). We can also detect schematic discrepancy that is defined below.

Definition 7.2. *Two schemas are* schematically discrepant *from each other iff some metadata in one schema correspond to the attribute values in the other schema. We call the meta-attributes whose values correspond to attribute values in other schemas* discrepant meta-attributes. □

The schemas of Figure 7.1 are schematically discrepant from each other. For example, the values of the attribute M# in $DB1$ correspond to the metadata of the relationship types in DB3. In this case, $m\#$ is a discrepant meta-attribute of the relationship types in $DB3$. The values of the attribute M# in $DB1$ correspond to the metadata of the entity types in $DB2$, and the values of the attribute S# in $DB1$ correspond to the metadata of the attributes S1_PRICE, ..., Sn_PRICE in $DB2$.

This definition of schematic discrepancy is an extension to that of [34] in which schematic discrepancy refers to the correspondence between attribute names (or relation names) and attribute values.

In Section 7.2, we will resolve schematic discrepancies by transforming metadata into attribute values, e.g., transforming $DB2$ and $DB3$ into a form of $DB1$ in Figure 7.1.

7.2 Resolution of schematic discrepancies in the integration of ER schemas

In schema integration, the 4 kinds of schematic discrepancies should be resolved in the order of context inheritance presented in Section 7.1, i.e., first for entity types, then relationship types, finally attributes of entity types and attributes of relationship types. The resolutions of the other semantic heterogeneities follow the resolution of schematic discrepancies. In general, given a set of ER schemas, we can integrate them in 4 steps:

1. Call the algorithms ResolveEnt, ResolveRel, ResolveEntAttr and ResolveRelAttr (introduced below) in order to resolve the schematic discrepancies of entity types, relationship types, attributes of entity types and attributes of relationship types.

2. Resolve the other semantic heterogeneities of naming conflicts, key conflicts, structural conflicts and constraint conflicts, using existing methods, e.g., [39].

3. Merge the transformed schemas. Equivalent entity types, relationship types and attributes are superimposed. Some constraint conflicts may need to be resolved during the merging [38].

4. Remove the redundant relationship types which can be derived from the others. Create special relationship types ISA, UNION, INTERSECT or DECOMPOSE among entity types of the integrated schema.

We present the 4 algorithms ResolveEnt, ResolveRel, ResolveEntAttr and ResolveRelAttr, i.e., the resolutions of schematic discrepancies for entity types, relationship types, attributes of entity types and attributes of relationship types one by one. Examples are provided to understand each algorithm. The resolution is implemented

by transforming discrepant meta-attributes into attributes of entity types. The transformation keeps the cardinalities of attributes and entity types, and therefore preserves functional dependencies and multivalued dependencies (proven in Section 7.3). Note that in the presence of context, the values of an attribute depend on not only the key values of the entity type/relationship type, but also the metadata of the attribute.

To simplify the presentation, we assume schema constructs only have discrepant meta-attributes, leaving out other meta-attributes that will not cause schematic discrepancies. Actually, non-discrepant meta-attributes will not be changed in schema transformation.

7.2.1 Resolving schematic discrepancies for entity types

Given an ER schema, we resolve the schematic discrepancies of the entity types of the schema in 2 steps. In Step 1, we resolve the schematic discrepancies of each entity type, and in Step 2, we merge the equivalent schema constructs in the transformed schema. Step 1 is further divided into 3 sub-steps. Given an entity type E, in Step 1.1, we transform the discrepant meta-attributes of E into the attributes of entity types, and relate the entity types in a relationship type. Then in Step 1.2, we handle the attributes of E according to the ways the attributes inherit the context of E. Finally in Step 1.3, we handle the relationship types involving E according to the ways the relationship types inherit the context of E.

We first show two examples of the resolution of schematic discrepancies of entity types, which focus on handling attributes and handling relationship types respectively, and then give the general algorithm.

Example 7.2. *In DB2 of Figure 7.1, the entity types JAN_PROD, . . . , DEC_PROD have the same discrepant meta-attribute m#. In Figure 7.2, we resolve the schematic*

discrepancies of these entity types in two steps.

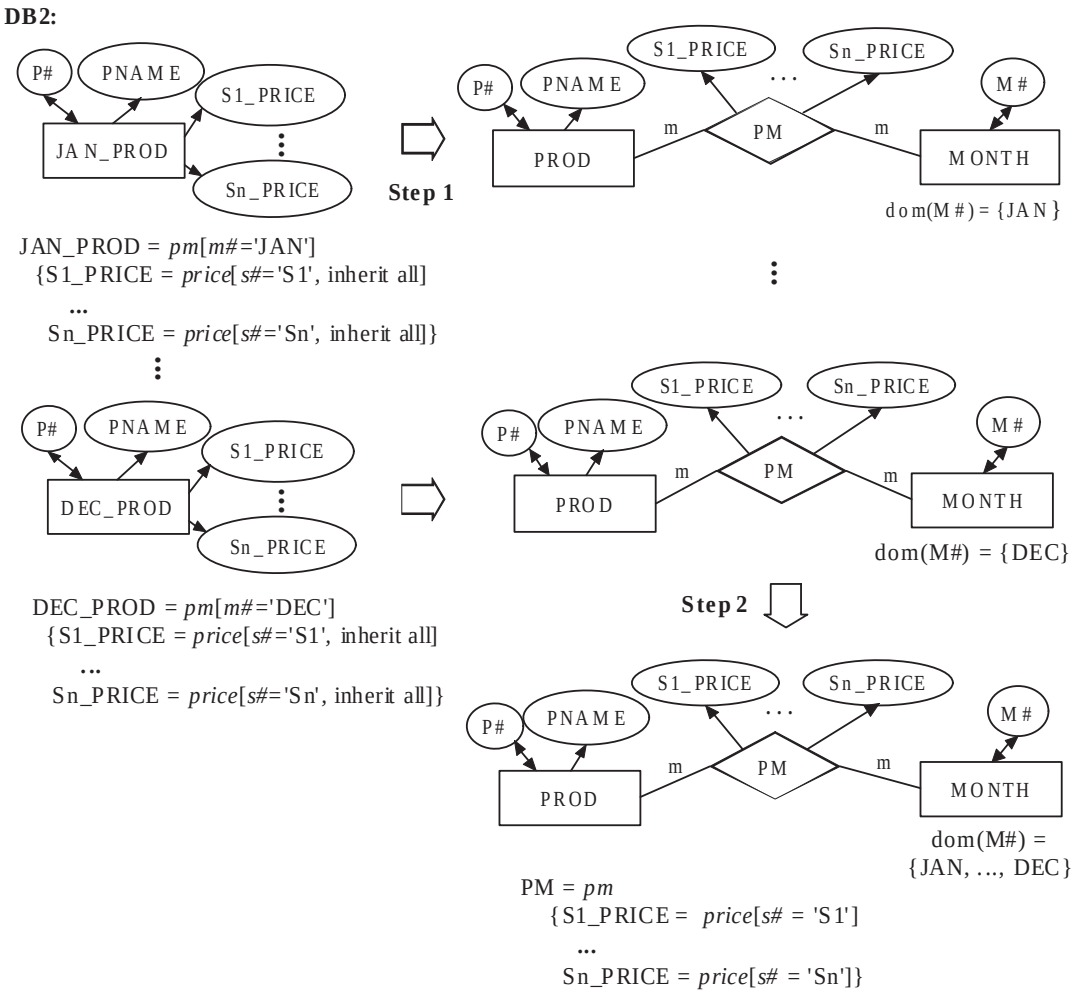


Figure 7.2: Resolve schematic discrepancies for entity types: handle attributes

In Step 1, for each entity type of DB2, say JAN_PROD = pm[m#='JAN'], we represent the discrepant meta-attribute m# as an attribute M# (with the only value 'JAN') of a new created entity type MONTH = month. As in the ontology, pm is a binary relationship type between the entity types product and month, after removing the context, we change the entity type JAN_PROD into an entity type PROD = product (with all the entities of JAN_PROD), and construct a relationship type PM = pm to associate the entity types PROD and MONTH.

Then we handle the attributes of JAN_PROD. As PNAME has nothing to do with the context of the entity type, it becomes an attribute of PROD. However, S1_PRICE, ..., Sn_PRICE inherit the context m#='JAN', i.e., their values depend on not only the product numbers, but also the month January. So they become the attributes of the relationship type PM. Note as the context of JAN_PROD that is the inherited context of the attributes S1_PRICE, ..., Sn_PRICE is removed, these attributes only have the self context s#='Si' left for i=1, ..., n (the discrepant meta-attribute s# will be resolved in Algorithm ResolveRelAttr later).

Similarly, we can resolve the schematic discrepancies of the other entity types FEB_PROD, ..., DEC_PROD.

Then in Step 2, the equivalent entity types, relationship types and attributes are merged respectively. Their domains are united. □

The schema transformation (Steps 1 and 2) of Figure 7.2 actually is the application of an "unite" operation (introduced in Section 3.1) on the entity types

$$JAN\_PROD(P\#, PNAME, S1\_PRICE, \ldots, Sn\_PRICE)$$
$$\vdots$$
$$DEC\_PROD(P\#, PNAME, S1\_PRICE, \ldots, Sn\_PRICE)$$

in $DB2$, such that the the metadata (i.e., months) of these entity types become the attribute values of the entity type $MONTH$, and the entity types $JAN\_PROD, \ldots,$ DEC_PROD are transformed to the relationship type

$$PM(P\#, M\#, S1\_PRICE, \ldots, Sn\_PRICE)$$

between $PROD$ and $MONTH$.

In general, the resolution of schematic discrepancies for entity types (the general algorithm will be given later) is an extension of the untie operator in the ER model.

As cardinality constraints represent functional dependencies and multivalued dependencies in the ER model, the propagation of cardinality constraints in the

transformation of ER schemas corresponds to the propagation of qualified functional dependencies (or qualified multivalued dependencies) in the transformation of relational schemas (introduced in Section 6.2).

For Figure 7.2, in $DB2$, we have two kinds of qualified functional dependencies. As the attribute $PNAME$ has nothing to do with the context (i.e., months) of the entity types, the functional dependency $P\# \rightarrow PNAME$ holds over the union of of the entity types $JAN\_PROD, \ldots, DEC\_PROD$, i.e., we can represent it as a qualified functional dependency

$$\{JAN\_PROD, \ldots, DEC\_PROD\}(P\# \rightarrow PNAME).$$

On the other hand, the attributes $S1\_PRICE, \ldots, Sn\_PRICE$ inherit the context (months) of the entity types, the functional dependency $P\# \rightarrow S1\_PRICE, \ldots, Sn\_PRICE$ holds in each entity type of $JAN\_PROD, \ldots, DEC\_PROD$, i.e., we can represent it as a set of qualified functional dependencies

$$M_i\_PROD(P\# \rightarrow S1\_PRICE, \ldots, Sn\_PRICE)$$

for each $M_i = JAN, \ldots, DEC$. Note that now the qualification (i.e., the set of entity types) of a qualified functional dependency is specified through the (inherited) contexts of attributes.

Then in the transformed schema of Figure 7.2, we can derive a functional dependency

$$P\# \rightarrow PNAME$$

on the $PROD$ entity type and a functional dependency

$$\{P\#, M\#\} \rightarrow \{S1\_PRICE, \ldots, Sn\_PRICE\}$$

on the PM relationship type, using a method similar to the derivation of qualified functional dependencies in application of an unite operation in the relational model.

In general, the propagation of cardinality constraints in the ER model (that will be discussed in Section 7.3) is an extension of the propagation of qualified functional dependencies in the relational model.

Then we show the other example in which we need to deal with relationship types in the resolution of schematic discrepancies of entity types.

Example 7.3. *In Figure 7.3, we give another ER schema DB4 modelling the similar information as those in Figure 7.1. In DB4, each entity type of JAN_PROD, . . . , DEC_PROD models the products supplied in one month, and each relationship type SUP_i, i=1, . . . , 12, models the supply relationships in the i-th month. Note in DB4, we have a constraint that is none in the schemas of Figure 7.1: "in each month, a product is uniquely supplied by one supplier." This constraint (i.e., a functional dependency P# → S#) is represented as a cardinality constraint on each relationship type SUP_i.*

In Figure 7.3, we resolve the schematic discrepancies of the entity types JAN_PROD, . . . , DEC_PROD in 3 steps. In Step 1, for each of these entity type, say JAN_PROD = pm[m#='JAN'], we transform the discrepant meta-attribute m# to an attribute M# of a new created entity type MONTH = month, and connect the entity types PROD and MONTH with a relationship type PM = pm.

Then we handle the relationship type SUP_1 that involves the entity type JAN_PROD. As SUP_1=supply[inherit all], i.e., it inherits the context m#='JAN' from JAN_PROD, and we have removed the context of JAN_PROD, SUP_1 becomes a ternary relationship type SUP=supply connecting the entity types PROD, MONTH and SUPPLIER.

Similarly, we can transform the entity types FEB_PROD, . . . , DEC_PROD and the relationship types SUP_2, . . . , SUP_{12}.

Then in Step 2, the equivalent entity types, relationship types and attributes are

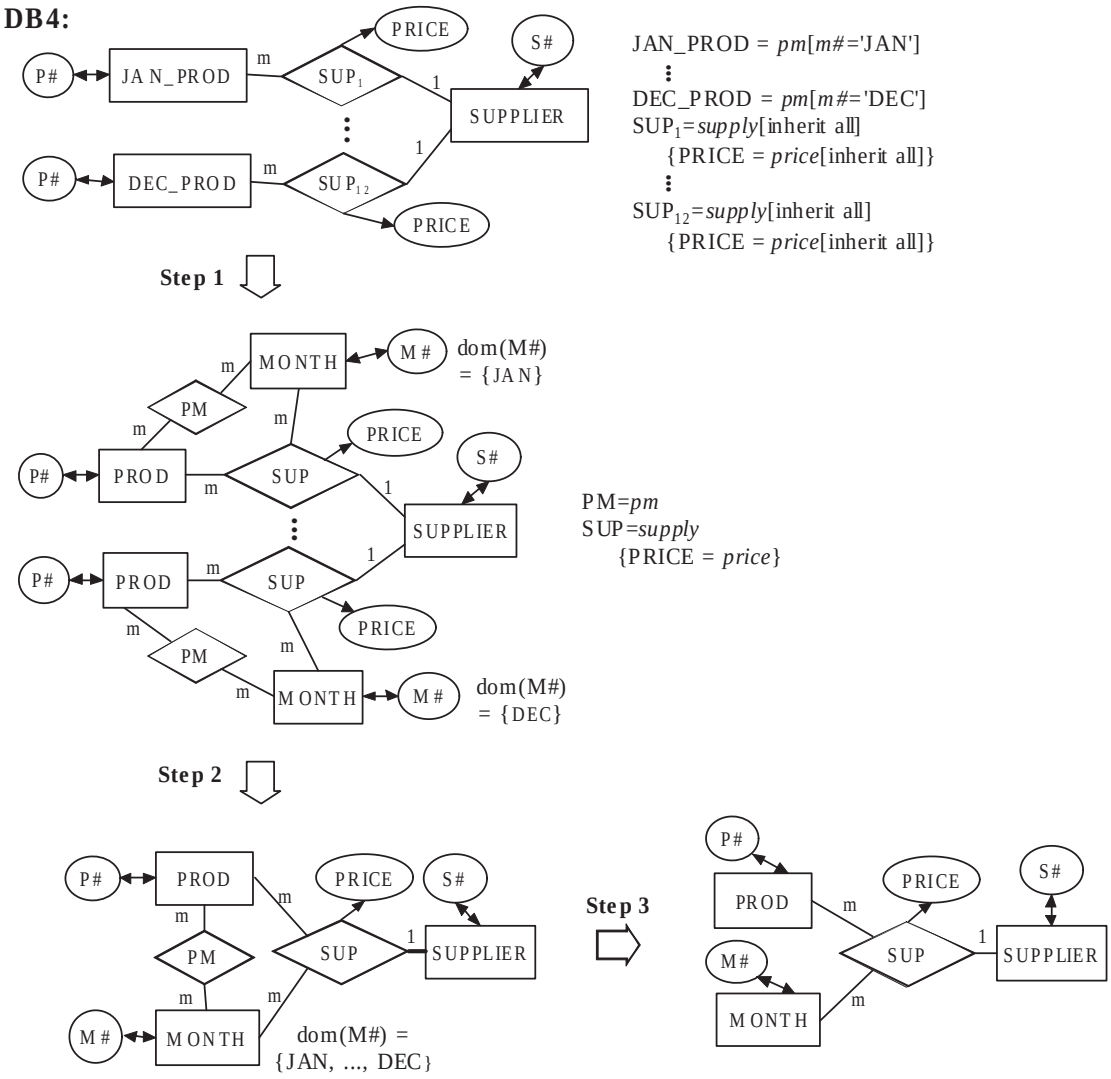


Figure 7.3: Resolve schematic discrepancies for entity types: handle relationship types

merged respectively. Their domains are united.

Finally in Step 3, as in the ontology, the relationship type pm is a projection of the relationship type supply, the relationship type PM of the transformed ER schema is redundant and therefore removed. Note that this step is not included in Algorithm ResolveEnt. Instead, it will be performed later in a main integration algorithm calling the resolution algorithms (Section 8.5).

Note that the cardinality constraints on the relationship types SUP_i's of DB4 are represented as an equivalent cardinality constraint (i.e., a functional dependency $\{P\#, M\#\} \rightarrow S\#$) on the relationship type SUP of the transformed schema. This issue will be studied in detail in Section 7.3. □

The general algorithm is given below.

Algorithm ResolveEnt

Given an ER schema DB, the algorithm produces a schema DB' transformed from DB such that all the discrepant meta-attributes of the entity types are transformed into the attributes of entity types.

Step 1 *Resolve the discrepant meta-attributes of an entity type.*

Let $E = T[C_1 = c_1, \ldots, C_l = c_l, inherit\ C_{l+1}, \ldots, C_m]$ be an entity type of DB, where T is a relationship type among m+1 entity types $T_1, \ldots, T_m, T_{m+1}$ in the ontology, and $C_1, \ldots, C_m$ are m discrepant meta-attributes that are the identifiers of $T_1, \ldots, T_m$. Each $C_i, i = 1, \ldots, m$, has a value of c_i. Let C_{m+1} be the identifier of T_{m+1}, such that the identifier of E, $K = C_{m+1}$.

Step 1.1 *Transform $C_1, \ldots, C_m$ into the attributes of entity types.*

Construct m+1 entity types $E_1 = T_1, \ldots, E_m = T_m, E_{m+1} = T_{m+1}$ with the identifiers $K_1 = C_1, \ldots, K_m = C_m, K = C_{m+1}$ (note that the identifier of E_{m+1} is the same as the identifier of E) if they do not exist.

Each E_i (i=1, ..., m) contains one entity with the identifier $C_i = c_i$. E_{m+1} contains all the entities of E.

Construct a relationship type $R = T$ connecting $E_1, \ldots, E_{m+1}$, such that $(c_1, \ldots, c_m, k) \in R[K_1, \ldots, K_m, K]$ iff $k \in E[K]$.

end Step

Step 1.2 *Handle the attributes of E.*

Let A be an attribute (not part of the identifier) of E. A corresponds to a type A_{ont} in the ontology, and has a self context (i.e., a set of meta-attributes with values) $selfCnt$.

if A is a many-to-one or many-to-many attribute **then**

case 1 A does not inherit any context of E:

A becomes an attribute of E_{m+1}, such that

$(k, a) \in E_{m+1}[K, A]$ iff $(k, a) \in E[K, A]$.

end case

case 2 $A = A_{ont}[selfCnt, inherit\ all]$, i.e., A inherits all the context $\{C_1 = c_1, \ldots, C_m = c_m\}$ from E:

Construct an attribute $A' = A_{ont}[selfCnt]$ of R, such that

$(c_1, \ldots, c_m, k, a) \in R[K_1, \ldots, K_m, K, A']$ iff $(k, a) \in E[K, A]$.

A' has the same cardinality as A.

end case

case 3 A inherits some context from E. Without losing generality, let $A = A_{ont}[selfCnt, inherit\ C_1, \ldots, C_j]$ for $1 \leq j < m$:

Construct a relationship type R' connecting E_{m+1} and $E_1, \ldots, E_j$.

Construct an attribute $A' = A_{ont}[selfCnt]$ of R', such that

$(c_1, \ldots, c_j, k, a) \in R'[K_1, \ldots, K_j, K, A']$ iff $(k, a) \in E[K, A]$.

A' has the same cardinality as A.

end case

else

/* *A is a one-to-one or one-to-many attribute, i.e., A determines the identifier of E in the context. We keep the inherited context of A, and delay the resolution of it in Algorithm ResolveEntAttr, the resolution for attributes of entity types, in which A will be transformed to the identifier of an entity type to preserve the cardinality constraint.* */

Construct an attribute $A' = A_{ont}[Cnt]$ of E_{m+1}, where Cnt is the self context of A' that is the union of the self and inherited contexts of A, such that

$(k, a) \in E_{m+1}[K, A']$ iff $(k, a) \in E[K, A]$.

end if

end Step

Step 1.3 *Handle the relationship types involving the entity type E in DB.*

Let $R1$ be a relationship type involving E in DB, and S be a sequence of the identifiers of all the entity types involved in $R1$. We transform $R1$ into a relationship type $R1'$ as below.

if $R1$ has no attributes, or only has many-to-one and many-to-many attributes **then**

case 1 $R1$ does not inherit any context of E:

Replace E with E_{m+1} in $R1$, and change $R1$ to $R1'$, such that

$s \in R1'[S]$ iff $s \in R1[S]$.

/**Note that the identifier of E is the same as the identifier of* E_{m+1}.*/

Represent each functional dependency on $R1$ (that is represented as a cardinality constraint of the participating entity types in $R1$) in $R1'$.

end case

case 2 $R1$ inherits all the context $\{C_1 = c_1, \ldots, C_m = c_m\}$ from E:

Construct $R1'$ involving $E_1, \ldots, E_m, E_{m+1}$ and all the entity types in $R1$ except E, such that

$(s, c_1, \ldots, c_m) \in R1'[S, K_1, \ldots, K_m]$ iff $s \in R1[S]$.

Let $\mathbb{A} \rightarrow \mathbb{B}$ be a functional dependency on $R1$, where $\mathbb{A}$ and $\mathbb{B}$ are two sets of the identifiers of some participating entity types in $R1$.

if K, the identifier of E, is in $\mathbb{A} \cup \mathbb{B}$ **then**

Represent a functional dependency $\mathbb{A}, K_1, \ldots, K_m \rightarrow \mathbb{B}$ in $R1'$.

else

Represent the same functional dependency $\mathbb{A} \rightarrow \mathbb{B}$ in $R1'$.

end if

end case

case 3 $R1$ inherits some context, say $\{C_1 = c_1, \ldots, C_j = c_j\}$ $(1 \leq j < m)$ from E:

Construct $R1'$ involving $E_1, \ldots, E_j$, E_{m+1} and all the entity types in $R1$ except E, such that

$(s, c_1, \ldots, c_j) \in R1'[S, K_1, \ldots, K_j]$ iff $s \in R1[S]$.

Let $\mathbb{A} \rightarrow \mathbb{B}$ be a functional dependency on $R1$, where $\mathbb{A}$ and $\mathbb{B}$ are two sets of the identifiers of some participating entity types in $R1$.

if $K \in \mathbb{A} \cup \mathbb{B}$ **then**

Represent a functional dependency $\mathbb{A}, K_1, \ldots, K_j \rightarrow \mathbb{B}$ in $R1'$.

else

Represent the same functional dependency $\mathbb{A} \rightarrow \mathbb{B}$ in $R1'$.

end if

end case

In each of the 3 cases, $R1'$ and $R1$ have the same attributes, correspond to

the same relationship type of the ontology, and have the same self context. $R1'$ has no inherited context.

else

/* $R1$ *has some one-to-one or one-to-many attributes. In order to preserve the cardinality constraints of the attributes of* $R1$*, we keep the inherited context of* $R1$ *in* $R1'$*. This context would be removed in Algorithm ResolveRel and ResolveRelAttr later.* */

Replace E with E_{m+1} in $R1$, and change $R1$ to $R1'$, such that $s \in R1'[S]$ iff $s \in R1[S]$.

The context of $R1'$ is the union of the self and inherited contexts of $R1$.

end if

end Step

end Step

Step 2 *Merge equivalent entity types and equivalent relationship types.*

for each set of equivalent entity types $\mathbb{E}$ **do**

Let E be the merged entity type.

The attribute set of E is the union of the attribute sets of all the entity types of $\mathbb{E}$.

for each set of equivalent attributes of some entity types of $\mathbb{E}$ **do**

Resolve the constraint conflicts in the attributes.

/* *Algorithms to resolve constraint conflicts are given in [38].* */

Unite the domains of these equivalent attributes.

end for

end for

for each set of equivalent relationship types $\mathbb{R}$ **do**

Let R be the merged relationship type.

The attribute set of R is the union of the attribute sets of all the relationship types of $\mathbb{R}$.

Resolve the constraint conflicts in the relationship types.

/ Algorithms to resolve constraint conflicts are given in [38]. */*

for each set of equivalent attributes of some relationship types of $\mathbb{R}$ **do**

Resolve the constraint conflicts in the attributes.

Unite the domains of these equivalent attributes.

end for

end for

end Step □

7.2.2 Resolving schematic discrepancies for relationship types

In the resolution of schematic discrepancies for relationship types, we should deal with a set of entity types (participating in a relationship type) instead of individual ones. The resolution can also be performed in 2 steps: first transform the discrepant meta-attributes of relationship types into the attributes of entity types (unlike Algorithm ResolveEnt, we don't need Step 1.3 in Algorithm ResolveRel), and then merge the equivalent schema constructs in the transformed schema. We first present an example below.

