Processing Queries and Merging Schemas in Support of Data Integration

Rachel Amanda Pottinger

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Rachel Amanda Pottinger

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

____

Philip A. Bernstein

____

Alon Y. Halevy

Reading Committee:

____

Philip A. Bernstein

____

Alon Y. Halevy

____

Renée J. Miller

Date: __________________________

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, or to the author.”

Signature_______________________________________

Date____

University of Washington

Abstract

Processing Queries and Merging Schemas in Support of Data Integration

Rachel Amanda Pottinger

Co-Chairs of the Supervisory Committee:

Affiliate Professor Philip A. Bernstein
Department of Computer Science and Engineering

Associate Professor Alon Y. Halevy
Department of Computer Science and Engineering

The goal of data integration is to provide a uniform interface, called a mediated schema, to a set of autonomous data sources, which allows users to query a set of databases without knowing the schemas of the underlying data sources.

This thesis describes two aspects of data integration: an algorithm for answering queries posed to a mediated schema and the process of creating a mediated schema. First, we present the MiniCon algorithm for answering queries in a data integration system and explain why MiniCon outperforms previous algorithms by up to several orders of magnitude.

Second, given two relational schemas for data sources, we propose an approach for using conjunctive queries to describe mappings between them. We analyze their formal semantics, show how to derive a mediated schema based on such mappings, and show how to translate user queries over the mediated schema into queries over local schemas. We then show a generic Merge operator that merges schemas and mappings regardless of data model or application. Finally, we show how to implement the derivation of mediated schemas using the generic Merge operator.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Portions of this thesis have been previously published. Most of Chapter 3 appeared in (Pottinger et al. 2001), and much of Chapter 5 appeared in (Pottinger et al. 2003).

This thesis could not have been written without the help of a great many people. I begin by thanking my advisors for all their time, their dedication, and the great fun that we had doing research together. In particular I thank Alon Halevy for teaching me what research is and how much fun it can be. I thank Phil Bernstein for helping me to learn how to do research with other people, honing my research (and writing) skills, and for giving instantaneous feedback. I thank the remaining member of my reading committee, Renée Miller, for her amazing ability to quickly grasp what I'm working on and her unfailing generosity with her time. The remaining members of my committee have also been great to work with. Dan Weld has helped me out when I had difficulties, and Dan Suciu showed me how much fun it is to have *two* database faculty in a department.

I also thank the graduate student members of the database group at the University of Washington. Special thanks go to Zack Ives for being a founding member of the group with me as well as a great friend. AnHai Doan has always been a sounding board and giver of advice that is as useful as it is entertaining. Marc Friedman did a great job at introducing me to the agony of having someone read your technical paper and helping me learn how to react. Peter Mork worked with me on merging ontologies, engaged me in many fruitful technical conversations and also provided fantastic moral support — you rule! Jayant Madhavan has also been fantastic to talk over ideas with. They, along with the other database students over the years, especially Nilesh Dalvi, Luna Dong, Mikhail Gubanov, Ashish Gupta, Yana Kadiyska, Isaac Kunen, Gerome Miklau, Bart Niswonger, and Igor Tatarinov, have been fantastically supportive in reading drafts, coming to practice talks, and discussing ideas over IHOP breakfasts and otherwise.

In addition to graduate students, I've had the chance to work with a number of fantastic undergraduates: Shiori Betzler, Ewa Jaslikowska, Jake Kreutzer, Hakim Weatherspoon, Seth Whelan, and Theodora Yeung were great students to work with and helped me learn how to direct undergraduate research.

Having Phil Bernstein — who works at Microsoft research — as an advisor, has meant that I've been lucky enough to have a second research group. I spent three wonderful summers as an intern in the Microsoft Research Database