Example 7.4. *In DB3 of Figure 7.1, the relationship types JAN_SUP, . . . , DEC_SUP have the same discrepant meta-attribute m#. In Figure 7.4, we resolve the schematic discrepancies of these relationship types in two steps.*

In Step 1, for each relationship type of DB3, say JAN_SUP = supply[m#='JAN'], we represent the meta-attribute m# as an attribute M# of a new created entity type MONTH. After removing the context, we change JAN_SUP into a ternary relationship type SUP = supply, to connect the entity types PROD, MONTH and

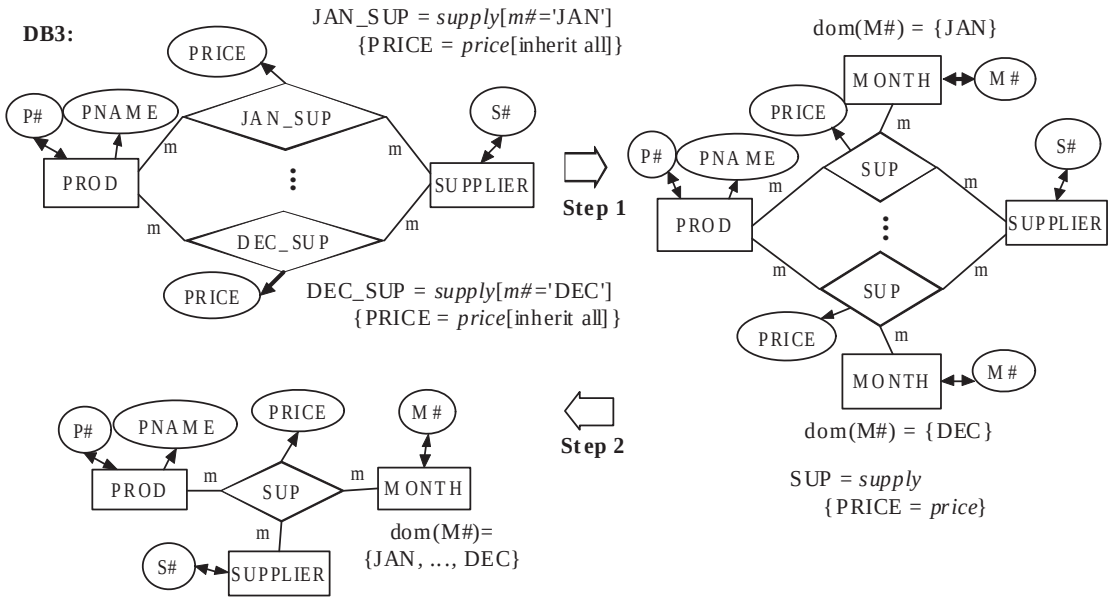


Figure 7.4: Resolve schematic discrepancies for relationship types

SUPPLIER.

Then we handle the attribute PRICE of the relationship type JAN_SUP. As PRICE = price[inherit all], i.e., its values depend on not only product numbers and supplier numbers, but also months, PRICE = price becomes an attribute of SUP in the transformed schema.

Similarly, we can transform the other relationship types FEB_SUP, . . . , DEC_SUP.

Then in Step 2, the equivalent entity types, relationship types and attributes are merged. Their domains are united. □

The schema transformation (Steps 1 and 2) of Figure 7.4 actually is the application of an "unite" operation (introduced in Section 3.1) on the relationship types

$$JAN\_SUP(P\#, S\#, PRICE)$$

$$\vdots$$

$$DEC\_SUP(P\#, S\#, PRICE)$$

in $DB3$, such that the the metadata (i.e., months) of these relationship types be-

come the attribute values of the entity type $MONTH$, and these relationship types are transformed to the relationship type

$$SUP(P\#, S\#, M\#, PRICE)$$

among $PROD$, $SUPPLIER$ and $MONTH$.

In general, the resolution of the schematic discrepancies for relationship types is an extension of the untie operator (see Section 3.1) in the ER model. Furthermore, the propagation of cardinality constraints in the schema transformation is an extension of the propagation of qualified functional dependencies in application of an unite operation (introduced in Section 6.2).

For Figure 7.4, given the functional dependencies

$$JAN\_SUP(P\#, S\# \rightarrow PRICE)$$
$$\vdots$$
$$DEC\_SUP(P\#, S\# \rightarrow PRICE)$$

on the relationship types $JAN\_SUP, \ldots, DEC\_SUP$ of $DB3$, we can derive a functional dependency on the relationship type SUP in the transformed schema, i.e.,

$$SUP(P\#, S\#, M\# \rightarrow PRICE).$$

The general algorithm ResolveRel is presented Appendix A.9. Note as the resolution of the schematic discrepancies for relationship types always follows the resolution for entity types, the relationship types input to Algorithm ResolveRel have no inherited context (see Step 1.3 of Algorithm ResolveEnt for the transformation of relationship types in the resolution of the schematic discrepancies of entity types).

7.2.3 Resolving schematic discrepancies for attributes of entity types

Given an ER schema, we resolve the schematic discrepancies of the attributes of entity types in two steps. In Step 1, given an attribute A of an entity type, we transform the discrepant meta-attributes of A into the attributes of entity types, and transform A to an attribute of a relationship type or the identifier of an entity type. Then in Step 2, we merge equivalent schema constructs of the transformed schema. We first explain the resolution algorithm by an example below.

Example 7.5. *In Figure 7.5, we give another ER schema DB5 modelling the similar information as those in Figure 7.1. In DB5, each of the* $12 \times n$ *attributes S1_JAN_PRICE, . . . , Sn_DEC_PRICE models the prices of the products supplied by one supplier in one month.*

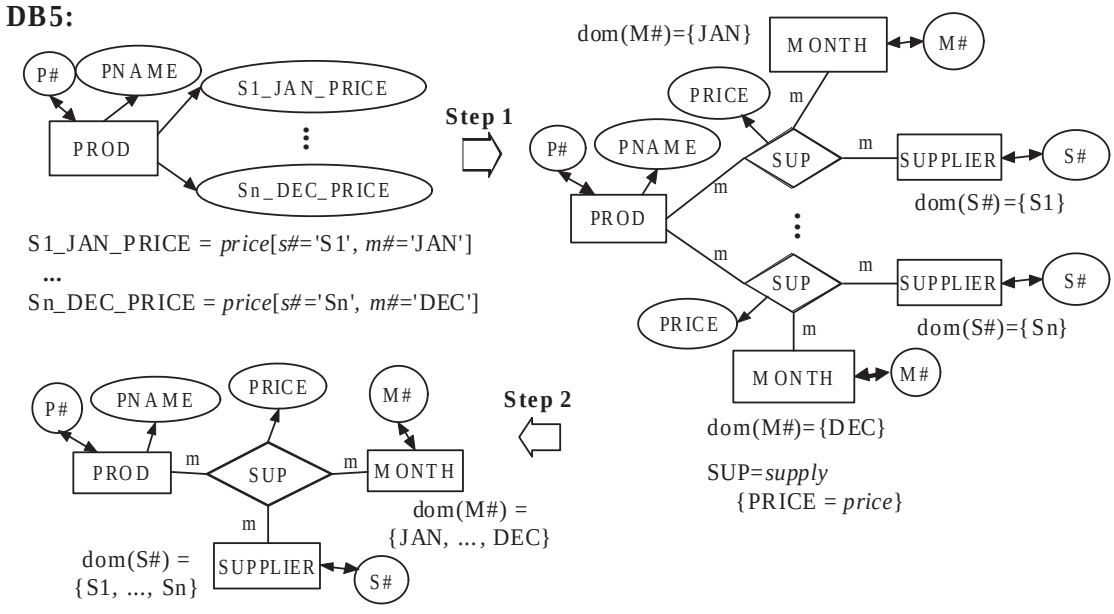


Figure 7.5: Resolve schematic discrepancies for attributes of entity types

In Figure 7.5, we resolve the schematic discrepancies of the attributes

S1_JAN_PRICE, ..., Sn_DEC_PRICE in two steps. In Step 1, for each of the attributes, say S1_JAN_PRICE = price[s#=S1, m#=JAN], we represent the discrepant meta-attributes s# and m# as the attributes of new created entity types SUPPLIER = supplier and MONTH = month. In the ontology, price is an attribute of the ternary relationship type supply. Then in the ER schema, we construct the relationship type SUP = supply to contain the attribute PRICE = price.

Similarly, we can transform the other attributes S1_FEB_PRICE, ..., Sn_DEC_PRICE.

Then in Step 2, we merge all the equivalent entity types, relationship types and attributes. Their domains are united. □

The resolution of schematic discrepancy in Figure 7.5 is actually an extended "fold" operation (see Section 3.1) on the attributes $S1\_JAN\_PRICE, \ldots,$ Sn_DEC_PRICE in $DB5$. Note that now these attributes have two (rather than one) meta attributes $s\#$ and $m\#$. In the transformation, the metadata (supplier numbers and months) of the attributes $S1\_JAN\_PRICE, \ldots, Sn\_DEC\_PRICE$ are transformed to the attribute values of the entity types $SUPPLIER$ and $MONTH$ in the relationship type SUP, and the values of the attributes $S1\_JAN\_PRICE, \ldots,$ Sn_DEC_PRICE become the values of the attribute $PRICE$ of the relationship type SUP in the transformed schema.

Furthermore, the propagation of cardinality constraints in the schema transformation of Figure 7.5 extends the propagation of qualified functional dependencies in application of a fold operation in the relational model (Section 6.2). For Figure 7.5, given a functional dependency

$$P\# \rightarrow \{S1\_JAN\_PRICE, \ldots, Sn\_DEC\_PRICE\}$$

represented as the cardinality constraints of the attributes of the entity type in $DB5$, we can derive a functional dependency

$$\{P\#, S\#, M\#\} \rightarrow PRICE$$

represented as a cardinality constraint on the relationship type SUP in the transformed schema.

The general algorithm ResolveEntAttr is presented in Appendix A.9. Note as the resolution of the schematic discrepancies for the attributes of entity types always follows the resolution for entity types, the attributes input to Algorithm ResolveEntAttr have no inherited context (see Step 1.2 of Algorithm ResolveEnt for the transformation of attributes in the resolution of the schematic discrepancies of entity types).

7.2.4 Resolving schematic discrepancies for attributes of relationship types

Given an ER schema, we resolve the schematic discrepancies of the attributes of relationship types in two steps, i.e., Step 1 of transforming the discrepant meta-attributes into the attributes of entity types and Step 2 of merging. Note unlike Algorithm ResolveEntAttr, in Algorithm ResolveRelAttr, we need to deal with a set of entity types involved in a relationship type instead of individual entity types. We first explain the resolution algorithm by an example below.

Example 7.6. *In the transformed schema of Figure 7.2, the attributes S1_PRICE, ..., Sn_PRICE of the relationship type PM represent the prices of the products supplied by the suppliers S1, ..., Sn in some months. These attributes have the same discrepant meta-attribute s#.*

In Figure 7.6, we resolve the schematic discrepancies of the attributes S1_PRICE, ..., Sn_PRICE in three steps.

In Step 1, for each attribute, say S1_PRICE = price[s#='S1'], we represent the

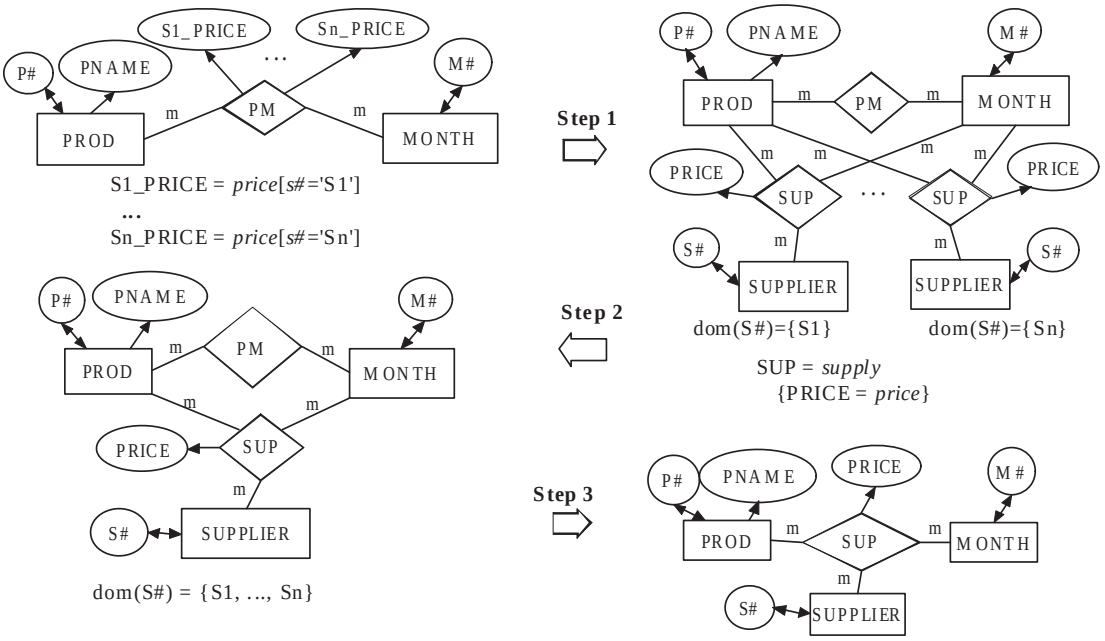


Figure 7.6: Resolve schematic discrepancies for attributes of relationship types

discrepant meta-attribute s# as an attribute S# of a new entity type SUPPLIER = supplier. In the ontology, price is an attribute of the ternary relationship type supply. Then in the ER schema, we construct the relationship type SUP = supply to contain the attribute PRICE = price.

Similarly, we transform the other attributes S2_PRICE, . . . , Sn_PRICE.

Then in Step 2, equivalent entity types, relationship types and attributes are merged respectively. Their domains are united.

Finally in Step 3, as in the ontology, the binary relationship type pm is a projection of the terary relationship type supply, in the ER schema, the relationship type PM is redundant and therefore removed. Note that this step is not included in Algorithm ResolveRelAttr. Instead, it will be performed later in a main integration algorithm calling the resolution algorithms (Section 8.5). □

The resolution of schematic discrepancy in Figure 7.6 is actually a "fold" op-

eration (see Section 3.1) on the attributes $S1\_PRICE, \ldots, Sn\_PRICE$ of the relationship type PM in $DB5$, such that the metadata (supplier numbers) of these attributes are transformed to the attribute values of the entity type $SUPPLIER$ in the relationship type SUP, and the values of these attributes become the values of the attribute $PRICE$ of the relationship type SUP in the transformed schema.

Furthermore, the propagation of cardinality constraints in the schema transformation of Figure 7.6 extends the propagation of qualified functional dependencies in application of a fold operation in the relational model (Section 6.2). For Figure 7.6, given a functional dependency

$$\{P\#, M\#\} \rightarrow \{S1\_PRICE, \ldots, Sn\_PRICE\}$$

represented as the cardinality constraints of the attributes of the original relationship type PM, we can derive a functional dependency

$$\{P\#, S\#, M\#\} \rightarrow PRICE$$

represented as a cardinality constraint on the relationship type SUP in the transformed schema.

The general algorithm ResolveRelAttr is presented in Appendix A.9. Note as the resolution of the schematic discrepancies for the attributes of relationship types always follows the resolution for relationship types, the attributes input to Algorithm ResolveRelAttr have no inherited context (see Step 1.2 of Algorithm ResolveRel for the transformation of attributes in the resolution of schematic discrepancies of relationship types).

7.3 Semantics preserving transformation

In this section, we will show that Algorithm ResolveEnt in Section 7.2.1, i.e., the resolution of the schematic discrepancies of entity types, preserves information and

cardinality constraints. The same property holds for the other three algorithms, which is omitted as the proofs are similar to that of Algorithm ResolveEnt.

7.3.1 Semantics preservation of Algorithm ResolveEnt

We first give a definition of information preservation in the transformation of ER schemas:

Definition 7.3 (Lossless transformation of ER schemas). *Given a schema transformation T on an ER schema E. Let I be any set of instances of the schema E, if there exist an inverse transformation T' of T, such that $T'(T(I)) = I$, then we say T is* information preserving. □

The resolution algorithm of schematic discrepancy for entity types, i.e., ResolveEnt is information preserving:

Theorem 7.1. *Algorithm ResolveEnt preserves the information of entity types, relationship types, attributes of entity types and attributes of relationship types.*

Proof. The transformation of Step 1 of Algorithm ResolveEnt is a one-to-one mapping from the instance set of each original schema construct (i.e., entity type, relationship type or attribute) onto the instance set of a transformed schema construct. This can be concluded from the necessary and sufficient conditions of the data transformation statements in the algorithm (i.e., the "iff" statements in Step 1 of Algorithm ResolveEnt).

Step 2 of merging equivalent schema constructs (implemented with the union or outer-join operations in data integration) also preserves the information. That is, there is a one-to-one mapping from the instance set of an original schema construct onto a subset of the instance set of the transformed schema construct. □

The same result of Theorem 7.1 holds for the other three algorithms, i.e., Algorithm ResolveRel, the resolution of the schematic discrepancies of relationship types, Algorithm ResolveEntAttr, the resolution of the schematic discrepancies of the attributes of entity types, and Algorithm ResolveRelAttr, the resolution of the schematic discrepancies of the attributes of relationship types.

Then we study the preservation of functional dependencies and multivalued dependencies in the schema transformation of Algorithm ResolveEnt.

Theorem 7.2. *The schema transformation of Algorithm ResolveEnt preserves the constraints of functional dependencies and multivalued dependencies.*

Proof. The proof is given in Appendix A.10. □

The same result of this theorem also holds for the other three resolution algorithms.

7.4 Schematic discrepancies in different models

7.4.1 Representing and resolving schematic discrepancies: from the relational model to ER

In comparison to the relational model, ER is a semantic rich model that provides more schema constructs to model an enterprise, and supports the representation of cardinality constraints. This makes ER schemas expressive, easy of representation and easy of understanding. On the other hand, this causes a diversity of schematic discrepancies on ER schemas. In particular, we resolved schematic discrepancies for entity types, relationship types, attributes of entity types and attributes of relationship types respectively. Furthermore, schematic discrepancy on ER schemas

is defined through context and more general (in terms of the number of meta-attributes of a schema construct) than the issue studied in the relational databases.

To resolve the schematic discrepancies for entity types and relationship types, we extended the unite operator in the ER model. That is, we "unite" a set of entity types (or relationship types) to a relationship type (or a new relationship type, resp.), such that the meta-attributes of the original entity types (or relationship types, resp.) are represented as the attributes of some entity types in the united relationship type, as shown in Figure 7.2 (or Figure 7.4, resp.). To resolve the schematic discrepancies for the attributes of entity types and attributes of relationship types, we extended the fold operator in the ER model. That is, we transform the meta-attributes of the original attributes to the attributes of entity types and the values of the original attributes to the values of a new attribute of a relationship type, as shown in Figure 7.5 or 7.6.

Qualified functional dependencies (and qualified multivalued dependencies) are represented as cardinality constraints with contexts in ER. Again, as a construct of an ER schema may have a set of meta-attributes, the constraints represented in the ER schema would be more general than the qualified functional dependencies studied in Chapter 6. The theory of the derivation of qualified functional dependencies was extended to propagate cardinality constraints in the transformation of ER schemas, as explained with Examples 7.2 to 7.6 in Section 7.2. For ER schemas, the propagation of cardinality constraints is involved in schema transformation rather than a separate process.

7.4.2 Extending the resolution in the integration of XML schemas

The resolution of schematic discrepancies in the integration of ER schemas can be extended to XML schemas in the ORASS model. That is, when schematic discrepancies occur, we just transform the discrepant meta-attributes into the attributes of new created object classes, and relate these object classes to original object classes with edges. The new challenges come from the hierarchical structure of XML.

For example, in Figure 7.7, suppose we want to integrate two ORASS schemas $S1$ and $S2$ modelling the same supply information, but with different structures. In Schema $S1$, it is represented as a path PROD/MONTH/SUPPLIER, i.e., for each product, listing the suppliers in each month. Note that the attributes of the object classes and relationship types in the schema are omitted for convenience. In Schema $S2$, it is represented as a set of paths JAN_PROD/SUPPLIER, ..., DEC_PROD/SUPPLIER. That is, each path represents the suppliers of products in one month. So the attribute values of months in Schema $S1$ correspond to the meta information of Schema $S2$. That is, the two schemas are schematically discrepant from each other.

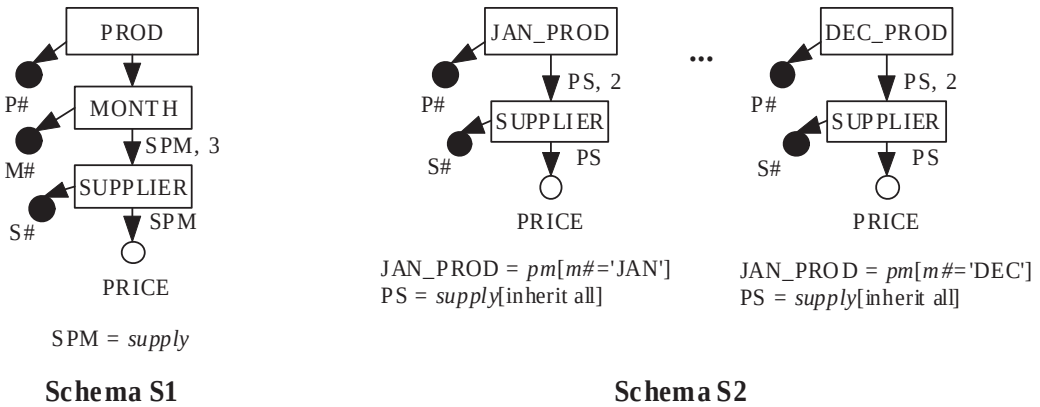


Figure 7.7: Two representations of the supply information in ORASS

To resolve the schematic discrepancy in the integration of $S1$ and $S2$, we may transform the meta information of months in $S2$ into the attribute values of a new created object class MONTH, and relate MONTH to the object classes of PROD (i.e., the object class transformed from JAN_PROD, ..., DEC_PROD after removing the meta information) and SUPPLIER. The problem is how to order the three object classes in a hierarchical path, e.g., MONTH/PROD/SUPPLIER, PROD/SUPPLIER/MONTH, etc. A reasonable choice would be PROD/MONTH/SUPPLIER, i.e., the same path as $S1$, as shown in Figure 7.8. Then in the integrated schema of $S1$ and $S2$, this path also becomes the integrated path of the source ones. The benefit of this choice is that when we integrate the data of the source schemas to populate the integrated schema, we do not need to reorder the objects of the path in $S1$.

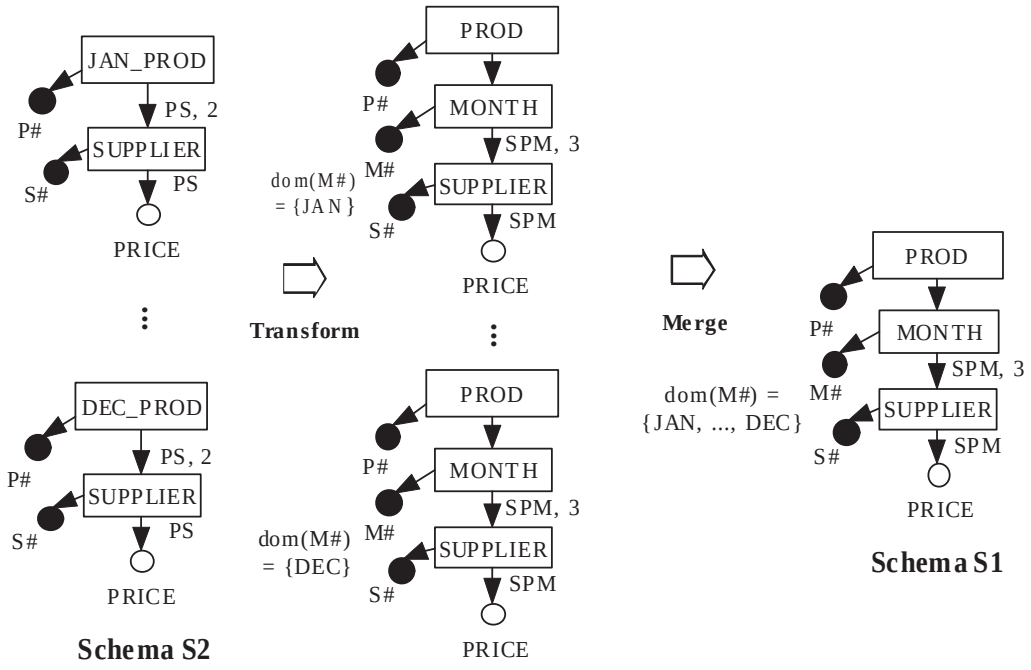


Figure 7.8: Transforming Schema S2 to S1

In a word, the resolution of schematic discrepancies in the integration of XML

schemas can be divided into two steps: first transform the discrepant meta attributes into attributes of object classes, and then integrate the transformed schemas (free of schematic discrepancy) into one with a "good" hierarchical structure. Although the first step can be adapted from the resolution for ER schemas, the second step is not trivial, and will be studied in detail in the next chapter.

7.5 Summary

We proposed a framework to represent the meta information (context) of a construct of an ER schema as a set of meta-attributes with metadata. Schematic discrepancy occurs when metadata in one schema correspond to attribute values in the other schema. Note that schematic discrepancy is now defined through context, which is more general than the issue studied in relational database. Qualified functional dependencies are represented by cardinality constraints with contexts in ER. Correspondingly, the resolution of schematic discrepancy in the ER model is an extension of the unite and fold operators in the relational model, and the propagation of cardinality constraints in the transformation of ER schemas is an extension of the propagation of qualified functional dependencies in the transformation of relational schemas. We resolved the schematic discrepancies of ER schemas by removing the context of schema constructs, i.e., transforming the meta-attributes that cause schematic discrepancies into attributes of entity types. The propagation of cardinality constraints is involved in the schema transformation. The resolution algorithms preserve the information and cardinality constraints in schema transformation. This work complements the previous work resolving other semantic heterogeneities in the integration of ER schemas. We have extended the the resolution techniques to XML schemas, while the hierarchical structures of XML remain

a challenge to be solved in the next chapter.

Chapter 8

Resolving hierarchical inconsistencies in the integration of XML schemas

An important consideration in schema integration is to preserve the information of source schemas. However, because of the inadequacy of DTD in XML schema integration (discussed in Section 2.2), the work based on DTD may loss information and generate an integrated schema that represents different meaning from the source schemas. Furthermore, existing work treated object classes as the semantic units, and might break relationship types among object classes in schema integration. We adopt ORASS to represent XML schemas, and treat relationship types as the semantic units in schema integration. The inconsistencies of the hierarchical structures of source schemas are resolved in merging relationship types. We first present some criteria of XML schema integration in applications, and then a method to integrate XML schema using ORASS.

8.1 Use cases and criteria of XML schema integration

XML schema integration plays an important role in building an integration system for either transaction or analytical processing purpose, but with different requirements. A system for transaction processing (usually a *mediated system*, e.g., [14, 15, 17, 18, 49]) has a virtual integrated view (i.e., a mediated schema) which provides a unified access to the heterogeneous data in sources. In this case, to integrate source schemas into a mediated schema, we have two criteria:

Criterion 1. *An integrated schema should preserve the information of source schemas, i.e., there is a one-to-one mapping from the instance set of each source schema onto a subset of the instance set of the integrated schema.*

Criterion 2. *An integrated schema should be concise, i.e., minimizing the number of redundant elements.*

These two criteria also apply to the schema integration in building a repository of XML schemas [60]. The purpose of such a system is to provide a centralized management of XML schemas to track the usage of schemas, and avoid proliferating redundant schemas.

On the other hand, a system for analytical processing (i.e., a *decision support system*) requires consolidating source data into a single physical store to provide fast, highly available and integrated access to related information. Data transformation/integration is a real process instead of performed on the fly. Source data are cleaned, transformed into a unified form, and then merged; redundancy is removed. Traditionally, in a decision support system, people need to integrate data with a magnitude of GB to TB. As the quantity of information available on the

Internet is rocketing, data transformation/integration itself becomes a bottle-neck in the system. In this case, schema integration is a preliminary step to guide the following data transformation. Besides the two criteria mentioned above, we have the 3rd one for schema integration in a decision support system:

Criterion 3. *Minimize the cost of data transformation.*

Another interesting application of XML data integration is the cache of the search engines on the Web. Currently, as the Web is HTML-based, search engines are based on information retrieval technology. When data sources are well defined and structured web sites (e.g., financial web sites, electronic libraries, and business sites using XML-based document exchange), more complex queries with some structural conditions can be supported. For example, given the title of a book, search the lowest price of the book from bookstore web sites. To speed up such queries, people may use robots crawling to cache book information. Unlike current caches which are simply repositories of HTML documents or URLs (with many redundancies and inconsistencies), the cache for an XML-based search engine should have integration capability to support complex queries. Because of the huge quantity of the information on the Web, the criteria of XML schema integration in this application are similar to those for analytical processing, i.e., Criterion 1, 2 and 3.

Another similar application is the XML Web cache in a Web-based information delivery system which provides Internet access to the information in large legacy infrastructures [21].

8.2 XML schema integration: using ORASS

As mentioned in Chapter 2, schema integration is usually performed on semantic rich models. We adopt ORASS as the canonical model in XML schema integration. In general, local (source) schemas could be DTDs, XML Schemas or even relational schemas. We first translate them into ORASS schemas, in which semantic enrichment is needed. The translation should be done semi-automatically. We have developed a tool to help domain experts to identify relationship types among object classes (the default relationship types are the binary ones between parent and child classes) and the attributes of relationship types. Then we specify the correspondences between the equivalent/similar constructs (i.e., object classes, relationship types and attributes) of the ORASS schemas. Finally, we transform and integrate the ORASS schemas, in which semantic heterogeneities are resolved. Once the integrated schema and the mappings from the component schemas to the integrated schema are created, we can do data integration or query processing through the integrated schema.

In general, given a set of XML schemas in ORASS, we integrate them in the following 7 steps. We say *two relationship types are at the same level* if their first nodes are either the same node in a schema tree, or two root nodes in two schema trees.

Step 1 Resolve schematic discrepancies (i.e., attribute names or class names in one schema correspond to attribute values in another) by transforming the attribute names or class names into attribute values.

Step 2 Resolve naming conflicts (i.e., homonyms and synonyms of object classes, relationship types and attributes) by renaming.

Step 3 Resolve structural conflicts (i.e., attributes in one schema correspond to

object classes in another schema) by transforming the attributes into object classes [74].

Step 4 Resolve the key conflicts of object classes from different schemas by selecting the common identifiers for equivalent object classes.

Step 5 Merge the schemas in a top-down way using an algorithm of merging relationship types (i.e., Algorithm MergeRel which will be introduced in Section 8.4). That is, for each source schema tree, we perform a depth first search on the object classes. The relationship types at the same level of the schema trees are merged with Algorithm MergeRel.

Step 6 Build class hierarchies and references among object classes.

Step 7 Remove the redundant relationship types that can be derived from others.

□

In this work, we studied the key techniques to merge relationship types (i.e., Step 5) in the integration of ORASS schemas. We first give an efficient method to reorder the objects in relationships in Section 8.3, and then an algorithm MergeRel to merge relationship types in Section 8.4. An example of XML schema integration will be given in Section 8.5.

8.3 Reordering the objects in relationships

In ORASS schemas, a relationship type is represented as a path, i.e., a hierarchy of object classes. Two object classes are *equivalent* to each other iff they model the same real world concept. Two relationship types are *equivalent* to each other iff they model the same association involving the same set of real world concepts. Note the sets of the instances of a pair of equivalent object classes (or equivalent

relationship types) may not be the same; instead, they can be related in one of 4 ways: *equal*, *subset*, *overlap* and *disjoint* [71]. The attribute sets of a pair of equivalent object classes (or equivalent relationship types) may not be the same also because of the heterogeneity and semi-structuredness of source data.

Two equivalent relationship types may have different hierarchical structures, i.e., different hierarchies of the participating object classes. To integrate equivalent relationship types, we need to transform them to consistent hierarchical structures. In this section, we present a method to reorder the objects in relationships, i.e., change the hierarchical orders of the objects in the relationships. Our method uses relational databases as temporary storage space, and is more efficient than the XQuery implementation.

To simplify the presentation, we use identifier values to represent the objects in relationships, and defer the allocation of attributes (of objects and relationships) until we have reordered the objects in the relationships. That is, the attributes of an object will be kept with the object, and the attributes of the relationship will be moved to the object at the lowest level in the transformed relationship.

We assume that in source XML data, the objects of a class have been sorted by the identifier values. Otherwise, adopting the different hierarchies of the object classes in an integrated relationship type makes little difference to the cost of data transformation using our method.

8.3.1 Reordering objects using relational databases

In this subsection, we introduce two methods to reorder objects in relationships: one uses relational databases to store and sort relationships, and the other one uses XQuery. We find the former method would be more efficient than the latter one in most cases.

Source XML data may be stored in different models, e.g., Element-Based Clustering model (or EBC) in which element nodes with the same tag name are clustered and organized as a list [53], Object Exchange Model (or OEM) [50], relational databases (or object-relational databases) [21], or native XML documents. No matter how XML data is stored in sources, in order to efficiently reorder the objects in relationships, we first get (using the wrappers on sources) and store the relationships in a flat table with the fields corresponding to the identifiers of the object classes involved in the relationship type, then sort the table by the identifier values of the object classes in a required order. Finally, we compute the relationships with the needed hierarchies of the objects by merging the objects with the same identifier values in the table. We hereby present an example to explain the process.

Example 8.1. *Suppose in Figure 2.2, we want to transform the relationship type S/P/M into P/S/M (S, P, M are the shorthands for SUPPLIER, PROD and MONTH), i.e., swapping SUPPLIER and PROD in Figure 2.2. Figure 8.1 (b) represents three relationships of the original relationship type, in which "s_1" represents an object of the class SUPPLIER with S# = s_1, and so on. The attribute PRICE of the relationship type will be handled after the reordering of objects, and is omitted here for convenience.*

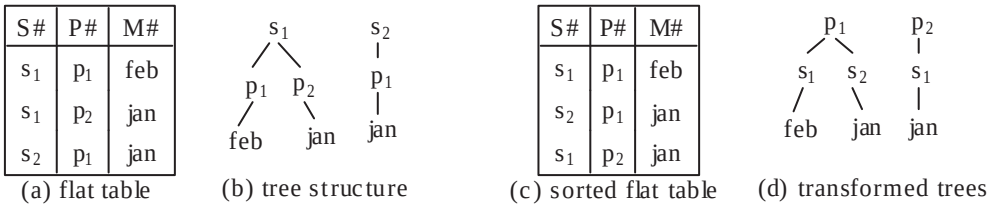


Figure 8.1: Reorder S/P/M into P/S/M: first sort the table by P#, S#, M#, then merge the objets with the same identifier values in the table

We first scan and store the original relationships in a flat table of Figure 8.1

(a). To transform S/P/M into P/S/M, we just sort the flat table by P#, S#, M#. This can be implemented by a sorting algorithm with a comparison function that, given two tuples, compares the P# values, and if there is a tie, compares the S# values, and if another tie occurs, compares the M# values. The result is Figure 8.1 (c). To construct the tree structure of these relationships, we scan the table, and merge the objects with same values of P#, S# and M# in the table in order, and get Figure 8.1 (d). □

Note that ORASS provides the following semantics that is necessary in reordering object classes in a relationship type:

1. relationship types. The object classes being reordered should be in a relationship type.

2. identifiers of object classes. We need to know the identifers of object classes to merge the equivalent objects in the transformed relationships.

3. attributes of relationship types. Attributes of a relationship type should always be attached to the lowest object class in the relationship type. For Figure 2.2, to reorder the object classes $PROD$ and $MONTH$ in the relationship type SPM, we should move the attribute $PRICE$ from $MONTH$ to $PROD$, as it is an attribute of the relationship type SPM.

Then we present another approach to reordering the objects in relationships, i.e., using XQuery. Suppose we do the same transformation as Example 8.1 with XQuery statements. The original XML data is given in Figure 2.4. The XQuery statements to implement the reordering is presented in Figure 8.2 (we assume that in the source data, the root elements or the elements under the same parent will not repeat). In Figure 8.2, we first get al.l the distinct values of P#; then for each product, we get al.l the suppliers supplying that product; finally, for each

product and each supplier supplying that product, we get al.l the months in which the product is supplied. Although not explicitly mention, users should know the necessary semantics (i.e., the relationship type SPM, the identifiers of the object classes and the attribute of the relationship type) in the original XML data in order to write such a query of restructuring.

```
for $P# in distinct-values(/SUPPLIER/PROD/@P#)
return <PROD P# = ‘‘{$P#}’’>
       {
       for $S in /SUPPLIER[PROD/@P# = $P#]
       return <SUPPLIER S# = ‘‘{$S/@S#}’’
              {
              for $M in $S/PROD[@P# = $P#]/MONTH
              return $M
              }
              </SUPPLIER>
       }</PROD>
```

Figure 8.2: XQuery statements to swap the elements SUPPLIER and PROD in the XML document section of Figure 2.4

Suppose the source data is stored in an XML document and SAX is used to parse the document. The evaluation of the XQuery statements would be costly. Even for the simple instance of Figure 2.4, six parses of the data are needed. The method using relational databases is more efficient which comprises two parses of the data (to read the relationships into a flat table and to output trees from the flat table), and the sorting of the tuples in the flat table.

8.3.2 Cost model

We consider the cost of data transformation in the integration of a set of equivalent relationship types with different hierarchical structures probably. From the last subsection, we know that the reordering of the objects in relationships needs to sort the relationships in a flat table by the identifier values of the object classes in a required order. Besides the cost of sorting, the integration of equivalent relationship

types involves some other costs, e.g., reading source relationships to flat tables, and combining the relationships of several sorted tables to get the integrated relationship type. The costs of reading and combination are constant given the numbers of the relationships in the sources. But the sorting cost may be variable if we adopt the different hierarchies of the object classes in the integrated relationship type. So as to Criterion 3, i.e., minimizing the cost of data transformation (see Section 8.1), we only need compare the sorting costs in deciding the hierarchical structure of an integrated relationship type.

To sort the relationships in a flat table, we adopt the classical method of "external merge sort" (the sort key consists of all the identifiers of the object classes involved in the relationship type, i.e., all the fields in the flat table). Given a table with n tuples (relationships) or N pages on a disk, let $B > 3$ be the number of buffer pages, the I/O cost of sorting is

$$2N \times \lceil 1 + \log_{B-1} \lceil N/B \rceil \rceil \qquad (1)$$

However, the cost may be even reduced if the original relationships are already sorted by the identifier values of some preceding object classes, and these object classes are not involved in the reordering.

In general, given a relationship type $O_1/O_2/\ldots/O_m$ with n relationships (or N pages for the table storing the relationships) sorted by the identifier values of $O_1, O_2, \ldots, O_m$, let k_j, $j = 1, \ldots, m$, be the average number of the objects of O_j under the same parent object (or the average number of the root objects of O_j) in the relationship type. Now we reorder $O_1/O_2/\ldots/O_m$ into $O'_1/O'_2/\ldots/O'_m$ such that $O_1 = O'_1, \ldots, O_i = O'_i$ but $O_{i+1} \neq O'_{i+1}$ for some $0 < i < m - 1$.

To implement the reordering, we should sort the relationships in the flat table. That is, we divide the relationships into $k_1 \times \ldots \times k_i$ groups such that the relation-

ships in each group have the same identifier values of $O_1, \ldots, O_i$. Then we only need to sort the relationships in each group as the relationships in the flat table are already sorted by the identifier values of $O_1, \ldots, O_i$ which remain unchanged in the transformed relationship type. Then the estimated I/O cost of sorting the relationships by the identifier values of $O'_1, \ldots, O'_m$ is

$$(k_1 \times \ldots \times k_i) \times 2 \times N/(k_1 \times \ldots \times k_i) \times \lceil 1 + \log_{B-1} \lceil N/(k_1 \times \ldots \times k_i)/B \rceil \rceil$$

$$= 2N \times \lceil 1 + \log_{B-1} \lceil N/(B \times k_1 \times \ldots \times k_i) \rceil \rceil \qquad (2)$$

Note when $i = 0$, i.e., the reordering changes the root node of a relationship type, then formula (2) is degenerated into (1). On the other hand, when $i = m$, i.e., no reordering occurs, then formula (2) has a value of 0. In the rest of the paper, we will use Formula (2) to estimate the cost of reordering the objects in relationships.

Given a set of equivalent relationship types $\mathbb{R}$ each element of which involves m object classes, our purpose is to integrate them into one relationship type R minimizing the total cost of transforming the relationships of $\mathbb{R}$ to the relationships of R. To decide the hierarchical structure of the integrated relationship type, should we compute the costs for all the $m!$ permutations of the m object classes? The answer is negative in most cases. We only need to select the hierarchical structure of the integrated relationship type from all the distinct hierarchies of the object classes in the relationship types of $\mathbb{R}$, as shown in the following proposition.

Proposition 8.1. *Given a set of equivalent relationship types $\mathbb{R}$, let R be the integrated relationship type of $\mathbb{R}$ minimizing the total cost of transforming the relationships of the types in $\mathbb{R}$ into the relationships of R, then the hierarchical structure of R must be the same as some relationship types in $\mathbb{R}$.*

Proof. This can be proven by induction.

Base case: the first object class of R is the same as the first object class of some relationship types of $\mathbb{R}$. Otherwise, to transform the relationships of each type R' of $\mathbb{R}$ to the relationships of R, we need to reorder all the objects in R', as R and R' have different root object classes. This takes more time than the case when R has the same root as some relationship types of $\mathbb{R}$.

Inductive hypothesis: the first i object classes of R are the same as the first i object classes of all the relationship types of $\mathbb{R}1$ for a non-empty set $\mathbb{R}1 \subseteq \mathbb{R}$.

We claim that the first $i+1$ object classes of R are the same as the first $i+1$ object classes of some relationship types of $\mathbb{R}1$. Otherwise, we can construct a relationship type R' that has the same first i object classes as R and the same $(i+1)$-th object class as all the relationship types of $\mathbb{R}2$ for a non-empty set $\mathbb{R}2 \subseteq \mathbb{R}1$. It takes the same cost to transform the relationships of the types in $\mathbb{R} - \mathbb{R}2$ to the corresponding ones of R' as the cost to transform them to the relationships of R, but takes less cost to transform the relationships of the types of $\mathbb{R}2$ to the corresponding ones of R' than the cost to transform them to the relationships of R. That is, R does not minimize the total transformation cost. This contradicts with our assumption. We conclude that the first $i+1$ object classes of R are the same as the first $i+1$ object classes of some relationship types of $\mathbb{R}$.

Consequently, when i is the number of all the object classes involved in the relationship types of $\mathbb{R}$, R is the same as some elements of $\mathbb{R}$. □

Before ending this section, we give an example of deciding the hierarchy of the object classes in an integrated relationship type.

Example 8.2. *Given two equivalent relationship types $R1$: S/P/M and $R2$: S/M/P, suppose for $R1$, the page number of the table storing the relationships is 1000, and the average numbers of the objects of S, P and M under the same parent in $R1$ are, respectively, 50, 40 and 12, and for $R2$, the page number of the table is 3200, and*

the average numbers of the objects of S, P and M under the same parent are 10, 60 and 12. Let the number of buffer pages $B = 5$. Applying Formula (2), we can compute the total reordering cost if we adopt $R1$ or $R2$ as the integrated relationship type:

1. *$R1$ as the integrated relationship type. We only need to transform all the relationships of $R2$ to the relationships of $R1$ (note that the root class S of $R2$ is the same as $R1$):*

   $0 + 2 \times 3200 \times \lceil 1 + \log_{5-1} \lceil 3200/(5 \times 10) \rceil \rceil = 25600$;

2. *$R2$ as the integrated relationship type. We only need to transform all the relationships of $R1$ to the relationships of $R2$:*

   $2 \times 1000 \times \lceil 1 + \log_{5-1} \lceil 1000/(5 \times 50) \rceil \rceil + 0 = 4000$.

Consequently, adopting $R2$ as the integrated relationship type dramatically reduces the data transformation cost. Intuitively, the benefit comes from two factors: (1) the cardinality of $R1$ is less than $R2$, and therefore transforming the relationships of $R1$ costs less; (2) the average number of the objects of S in $R1$ is bigger than that in $R2$, and therefore the sorting of the tuples in the flat table of $R1$ costs less than $R2$. □

Proposition 8.1 restricts the search space in computing an integrated schema of a set of equivalent relationship types. More generally, several relationship types from source schemas may contain some equivalent object classes constituting equivalent sub relationship types which should be merged. We will study this general issue in the next section.

8.4 Merging relationship types

The merging of relationship types should meet the three criteria raised in Section 8.1. We assume that the schema matching information of equivalent object classes and equivalent (sub) relationship types are already known by matching techniques [63] and human input.

8.4.1 Definitions

We first give an example to provide some intuition of the issue, and introduce some concepts.

Example 8.3. *In Figure 8.3 (a), there are three relationship types R1, R2 and R3 among object classes A (SUPPLIER), B (PROD), C (MONTH) and D (PROJECT), where the sub relationship type among A, B and D in R1 is equivalent to R2, and the sub relationship type among A, B and C in R1 is equivalent to R3.*

We have four ways to merge the 3 relationship types, and get four candidate integrated schemas (I) to (IV) in Figure 8.3 (b). E.g., Schema (I) is obtained by first swapping C and D in R1, then merging it with R2 on A, B and D, then swapping B and A in R3, and merging them together on A and B. Note that the two C (one from R1 and the other one from R3) of Schema (I) cannot be merged; otherwise some relationships of R3 which are not in R1 would be lost.

Note that each of the four candidate integrated schemas preserves the information of the source relationship types R1, R2 and R3. □

Then the problem is, from those candidate integrated schemas, select one according to the three criteria in Section 8.1. The method will be introduced in the next sub-section. We first define some concepts below.

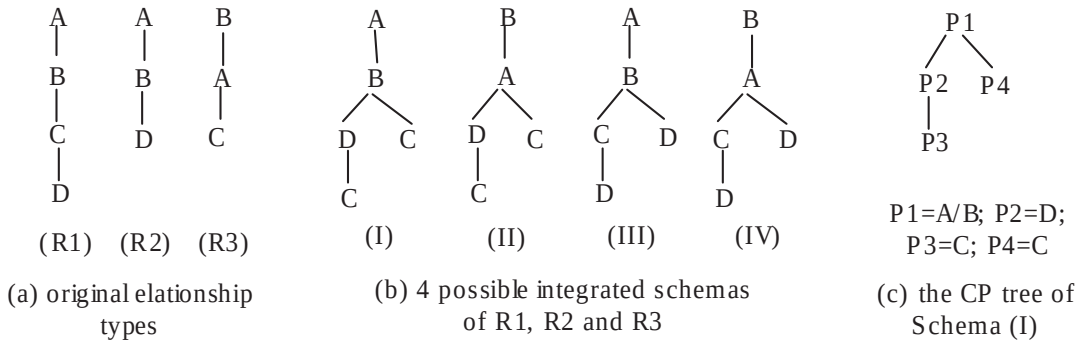


Figure 8.3: different ways to merge relationship types

Definition 8.1 (Cognate path). *In an integrated schema S of a set of relationship types, we call a path of S a* cognate path *if all the object classes within the path are from the same set of source relationship types. A cognate path is* non-trivial *if it contains more than one object class.* □

For example, in the integrated schema (I) of Figure 8.3 (b), A/B is a cognate path as A and B are from the same set of source relationship types {R1, R2, R3} (no matter what hierarchical orders of A and B in the source relationship types). However, A/B/D is not a cognate path, as A and B are from {R1, R2, R3}, but D is from {R1, R2}.

Lemma 8.1. *In an integrated schema of a set of relationship types, the reordering of some object classes in a cognate path preserves the information of all the relationship types.*

Proof. Given an integrated schema S of a set of relationship types $\mathbb{R}$, let $O_1/O_2/\ldots/O_n$ be a cognate path of S. Then the object classes $O_1, O_2, \ldots, O_{n-1}$ on the path have no branches. Otherwise, some classes of $O_1, O_2, \ldots, O_n$ would come from different relationship types of $\mathbb{R}$. Let $\mathbb{R}1 \subseteq \mathbb{R}$ be the set of the relationship types from which the cognate path $O_1/O_2/\ldots/O_n$ comes. Then the reordering of some object classes in the cognate path only affects the relationship types of $\mathbb{R}1$. For any rela-

tionship r of a relationship type of $\mathbb{R}1$, it becomes r' in S such that r and r' are equivalent to each other, but have different hierarchical structures on the objects of $O_1, O_2, \ldots, O_n$. That is, there is a one-to-one mapping from the instance set of each relationship type of $\mathbb{R}1$ onto a subset of the relationships of S. We conclude that the reordering of some object classes in the path $O_1/O_2/\ldots/O_n$ of S preserves the information of all the relationship types of $\mathbb{R}$. □

On the other hand, the reordering of the object classes from two cognate paths would break relationship types, and therefore is not allowed. E.g., in Schema (I) of Figure 8.3 (b), the swap of B and D will break the relationship type A/B/C (i.e., R3).

Definition 8.2 (Variation of cognate path). *Given a cognate path P, we call a path P' a* variation *of P iff P' contains the same set of object classes as P, but probably has a different hierarchical structure.* □

Definition 8.3 (Beneficial cognate path). *Given an integrated schema S of a set of relationship types, we say a cognate path P of S is* beneficial *if the path from the root of S to some object class of P is the same as the paths from the roots of some source relationship types.* □

For example, in Schema (I) of Figure 8.3 (b), the cognate path A/B is beneficial, as it is the same as the paths from the roots of R1 and R2 in Figure 8.3 (a); the trivial cognate path D is beneficial, as A/B/D is the same as R2. Consequently, adopting Schema (I) as the integrated schema can save the data transformation cost of transforming the relationships of R1 and R2 (actually no cost for R2), but not R3 whose root is different from the root of Schema (I) (see Section 8.3.2 for the computation of data transformation costs).

Definition 8.4 (Cognate path tree). *Given an integrated schema tree S of a set of relationship types, we obtain the* cognate path tree *(or* CP tree*) of S by using a node to represent each cognate path in S and preserving the edges between the cognate paths in S.* □

For example, in Figure 8.3, the CP tree of Schema (I) is Figure 8.3 (c). Note in this CP tree, P3 is from R1 and P4 from R3.

A CP tree provides an overview of an integrated schema, treating cognate paths as units to be handled in our algorithm. As there is a one-to-one mapping from an integrated schema tree to a CP tree, we may use the two concepts (i.e., an integrated schema tree and its CP tree) interchangeably in the rest of the paper.

Definition 8.5 (Merge-able schema trees). *Two integrated schema trees $S1$ and $S2$ of some relationship types are* (k-)merge-able *iff they have the same k object classes $O_1, O_2, \ldots, O_k$ in the cognate paths from their root nodes. $O_1, O_2, \ldots, O_k$ are called the* merge-able object classes *between $S1$ and $S2$.* □

For example, in Figure 8.3 (a), $R1$ and $R2$ are merge-able as they have the same object classes A, B and D. They can be merged to a schema (path) $S = A/B/D/C$ consisting of two cognate paths $P1 = A/B/D$ and $P2 = C$. Then S and $R3$ are merge-able as they have the same object classes A and B in $P1$ of S and in $R3$. Consequently, we can merge them to Schema (I) of Figure 8.3 (b).

Definition 8.6 (Data transformation cost of integrated schema). *Given an integrated schema S of a set of relationship types $\mathbb{R}$, the* data transformation cost of S *is the total cost of transforming the relationships of the types in $\mathbb{R}$ into the relationships of S.* □

The cost of transforming the relationships of a type can be computed with Formula (2) of Section 8.3.2. Given an integrated schema S of a set of relationship

types, the reordering of the object classes in some individual cognate paths of S will not break the relationship types (Lemma 8.1), but may change the data transformation cost of S.

Definition 8.7 (Variation of integrated schema). *Given an integrated schema S of a set of relationship types, we call a schema obtained by reordering some object classes in some individual cognate paths of S a* variation *of S (S is also a variation of itself).* □

From Lemma 8.1, we know that the variations of an integrated schema represent the same set of relationship types, but probably have different hierarchical structures.

Definition 8.8 (Minimum cost tree). *In an integrated schema tree S of a set of relationship types, let $\mathbb{S}$ be the set of all the variations of S. We call $S' \in \mathbb{S}$ the* minimum cost tree *of S iff S' minimizes the data transformation costs of all the elements of $\mathbb{S}$.* □

For example, in Figure 8.3 (b), Schema (II) is a variation of (I). The minimum cost tree of Schema (I) is Schemas (I) or (II) which has less cost to transform the relationships of $R1$, $R2$ and $R3$ of Figure 8.3 (a) to the relationships of the integrated schema.

8.4.2 Algorithm

Recall the three criteria of schema integration (Section 8.1), i.e., information preserving, minimization of both redundancy and the cost of data transformation. Our algorithm of merging relationship types are based on the three criteria. Although Criterion 1 should be always satisfied, Criterion 2 and 3 may not be achieved at the same time (e.g., data transformation is unnecessary at all if we do not merge

any relationship types). Criterion 2 usually has a higher priority than Criterion 3. The reason is that a concise integrated schema (i.e., meeting Criterion 2) facilitates the pose of queries against the integrated schemas, and reduces the redundancy of integrated data. Note redundancy not only wastes storage space, but also causes update, insertion and deletion anomalies. [1]

Given a set of relationship types, we merge them by a 3-step algorithm MergeRel. Step 1 is for Criterion 2, the conciseness of integrated schemas. In this step, we try to merge the equivalent object classes of equivalent (sub) relationship types as many as possible. It produces several candidate integrated schemas whose minimum cost trees are computed in Step 2. The schema with the minimum cost of data transformation will be chosen as the final integrated schema. Finally in Step 3, we handle the attributes of object classes and attributes of relationship types for the integrated schema. We first explain the first 2 steps by the following example, and then give the general algorithm.

Example 8.4. *In this example, we integrate the three relationship types $R1$, $R2$ and $R3$ of Figure 8.3 (a) in two steps. In Step 1, we have two strategies to merge the relationship types (note in this step, we don't care the variations of cognate paths):*

Strategy 1. *First merge $R1$ and $R2$, then $R3$, and get Schema (I) of Figure 8.3 (b), or*

Strategy 2. *First merge $R1$ and $R3$, then $R2$, and get Schema (III) of Figure 8.3 (b).*

Then for each of the candidate integrated schemas obtained from Step 1, we compute the minimum cost tree for it in Step 2. For Strategy 1, given the CP tree

[1] Given more semantics, e.g., functional dependencies, we may generate more efficient integrated schemas.

of Figure 8.3 (c), we decide the hierarchical structures of the cognate paths in a top-down way. First for the cognate path P1, we have two variations, A/B and B/A. Adopting A/B benefits the transformations of the relationships of R1 and R2, while adopting B/A benefits the transformation of the relationships of R3. We study the two variations one by one.

For the variation A/B, as the following cognate paths P2, P3 and P4 are all trivial, no variations of them need to be considered. We compute the total cost (say c1) to transform the relationships of R1, R2 and R3 into the relationships of Schema (I), using Formula (2) of Section 8.3.2.

For the variation B/A of P1 (the integrated schema becomes (II) of Figure 8.3 (b)), we compute the cost (say c2) to transform the relationships of R1, R2 and R3 into the relationships of Schema (II). Then we compare c1 and c2, and choose the schema with the less cost, say Schema (II), as the integrated schema for Strategy 1.

Then in a similar way, for Strategy 2, we compare the data transformation costs of Schemas (III) and (IV), and get the integrated schema, say Schema (IV) with the less cost. Finally, we compare the data transformation costs of Schemas (II) and (IV), and choose the one with the less cost as the final integrated schema of R1, R2 and R3. □

The general algorithm MergeRel is presented below. In the algorithm, we adopt a greedy strategy for Step 1 of merging relationship types. The idea is similar to the classical "agglomerative hierarchical clustering", if we treat the merging problem as clustering, and the number of merge-able object classes as the similarity between two schema trees. That is, in each iteration of a loop, we merge the relationship types (or schema trees) with the maximum number of merge-able object classes, until no two trees are merge-able. Then in Step 2, we call an algorithm MCT (given

after MergeRel) to compute the minimum cost trees for the candidate integrated schemas produced in Step 1, and select the schema with the minimum data transformation cost as the final integrated schema. Finally in Step 3, we represent the attributes for the integrated schema. Note that the integrated schema of a set of relationship types may be a set of trees instead of one tree.

Algorithm MergeRel

Input: A set of relationship types $\mathbb{R}$.

Output: An integrated schema of the relationship types of $\mathbb{R}$.

Step 1. *In this step, we merge the relationship types of* $\mathbb{R}$ *in a loop.*

In each iteration of the loop, let $\mathbb{T}$ be the set of schema trees obtained from the last iteration (initially $\mathbb{T}$ is equal to $\mathbb{R}$).

while some trees in $\mathbb{T}$ are merge-able **do**

Let k be the maximum number of the merge-able object classes between any two trees of $\mathbb{T}$. Let M be a set of merge-able object classes with the cardinality k, and $\mathbb{T}_M$ be the maximum set of trees merge-able on M in $\mathbb{T}$.

for each tree of $\mathbb{T}_M$ **do**

Push the object classes of M to the top of the tree, and permute these object classes in a fixed hierarchy which is decided in random.

end for

Merge these trees on the object classes of M. The domain of each object class of M in the merged schema is the union of the domains of the corresponding object classes of the trees in $\mathbb{T}_M$.

Note that the set M may not be unique, and the different choices of M lead to different integrated schemas whose minimum cost trees will be computed in Step 2.

end while

At the end of Step 1, we get a set of candidate integrated schemas. Each schema (probably consisting of a set of trees) is an integration result of $\mathbb{R}$ corresponding to a sequence of the choices of M during the merging.

Step 2. *In this step, we compute the minimum cost trees for the integrated schemas.*

for each integrated schema S obtained in Step 1 **do**

for each CP tree T of a schema tree in S **do**

Let $r(T)$ be the root of T. Call Algorithm MCT$(r(T), c, T')$ (given below) to compute a minimum cost tree T' of T with the data transformation cost c.

end for

The data transformation cost of S is then the sum of the costs of the trees in S.

end for

The schema with the minimum data transformation cost is selected to be the final integrated schema of $\mathbb{R}$.

Step 3. *In this step, we represent the attributes of the integrated schema.*

for each object class O in the integrated schema, let $\mathbb{O}$ be the set of the corresponding object classes in the relationship types of $\mathbb{R}$, **do**

for each set of equivalent attributes $\mathbb{A}$ of some classes of $\mathbb{O}$ **do**

Represent an attribute A (equivalent to the attributes of $\mathbb{A}$) of O.

if there are some constraint conflicts (e.g., domain mismatch and cardinality conflicts) among the attributes of $\mathbb{A}$, **then**

Resolve these conflicts with some existing techniques, e.g., [38].

end if

The domain of A is the union of the domains of the attributes of $\mathbb{A}$.

end for

end for

for each relationship type R in the integrated schema, let $\mathbb{R}1$ be the set of the corresponding relationship types in $\mathbb{R}$, **do**

for each set of equivalent attributes $\mathbb{A}$ of some relationship types of $\mathbb{R}1$, **do**

Represent an attribute A (equivalent to the attributes of $\mathbb{A}$) of R under the object class at the lowest level of R.

if there are some constraint conflicts among the attributes of $\mathbb{A}$, **then**

Resolve these conflicts with some existing techniques, e.g., [38].

end if

The domain of A is the union of the domains of the attributes of $\mathbb{A}$.

end for

end for □

We then give Algorithm MCT below to compute the minimum cost tree of a schema tree. Algorithm MCT is developed based on an observation: given an integrated schema tree S of a set of relationship types, for P a cognate path in the schema tree, if P is beneficial, it saves the cost of transforming the relationships of some source relationship types to the relationships of S; otherwise, the hierarchical structure of P has nothing to do with the data transformation cost (a formal proof will be given later), and therefore can be decided in random.

The execution of Algorithm MCT can be divided into two stages: top-down and bottom-up stages. In the top-down stage, we recursively selects the hierarchical structures of the cognate paths in the integrated schema (Line 5). When reaching a leaf node (Line 10) or some node without beneficial variations (Line 19), we compute the data transformation cost for the corresponding path (Line 12 to 17) or sub-tree (Line 21 to 22), and return the minimum cost trees for those sub

Algorithm:MCT(P, c, T)

Input: An integrated schema tree S of a set of relationship types $\mathbb{R}$, the CP tree T_S of S, and P the root node of T_S.

Output: A minimum cost tree T of T_S with the data transformation cost c.

1: **if** P has some variations that are beneficial **then**
2: **if** P is a non-leaf node in T_S **then**
3: **for** each beneficial variation of P **do**
4: **for** each child node $P.child[i]$ of P, $i = 1, \ldots, n$ **do**
5: MCT($P.child[i], c_i, T_i$)
6: **end for**
7: **end for**
8: Let P_{opt} be a variation of P minimizing $c = c_1 + c_2 + \ldots + c_n$, and let T'_i, $i = 1, \ldots, n$, be the returned minimum cost trees rooted at $P_{opt}.child[i]$.
9: Construct the CP tree T that has the root P_{opt} and n sub-trees $T'_1, \ldots, T'_n$ under P_{opt}.
10: **else**
11: */* P is a leaf node.*/*
12: **for** each beneficial variation of P **do**
13: Let PATH be the path in S corresponding to the path from the root to P in T_S.
14: Compute the data transformation cost c of PATH by Formula (2) of Section 8.3.2.
15: **end for**
16: Let P_{opt} be a variation of P minimizing c.
17: Construct the CP tree T that has the only node P_{opt}.
18: **end if**
19: **else**
20: */* P has no beneficial variation. In this case, the hierarchical structures of P and the descendants of P are not important.*/*
21: Construct the CP tree T as the same (sub-)tree rooted at P in T_S.
22: Compute the data transformation cost c of the tree consisting of the path from the root to P and the (sub-)tree rooted at P in T_S.
23: **end if** □

structures. Then in the bottom-up stage, we compute the data transformation costs and construct the minimum cost trees rooted at the parent nodes given the costs and the minimum cost trees rooted at the child nodes (Line 8 to 9).

We refer readers to Example 8.4 for the intuition of Algorithm MergeRel and MCT. In the next subsection, we will evaluate the algorithms by the three criteria.

8.4.3 Evaluation of Algorithm MergeRel

We evaluate the algorithm MergeRel of merging relationship types by the three criteria given in Section 8.1. As to Criterion 1, we have:

Theorem 8.1. *Algorithm MergeRel preserves the information of object classes, relationship types, the attributes of object classes and the attributes of relationship types.*

Proof. Given a set of relationship types $\mathbb{R}$, let S be the integrated schema with Algorithm MergeRel.

For each object class O of S, let $\mathbb{O}$ be the set of the corresponding object classes in the relationship types of $\mathbb{R}$. The domain of O is the union of the domains of the classes in $\mathbb{O}$. That is, the merging preserves the objects of each class in $\mathbb{O}$.

As to the preservation of relationships, we consider Step 1 and 2 of Algorithm MergeRel. In Step 1, in each iteration of the loop, to merge two trees (or paths), we first reorder the object classes in the cognate paths from the roots of the two trees, which preserves relationships (according to Lemma 8.1). Then we merge the mergeable object classes of the two trees, which makes no difference to relationships. Then in Step 2, we call Algorithm MCT to compute the minimum cost trees of the candidate integrated schemas, in which we may reorder the object classes in some cognate paths to get beneficial paths. This also preserves relationships, according to Lemma 8.1.

We omit the proof of the preservation of the attributes, which requires a thorough investigation of the resolution of constraint conflicts among attributes. This is not the focus of this paper. We refer interested readers to [38]. □

As to Criterion 2, the minimization of redundancy, Algorithm MergeRel (Step 1) adopts a greedy strategy to merge object classes in equivalent (sub) relationship types. However, it cannot be ensured that the integrated schema minimizes the number of redundant object classes, although our experiment shows that it almost always produces concise integrated schemas. It may take much time to compute the "optimal" result according to Criterion 2. Actually, even adopting our greedy method, we find the search space may be exponential to the number of source relationship types for some extreme cases. This step becomes the bottle-neck of the merging algorithm, and requires further research on it. Fortunately, the efficiency of schema integration is not as critical as the efficiency of data transformation or query processing, as schema integration would be performed infrequently and off-line in many cases.

As to Criterion 3, we have the following result:

Theorem 8.2. *Algorithm MergeRel (Step 2) computes an integrated schema minimizing the data transformation costs in the set of the schemas produced by Step 1 of the algorithm.*

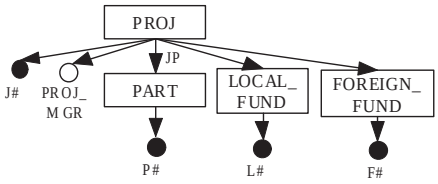
Proof. The proof is given in Appendix A.11. □

8.5 XML schema integration by example

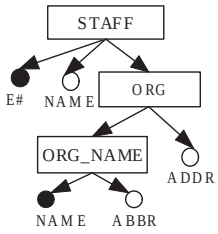
The methods of reordering object classes and merging relationship types introduced in the last two sections provide the basic operations for XML view management [12] and XML schema and data integration. In Section 8.2, we introduced 7 steps to

integrate XML schemas in ORASS, using our method to merge relationship types and some existing techniques to resolve semantic heterogeneities. We herby give an example to explain some of these steps.

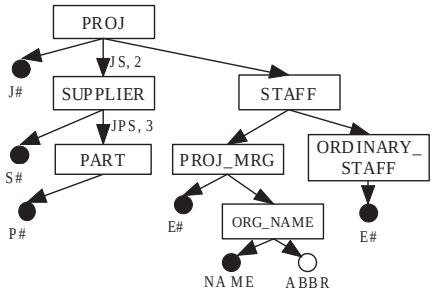
Example 8.5. *Given the 4 schemas of Figure 8.4 (the schemas are adapted from [74]), we integrate them following the 7 steps (see Section 8.2). Without loss of generality and to simplify the discussion, we assume that there are no schematic discrepancy (Step 1), no naming conflict (Step 2) and no key conflict (Step 4) among these schemas, and the schemas only include few attributes.*



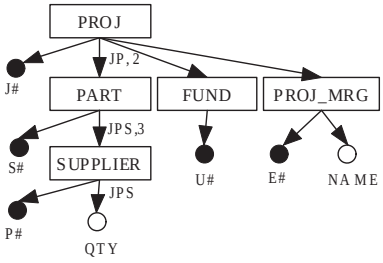
(a) Schema S1



(b) Schema S2



(c) Schema S3



(d) Schema S4

Figure 8.4: Source schemas

Step 3 *Resolve attribute-class conflicts. The attribute PROJ_MGR (i.e., project manager) in Schema S1 corresponds to the object classes PROJ_MGR in Schema S3 and S4. We transform it into an object class in S1.*

Step 5 *Merge relationship types. The relationship type JP of Schema S1, JPS of S3 and JPS of S4 are at the same level and have equivalent (sub-)structures. We merge them using Algorithm MergeRel.*

In Step 1 of Algorithm MergeRel, we first merge the JPS relationship types of S3 and S4, then JP of S1, and get an integrated schema (path) PROJ/PART/SUPPLIER constituted of two cognate paths P1=PROJ/PART from S1, S3 and S4 and P2=SUPPLIER from S3 and S4.

Then in Step 2 of Algorithm MergeRel, we compute the minimum cost tree of the integrated schema obtained in the last step. For P1, the path PROJ/PART is the only beneficial path which benefits the transformations of the relationships of all the three relationship types. P2 is a trivial cognate path. Consequently, we get the final integrated schema (path) PROJ/PART/SUPPLIER. This schema preserves the relationships of the source relationship types of S1, S3 and S4, has no redundancy and the minimal cost of data transformation. Actually, we only need to reorder the objects of SUPPLIER and PART in the relationship type JPS of S3 in data transformation.

Similarly, the relationship types between PROJ and PROJ_MGR in Schema S1 and S4 are merged.

Step 6 *Build class hierarchies and references. In the merged schema (say S5) of S1, S3 and S4, the object class FUND (from Schema S4) is a generalization of LOCAL_FUND and FOREIGN_FUND (from S1), and STAFF (from S3) is a generalization of PROJ_MGR (from S1, S3 and S4) and ORDINARY_STAFF (from S3). ISA relationship types are created to link them. The result is shown as Figure 8.5 (attributes are all omitted for convenience).*

Note the object class STAFF of S5 in Figure 8.5 has an inclusion dependency: its objects are the staff involved in some projects. The same object class of Schema S2 has not this constraint, and therefore cannot be merged into S5; otherwise, some staff of S2 may be lost in the integrated schema. Instead, we

use a reference (dashed arrows) to indicate that the staff members of S5 is in the object class STAFF of S2.

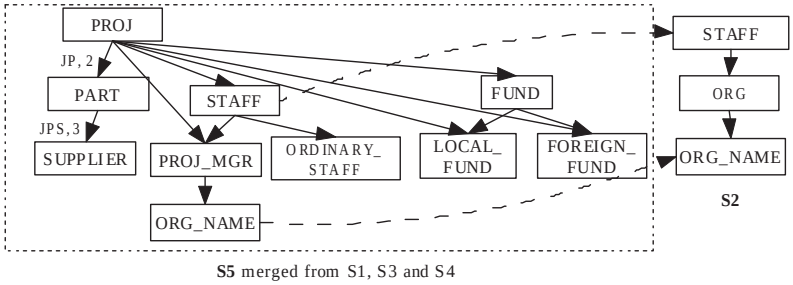


Figure 8.5: Intermediate integrated schema of S1 to S4 after Step 6

Step 7 *Remove the redundant relationship types. In Schema S5 of Figure 8.5, we remove the redundant relationship type from PROJ to PROJ_MGR (this relationship type can be derived from the relationship type from PROJ to STAFF and the relationship type from STAFF to PROJ_MGR), the relationship type from PROJ to LOCAL_FUND, the relationship type from PROJ to FOREIGN_FUND, and the relationship type from PROJ_MGR to ORG_NAME in S5. Figure 8.6 is the final integrated schema.* □

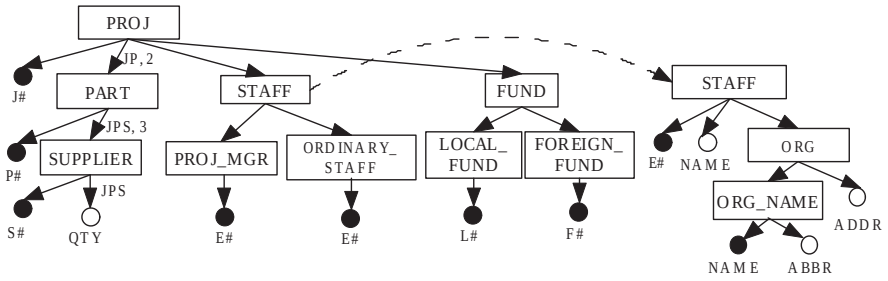


Figure 8.6: Integrated schema of S1 to S4 by our approach

Note that in Step 5, we adopt a rather strict method to integrate schemas, i.e., only relationship types at the same level would be merged, as two relationship types

at the same level have the same context information (i.e., inclusion dependency). However, this restriction may be loosen if we allow merging equivalent (sub) relationship types of different levels. Then we could obtain more concise integrated schemas, but lose some information.

8.6 Comparison with other approaches to XML schema integration

In [74], they treated object classes as first-class citizens, and focused on the resolution of attribute-class conflicts, classification inconsistencies and ancestor-descendant conflicts in the integration of XML schemas in ORASS. To integrate schemas, they assigned each source schema a weight of importance, and tried to keep the characteristics of source schemas with larger weights in the integrated schemas. This work is an extension to the previous work. The integration approach of this work is based on the merging of relationships, which preserves more information. Instead of using the generic criterion of "weight" assigned by users, we adopt three concrete criteria for schema integration.

Figure 8.7 is the integrated schema of Schemas S1 to S4 in Figure 8.4 by the approach of [74], which has some problems in comparison to Figure 8.6:

- The binary relationship type JP of Schema S1 is lost in Figure 8.7. E.g., a relationship between a project and a part that occurs in S1 but not in S3 and S4 would be lost. Note by assigning a high weight to S1, we may preserve JP, but some other relationships may be lost. The reason is that their work was based on the merging of object classes. Relationship types are added to link object classes. When several relationship types point to one object class, the relationship types with small weights would be removed (i.e., lost) in the

integrated schema. Our approach is based on the merging of relationship types, and therefore preserve information.

- Schema S2 is merged with the other schemas. This introduces an unnecessary constraint to the objects of Schema S2, and may cause some loss of information. E.g., a staff member of S2 that does not participate in any project would be lost in the integrated schema. The reason is that they merge equivalent object classes, not considering the ancestors (that represent the context information) of the classes. In our approach, we merge the equivalent object classes with the same ancestors, and therefore have not this problem.

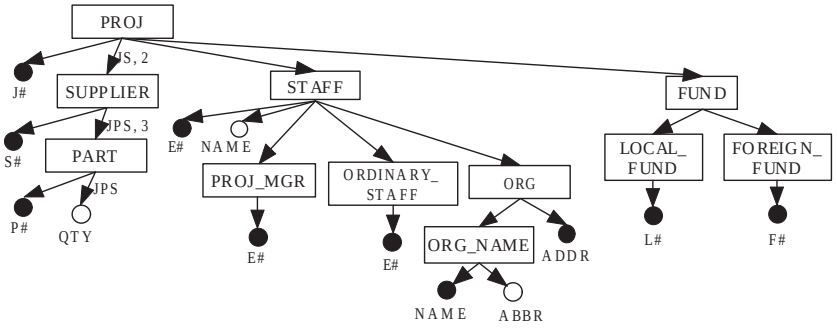


Figure 8.7: Integrated schema of S1 to S4 by the approach of [74]

Unlike our semantic approach, some work in XML schema integration was based on the grammar model of DTD. For example, [65] devised a semantic approach to integrate a set of DTDs into a common conceptual schema. That work has some deficiencies compared with ours: (1) they handled only binary relationship types between XML elements; (2) they did not integrate equivalent relationship types in different hierarchical orders; (3) the integrated schema may contain some unnecessary circles for redundant relationship types. [29] applied a tree grammar inference technique to generate a set of tree grammar rules from source DTDs. For

the inadequacies of DTD in schema integration (Section 2.2), their approach may lose semantics and the integrated schema may be redundant.

The example below shows the differences between the integration approaches based on ORASS and DTD. Figure 8.8 is the integrated schema of Schemas S1 to S4 by the approach of [29], which has some problems in comparison to Figure 8.6:

- Some relationship types are redundant, e.g., the one from PROJ to PART, the one from PART to SUPPLIER, and the one from PROJ to PROJ_MGR. The reason is that their approach did not check the recursive definitions of elements which are redundant in an integrated DTD. Actually as DTD does not support n-ary ($n > 2$) relationship types in hierarchical paths, the approaches based on DTD can hardly eliminate the redundant relationship types without any loss of information.

- QTY, an attribute of the ternary relationship type JPS, becomes an attribute under the object class SUPPLIER. In other words, QTY becomes an attribute of SUPPLIER or an attribute of the binary relationship type between PROJ and SUPPLIER. This is wrong as quantities are dependent on products, suppliers and months. The underlying reason is that DTD does not support the attributes of relationship types.

- Schema S2 is merged with the other schemas. The same problem occurred as mentioned above.

- They did not consider class hierarchies, e.g., the ISA relationships between FUND and LOCAL_FUND/GLOBAL_FUND.

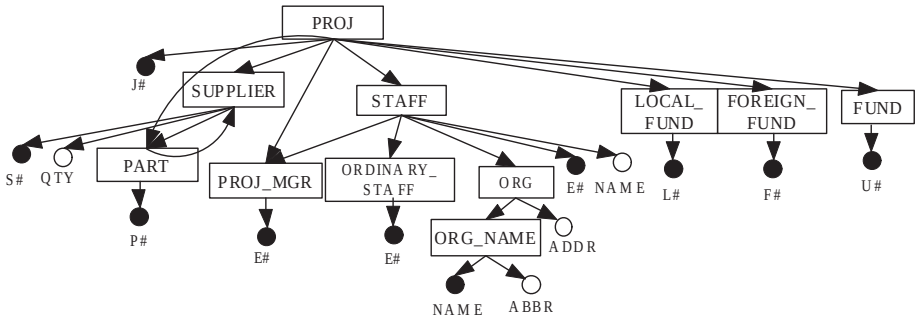


Figure 8.8: Integrated schema of S1 to S4 by the approach of [29]

8.7 Summary

In some XML data integration systems, e.g., Xyleme [17], Nimble [18], LoPiX [49] and YAT [14, 15], the developers either provided an XML query language for users to write integrated schemas by hand, or assumed that an integrated schema and the mapping from source schemas to the integrated schema have been given already. They focused on query processing through the integrated schema. However, our purpose was to developed an approach to automatically integrate a set of XML schemas given the schema matching information. The integrated schema should satisfy three criteria emanating from applications, i.e., information preservation, minimization of both redundancy and the cost of data transformation. Most of the previous work was based on DTD, and therefore could not handle n-ary ($n > 2$) relationship types among object classes and attributes of relationship types. Yang et al. [74] used ORASS to represent XML schemas. However, their approach was based on the merging of object classes, and may loss some relationship types. They used the user-specified weight of importance to assess source schemas and to compute integrated schemas. However, the criterion of weight was sometimes too vague to be specified. In this work, we also adopted ORASS to represent XML schemas for the rich semantics of the model. Our work was based on the merging

of relationship types, and preserved the information of object classes, relationship types, attributes of object classes and attributes of relationship types. In particular, we presented Algorithm MergeRel to merge equivalent (sub) relationship types with different hierarchical structures. Consequently, our approach can produce better integrated schemas than Yang et al.'s. Furthermore, our approach was based on the three specific criteria that were customized for the applications introduced in Section 8.1.

Chapter 9

Conclusion

9.1 Summary of contributions

This research attempted to resolve some semantic heterogeneities and derive constraints in the integration of relational, ER or XML schemas.

Schematic discrepancy occurs when meta information in one schema correspond to data values in another schema. In relational databases, researchers [34] have developed the 4 restructuring operators fold, unfold, unite and split to transform discrepant schemas into consistent ones. In this work, we developed a theory for discrepant schema transformations. We gave the algebraic laws, i.e., reconstructibility and commutativity, of the 4 restructuring operators. We found fold and unfold operations are not reconstructible unless the original relations satisfy some functional dependencies (i.e., Properties 3 and 4 in Section 5.1.1). A discrepant schema transformation is lossless if the operations of it are all reconstructible. A lossless discrepant schema transformation can be simplified to a non-redundant transformation that contains only necessary operations. This theoretical work could be used to improve the integration/interoperation of multiple and heterogeneous databases. Furthermore, as a multidatabase query can be implemented with the restructuring

operators and relational algebra [34], our work (together with the existing theory of relational algebra) could be used to optimize the query.

Conventional functional dependencies are inadequate to represent constraints in multiple and heterogeneous databases. We extended functional dependencies to qualified functional dependencies, i.e., functional dependencies holding over a set of relations or a set of horizontal partitions of relations, to represent some common constraints in multidatabase, and studied the derivation and uses of qualified functional dependencies in multidatabases. We presented a set of inference rules to derive new qualified functional dependencies from given ones in a set of fixed relational schemas. We proved that these rules are sound, complete and irreducible. We gave an algorithm CLOSURE (in Section 6.1.3) to compute an attribute closure under the qualification of some attribute values in a set of relations with respect to a set of qualified functional dependencies. We gave the propagation rules to derive the qualified functional dependencies on transformed schemas from the qualified functional dependencies on original schemas in application of a (set of) restructuring operator(s). In discrepant schema transformations, we are mostly interested in simple qualified functional dependencies that are the qualified functional dependencies represented as functional dependencies in a canonical schema (a canonical schema is one in which all interesting information is modelled as attribute values, e.g., $DB1$ in Figure 1.1). Based on the inference and propagation rules, we proposed algorithms to derive simple qualified functional dependencies in discrepant schema transformations. The algorithms are sound and complete. The efficient derivation algorithm runs in polynomial time for some large and significant classes of inputs. Our theory of qualified functional dependency derivation is useful in data integration/mediation systems and multidatabase interoperation. For example, we can use it to verify lossless transformations, normalize integrated schemas, detect

duplicate and inconsistency, verify data integrity in materialized view maintenance, and optimize queries in data integration/mediation. SchemaSQL is a multidatabase language which was used to solve many problems [54]. However, we found that a SchemaSQL view may be ambiguous. By deriving qualified functional dependencies for a view, we can verify the its uniqueness.

In comparison with the relational model, the ER approach provides adequate schema constructs to model an enterprise, and supports cardinality constraints. This makes ER an appropriate model in the view integration of database design and the database-schema integration of federated database systems. In the integration of ER schemas with schematic discrepancies, we explicitly represent the meta information (i.e., context) of each schema construct as a set of meta-attributes with values, and resolve schematic discrepancies by transforming the meta attributes into attributes of entity types. The schema transformation was proven to be both information preserving and constraint (functional dependency and multivalued dependency) preserving. Note that as the ER model supports cardinality constraints, the derivation of constraints is involved in schema transformation rather than a separate process. The resolution of schematic discrepancies for entity types and relationship types is an extension of the unite operator, and the resolution of schematic discrepancies for the attributes of entity types and attributes of relationship types is an extension of the fold operator in the ER model. Qualified functional dependencies (and qualified multivalued dependencies) are represented as cardinality constraints with contexts in ER. The theory of the derivation of qualified functional dependencies was extended to propagate cardinality constraints in the transformation of ER schemas. This work complements the previous work resolving other semantic heterogeneities in the integration of ER schemas.

As XML becomes more and more a de facto standard to represent and exchange

data on the Web, XML schema and data integration becomes a hot topic in both academy and industry. The resolutions of schematic discrepancies and other semantic heterogeneities for the relational and ER models can be extended to XML. However, the hierarchical structure of XML brings new challenges in the integration of XML schemas, which becomes the focus of our work. Most of the previous work was based on DTD, and therefore could not handle n-ary ($n > 2$) relationship types among object classes and attributes of relationship types. Yang et al. [74] used ORASS to represent XML schemas. However, their approach was based on the merging of object classes, and may loss some relationship types. They used the user-specified weight of importance to assess source schemas and to compute integrated schemas. However, the criterion of weight was sometimes too vague to be specified. In this work, we also adopted ORASS to represent XML schemas for the rich semantics of the model. Our work was based on the merging of relationship types, and preserved the information of object classes, relationship types, attributes of object classes and attributes of relationship types. In particular, we presented Algorithm MergeRel to merge equivalent (sub) relationship types with different hierarchical structures. We identified three specific criteria of schema integration in different applications, i.e., information preserving, minimization of both redundancy and the cost of data transformation. Our approach was based on the three specific criteria that were customized for the applications introduced in Section 8.1. Furthermore, to resolve the hierarchical inconsistencies among XML schemas, we gave an efficient method to reorder the objects in relationships (i.e., changing the hierarchical positions of the objects in the relationships which are represented as hierarchical paths in ORASS) using relational databases. Our work is useful in XML-based information integration systems and generic model management systems. The methods of reordering object classes and merging relationship types can

also be used in XML view management.

9.2 Future work

There are some strong assumptions behind the restructuring operators. For example, in application of fold and unite operations, we suppose that we know in advance that the attribute names or relation names are the values of some attributes. Such information may be hard to know in practice. Are there any ways to get these information automatically? For another example, in application of unfold and split operations on some attributes, we assume that the domains of the attributes are finite. How about the cases when the domains are infinite? These problems need to be resolved when we apply the restructuring operators in practice.

We studied the derivation of qualified functional dependencies together with functional dependencies, cardinality constraints in discrepant schema transformations. It would be interesting and useful to incorporate some other constraints, e.g., inclusion dependencies, multivalued dependencies, etc, in the framework.

A promising application of our theory of qualified functional dependency is in the optimization of queries in a multidatabase system, i.e., using the semantics of qualified functional dependencies and other constraints to add/remove some conditions in a multidatabase query.

We can also develop a normalization theory of multidatabases based on qualified functional dependencies, as the normalization theory of individual databases based on functional dependencies.

XML schema integration is a preceding procedure of data integration or mediation that requires further research. For example, given an integrated schema and the mappings from the integrated schema to source schemas, how can we query the

data in the sources through the integrated schema.

Appendix A

Appendix

A.1 Commutativity of restructuring operations

Property 5 (Commutativity of fold and unite operations). *Given a set of relations* $\{d_i(A_1, \ldots, A_n, b_1, \ldots, b_m) | i = 1, \ldots, l\}$*, such that the relation names* $d_1, \ldots, d_l$ *are from the domain of an attribute* D*, and in each relation of* d_i*, the attribute names* $b_1, \ldots, b_m$ *are from the domain of an attribute* B *and their values are from the domain of another attribute* C*, we have:*

$$\mathit{unite}(\{\mathit{fold}(d_i, B, C) | i = 1, \ldots, l\}, D) = \mathit{fold}(\mathit{unite}(\{d_1, \ldots, d_l\}, D), B, C). \quad \square$$

Property 6 (Commutativity of fold and split operations). *Given a relation* $R(A_1, \ldots, A_n, b_1, \ldots, b_m, D)$*, such that the attribute names* $b_1, \ldots, b_m$ *are from the domain of an attribute* B *and their values are from the domain of another attribute* C*, we have:*

$$\mathit{split}(\mathit{fold}(R, B, C), D) = \{\mathit{fold}(R_i, B, C) | R_i \in \mathit{split}(R, D)\}. \quad \square$$

Property 7 (Commutativity of two fold operations). *Given a relation*

$R(A_1, \ldots, A_n, b_1, \ldots, b_m, b'_1, \ldots, b'_l)$, such that the attribute names $b_1, \ldots, b_m$ are from the domain of an attribute B and their values are from the domain of an attribute C, and the attribute names $b'_1, \ldots, b'_l$ are from the domain of an attribute B' and their values are from the domain of an attribute C', we have:

$$fold(fold(R, B, C), B', C') = fold(fold(R, B', C'), B, C). \quad \square$$

Property 8 (Commutativity of fold and unfold operations). *Given a relation $R(A_1, \ldots, A_n, b_1, \ldots, b_m, B', C')$, such that the attribute names $b_1, \ldots, b_m$ are from the domain of an attribute B and their values are from the domain of an attribute C, we have:*

$$unfold(fold(R, B, C), B', C') = fold(unfold(R, B', C'), B, C). \quad \square$$

Property 9 (Commutativity of unfold and unite operations). *Given a set of relations $\{d_i(A_1, \ldots, A_n, B, C) | i = 1, \ldots, l\}$, such that the relation names $d_1, \ldots, d_l$ are from the domain of an attribute D, we have:*

$$unite(\{unfold(d_i, B, C) | i = 1, \ldots, l\}, D) = unfold(unite(\{d_1, \ldots, d_l\}, D), B, C).$$

Property 10 (Commutativity of unfold and split operations). *Given a relation $R(A_1, \ldots, A_n, B, C, D)$, we have:*

$$split(unfold(R, B, C), D) = \{unfold(R_i, B, C) | R_i \in split(R, D)\}. \quad \square$$

Property 11 (Commutativity of two unfold operations). *Given a relation*

$R(A_1, \ldots, A_n, B, C, B', C')$, *we have:*

$$\textit{unfold}(\textit{unfold}(R, B, C), B', C') = \textit{unfold}(\textit{unfold}(R, B', C'), B, C). \quad \square$$

A.2 Proof of Lemma 5.1

We prove this lemma by induction. Let $T =< T_1, \ldots, T_n >$ be a transformation consisting of n steps, such that each step is one or a set of restructuring operation(s).

Base case: $n = 2$. If T consists of two operations unite$(\mathbb{R}, B)$ and split(R, B), then according to the reconstructibility of unite (Property 2) or split (Property 1), T is the same as the "identical" transformation. That is, the unite and split operations are reverse operations. Otherwise, if T consists of two sets of operations $\{\text{fold}(R, B, C)|R \in \mathbb{R}\}$ and $\{\text{unfold}(S, B, C)|S \in \mathbb{S}\}$, then according to the reconstructibility of fold (Property 3) or unfold (Property 4), T is the same as the identical transformation. That is, the fold and unfold operations are reverse operations.

Induction step: suppose the lemma is true for any transformation T with i steps. Then for $T =< T_1, \ldots, T_i, T_{i+1} >$, we need to prove that for any of the following 4 cases, T_1 and T_{i+1} are a pair of reverse operations:

1. $T_1 = \text{unite}(\mathbb{R}, B)$, and $T_{i+1} = \text{split}(R, B)$,
2. $T_1 = \text{split}(R, B)$, and $T_{i+1} = \text{unite}(\mathbb{R}, B)$,
3. $T_1 = \{\text{fold}(R, B, C)|R \in \mathbb{R}\}$, and $T_{i+1} = \{\text{unfold}(S, B, C)|S \in \mathbb{S}\}$,
4. $T_1 = \{\text{unfold}(S, B, C)|S \in \mathbb{S}\}$, and $T_{i+1} = \{\text{fold}(R, B, C)|R \in \mathbb{R}\}$.

We only prove the 1st case, i.e., $T_1 = \text{unite}(\mathbb{R}, B)$, and $T_{i+1} = \text{split}(R, B)$. The other 3 cases can be proven similarly.

If the operations of $T_2, T_3, \ldots, T_i$ do not have the parameter of attribute B, then using the commutativity of unite operations (Properties 5 and 9), we can swap T_1 and $T_2, T_3, \ldots, T_i$ one by one, and then cancel T_i and T_{i+1} together using the reconstructibility of unite (Property 2). Then we get a new transformation T' with $i-1$ steps, such that T' do the same transformation as T and there is a one-to-one mapping from the set of all the operations of T except T_1 and T_{i+1} to the set of all the operations of T', such that h preserves the operations and the important operation orders of T. So T_1 and T_{i+1} are a pair of reverse operations in T.

Otherwise, if some operations of $T_2, T_3, \ldots, T_i$ have an parameter of attribute B, without loss of generality, let $T_j = \text{unfold}(\mathbb{R}1, B, C)$ for some $2 \leq j \leq i$ where $\mathbb{R}1$ is a set of relations produced by T_{j-1}. We claim that there must be an operation $T_k = \text{fold}(\mathbb{R}2, B, C)$ for some $j+1 \leq k \leq i$ where $\mathbb{R}2$ is a set of relations produced by T_{k-1}. Otherwise, the attribute values of B would be the attribute names of the transformed relations after T_j. That is, B would not be an attribute in R on which $T_{i+1} = \text{split}(R, B)$ is performed, which is wrong.

According to the induction assumption, T_j and T_k are a pair of reverse operations in the transformation $T1 =< T_j, T_{j+1}, \ldots, T_k >$, and we can find a transformation $T1'$ that does the same transformation as $T1$, and has $k-j-1$ steps at most. Replacing $T1$ with $T1'$ in T, we get a transformation T' that does the same transformation as T, and has $i-1$ steps at most. We can construct a one-to-one mapping h from the set of some operations of T to the set of all the operations of T', such that h maps the operations $T_1, \ldots, T_{j-1}, T_{k+1}, \ldots, T_{i+1}$ of T to the same operations of T', and maps all the operations of $T1$ in T to some operations of $T1'$ in T'. Note h preserves the operations and the important operation orders of T.

As T' has $i-1$ steps at most, according to the induction assumption, unite$(\mathbb{R}, B)$

and split(R, B), i.e., the first and last operations of T', are a pair of reverse operations in T'. That is, we can find an transformation T'' that does the same transformation as T', and a one-to-one mapping h' from the set of some operations of T' except unite$(\mathbb{R}, B)$ and split(R, B) to the set of all the operations of T'', such that h' preserves the operations and the important operation orders of T'.

Then we can construct a mapping $h' \circ h$ from the set of some operations of T except unite$(\mathbb{R}, B)$ and split(R, B) to the set of all the operations of T''. The mapping preserves the operations and the important operation orders of T. Consequently, unite$(\mathbb{R}, B)$ and split(R, B) is a pair of reverse operations in T.

A.3 Proof of Lemma 5.2

($\Rightarrow$) If T is a non-redundant transformation, then T does not contain any pair of reverse operations. Otherwise, we can remove the reverse operations by the three steps given before Lemma 5.2 in Section 5.2.

($\Leftarrow$) Suppose T does not include any pair of reverse operations, but is a redundant transformation. That is $\exists T' \preceq T$, T' implements the same transformation as T, but $T \preceq T'$ is not true. That is, for some operation (or operation set) $T_i \in T, \neg\exists T'_j \in T'$, such that T_i and T'_j are of the same kind of operators, and have the same parameters of attributes. In this case, we claim that no matter what kind of operation T_i is, T' does not implement the same transformation as T. This will contradict our assumption. We first show this for the unite operation.

Suppose unite$(\mathbb{R}, B) \in T$, and unite$(\mathbb{S}, B) \notin T'$, where $\mathbb{R}$ and $\mathbb{S}$ are two sets of relations whose names are from $dom(B)$. We can infer that the values of the attribute B are modelled as the names of the original relations of T (T'). As T does not contain any pair of reverse operations, and unite$(\mathbb{R}, B) \in T$, split(R, B)

$\notin T$ for any relation name R. Then in the target transformed relations of T, the values of B are modelled as attribute names or attribute values, but not relation names. However in T', as we never perform a unite operation on B, the values of B are modelled as relation names in the target transformed relations. Then the transformed relations of T and T' are different. This contradicts our assumption that T and T' implement the same transformation.

The proofs for the other three kinds of operations that T_i can be are similar, which are omitted here. We conclude if T has no reverse operations, then it is a non-redundant transformation.

A.4 Proof of Theorem 6.1

For the soundness, we only prove Rule A9:

(A9) If $\mathbb{R}(\overline{X} \rightarrow Y)$ and $\mathbb{R}(\overline{X1}, Y \rightarrow Z)$ hold for $\overline{X1}$ a set (possibly an empty set) of some qualification attributes in $\overline{X}$, then $\mathbb{R}(\overline{X} \rightarrow Z)$ holds.

Given any two tuples $t1$ and $t2$ from the relation(s) of $\mathbb{R}$, such that the attribute values of $t1$ and $t2$ satisfy the conditions specified in $\overline{X}$. That is, if attribute $A \in \overline{X}$, then $t1.A = t2.A$; if qualification attribute $A_{\sigma=S} \in \overline{X}$, then $t1.A \in S$ and $t2.A \in S$. As the qualified functional dependency $\mathbb{R}(\overline{X} \rightarrow Y)$ holds, the attribute values of Y are the same in $t1$ and $t2$. As $\overline{X1}$ is a subset of $\overline{X}$, $t1$ and $t2$ also satisfies the conditions of $\overline{X1}$. As the qualified functional dependency $\mathbb{R}(\overline{X1}, Y \rightarrow Z)$ holds, the attribute values of Z are the same in $t1$ and $t2$. Consequently, the qualified functional dependency $\mathbb{R}(\overline{X} \rightarrow Z)$ holds. The proof of the soundness of the other inference rules are omitted.

The irreducibility can be proven by showing that if we delete any one of the 10 rules, we can find a set of relational schemas and a set of qualified functional

dependencies, such that the set of the dependencies derived by the left 9 rules is a proper subset of the set of the dependencies derived by the 10 rules.

For Rule A1, given two relational schemas $R1(A, B)$ and $R2(A, B)$, let $\{R1, R2\}(A \rightarrow B)$ be a qualified functional dependency on them. We prove that A1 is irreducible by showing that using Rules A2 to A10, we cannot derive the functional dependency $R1(A \rightarrow B)$ (or $R2(A \rightarrow B)$ which can be derived by A1). To show this, we just compute the set of all the qualified functional dependencies derived using Rules A2 to A10. The qualified functional dependency $R1(A \rightarrow B)$ is not in the set. Actually, without Rule A1, we can only derive the trivial qualified functional dependencies on $R1$ by Rules A5 and A7.

The irreducibilities of the other rules can be proven similarly. In the rest of this section, we prove the completeness of the inference rules.

We define a *satisfaction relationship* "$\sqsupseteq$" between two mixed sets of regular and qualification attributes $\overline{X}$ and $\overline{Y}$. Intuitively, $\overline{X} \sqsupseteq \overline{Y}$, read "$\overline{X}$ satisfies $\overline{Y}$" if, $\overline{X}$ specifies more restrictive conditions than $\overline{Y}$ when they are on the left hand sides of qualified functional dependencies. Specifically, if $\overline{X}$ and $\overline{Y}$ only contain regular attributes, and $\overline{X} \supseteq \overline{Y}$, then $\overline{X} \sqsupseteq \overline{Y}$, as $\overline{X}$ requires more attributes to be equal than $\overline{Y}$ when they appear on the left hand sides of (qualified) functional dependencies.

Definition A.1 (Satisfaction relationship). *Given two mixed sets of regular and qualification attributes $\overline{X}$ and $\overline{Y}$, we define $\overline{X} \sqsupseteq \overline{Y}$ provided: (a) for any qualification attribute $A_{\sigma=S1} \in \overline{Y}$ for $S1 \subseteq dom(A)$, there's a qualification attribute $A_{\sigma=S2} \in \overline{X}$ such that $S2 \subseteq S1$, and (b) for any regular attribute $A \in \overline{Y}$, either $A \in \overline{X}$ or $A_{\sigma=\{a\}} \in \overline{X}$ for some $a \in dom(A)$.* $\square$

The following rules hold for the $\sqsupseteq$ relation:

1. *Transitivity.* $\overline{X} \sqsupseteq \overline{Y}, \overline{Y} \sqsupseteq \overline{Z} \Rightarrow \overline{X} \sqsupseteq \overline{Z}$

2. *Augmentation.* $\overline{X} \cup \overline{Y} \sqsupseteq \overline{X}$

3. *Additivity.* $\overline{X} \sqsupseteq \overline{Y}, \overline{X} \sqsupseteq \overline{Z} \Rightarrow \overline{X} \sqsupseteq \overline{Y} \cup \overline{Z}$

The attribute closure of a set of attributes X under a more restrictive condition would contain more attributes, as described in the lemma below (the proof is omitted).

Lemma A.1. *Given two sets of relations* $\mathbb{R}1$ *and* $\mathbb{R}2$*, let* $W1$ *and* $W2$ *be two sets of qualification attributes, and* X *be a set of attributes. If* $\mathbb{R}1 \subseteq \mathbb{R}2$ *and* $W1 \sqsupseteq W2$*, then* $X^+_{\mathbb{R}1,W1} \supseteq X^+_{\mathbb{R}2,W2}$. $\square$

For example, given 2 qualified functional dependencies $\mathbb{R}(A, B_{\sigma=\{b_1,b_2\}} \rightarrow C)$ and $\mathbb{R}(A, B_{\sigma=\{b_1\}} \rightarrow D)$. Let $W1 = \{B_{\sigma=\{b_1\}}\}$ and $W2 = \{B_{\sigma=\{b_1,b_2\}}\}$. Then $W1 \sqsupseteq W2$ as $W1$ specifies a more restricted condition (i.e., attribute B can only take value b_1) than $W2$ (i.e., attribute B can take value b_1 or $b2$) when they occur on the left hand side of qualified functional dependencies. We compute $\{A\}^+_{\mathbb{R},W1} = \{C, D\}$ and $\{A\}^+_{\mathbb{R},W2} = \{C\}$. So $\{A\}^+_{\mathbb{R},W1} \supseteq \{A\}^+_{\mathbb{R},W2}$.

Then we define the projection of attributes below.

Definition A.2. *Given a mixed set of regular and qualification attributes* $\overline{X}$ *and a set of regular attributes* Y*, we define the* projection of $\overline{X}$ onto Y *to be a subset of* $\overline{X}$ *consisting of the regular attributes also in* Y *and the qualification attributes qualifying the attributes of* Y*:*

$$\overline{X}[Y] = \{A | A \in \overline{X} \cap Y\} \cup \{A_{\sigma=S} | A_{\sigma=S} \in \overline{X} \text{ and } A \in Y\}. \quad \square$$

For example, let $\overline{X} = \{A_{\sigma=S1}, B_{\sigma=S2}, C, D\}$ and $Y = \{A, C\}$, then $\overline{X}[Y] = \{A_{\sigma=S1}, C\}$.

The inference rules A1 to A10 are complete. We will prove this by showing that if F is a set of qualified functional dependencies that hold over a set of relation schemas $\mathbb{S}$, and f is a qualified functional dependency which cannot be proved by the inference rules, then there must be an instance of $\mathbb{S}$ over which the dependencies of F all hold but f does not; that is, F does not logically imply f. Algorithm CONSTRUCT_INSTANCE below is proposed to construct such an instance of $\mathbb{S}$ for the given F and f. We prove the correctness of the algorithm in the following 2 lemmas.

Lemma A.2. *In an iteration of Algorithm CONSTRUCT_INSTANCE, QA is the current set of qualification attributes, and XC is the current attribute closure. Let QA' and XC' be the QA and XC after the execution of the algorithm. In the returned relations of $R1$ and $R2$, all the attributes in XC' have the same non-null values, while the other attributes have different values.*

Proof. First we prove that at the beginning of the k-th iteration of the while loop (Lines 6 to 17), those attributes with the same non-null values in $R1$ and $R2$ belong to XC.

Base case: k=1. the condition holds as all attributes have the null value.

Induction step: Let QA_k and XC_k be the values of QA and XC at the beginning of the k-th iteration. Suppose at the beginning of the k-th iteration, the attributes with the same non-null values in $R1$ and $R2$ belong to XC_k. As $QA_{k+1} \sqsupseteq QA_k$, $XC_{k+1} \supseteq XC_k$ (Lemma A.1). For each attribute A with the same value in the relations of $R1$ and $R2$, if it is in XC_k, it is also in XC_{k+1}; otherwise, according to the induction hypothesis, the algorithm assigned the same value to A in $R1$ and $R2$ during the k-th iteration (Line 9). So $A \in XC_{k+1}$ according to the condition of Line 8.

When the loop terminates, we have: all the attributes in $R1$ and $R2$ have non-

Algorithm:CONSTRUCT_INSTANCE

Input: Let F be a set of qualified functional dependencies that hold over a set of relational schemas $\mathbb{S}$ with the same set of attributes U, and suppose f, $\mathbb{R}(W, X \rightarrow Y)$, cannot be derived by the inference rules A1 to A10, where $\mathbb{R} \subseteq \mathbb{S}$, W is a set of qualification attributes, and X and Y are 2 sets of regular attributes from U.

Output: An instance of $\mathbb{S}$ over which the dependencies of F all hold, while f does not.

1: Find two relational schemas $R1, R2 \in \mathbb{R}$, such that $Y \nsubseteq X^{+}_{\{R1,R2\},W}$;
2: /* $R1$ *and* $R2$ *may refer to the same relational schema if* $\mathbb{R}$ *comprises a single relational schema. The following statements construct a tuple for each relation of* $R1$ *and* $R2$*. The other relations of* $\mathbb{S}$ *are empty.* */
3: Initialize each attribute in $R1$ and $R2$ with the *null* value;
4: $QA := W$; /* QA *is the current set of qualification attributes.* */
5: $XC := X^{+}_{\{R1,R2\},QA}$;
6: **while** $\exists A \in U, R1[A] = R2[A] = null$ or $\exists A \in XC, R1[A] \neq R2[A]$ **do**
7: /* $R1[A]$ *refers to the value of attribute* A *in the relation of* $R1$*.* */
8: **if** $A \in XC$ and ($R1[A] \neq R2[A]$ or $R1[A] = R2[A] = null$) **then**
9: $R1[A] := a$ and $R2[A] := a$ such that $a \in dom(A)$, $\{A_{\sigma=\{a\}}\} \sqsupseteq QA[A]$, and $Y \nsubseteq X^{+}_{\{R1,R2\},QA\cup\{A_{\sigma=\{a\}}\}}$;
10: $QA := QA \cup \{A_{\sigma=\{a\}}\}$; /*Note that if QA contains a qualification attribute $A_{\sigma=S}$, then the qualification attribute is replaced by $A_{\sigma=\{a\}}$ in the new QA. */
11: **else**
12: /* i.e., $A \notin XC$ and $R1[A] = R2[A] = null$ */
13: $R1[A] := a_1$ and $R2[A] := a_2$ such that $a_1, a_2 \in dom(A)$, $a_1 \neq a_2$, $\{A_{\sigma=\{a_1,a_2\}}\} \sqsupseteq QA[A]$, and $Y \nsubseteq X^{+}_{\{R1,R2\},QA\cup\{A_{\sigma=\{a_1,a_2\}}\}}$;
14: $QA := QA \cup \{A_{\sigma=\{a_1,a_2\}}\}$;
15: **end if**
16: $XC := X^{+}_{\{R1,R2\},QA}$;
17: **end while**
18: **return** the set of the relations $\{R1, R2\}$. /* *The other relations of* $\mathbb{S}$ *are all empty.*/

null values, all the attributes of XC' have the same values in $R1$ and $R2$ (the condition of the while loop), and all the attributes with the same values in $R1$ and $R2$ belong to XC' (loop invariant proven above). That is, the attributes not in XC' have different values in $R1$ and $R2$.

Finally, the while loop will terminate, as in each iteration, the algorithm assigns values to one attribute, and it will assign values to one attribute at most twice. □

Lemma A.3. *Given the input of Algorithm CONSTRUCT_INSTANCE, it constructs an instance of* $\mathbb{S}$ *over which the qualified functional dependencies of* F *all hold but* f *doest not.*

Proof. (1) The qualified functional dependencies of F all hold over the set of the relations $\{R1, R2\}$ returned by the algorithm.

Let QA' and XC' be the values of QA and XC after the execution of the algorithm. We have proven that in the returned relations $R1$ and $R2$, all the attributes in XC' have the same values, while the others have different values. Suppose a qualified functional dependency $\mathbb{R}1(W1, X1 \rightarrow Y1)$ in F does not hold over $\{R1, R2\}$, where $W1$ is a set of qualification attributes, and $X1$ and $Y1$ are sets of regular attributes. That is, $\{R1, R2\} \subseteq \mathbb{R}1$, $QA' \sqsupseteq W1 \cup X1$, and $Y1$ cannot be a subset of XC'. As the attributes of $X1$ have the same values in $R1$ and $R2$, $X1 \subseteq XC' = X^{+}_{\{R1,R2\},QA'}$ (Lemma A.2). That is, $\{R1, R2\}(QA', X \rightarrow X1)$ holds. Then we can derive $\{R1, R2\}(QA', X \rightarrow Y1)$. But then by the definition of attribute closure, $Y1 \subseteq XC'$, which we assumed not to be the case. We conclude by contradiction that each qualified functional dependency $\mathbb{R}1(W1, X1 \rightarrow Y1)$ in F holds over the set of the relations $\{R1, R2\}$.

(2) $f : \mathbb{R}(W, X \rightarrow Y)$ does not hold over $\{R1, R2\}$.

We first try to prove that at the beginning of the k-th iteration, $QA \sqsupseteq W$ and $Y \nsubseteq XC$.

Base case: k=1. $QA \sqsupseteq W$ as $QA = W$. We can always find two relations $R1, R2 \subseteq \mathbb{R}$, such that $Y \nsubseteq XC = X^+_{\{R1,R2\},QA}$; otherwise, $\{Ri, Rj\}(W, X \rightarrow Y)$ for any $Ri, Rj \in \mathbb{R}$ can be deduced from F, then we can infer $\mathbb{R}(W, X \rightarrow Y)$ by the inference rule A2. It contradicts the assumption that f cannot be deduced by the inference rules.

Induction step: Let QA_k and XC_k be the values of QA and XC at the beginning of the k-th iteration. Suppose at the beginning of the k-th iteration, $QA_k \sqsupseteq W$ and $Y \nsubseteq XC_k$. Let $QA_{k+1} = QA_k \cup \{A_{\sigma=S}\}$, then $QA_{k+1} \sqsupseteq QA_k$. According to the induction hypothesis $QA_k \sqsupseteq W$, we have $QA_{k+1} \sqsupseteq W$. In what follows, we will prove Y is not a subset of XC_{k+1} by contradiction. According to the induction hypothesis, $\{R1, R2\}(QA_k, X \rightarrow Y)$ cannot be deduced from F by the inference rules. We consider two cases corresponding to the conditions of Lines 8 and 11:

Case 1: $A \in XC_k$ and $QA_{k+1} = QA_k \cup \{A_{\sigma=\{a\}}\}$. We can always find the value a of attribute A, such that $\{R1, R2\}(QA_{k+1}, X \rightarrow Y)$ cannot be deduced from F by the inference rules; otherwise, $\{R1, R2\}(QA_k, A_{\sigma=\{a\}}, X \rightarrow Y)$ can be deduced for each a such that $\{A_{\sigma=\{a\}}\} \sqsupseteq QA_k[A]$. So $\{R1, R2\}(QA_k, A, X \rightarrow Y)$ can be inferred by the inference rules. As $A \in XC_k$, i.e., $\{R1, R2\}(QA_k, X \rightarrow A)$ holds, we can deduce $\{R1, R2\}(QA_k, X \rightarrow Y)$, which we assumed not to be the case.

Case 2: $A \notin XC_k$ and $QA_{k+1} = QA_k \cup \{A_{\sigma=\{a_1,a_2\}}\}$. We can always find two values a_1, a_2 of attribute A, such that $\{R1, R2\}(QA_{k+1}, X \rightarrow Y)$ cannot be deduced from F by the inference rules; otherwise, $\{R1, R2\}(QA_k, A_{\sigma=\{a_i,a_j\}}, X \rightarrow Y)$ can be deduced for any a_i, a_j such that $\{A_{\sigma=\{a_i,a_j\}}\} \sqsupseteq QA_k[A]$. So $\{R1, R2\}(QA_k, X \rightarrow Y)$ can be inferred by the inference rules, which we assumed not to be the case.

When the loop terminates, $QA' \sqsupseteq W$ and $Y \nsubseteq XC'$. So there must be an attribute in Y which has different values in the relations of $R1$ and $R2$. As $X \subseteq XC'$, all the attributes of X have the same values in $R1$ and $R2$. Consequently,

$\mathbb{R}(W, X \rightarrow Y)$ does not hold over the set of the relations $\{R1, R2\}$. Note that the loop will terminate as mentioned in the proof of Lemma A.2. □

From Lemma A.3, we know that the rules A1 to A10 are complete to infer qualified functional dependencies in fixed relational schemas.

A.5 Proof of Theorem 6.2

We first give a lemma below.

Lemma A.4. *In Algorithm CLOSURE, for any* $x_1 \in dom(X_1), \ldots, x_n \in dom(X_n)$, *such that* $\{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\} \sqsupseteq W[X_1, \ldots, X_n]$, *let* $W1 = W \cup \{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\}$, *the inner loop (Line 6 to 10) correctly computes* $closure1$, *i.e.,* $X^+_{\mathbb{R},W1}$ *w.r.t.* F.

Proof. First, if A is in the set $closure1$ produced by the inner loop, then $A \in X^+_{\mathbb{R},W1}$. This can be proven easily by induction. We now prove the converse: if $A \in X^+_{\mathbb{R},W1}$, then A is in $closure1$ produced by the inner loop.

Suppose $A \in X^+_{\mathbb{R},W1}$, but A is not in $closure1$ produced by the inner loop. Consider 2 relational schemas $R1, R2 \in \mathbb{R}$ ($R1$ and $R2$ may refer to the same relational schema if $\mathbb{R}$ comprises only one relation). $R1$ and $R2$ each has a tuple that agree on the attributes of $closure1$ and disagree on all the other attributes, and the attribute values satisfy the qualification of $W1$. We claim that the set of the relations $\{R1, R2\}$ satisfies F. If not, let $\mathbb{R}1(V, Y \rightarrow Z)$ (V is a set of qualification attributes, and Y and Z are 2 sets of regular attributes from U) be a dependency of F that is violated by $\{R1, R2\}$. That is, $\{R1, R2\} \subseteq \mathbb{R}1$ and $\{A_{\sigma=R1[A]\cup R2[A]} | A \in U\} \sqsupseteq V \cup Y$, and Z cannot be a subset of $closure1$.

We claim that $V \subseteq \{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\} \cup \{A_{\sigma=\{a\}} | A_{\sigma=\{a\}} \in W\}$. First, as the qualification attributes of V take single values according to our restriction

on F, the attributes of V are all from $closure1$; otherwise, an attribute of V has different values in $R1$ and $R2$, and then the attribute values of $R1$ and $R2$ cannot satisfy the qualification of V. Second, the attributes of V will not be on the right hand side of a qualified functional dependency, according to the condition that there is no intersection between the sets of regular and qualification attributes of the qualified functional dependencies of F, so the attributes of V will not be added into $closure1$ during the inner loop (Lines 6 to 10). So all the attributes qualified in V can only be added into $closure1$ by the statement of Line 4 in the algorithm, i.e., $V \subseteq \{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\} \cup \{A_{\sigma=\{a\}} | A_{\sigma=\{a\}} \in W\}$.

So $W1 \sqsupseteq V$. As $Y \subseteq closure1$, $closure1 \sqsupseteq Y$. So $W1 \cup closure1 \sqsupseteq V \cup Y$. So $Z \subseteq clsoure1$ (Lines 7 to 9). We conclude by contradiction that each qualified functional dependency of F holds over $\{R1, R2\}$.

Thus, the relation set $\{R1, R2\}$ must also satisfy $\mathbb{R}(W1, X \rightarrow A)$. The reason is that we assume $A \in X^{+}_{\mathbb{R},W1}$, i.e., $\mathbb{R}(W1, X \rightarrow A)$ follows from F by the inference rules A1 to A10. Since the inference rules are sound, any relation set satisfying F satisfies $\mathbb{R}(W1, X \rightarrow A)$. But the only way $\mathbb{R}(W1, X \rightarrow A)$ could hold over $\{R1, R2\}$ is if A is in $closure1$, for if not, then the attribute values of $R1$ and $R2$ which satisfy $W1 \cup X$, would disagree on A and violate $\mathbb{R}(W1, X \rightarrow A)$. □

Then we can prove Theorem 6.2, the correctness of Algorithm CLOSURE, below.

In the algorithm, for any $x_1 \in dom(X_1), \ldots, x_n \in dom(X_n)$, such that $\{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\} \sqsupseteq W[X_1, \ldots, X_n]$, let $W1 = W \cup \{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\}$. If $A \in closure$, then $A \in X^{+}_{\mathbb{R},W1}$, i.e., $\mathbb{R}(W1, X \rightarrow A)$ holds for each $W1$. So $\mathbb{R}(W, X \rightarrow A)$ can be deduced by the inference rules. That is $A \in X^{+}_{\mathbb{R},W}$.

On the other hand, if $A \in X^{+}_{\mathbb{R},W}$, then $A \in X^{+}_{\mathbb{R},W1}$ for each $W1$. According to Lemma A.4, A will be returned by each iteration of the inner loop (Lines 6 to 10).

So A will be in the set *closure* returned by the algorithm.

A.6 Proof of Theorem 6.3

We first give a lemma below.

Lemma A.5. *Given a discrepant schema transformation T consisting of unite and fold operations, for each simple qualified functional dependency f on the original relations of T, we can infer an equivalent qualified functional dependency on the target transformed relations using Algorithm NAIVE_PROPAGATE.*

Proof. This can be concluded from Lemmas 6.2 and 6.3. □

Then we prove Theorem 6.3. Algorithm NAIVE_PROPAGATE is sound because each qualified functional dependency computed by it holds over $\mathbb{R}_k$. This can be concluded from the soundness of the inference rules and propagation rules. We prove the completeness below.

Given a discrepant schema transformation T consisting of k steps, let $\mathbb{R}_0$ and $\mathbb{R}_k$ be the original and target transformed relations of T, and $\mathbb{R}_1, \ldots, \mathbb{R}_{k-1}$ be the intermediate transformed relations. Let F_0 be the set of the qualified functional dependencies on $\mathbb{R}_0$, and F_k be the set of the qualified functional dependencies on $\mathbb{R}_k$ derived using NAIVE_PROPAGATE. We can prove the completeness of Algorithm NAIVE_PROPAGATE by showing that if a simple qualified functional dependency $f \notin F_k^+$, then there must be an instance r_0 of $\mathbb{R}_0$ in which the dependencies of F_0 all hold, but in the target transformed relations r_k (an instance of $\mathbb{R}_k$), f doest not hold. That is, F_0 does not logically imply f.

Without loss of generality, we assume the given lossless transformation T is non-redundant, which is implemented in two phases: (1) transform the original relation set $\mathbb{R}_0$ into some $\mathbb{R}_i$ for some $0 \leq i \leq k$ using unite and fold operations,

such that the relation names and attribute names in $\mathbb{R}_0$ become attribute values in $\mathbb{R}_i$; (2) transform $\mathbb{R}_i$ into $\mathbb{R}_k$ using unfold and split operations.

Suppose a simple qualified functional dependency f for $\mathbb{R}_k$ cannot be derived from F_0 using Algorithm NAIVE_PROPAGATE. According to Lemma A.5, we can infer an equivalent qualified functional dependency g of f for $\mathbb{R}_i$ (Note that $\mathbb{R}_i$ can be transformed from $\mathbb{R}_k$ using unite and fold operations as T is a lossless transformation). We can also infer a set of qualified functional dependencies F_i for $\mathbb{R}_i$ which are equivalent to F_0 according to Lemma A.5. We claim F_i does not imply g. This can be proven by contradiction. Suppose F_i implies g. As the inference rules A1 to A10 are complete, we can infer g from F_i. As we inferred F_i from F_0, and f from g, we can infer f from F_0 by Algorithm NAIVE_PROPAGATE. This contradicts our assumption.

Consequently, there exists an instance r_i of $\mathbb{R}_i$, such that the qualified functional dependencies in F_i all hold but g does not. Let r_0 and r_k be the original and target transformed relations of r_i. Then F_0 holds over r_0 as F_i holds over r_i. However, f does not hold over r_k as g does not hold over r_i.

A.7 Quick propagation rules and Algorithm EFFICIENT_PROPAGATE

We present the quick propagation rules for each operator as an algorithm below. To simplify the presentation, in the following algorithms, we assume each input qualified functional dependency has a single attribute on the right. Although we require original qualified functional dependencies to be simple and therefore only contain regular attributes, the input qualified functional dependencies of a quick propagation rule may have qualification attributes, as a simple qualified functional

dependency may be changed into one with qualification attributes during the unite and fold operations in a schema operations in a schema transformation.

Algorithm INFER_SPLIT: *Quick propagation of qualified functional dependencies for a split operation.*

INPUT: Let $R(A_1, \ldots, A_n, B)$ be an original relation such that $dom(B) = \{b_1, \ldots, b_m\}$, and $b_i(A_1, \ldots, A_n)$ for each $i = 1, \ldots, m$ be the transformed relations using split(R, B). Let F be a set (not necessary a closure) of qualified functional dependencies on R.

OUTPUT: a set of qualified functional dependencies, G, holding over the set of the transformed relations $\{b_1, \ldots, b_m\}$.

METHOD: Let $\overline{X}$ and $\overline{Y}$ be 2 mixed sets of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $A \in \{A_1, \ldots, A_n\}$. We compute the qualified functional dependencies in G using the following rules:

1. If $R(\overline{X} \rightarrow A) \in F$, then $dom(B)(\overline{X} \rightarrow A) \in G$.
2. If $R(\overline{X}, B \rightarrow A) \in F$, then $b_i(\overline{X} \rightarrow A) \in G$ for each $i = 1, \ldots, m$.
3. If $R(\overline{X}, B_{\sigma=\{b_i\}} \rightarrow A) \in F$ for some $1 \leq i \leq m$, then $b_i(\overline{X} \rightarrow A) \in G$.
4. If $R(\overline{X} \rightarrow B) \in F$ and $R(\overline{Y}, B \rightarrow A) \in F$, then $dom(B)(\overline{X}, \overline{Y} \rightarrow A) \in G$.

□

The four rules are derived from the inference rules A1 to A10 and the propagation rule P1. Note that for Rule 4, although $R(\overline{X} \rightarrow B)$ cannot be changed to any dependency in G, it and $R(\overline{Y}, B \rightarrow A)$ together imply $R(\overline{X}, \overline{Y} \rightarrow A)$ which can be changed to $dom(B)(\overline{X}, \overline{Y} \rightarrow A)$ in G.

Algorithm INFER_UNITE: *Quick propagation of qualified functional dependencies for a unite operation.*

INPUT: Let $b_i(A_1, \ldots, A_n)$ for each $i = 1, \ldots, m$ be original relations such that $dom(B) = \{b_1, \ldots, b_m\}$ for an attribute B, and $R(A_1, \ldots, A_n, B)$ be the transformed relation using unite($\{b_1, \ldots, b_m\}, B$). Let F be a set (not necessary a closure) of qualified functional dependencies holding over the set of the original relations $\{b_1, \ldots, b_m\}$.

OUTPUT: a set of qualified functional dependencies G holding in the transformed relation R.

METHOD: Let $\overline{X}$ be a mixed set of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $A \in \{A_1, \ldots, A_n\}$. We compute the qualified functional dependencies in G using the following rules:

1. If $dom(B)(\overline{X} \rightarrow A) \in F$, then $R(\overline{X} \rightarrow A) \in G$.

2. If $\mathbb{R}(\overline{X} \rightarrow A) \in F$ for a set of relation names $\mathbb{R} \subset \{b_1, \ldots, b_m\}$, then $R(\overline{X}, B_{\sigma=\mathbb{R}} \rightarrow A) \in G$. $\square$

Algorithm INFER_UNFOLD: *Quick propagation of qualified functional qualified functional dependencies for a set of unfold operations.*

INPUT: Let $R_i(A_1, \ldots, A_n, B, C)$ for each $i = 1, \ldots, l$ be original relations, and $S_i(A_1, \ldots, A_n, b_1, \ldots, b_m)$ for each $i = 1, \ldots, l$ be the transformed relations using unfold(R_i, B, C). Let F be a set (not necessary a closure) of qualified functional dependencies holding over the set of the original relations $\{R_1, \ldots, R_l\}$.

OUTPUT: a set of qualified functional dependencies G holding over the set of the transformed relations $\{S_1, \ldots, S_l\}$.

METHOD: Let $\overline{X}$, $\overline{Y}$ and $\overline{Z}$ be mixed sets of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $A \in \{A_1, \ldots, A_n\}$. Let $\mathbb{R}1$, $\mathbb{R}2$, $\mathbb{R}3 \subseteq \{R_1, \ldots, R_l\}$ and $\mathbb{S}1$, $\mathbb{S}2$, $\mathbb{S}3 \subseteq \{S_1, \ldots, S_l\}$, such that the relations of $\mathbb{S}1$ ($\mathbb{S}2$ or $\mathbb{S}3$) are transformed from

the relations of $\mathbb{R}1$ ($\mathbb{R}2$ or $\mathbb{R}3$). We compute the qualified functional dependencies in G using the following rules:

1. If $\mathbb{R}1(\overline{X}, B_{\sigma=\{b_i\}} \rightarrow C) \in F$, then $\mathbb{S}1(\overline{X} \rightarrow b_i) \in G$.
2. If $\mathbb{R}1(\overline{X}, B \rightarrow C) \in F$ or $\mathbb{R}1(\overline{X} \rightarrow C) \in F$, then $\mathbb{S}1(\overline{X} \rightarrow b_i) \in G$ for each $b_i \in dom(B)$.
3. If $\mathbb{R}1(\overline{X}, B_{\sigma=\{b_i\}}, C \rightarrow A) \in F$, then $\mathbb{S}1(\overline{X}, b_i \rightarrow A) \in G$.
4. If $\mathbb{R}1(\overline{X}, B, C \rightarrow A) \in F$ or $\mathbb{R}1(\overline{X}, C \rightarrow A) \in F$, then $\mathbb{S}1(\overline{X}, b_i \rightarrow A) \in G$ for each $b_i \in dom(B)$.
5. If $\mathbb{R}1(\overline{X} \rightarrow A) \in F$, then $\mathbb{S}1(\overline{X} \rightarrow A) \in G$.
6. If $\mathbb{R}1(\overline{X} \rightarrow B) \in F$ and $\mathbb{R}2(\overline{Y}, B \rightarrow A) \in F$, then $\mathbb{S}1 \cap \mathbb{S}2(\overline{X}, \overline{Y} \rightarrow A) \in G$.
7. If $\mathbb{R}1(\overline{X} \rightarrow C) \in F$ and $\mathbb{R}2(\overline{Y}, C \rightarrow A) \in F$, then $\mathbb{S}1 \cap \mathbb{S}2(\overline{X}, \overline{Y} \rightarrow A) \in G$.
8. If $\mathbb{R}1(\overline{X} \rightarrow B) \in F$, $\mathbb{R}2(\overline{Y}, B \rightarrow C) \in F$ and $\mathbb{R}3(\overline{Z}, C \rightarrow A) \in F$, then $\mathbb{S}1 \cap \mathbb{S}2 \cap \mathbb{S}3(\overline{X}, \overline{Y}, \overline{Z} \rightarrow A) \in G$.
9. If $\mathbb{R}1(\overline{X} \rightarrow C) \in F$, $\mathbb{R}2(\overline{Y}, C \rightarrow B) \in F$ and $\mathbb{R}3(\overline{Z}, B \rightarrow A) \in F$, then $\mathbb{S}1 \cap \mathbb{S}2 \cap \mathbb{S}3(\overline{X}, \overline{Y}, \overline{Z} \rightarrow A) \in G$. $\square$

Algorithm INFER_FOLD: *Quick propagation of qualified functional qualified functional dependencies for a set of fold operations.*

INPUT: Let $R_i(A_1, \ldots, A_n, b_1, \ldots, b_m)$ for each $i = 1, \ldots, l$ be original relations, and $S_i(A_1, \ldots, A_n, B, C)$ for each $i = 1, \ldots, l$ be the transformed relations using fold(R_i, B, C). Let F be the set (not necessary a closure) of qualified functional dependencies holding over the set of the original relations $\{R_1, \ldots, R_l\}$.

OUTPUT: a set of qualified functional dependencies G holding over the set of the transformed relations $\{S_1, \ldots, S_l\}$.

METHOD: Let $\overline{X}$ be a mixed set of regular and qualification attributes from $\{A_1, \ldots, A_n\}$, and $A \in \{A_1, \ldots, A_n\}$. Let $\mathbb{R} \subseteq \{R_1, \ldots, R_l\}$ and $\mathbb{S} \subseteq \{S_1, \ldots, S_l\}$, such that the relations of $\mathbb{S}$ are transformed from the relations of $\mathbb{R}$. We compute the qualified functional dependencies in G using the following rules:

1. If $\mathbb{R}(\overline{X} \rightarrow b_i) \in F$, then $\mathbb{S}(\overline{X}, B_{\sigma=\{b_i\}} \rightarrow C) \in G$.

2. If $\mathbb{R}(\overline{X}, b_i \rightarrow A) \in F$, then $\mathbb{S}(\overline{X}, B_{\sigma=\{b_i\}}, C \rightarrow A) \in G$.

3. If $\mathbb{R}(\overline{X} \rightarrow A) \in F$, then $\mathbb{S}(\overline{X} \rightarrow A) \in G$. $\square$

Algorithm EFFICIENT_PROPAGATE

INPUT: Let $T =< T_1, \ldots, T_n >$ be a discrepant schema transformation for each T_i a (or a set of) restructuring operation(s). Let F_0 be the set (not necessary a closure) of qualified functional dependencies holding over the set of the original relations of T.

OUTPUT: A set of qualified functional dependencies F_n holding over the set of the transformed relations of T.

METHOD:

for i from 0 to $n-1$ **do**

Let $\mathbb{R}_i$ be the set of original (input) relations of the transformation step T_{i+1}, and F_i be the set of qualified functional dependencies holding over $\mathbb{R}_i$.

case 1 T_{i+1} is a split operation:

call Algorithm INFER_SPLIT to get F_{i+1}

end case

case 2 T_{i+1} is a unite operation:

call Algorithm INFER_UNITE to get F_{i+1}

end case

case 3 T_{i+1} is a (or a set of) unfold operation(s):

call Algorithm INFER_UNFOLD to get F_{i+1}

end case

case 4 T_{i+1} is a (or a set of) fold operation(s):

call Algorithm INFER_FOLD to get F_{i+1}

end case

end for

A.8 Proof of Theorem 6.4

During a discrepant schema transformation, a simple qualified functional dependency may become a qualified functional dependency with qualification attributes. The following lemma describes the form of the qualified functional dependencies derived during the execution of Algorithm EFFICIENT_PROPAGATE.

Lemma A.6. *Given a lossless transformation and a set of simple qualified functional dependencies for the original relations of the transformation, in execution of Algorithm EFFICIENT_PROPAGATE, the set of qualified functional dependencies F produced by an inference algorithm (INFER_SPLIT, INFER_UNITE, INFER_UNFOLD or INFER_FOLD) satisfies 4 conditions:*

1. *The qualification attributes of the qualified functional dependencies in F are restricted to take single values.*

2. *Let $Z1$ be the set of all the regular attributes of the qualified functional dependencies in F, and $Z2$ be the set of all the attributes occurring in the*

qualification attributes of the qualified functional dependencies in F. Then $Z1 \cap Z2 = \emptyset$.

3. *Condition (1) of Definition 6.3.*

4. *Condition (3) of Definition 6.3.*

Proof. The lemma can be proven by induction on the transformation steps. □

Given a set of qualified functional dependencies F, if it implies a qualified functional dependency f, then there is a subset of the qualified functional dependencies of F from which we can infer f by the inference rules A1 to A10. Definitions A.3, A.4 and Lemma A.7 give a way to infer f from F.

Definition A.3. *Given a set of qualified functional dependencies F, a* derivation sequence *s for a qualified functional dependency $\mathbb{R}(\overline{X} \rightarrow A)$ is a sequence of qualified functional dependencies from F:*

$$\mathbb{R}_1(\overline{Y_1} \rightarrow A_1), \ldots, \mathbb{R}_m(\overline{Y_m} \rightarrow A_m)$$

such that $\mathbb{R} \subseteq \mathbb{R}_i, \overline{X} \sqsupseteq \overline{Y_1}, A_m = A$, and $\overline{X} \cup \{A_1, \ldots, A_{i-1}\} \sqsupseteq \overline{Y_i}$ for $i = 1, \ldots, m$. We say s lies in a set of attributes *U iff the attributes occurring in the qualified functional dependencies of s are all from U.* □

Definition A.4. *Given a set of qualified functional dependencies F and a qualified functional dependency $\mathbb{R}(W, X \rightarrow A)$ with the set of qualification attributes W and the set of regular attributes $X = \{X_1, \ldots, X_n\}$. A* derivation sequence set *S for $\mathbb{R}(W, X \rightarrow A)$ is a set of derivation sequences from F:*

$$S = \{s | s \text{ is a derivation sequence for } \mathbb{R}(W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}} \rightarrow A) \text{ for any } x_1 \in dom(X_1), \ldots, x_n \in dom(X_n) \text{ such that } \{X_{1\,\sigma=\{x_1\}}, \ldots, X_{n\,\sigma=\{x_n\}}\} \sqsupseteq W[X_1, \ldots, X_n]\}.$$ □

The following example explains the concept of derivation sequence set.

Example A.1. *Given a relation $R(A, B, C)$, let F be a set of qualified functional dependencies consisting of qualified functional dependencies $R(A_{\sigma=\{a\}} \rightarrow B)$ for each $a \in dom(A)$ and a functional dependency $B \rightarrow C$ on R. Let $f : A \rightarrow C$ be a functional dependency on R. Our purpose is to verify whether f follows from F. Instead of computing F^+, we try to find a derivation sequence set S for f:*

$$\{R(A_{\sigma=\{a\}} \rightarrow B), R(B \rightarrow C) |\ a \in dom(A)\}.$$

Note that the dependencies in the derivation sequences are from F, and each sequence in S contributes to a "component" of f: $R(A_{\sigma=\{a\}} \rightarrow C)$. Consequently, we know F implies f. The general result is presented in the following lemma. □

Lemma A.7. *Given a set of qualified functional dependencies F satisfying the four conditions of Lemma A.6 and a qualified functional dependency f, $f \in F^+$ iff f is a trivial dependency or there exists a derivation sequence set S for f.*

Proof. Let $f = \mathbb{R}(W, X \rightarrow A)$ be a non-trivial dependency. If there exists a derivation sequence set S for f, using the inference rules A1 to A10, we can derive f from the qualified functional dependencies of the derivation sequence set, i.e., $f \in F^+$.

On the other hand, if $f \in F^+$, $A \in X^+_{\mathbb{R},W}$. The computation of $X^+_{\mathbb{R},W}$ with Algorithm CLOSURE is actually a process to construct a derivation sequence set for f. □

We are ready to prove the completeness of Algorithm EFFICIENT_PROPAGATE now. We will prove this by showing that Algorithm EFFICIENT_PROPAGATE generates the same result as NAIVE_PROPAGATE. In what follows, we first prove

the completeness of the four inference algorithms INFER_SPLIT, INFER_UNITE, INFER_UNFOLD and INFER_FOLD.

Lemma A.8. *The algorithms INFER_SPLIT, INFER_UNITE, INFER_UNFOLD and INFER_FOLD are complete if the input qualified functional dependencies satisfy the 4 conditions in Lemma A.6.* □

Proof. We only prove the completeness of Algorithm INFER_SPLIT. The proofs for the other inference algorithms are similar and omitted. Let $R(A_1, \ldots, A_n, B)$ be an original relation, and $b_i(A_1, \ldots, A_n)$ for each $i = 1, \ldots, m$ be the transformed relations using split(R, B). Let F be a set (not necessary a closure) of qualified functional dependencies in R. Let G and G' be the sets of the qualified functional dependencies computed by Algorithm NAIVE_PROPAGATE and INFER_SPLIT respectively. We will prove G is equivalent to G'. It is easy to see that $G' \subseteq G^+$ as the rules of Algorithm INFER_SPLIT are derived from the inference rules A1 to A10 and propagation rules P1 to P4. We will prove $G \subseteq G'^+$ as follows.

For any $g = \mathbb{R}(W, X \rightarrow A) \in G$, where $\mathbb{R} \subseteq \{b_1, \ldots, b_m\}$, W is a set of qualification attributes and $X = \{X_1, \ldots, X_l\}$ a set of regular attributes from $\{A_1, \ldots, A_n\}$, and $A \in \{A_1, \ldots, A_n\}$, let $f = R(B_{\sigma=\mathbb{R}}, W, X \rightarrow A) \in F^+$ be the original qualified functional dependency of g. According to Lemma A.7, there's a derivation sequence set S from F for f. Let $s : R(Y_1 \rightarrow C_1), \ldots, R(Y_k \rightarrow C_k)$, be a derivation sequence in S for the qualified functional dependency

$$R(B_{\sigma=\mathbb{R}}, W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}} \rightarrow A)$$

for $x_1 \in dom(X_1), \ldots, x_l \in dom(X_l)$ satisfying $\{X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}}\} \sqsupseteq W[X_1, \ldots, X_l]$. Then we will construct a derivation sequence s' with the dependencies from G' for

$$\mathbb{R}(W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}} \rightarrow A)$$

by transforming the qualified functional dependencies of s.

For each $i = 1, \ldots, k$, let s_i and s'_i be the i-th qualified functional dependency in s and s' resp., we consider 3 cases of $s_i = R(\overline{Y_i} \rightarrow C_i)$:

Case 1: s_i lies in $\{A_1, \ldots, A_n\}$. Then we can derive $s'_i = dom(B)(\overline{Y_i} \rightarrow C_i)$ from s_i using Rule (1) of INFER_SPLIT.

Case 2: s_i contains either $B_{\sigma=\{b\}}$ for some $b \in dom(B)$, or B in its Left Hand Side (LHS), but all s_j with $1 \leq j < i$ does not contain B in their Right Hand Side (RHS). In this case, f must have a form of $R(B_{\sigma=\{b\}}, W, X \rightarrow A)$ (and $g = b(W, X \rightarrow A)$), as otherwise s_i would not be in s for $\{W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}}\} \cup \{C_1, \ldots, C_{i-1}\}$ does not satisfy $\overline{Y_i}$. Then we derive $s'_i = b(\overline{Y'_i} \rightarrow C_i)$ from s_i for $\overline{Y'_i} = \overline{Y_i} - \{B_{\sigma=\{b\}}\}$ or $\overline{Y'_i} = \overline{Y_i} - \{B\}$, using Rule (3) or (2) of INFER_SPLIT.

Case 3: s_i contains B in its RHS, i.e., $R(\overline{Y_i} \rightarrow B)$. Without loss of generality, we assume s contains exactly one such qualified functional dependency (it would be no need to derive B more than once). We derive the dependencies in s' from those in s by applying Rule (4) of Algorithm INFER_SPLIT: for each j, $i < j \leq k$, if $s_j = R(\overline{Y_j} \rightarrow C_j)$ contains B in its LHS, then derive $s'_j = dom(B)(\overline{Y_i}, \overline{Y'_j} \rightarrow C_j)$ for $\overline{Y'_j} = \overline{Y_j} - \{B\}$. Note that in this case, s_i itself would not be changed into any dependency in s'.

Clearly, s' is a derivation sequence from G' for $\mathbb{R}(W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}} \rightarrow A)$. It follows that $\mathbb{R}(W, X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}} \rightarrow A) \in G'^{+}$. As this is true for any $x_1 \in dom(X_1), \ldots, x_n \in dom(X_n)$ satisfying $\{X_{1\,\sigma=\{x_1\}}, \ldots, X_{l\,\sigma=\{x_l\}}\} \sqsupseteq W[X_1, \ldots, X_l]$, we have $\mathbb{R}(W, X \rightarrow A) \in G'^{+}$. That is, $\forall g \in G, g \in G'^{+}$. So $G \subseteq G'^{+}$. As Algorithm NAIVE_PROPAGATE is complete (Theorem 6.3), so is INFER_SPLIT. □

As Algorithm EFFICIENT_PROPAGATE is actually a sequence of calls of the four inference algorithms, from Lemmas A.6 and A.8, we know that Algorithm

EFFICIENT_PROPAGATE is complete. The soundness of the algorithm can be concluded from the soundness of the quick propagation rules. Finally, we proved that Theorem 6.4 is true. That is, Algorithm EFFICIENT_PROPAGATE is sound and complete to infer simple qualified functional dependencies in a lossless discrepant schema transformation.

A.9 Resolution algorithms of schematic discrepancies in the integration of ER schemas

Algorithm ResolveRel

Given an ER schema DB, the algorithm produces a schema DB' transformed from DB such that all the discrepant meta-attributes of relationship types are transformed into the attributes of entity types.

Step 1 *Resolve the discrepant meta-attributes of a relationship type.*

Let $R = T[C_1 = c_1, \ldots, C_m = c_m]$ be a relationship type among entity types $E_{m+1}, E_{m+2}, \ldots, E_n$ in DB, where T is a relationship type among n entity types $T_1, \ldots, T_n$ in the ontology, $C_1, \ldots, C_m$ are m discrepant meta-attributes that are identifiers of $T_1, \ldots, T_m$, and each $E_i = T_i$ for $i = m+1, \ldots, n$ has an identifier $K_i = C_i$. Let T' (a projection of T) be a relationship type among the entity types $T_{m+1}, \ldots, T_n$ in the ontology.

/ Note that the inherited context of R has been removed in Algorithm ResolveEnt if any.*/*

Step 1.1 *Transform $C_1, \ldots, C_m$ into attributes of entity types.*

Construct m entity types $E_1 = T_1, \ldots, E_m = T_m$ with the identifiers $K_1 = C_1, \ldots, K_m = C_m$ if they do not exist.

Each E_i $(i = 1, \ldots, m)$ contains one entity with the identifier $C_i = c_i$.

Construct a relationship type $R' = T$ connecting $E_1, \ldots, E_n$, such that $(c_1, \ldots, c_m, c_{m+1}, \ldots, c_n) \in R'[K_1, \ldots, K_m, K_{m+1}, \ldots, K_n]$ iff $(c_{m+1}, \ldots, c_n) \in R[K_{m+1}, \ldots, K_n]$.

Let $\mathbb{A} \rightarrow \mathbb{B}$ be a functional dependency on R, where $\mathbb{A}$ and $\mathbb{B}$ are two sets of the identifiers of some participating entity types in R. Represent a functional dependency $\mathbb{A}, K_1, \ldots, K_m \rightarrow \mathbb{B}$ in R'.

end Step

Step 1.2 *Handle the attributes of R.*

Let A be an attribute of R. A corresponds to A_{ont} in the ontology, and has a self context $selfCnt$.

if A is a many-to-one or many-to-many attribute **then**

case 1 attribute A does not inherit any context of R:

A becomes an attribute of a new relationship type $R'' = T'$ among $E_{m+1}, \ldots, E_n$, such that

$(c_{m+1}, \ldots, c_n, a) \in R''[K_{m+1}, \ldots, K_n, A]$ iff

$(c_{m+1}, \ldots, c_n, a) \in R[K_{m+1}, \ldots, K_n, A]$.

end case

case 2 attribute $A = A_{ont}[selfCnt, inherit\ all]$ inherits all the context $\{C_1 = c_1, \ldots, C_m = c_m\}$ from R:

Construct an attribute $A' = A_{ont}[selfCnt]$ of R', such that

$(c_1, \ldots, c_m, c_{m+1}, \ldots, c_n, a) \in R'[K_1, \ldots, K_m, K_{m+1}, \ldots, K_n, A']$ iff

$(c_{m+1}, \ldots, c_n, a) \in R[K_{m+1}, \ldots, K_n, A]$.

A' has the same cardinality as A.

end case

case 3 A inherits some context, say $\{C_1 = c_1, \ldots, C_j = c_j\}$ $(1 \leq j < m)$ from R:

Construct a relationship type R'' connecting the entity types $E_1, \ldots, E_j$, $E_{m+1}, \ldots, E_n$.

Construct an attribute $A' = A_{ont}[selfCnt]$ of R'', such that

$(c_1, \ldots, c_j, c_{m+1}, \ldots, c_n, a) \in R''[K_1, \ldots, K_j, K_{m+1}, \ldots, K_n, A']$ iff

$(c_{m+1}, \ldots, c_n, a) \in R[K_{m+1}, \ldots, K_n, A]$.

A' has the same cardinality as A.

end case

else

/ A is a one-to-one or one-to-many attribute, i.e., A determines the identifier of R in the context. We keep the inherited context of A, and delay the resolution of it in Algorithm ResolveRelAttr, in which A will be transformed to the identifier of an entity type to preserve the cardinality constraint. */*

Construct an attribute $A' = A_{ont}[Cnt]$ of the relationship type $R'' = T'$, where Cnt is the self context of A' that is the union of the self and inherited contexts of A, such that

$(c_{m+1}, \ldots, c_n, a) \in R''[K_{m+1}, \ldots, K_n, A']$ iff

$(c_{m+1}, \ldots, c_n, a) \in R[K_{m+1}, \ldots, K_n, A]$.

end if

end Step

end Step

Step 2 *Merge equivalent constructs.*

Merge equivalent entity types, relationship types and attributes respectively. Their domains are united.

end Step □

Algorithm ResolveEntAttr

Given an ER schema DB, the algorithm produces a schema DB' transformed from DB such that all the discrepant meta-attributes of the attributes of the entity types in DB are transformed into the attributes of entity types.

Step 1 *Resolve the discrepant meta-attributes of an attribute of an entity type.*

Given an entity type $E = T$ (with the identifier K) of DB, let $A = A_{ont}[C_1 = c_1, \ldots, C_m = c_m]$ be an attribute of E, for A_{ont} an attribute of a relationship type T_R, and $C_1, \ldots, C_m$ the discrepant meta-attributes that are identifers of entity types $T_1, \ldots, T_m$ in the ontology. T_R is a relationship type among $T_1, \ldots, T_m$ and T in the ontology.

/* *Note that the inherited context of A has been removed in Algorithm ResolveEnt if any.*/

Construct an entity type $E_i = T_i$ with the identifier $K_i = C_i$ for each $i = 1, \ldots, m$ if they do not exist. Each E_i contains one entity with the identifier $C_i = c_i$.

if A is a many-to-one or many-to-many attribute **then**

Construct a relationship type $R = T_R$ connecting the entity types $E_1, \ldots, E_m$ and E.

Attribute $A' = A_{ont}$ becomes an attribute of R, such that

$(c_1, \ldots, c_m, k, a) \in R[K_1, \ldots, K_m, K, A']$ iff $(k, a) \in E[K, A]$.

else

/*A *is a one-to-one or one-to-many attribute, i.e., A and the meta-attributes* $C_1, \ldots, C_m$ *together determine the identifier* K *of* E. *A should be modelled as the identifier of an entity type to preserve the cardinality constraint.*/

Construct $E_{A'}$ with the identifier $A' = A_{ont}$.

Construct a relationship type R' connecting the entity types $E_1, \ldots, E_m$, E and $E_{A'}$, such that

$(c_1, \ldots, c_m, k, a) \in R[K_1, \ldots, K_m, K, A']$ iff $(k, a) \in E[K, A]$.

Represent a functional dependency $K_1, \ldots, K_m, A' \rightarrow K$ as the cardinality constraint on R.

if A is a one-to-one attribute **then**

Represent a functional dependency $K_1, \ldots, K_m, K \rightarrow A'$ on R.

end if

end if

end Step

Step 2 *Merge equivalent constructs.*

Merge equivalent entity types, relationship types and attributes respectively. Their domains are united.

end Step □

Algorithm ResolveRelAttr

Given an ER schema DB, the algorithm produces a schema DB' transformed from DB such that all the discrepant meta-attributes of the attributes of the relationship types in DB are transformed into the attributes of entity types.

Step 1 *Resolve the discrepant meta-attributes of an attribute of a relationship type.*

In DB, let R (with the identifier K_R) be a relationship type among m entity types $E_1 = T_1, \ldots, E_m = T_m$ with the identifiers $K_1 = C_1, \ldots, K_m = C_m$, and let $A = A_{ont}[C_{m+1} = c_{m+1}, \ldots, C_n = c_n]$ be an attribute of R, where $C_{m+1}, \ldots, C_n$ are discrepant meta-attributes that are identifers of entity types $T_{m+1}, \ldots, T_n$, and A_{ont} is an attribute of a relationship type T among n entity

types $T_1, \ldots, T_n$ in the ontology.

/ Note that the inherited context of A has been removed in Algorithm ResolveRel if any.*/*

Construct an entity types $E_i = T_i$ with an identifier $K_i = C_i$ for each $i = m+1, \ldots, n$ if it does not exist. Each E_i contains one entity with the identifier c_i.

if A is a many-to-one or many-to-many attribute **then**

Construct a relationship type $R' = T$ connecting the entity types $E_1, \ldots, E_n$.

Attribute $A' = A_{ont}$ becomes an attribute of R', such that

$(c_1, \ldots, c_m, c_{m+1}, \ldots, c_n, a) \in R'[K_1, \ldots, K_m, K_{m+1}, \ldots, K_n, A']$ iff

$(c_1, \ldots, c_m, a) \in R[K_1, \ldots, K_m, A]$.

else

*/*A is a one-to-one or one-to-many attribute, i.e., A and the meta-attributes $C_{m+1}, \ldots, C_n$ together determine the identifier of R. A should be modelled as the identifier of an entity type to preserve the cardinality constraint.*/*

Construct $E_{A'}$ with the identifier $A' = A_{ont}$.

Construct a relationship type R' connecting the entity types $E_1, \ldots, E_n$ and $E_{A'}$, such that

$(c_1, \ldots, c_m, c_{m+1}, \ldots, c_n, a) \in R'[K_1, \ldots, K_m, K_{m+1}, \ldots, K_n, A']$ iff

$(c_1, \ldots, c_m, a) \in R[K_1, \ldots, K_m, A]$.

Represent a functional dependency $K_{m+1}, \ldots, K_n, A' \rightarrow K_R$ as the cardinality constraint on R'.

if A is a one-to-one attribute **then**

Represent a functional dependency $K_{m+1}, \ldots, K_n, K_R \rightarrow A'$ on R'.

end if

end if

end Step

Step 2 *Merge equivalent constructs.*

Merge equivalent entity types, relationship types and attributes respectively. Their domains are united.

end Step □

A.10 Proof of Theorem 7.2

We prove Theorem 7.2 through 5 lemmas in the rest of this section. In ER schemas, cardinality constraints (in particular, the cardinalities of the attributes of entity types, the cardinalities of the entity types in a relationship type, or the cardinalities of the attributes of relationship types) may represent functional dependencies/multivalued dependencies, as mentioned in Section 2.1. Lemma A.9 is on the preservation of the functional dependencies and multivalued dependencies represented as the cardinality constraints of the attributes of entity types. Lemmas A.10 and A.11 are on the preservation of the functional dependencies represented as the cardinality constraints of the entity types in a relationship type. Lemmas A.12 and A.13 are on the preservation of the functional dependencies and multivalued dependencies represented as the cardinality constraints of the attributes of a relationship type.

We first show an example of the preservation of the functional dependencies that are the cardinality constraints of the attributes of entity types below.

Example A.2. *In Figure 7.2, in each of the entity types* $JAN\_PROD, \ldots, DEC\_PROD$ *of DB2, the attributes* $S1\_PRICE, \ldots, Sn\_PRICE$ *inherit all the context of the entity type. After Algorithm ResolveEnt, the discrepant meta-attribute m# becomes the attribute M# of the entity type MONTH in the transformed schema, and*

$S1\_PRICE, \ldots, Sn\_PRICE$ become the attributes of the relationship type PM. We have the following result:

A functional dependency $P\# \rightarrow \{S1\_PRICE, \ldots, Sn\_PRICE\}$ holds in each entity type of DB2 iff a functional dependency $\{P\#, M\#\} \rightarrow \{S1\_PRICE, \ldots, Sn\_PRICE\}$ holds in the relationship type PM of the transformed schema.

On the other hand, in each entity type of DB2, the attribute $PNAME$ has nothing to do with the context of the entity type, i.e., a product name is only dependent on the product number, independent of the months in which the product is supplied. We have the following result:

A functional dependency $P\# \rightarrow PNAME$ holds in each entity type of DB2 iff the same functional dependency $P\# \rightarrow PNAME$ holds in the entity type $PROD$ of the transformed schema. □

In general, we have the following result:

Lemma A.9. *Algorithm ResolveEnt preserves the functional dependencies and multivalued dependencies represented as the cardinality constraints of the attributes of entity types.*

Proof. Recall Step 1.2 of Algorithm ResolveEnt in which we transform the attributes of entity types in the resolution of schematic discrepancies of entity types. We first claim that for each case of Step 1.2, the cardinality constraints of attributes can be preserved in the transformed schema, then prove a typical case. In Algorithm ResolveEnt, equivalent schema constructs will be merged in Step 2. Without losing generality, we consider a set of entity types (instead of individual ones) that correspond to the same ontology type, and have the same set of meta-attributes but different metadata, such as the entity types $JAN\_PROD, \ldots, DEC\_PROD$ of $DB2$ in the above example. Such entity types will be transformed to equivalent relationship types and merged in the algorithm.

In general, in an ER schema DB, let $\mathbb{E}$ be a set of entity types with the same identifier K, the same attribute A and the same meta-attributes, i.e.,

$$\mathbb{E} = \{E | E = T[C_1 = c_1, \ldots, C_l = c_l, inherit\ C_{l+1}, \ldots, C_m],$$
$$c_1 \in dom(C_1), \ldots, c_l \in dom(C_l)\}.$$

Let A correspond to an attribute A_{ont} in the ontology, and have a self context $selfCnt$, i.e., a set of meta-attributes with values.

Let DB' be the schema transformed from DB by Algorithm ResolveEnt, in which $C_1, \ldots, C_m$ become the identifiers $K_1, \ldots, K_m$ of entity types $E_1, \ldots, E_m$, and an entity type E_{m+1} with the identifier K is created to contain all the entities of the entity types of $\mathbb{E}$. We claim that:

case 1 A is a many-to-one or many-to-many attribute in each entity type of $\mathbb{E}$:

case 1.1 $A = A_{ont}[selfCnt]$ does not inherit any context from the entity types:

A becomes a non-identifier attribute of the entity type E_{m+1} in the transformed schema DB'.

if A is a many-to-one attribute **then**

A functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$ iff a functional dependency $K \rightarrow A$ holds in E_{m+1}.

else

*/*A is a many-to-many attribute.*/*

A multivalued dependency $K \twoheadrightarrow A$ holds in each entity type of $\mathbb{E}$ iff a multivalued dependency $K \twoheadrightarrow A$ holds in E_{m+1}.

end if

end case

case 1.2 $A = A_{ont}[selfCnt, inherit\ all]$ inherits all the context of the entity types:

$A' = A_{ont}[selfCnt]$ becomes an attribute of a new-created relationship type R among m+1 entity types $E_1, \ldots, E_{m+1}$ in DB'.

if A is a many-to-one attribute **then**

A functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$ iff a functional dependency $K_1, \ldots, K_m, K \rightarrow A'$ holds in R.

else

/*A *is a many-to-many attribute.*/

A multivalued dependency $K \twoheadrightarrow A$ holds in each entity type of $\mathbb{E}$ iff a multivalued dependency $K_1, \ldots, K_m, K \twoheadrightarrow A'$ holds in R.

end if

end case

case 1.3 $A = A_{ont}[selfCnt, inherit\ C_1, \ldots, C_j]$ $(1 \leq j < m)$ inherits some context of the entity types.

$A' = A_{ont}[selfCnt]$ becomes an attribute of a new-created relationship type R' among j+1 entity types $E_1, \ldots, E_j$ and E_{m+1} in DB'.

if A is a many-to-one attribute **then**

A functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$ iff a functional dependency $K_1, \ldots, K_j, K \rightarrow A'$ holds in R'.

else

/*A *is a many-to-many attribute.*/

A multivalued dependency $K \twoheadrightarrow A$ holds in each entity type of $\mathbb{E}$ iff a multivalued dependency $K_1, \ldots, K_j, K \twoheadrightarrow A'$ holds in R'.

end if

end case

end case

case 2 A is a one-to-one or one-to-many attribute:

$A' = A_{ont}[Cnt]$ becomes an attribute of E_{m+1} in DB', where Cnt is the self context of A' that is the union of the self context (i.e., $selfCnt$) and inherited context (say $inhrtCnt$) of A.

A functional dependency $A \rightarrow K$ holds in each entity type of $\mathbb{E}$ which has the context $inhrtCnt$ iff $A' \rightarrow K$ holds in E_{m+1}.

if A is a one-to-one attribute **then**

A functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$ which has the context $inhrtCnt$ iff a functional dependency $K \rightarrow A'$ holds in E_{m+1}.

else

A multivalued dependency $K \twoheadrightarrow A$ holds in each entity type of $\mathbb{E}$ which has the context $inhrtCnt$ iff a multivalued dependency $K \twoheadrightarrow A'$ holds in E_{m+1}.

end if

end case

Then we prove the above claim. We only prove the case when A is a many-to-one attribute that inherits all the context of the entity types, i.e, a sub-case of Case 1.2. The other cases can be proven in a similar way.

(=>) If a functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$, then a functional dependency $K_1, \ldots, K_m, K \rightarrow A'$ holds in R.

Suppose we are given two tuples $(c_1, \ldots, c_m, k, a), (c_1, \ldots, c_m, k, a') \in R[K_1, \ldots, K_m, K, A']$. As the two tuples have the same values on $K_1, \ldots, K_m$ (i.e., $C_1, \ldots, C_m$), they must come from the same entity type of $\mathbb{E}$. As a functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$, we have $a = a'$. Consequently, $K_1, \ldots, K_m, K \rightarrow A'$ holds in R.

(<=) If a functional dependency $K_1, \ldots, K_m, K \rightarrow A'$ holds in R, then a functional dependency $K \rightarrow A$ holds in each entity type of $\mathbb{E}$.

For each entity type $E = T[C_1 = c_1, \ldots, C_l = c_l, inherit\ C_{l+1}, \ldots, C_m]$ in $\mathbb{E}$,

given two tuples $(k, a), (k, a') \in E[K, A]$, by Algorithm ResolveEnt, we can transform them to two tuples $(c_1, \ldots, c_m, k, a), (c_1, \ldots, c_m, k, a') \in R[K_1, \ldots, K_m, K, A']$. As a functional dependency $K_1, \ldots, K_m, K \rightarrow A'$ holds in R, we have $a = a'$. Consequently, $K \rightarrow A$ holds in E. □

In the ER approach, functional dependencies/multivalued dependencies can be represented by not only the cardinalities of the attributes of entity types, but also the cardinalities of the entity types and the cardinalities of the attributes in a relationship type. In the rest of this section, we first show an example of functional dependencies represented as the cardinality constraints in relationship types, then present 4 lemmas to generalize the results.

Example A.3. *In the schema DB4 of Figure 7.3, a relationship type SUP_1 (the other relationship types are similar) inherits the context m#='JAN' from its participating entity type JAN_PROD. After Algorithm ResolveEnt, the discrepant meta-attribute m# becomes an attribute M# of the entity type MONTH in the transformed schema, and the relationship types SUP_i's are transformed and merged into a ternary relationship type SUP among the entity types PROD, MONTH and SUPPLIER. We have the following result:*

A functional dependency $P\# \rightarrow S\#$ holds in each relationship type of DB4 iff a functional dependency $\{P\#, M\#\} \rightarrow S\#$ holds in the relationship type SUP of the transformed schema.

In the schema of DB4, the relationship type SUP_1 (the other relationship types are similar) has an attribute PRICE with the inherited context $m\# = `JAN'$. After Algorithm ResolveEnt, PRICE becomes an attribute of the relationship type SUP in the transformed schema. We have the following result:

A functional dependency $P\# \rightarrow PRICE$ holds in each relationship type of DB4 iff a functional dependency

$\{P\#, M\#\} \rightarrow PRICE$ *holds in the relationship type* SUP *of the transformed schema.* □

Recall Step 1.3 of Algorithm ResolveEnt in which we transform relationship types in the resolution of schematic discrepancies for entity types. According to whether a relationship type has any one-to-one or one-to-many attributes, the transformation methods would be different. Correspondingly, when presenting the issue of functional dependency preservation, we also divide the two cases. In particular, Lemma A.10 and Lemma A.12 are, respectively, on the preservation of the cardinalities of entity types and the cardinalities of attributes when relationship types only have many-to-one and many-to-many attributes. On the other hand, Lemma A.11 and Lemma A.13 are, respectively, on the preservation of the cardinalities of entity types and the cardinalities of attributes when relationship types have some one-to-one or one-to-many attributes.

Lemma A.10. *When relationship types only have many-to-one and many-to-many attributes, Algorithm ResolveEnt preserves the functional dependencies represented as the cardinality constraints of the entity types in the relationship types.*

Proof. We first claim that for any of the three cases of Step 1.3 in Algorithm ResolveEnt, the cardinality constraints of the entity types in a relationship type can be preserved in the transformed schema. Note a relationship type may involve several entity types with discrepant meta-attributes all of which need to be resolved. For ease of presentation, we will not distinguish the three cases as in Step 1.3 of the algorithm, but rather generalize the cases. In particular, we give the general form of a functional dependency on a relationship type of a transformed schema and the corresponding functional dependencies in the original schema, and show that the functional dependency of the transformed schema and the functional dependencies

of the original schema are equivalent to each other. In Step 2 of Algorithm ResolveEnt, equivalent schema constructs will be merged. Without losing generality, we consider a set of relationship types that correspond to the same ontology type, and have the same self context but not necessary the same inherited context. Such relationship types will be transformed to equivalent relationship types and merged in Algorithm ResolveEnt.

In general, in an ER schema DB, let $\mathbb{R} = \{R_1, \ldots, R_n\}$ be a set of relationship types corresponding to the same ontology type within the same self context, such that each relationship type has no attributes, or only has many-to-one and many-to-many attributes.

Let DB' be the schema transformed from DB by Algorithm ResolveEnt, in which all the relationship types of $\mathbb{R}$ are transformed and merged into a relationship type R'. We claim that:

A functional dependency $K_1, \ldots, K_m \rightarrow K_{m+1}$ holds in R' for $K_1, \ldots, K_{m+1}$ the identifiers of the $m+1$ entity types involved in R', iff in each relationship type $R_i \in \mathbb{R}$, a functional dependency $K_1^i, \ldots, K_l^i \rightarrow K_{l+1}^i$ holds for $K_1^i, \ldots, K_{l+1}^i$ the identifiers of the entity types $E_1^i, \ldots, E_{l+1}^i$ involved in R_i, such that:

1. K_{m+1} is equivalent to K_{l+1}^i;

2. for each $j = 1, \ldots, m$, $K_j \in \{K_1^i, \ldots, K_l^i\}$, i.e., K_j is equivalent to some element of $\{K_1^i, \ldots, K_l^i\}$, or K_j corresponds to a discrepant meta-attribute of an entity type of $E_1^i, \ldots, E_{l+1}^i$;

3. for each $j = 1, \ldots, l+1$, $K_j^i \in \{K_1, \ldots, K_{m+1}\}$, and all the discrepant meta-attributes of E_j^i are represented as the identifiers in $\{K_1, \ldots, K_m\}$.

Then we prove the above claim:

(=>) If a functional dependency $K_1, \ldots, K_m \rightarrow K_{m+1}$ holds in R', then a functional dependency $K_1^i, \ldots, K_l^i \rightarrow K_{l+1}^i$ holds in each relationship type $R_i \in \mathbb{R}$.

Suppose we are given two tuples $(k_1^i, \ldots, k_l^i, k_{l+1}^i)$, $(k_1^i, \ldots, k_l^i, k_{l+1}^i{}') \in R_i[K_1^i, \ldots, K_l^i, K_{l+1}^i]$. These two tuples correspond to $(k_1, \ldots, k_m, k_{m+1})$, $(k_1, \ldots, k_m, k'_{m+1}) \in R'[K_1, \ldots, K_m, K_{m+1}]$, which satisfy the three conditions of the above claim. As the functional dependency $K_1, \ldots, K_m \rightarrow K_{m+1}$ holds in R', $k_{m+1} = k'_{m+1}$. As k_{m+1} is equivalent to k_{l+1}^i and k'_{m+1} is equivalent to $k_{l+1}^i{}'$, $k_{l+1}^i = k_{l+1}^i{}'$. So a functional dependency $K_1^i, \ldots, K_l^i \rightarrow K_{l+1}^i$ holds in each relationship type R_i.

(<=) If a functional dependency $K_1^i, \ldots, K_l^i \rightarrow K_{l+1}^i$ holds in each relationship type $R_i \in \mathbb{R}$, then a functional dependency $K_1, \ldots, K_m \rightarrow K_{m+1}$ holds in R'.

Suppose we are given two tuples $(k_1, \ldots, k_m, k_{m+1}), (k_1, \ldots, k_m, k'_{m+1}) \in R'[K_1, \ldots, K_m, K_{m+1}]$. These two tuples correspond to $(k_1^i, \ldots, k_l^i, k_{l+1}^i), (k_1^i, \ldots, k_l^i, k_{l+1}^i{}')$ $\in R_i[K_1^i, \ldots, K_l^i, K_{l+1}^i]$ for some relationship type $R_i \in \mathbb{R}$, which satisfy the three conditions of the above claim. As the functional dependency $K_1^i, \ldots, K_l^i \rightarrow K_{l+1}^i$ holds in the relationship type R_i, $k_{l+1}^i = k_{l+1}^i{}'$. As k_{m+1} is equivalent to k_{l+1}^i and k'_{m+1} is equivalent to $k_{l+1}^i{}'$, $k_{m+1} = k'_{m+1}$. So the functional dependency $K_1, \ldots, K_m \rightarrow K_{m+1}$ holds in R'. □

Note in the proof of Lemma A.10, two relationship types of $\mathbb{R}$ may involve different sets of entity types because of the interplay of data and metadata (but after Algorithm ResolveEnt, these relationship types will be transformed to equivalent ones). Consequently, a functional dependency on R' may correspond to different functional dependencies on the relationship types of $\mathbb{R}$.

Lemma A.11. *When relationship types have some one-to-one or one-to-many attributes, Algorithm ResolveEnt preserves the functional dependencies represented as the cardinality constraints of the entity types in the relationship types.*

Proof. When a relationship type has some one-to-one or one-to-many attributes, the inherited context of the relationship type would be kept in the transformed relationship type (see Step 1.3 of Algorithm ResolveEnt). In this case, a set of relationship types corresponding to the same ontology type within the same self context would not be necessarily transformed into equivalent relationship types. Further, they should also have the same inherited context to be merged.

In general, in an ER schema DB, let $\mathbb{R} = \{R_1, \ldots, R_n\}$ be a set of relationship types corresponding to the same ontology type within the same context, such that each relationship type has some one-to-one or one-to-many attributes.
Let DB' be the schema transformed from DB by Algorithm ResolveEnt, in which all the relationship types of $\mathbb{R}$ are transformed and merged into a relationship type R'. We claim that:

A functional dependency $\mathbb{A} \rightarrow \mathbb{B}$ holds in each relationship type of $\mathbb{R}$ for $\mathbb{A}$ and $\mathbb{B}$ two distinct sets of the identifiers of some entity types involved in each relationship type of $\mathbb{R}$ iff the same functional dependency $\mathbb{A} \rightarrow \mathbb{B}$ holds in R'.

The proof of the claim is omitted. □

Lemma A.12. *When relationship types only have many-to-one and many-to-many attributes, Algorithm ResolveEnt preserves the functional dependencies and multivalued dependencies represented as the cardinality constraints of the attributes of the relationship types.*

Proof. If a relationship type only has many-to-one and many-to-many attributes, given an attribute of the relationship type, Algorithm ResolveEnt will remove some of its context inherited from the entity types involved in the relationship type, and move the attribute to a new relationship type (see Step 1.3 of Algorithm ResolveEnt). As long as we keep the cardinality of the attribute, the functional dependency/multivalued dependency are also preserved, but may be represented

in different forms. Note in a relationship type, although the cardinalities of entity types can only represent functional dependencies, the cardinalities of attributes can represent functional dependencies (if they are many-to-one attributes) and multivalued dependencies (if they are many-to-many attributes).

In general, in an ER schema DB, let $\mathbb{R} = \{R_1, \ldots, R_n\}$ be a set of relationship types that correspond to the same ontology type, have the same self context and the same attribute A, such that each relationship type only has many-to-one and many-to-many attributes.

Let DB' be the schema transformed from DB by Algorithm ResolveEnt, in which all the relationship types of $\mathbb{R}$ are transformed and merged into a relationship type R' with the attribute A. We claim:

A functional dependency $K_1, \ldots, K_m \rightarrow A$ (or a multivalued dependency $K_1, \ldots, K_m \twoheadrightarrow A$) holds in R' for $K_1, \ldots, K_m$ the identifiers of the m entity types involved in R', iff in each relationship type $R_i \in \mathbb{R}$, a functional dependency $K_1^i, \ldots, K_l^i \rightarrow A$ (or a multivalued dependency $K_1^i, \ldots, K_l^i \twoheadrightarrow A$) holds for $K_1^i, \ldots, K_l^i$ the identifiers of the entity types $E_1^i, \ldots, E_l^i$ involved in R_i, such that:

1. for each $j = 1, \ldots, m$, $K_j \in \{K_1^i, \ldots, K_l^i\}$, or K_j corresponds to a discrepant meta-attribute of an entity type of $E_1^i, \ldots, E_l^i$;

2. for each $j = 1, \ldots, l$, $K_j^i \in \{K_1, \ldots, K_m\}$, and all the discrepant meta-attributes of E_j^i are represented as the identifiers in $\{K_1, \ldots, K_m\}$.

The proof of the claim is omitted. □

Lemma A.13. *When relationship types have some one-to-one or one-to-many attributes, Algorithm ResolveEnt preserves the functional dependencies and multivalued dependencies represented as the cardinality constraints of the attributes of the relationship types.*

Proof. If a relationship type R has some one-to-one or one-to-many attributes, we should consider two kinds of dependencies: the identifier of R determines an attribute of R (i.e., a functional dependency or multivalued dependency), and the attribute determines the identifier of R (i.e., a functional dependency).

In general, in an ER schema DB, let $\mathbb{R} = \{R_1, \ldots, R_n\}$ be a set of relationship types that correspond to the same ontology type, have the same context and the same attribute A, such that each relationship type has some one-to-one or one-to-many attributes.

Let DB' be the schema transformed from DB by Algorithm ResolveEnt, in which all the relationship types of $\mathbb{R}$ are transformed and merged into a relationship type R' with the attribute A. We claim:

A functional dependency $K \rightarrow A$ (or a multivalued dependency $K \twoheadrightarrow A$) for K the identifier of each relationship type of $\mathbb{R}$ holds in each relationship type of $\mathbb{R}$ iff the same functional dependency $K \rightarrow A$ (or the same multivalued dependency $K \twoheadrightarrow A$) holds in R'.

Furthermore, if A is a one-to-one or one-to-many attribute, we also have a result below:

A functional dependency $A \rightarrow K$ for K the identifier of each relationship type of $\mathbb{R}$ holds in each relationship type of $\mathbb{R}$ iff the same functional dependency $A \rightarrow K$ holds in R'.

The proof of the claim is omitted. □

This completes the proof of Theorem 7.2. In a similar way, we can prove that any of the other three algorithms of ResolveRel, ResolveEntAttr and ResolveRelAttr preserves functional dependencies and multivalued dependencies.

A.11 Proof of Theorem 8.2

To prove the theorem, we should first prove the correctness of Algorithm MCT. We first define *conditioned minimum cost trees* below. Unlike a minimum cost tree that minimizes the data transformation costs of all the variations of an integrated schema tree, a conditioned minimum cost tree minimizes the costs of all the variations with the same fixed hierarchical structures of some cognate paths.

Definition A.5 (Conditioned minimum cost tree). *In a CP tree T obtained by merging a set of relationship types, let $P_1, \ldots, P_k$ be the nodes (i.e., cognate paths) of T. Given the hierarchical structures of $P_1, \ldots, P_k$, let $\mathbb{T}$ be the set of all the variations of T with the given hierarchical structures of $P_1, \ldots, P_k$. We call $T' \in \mathbb{T}$* a conditioned minimum cost tree of T with respect to $\{P_1, \ldots, P_k\}$, *if T' minimizes the data transformation costs of all the elements of $\mathbb{T}$.* □

The following two lemmas reveal the effect of beneficial cognate paths on the data transformation cost of an integrated schema.

Given a tree T and a node P in T, we denote $T(P)$ to be the tree consisting of the path from the root to P and the subtree rooted at P in T.

Lemma A.14. *Given a CP tree T obtained by merging a set of relationship types, let P be a node in T, and $P_1/\ldots/P_k$ be the path from the root to the parent of P in T. Given the hierarchical structures of $P_1, \ldots, P_k$, if P has some beneficial variations, then in the conditioned minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$, P must be beneficial.*

Proof. This lemma can be proven by contradiction. Suppose in the conditioned minimum cost tree T' of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$, P is not beneficial. Then we can find a tree T'' that is the same as T' except the hierarchical structure of P

which is beneficial. Then the data transformation cost of T'' is less then T'. This contradicts with the assumption that T' is the conditioned minimum cost tree. □

Lemma A.15. *Given a CP tree T obtained by merging a set of relationship types, let P be a node in T, and $P_1/\ldots/P_k$ be the path from the root to the parent of P in T. Given the hierarchical structures of $P_1, \ldots, P_k$, if P has no beneficial variations, then the hierarchical structure of P and the structures the descendants of P in T have nothing to do with the data transformation cost of the conditioned minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$.*

Proof. Given any variation T' of $T(P)$ with the given hierarchical structures of $P_1, \ldots, P_k$, as P has no beneficial variations, we can compute the data transformation cost of T' according to the beneficial cognate paths of $P_1, \ldots, P_k$ (see Formula (2) of Section 8.3.2), which have nothing to do with P and the descendants of P. □

Lemma A.16. *Algorithm MCT correctly computes a minimum cost tree.*

Proof. Given a CP tree T_S obtained by merging a set of relationship types, let P be a node of T_S, and $P_1/\ldots/P_k$ be the path from the root to the parent of P in T_S. We claim that Algorithm MCT(P, c, T) computes the conditioned minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$.

Base case: (Case i) P has beneficial variations and is a leaf node. According to Lemma A.14, to compute the conditioned minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$, we only need to consider the beneficial variations of P, as shown in Algorithm MCT (Line 12 to 17).

(Case ii) P has no beneficial variations. In this case, the hierarchical structures of P and the structures of the descendants of P have nothing to do with the data transformation cost, according to Lemma A.15. Algorithm MCT (Line 21 to 22) correctly computes the minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$.

Inductive step: P has beneficial variations and is a non-leaf node.

Inductive hypothesis: For any child element P' of P, Algorithm MCT returns the conditioned minimum cost tree of $T(P')$ w.r.t. $\{P_1, \ldots, P_k, P\}$.

Given the hierarchical structures of $P_1, \ldots, P_k$ and a beneficial variation of P, for each child P' of P, Algorithm MCT (Line 5) correctly computes the minimum cost tree of $T(P')$ w.r.t. $\{P_1, \ldots, P_k, P\}$, according to the inductive hypothesis. We claim that $T(P)$ is the conditioned minimum cost tree w.r.t. $\{P_1, \ldots, P_k, P\}$ at that time. Otherwise, suppose $T(P)'$ is the conditioned minimum cost tree whose data transformation cost is less than $T(P)$. Then in $T(P)'$, for each subtree $T1'$ rooted at a child of P that is different from the corresponding subtree $T1$ of $T(P)$, we can replace $T1'$ with $T1$, and get a tree that has at most the same cost as $T(P)'$. This contradicts with the assumption that $T(P)'$ has less data transformation cost than $T(P)$.

Then according to Lemma A.14, after considering all the beneficial variations of P (Line 3), we can get the conditioned minimum cost tree of $T(P)$ w.r.t. $\{P_1, \ldots, P_k\}$ (Line 8 to 9).

Consequently, when the algorithm completes, it returns the minimum cost tree of T_S. Note at that time P is the root node and $\{P_1, \ldots, P_k\}$ is empty. □

Since Algorithm MCT correctly computes a minimum cost tree for a schema tree, Algorithm MergeRel (Step 2) can produce an integrated schema minimizing the data transformation costs in the set of the schemas produced by Step 1 of the algorithm. That is, Theorem 8.2 is true.

Bibliography

[1] XML Schema, W3C recommendation. http://www.w3c.org/XML/Schema, 2001.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*, chapter 8, 10, pages 173–187, 216–235. Addison-Wesley, 1995.

[3] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

[4] Carlo Batini and Maurizio Lenzerini. A methodology for data schema integration in the entity-relationship model. *IEEE Trans. on Software Engineering*, 10(6), 1984.

[5] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Juliana Freire, and Rajeev Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, pages 277–288, 2003.

[6] Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.

[7] Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano. Semantic integration of heterogeneous information sources. *Data Knowl. Eng.*, 36(3):215–249, 2001.

[8] Paul De Bra. *Horizontal Decompositions in the Relational Database Model*. PhD thesis, University of Antwerp, 1987.

[9] Paul De Bra and Jan Paredaens. Conditional dependencies for horizontal decompositions. In *International Colloquium on Automata, Languages and Programming*, 1983.

[10] Giacomo Cabri, Francesco Guerra, Maurizio Vincini, Sonia Bergamaschi, Letizia Leonardi, and Franco Zambonelli. Momis: Exploiting agents to support information integration. *Int. J. Cooperative Inf. Syst.*, 11(3):293–314, 2002.

[11] Yabing Chen, Mong Li Lee, and Tok Wang Ling. Automatic generation of SQLX view definitions from ORA-SS views. In *DASFAA*, pages 476–481, 2004.

[12] Yabing Chen, Tok Wang Ling, and Mong Li Lee. Designing valid XML views. In *International Conference on Conceptual Modeling (ER)*, pages 463–478, 2002.

[13] Yabing Chen, Tok Wang Ling, and Mong Li Lee. Automatic generation of XQuery view definitions from ORA-SS views. In *International Conference on Conceptual Modeling (ER)*, pages 158–171, 2003.

[14] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient XML integration. In *SIGMOD*, 2000.

[15] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *SIGMOD*, 1998.

[16] Jos de Bruijn, Francisco Martin-Recuerda, Dimitar Manov, and Marc Ehrig. State-of-the-art survey on ontology merging and aligning. Technical report, Digital enterprise research institute, Univ. of Innsbruck, http://www.aifb.uni-karlsruhe.de/WBS/meh/publications/debruijn04state.pdf, 2004.

[17] Claude Delobel, Chantal Reynaud, Marie-Christine Rousset, Jean-Pierre Sirot, and Dan Vodislav. Semantic integration in Xyleme: a uniform tree-based approach. *Data & Knowledge Engineering*, 44(3):267–298, 2003.

[18] Daniel S. Weld Denise Draper, Alon Y. Halevy. The Nimble XML data integration system. In *ICDE*, 2001.

[19] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Y. Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies*, pages 385–404. 2004.

[20] Fernando Berzal Galiano, Juan C. Cubero, Fernando Cuenca, and Juan Miguel Medina. Relational decomposition through partial functional dependencies. *Data & Knowledge Engineering*, 43(2):207–234, 2002.

[21] Anthony Tomasic Georges Gardarin, Antoine Mensch. An introduction to the e-XML data integration suite. In *EDBT*, 2002.

[22] Georg Gottlob. Computing covers for embedded functional dependencies. In *SIGMOD*, 1987.

[23] Marc Gyssens, Laks V. S. Lakshmanan, and Iyer N. Subramanian. Tables as a paradigm for querying and restructuring. In *PODS*, pages 93–103, 1996.

[24] Qi He and Tok Wang Ling. Extending and inferring functional dependencies in schema transformation. In *CIKM*, pages 12–21, 2004.

[25] Qi He and Tok Wang Ling. Resolving schematic discrepancy in the integration of entity-relationship schemas. In *International Conference on Conceptual Modeling (ER)*, pages 245–258, 2004.

[26] Qi He and Tok Wang Ling. An ontology based approach to the integration of entitycrelationship schemas. *Data & Knowledge Engineering*, 58(3):299–326, 2006.

[27] Chun-Nan Hsu and Craig A. Knoblock. Semantic query optimization for query plans of heterogeneous multidatabase systems. *TKDE*, 12(6):959–978, 2000.

[28] W. H. Inmon. *Building the Data Warehouse*. Wiley, second edition, 1996.

[29] Euna Jeong and Chun-Nan Hsu. Induction of integrated view for XML data with heterogeneous DTDs. In *CIKM*, 2001.

[30] Vipul Kashyap and Amit P. Sheth. Semantic and schematic similarity between database objects: a context-based approach. *The VLDB Journal*, 5, 1996.

[31] Andreas Koeller and Elke A. Rundensteiner. Incremental maintenance of schema-restructuring views in SchemaSQL. *TKDE*, 16(9):1096–1111, 2004.

[32] Ravi Krishnamurthy, Witold Litwin, and William Kent. Language features for interoperability of databases with schematic discrepancies. In *SIGMOD Conference*, pages 40–49, 1991.

[33] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. Schemasql - a language for interoperability in relational multi-database systems. In *VLDB*, pages 239–250, 1996.

[34] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. On efficiently implementing SchemaSQL on a SQL database system. In *VLDB*, pages 471–482, 1999.

[35] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL-an extension to SQL for multidatabase interoperability. *TODS*, 2001.

[36] James A. Larson, Shamkant B. Navathe, and Ramez Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Trans. Software Eng.*, 15(4), 1989.

[37] Ora Lassila and Ralph R. Swick. Resource description framework (RDF) model and syntax specification. W3C recommendation. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, 1999.

[38] Mong Li Lee and Tok Wang Ling. Resolving constraint conflicts in the integration of ER schemas. In *International Conference on Conceptual Modeling (ER)*, pages 394–407, 1997.

[39] Mong Li Lee and Tok Wang Ling. A methodology for structural conflicts resolution in the integration of entity-relationship schemas. *Knowledge and Information Sys.*, 5:225–247, 2003.

[40] Mong Li Lee, Tok Wang Ling, and Wai Lup Low. Designing functional dependencies for XML. In *EDBT*, pages 124–141, 2002.

[41] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107, 1994.

[42] Tok Wang Ling. Extending classical functional dependencies for physical database design (lecture notes). http://www.comp.nus.edu.sg/ lingtw/cs4221/extended.fds.pdf, 2001.

[43] Tok Wang Ling, Mong Li Lee, and Gillian Dobbie. *Semistructured Database Design*. Springer, 2005.

[44] Mengchi Liu and Tok Wang Ling. A data model for semistructured data with partial and inconsistent information. In *EDBT*, pages 317–331, 2000.

[45] Wai Lup Low, Mong Li Lee, and Tok Wang Ling. A knowledge-based approach for duplicate elimination in data cleaning. *Information Systems*, 26:585–606, 2001.

[46] Daofeng Luo, Ting Chen, Tok Wang Ling, and Xiaofeng Meng. On view transformation support for a native XML DBMS. In *DASFAA*, pages 226–231, 2004.

[47] Alexander Maedche, Boris Motik, Nuno Silva, and Raphael Volz. MAFRA a mapping framework for distributed ontologies. In *the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW-2002*, 2002.

[48] Nelson Mendonça Mattos. Integrating information for on demand computing. In *VLDB*, 2003.

[49] Wolfgang May. Lopix: a system for XML data integration and manipulation. In *VLDB*, 2001.

[50] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[51] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.

[52] Eduardo Mena, Arantza Illarramendi, Vipul Kashyap, and Amit P. Sheth. Observer: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Distributed and Parallel Databases*, 8(2):223–271, 2000.

[53] Xiaofeng Meng, Daofeng Luo, Mong Li Lee, and Jing An. OrientStore: A schema based native XML storage system. In *VLDB*, pages 1057–1060, 2003.

[54] Renée J. Miller. Using schematically heterogeneous structures. In *SIGMOD*, pages 189–200, 1998.

[55] Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB*, 1993.

[56] Wei Ni and Tok Wang Ling. GLASS: a graphical query language for semi-structured data. In *DASFAA*, pages 363–370, 2003.

[57] Wei Ni and Tok Wang Ling. Translate graphical XML query language to SQLX. In *DASFAA*, pages 907–913, 2005.

[58] Natalya Fridman Noy and Mark A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI/IAAI*, pages 450–455, 2000.

[59] Borys Omelayenko and Dieter Fensel. A two-layered integration approach for product information in B2B e-commerce. In *the Second Intenational Conference on Electronic Commerce and Web Technologies (EC WEB-2001)*, 2001.

[60] David Plotkin. Building the XML repository (presentation slides). http://www.intelligenteai.com/XMLRepository/, 2001.

[61] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[62] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, 2003.

[63] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[64] M. P. Reddy, Bandreddi E. Prasad, and Amar Gupta. Formulating global integrity constraints during derivation of global schema. *Data & Knowledge Engineering*, pages 241–268, 1995.

[65] Patricia Rodríguez-Gianolli and John Mylopoulos. A semantic approach to xml-based data integration. In *ER*, pages 117–132, 2001.

[66] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL web ontology language guide, W3C recommendation. http://www.w3.org/TR/owl-guide/, 2004.

[67] Yann Dupont Stefano Spaccapietra, Christine Parent. Model independent assertions for integration of heterogeneous schemas. *VLDB*, 1992.

[68] Wei Sun and Clement T. Yu. Semantic query optimization for tree and chain queries. *TKDE*, 6(1):136–151, 1994.

[69] Aris Tsois and Timos K. Sellis. The generalized pre-grouping transformation: aggregate query optimization in the presence of dependencies. In *VLDB*, 2003.

[70] Mike Uschold. Creating, integrating, and maintaining local and global ontologies. In *the First Workshop on Ontology Learning (OL-2000)*, 2000.

[71] Mark W. W. Vermeer and Peter M. G. Apers. On the applicability of schema integration techniques to database interoperation. In *ER*, 1996.

[72] Mark W. W. Vermeer and Peter M. G. Apers. The role of integrity constraints in database interoperation. In *VLDB*, 1996.

[73] Pepijn R. S. Visser and Zhan Cui. On accepting heterogeneous ontologies in distributed architectures. In *the ECAI98 workshop on applications of ontologies and problem-solving methods*, 1998.

[74] Xia Yang, Mong Li Lee, and Tok Wang Ling. Resolving structural conflicts in the integration of XML schemas: a semantic approach. In *ER*, pages 520–533, 2003.

[75] Xia Yang, Mong Li Lee, Tok Wang Ling, and Gillian Dobbie. A semantic approach to query rewriting for integrated xml data. In *ER*, 2005.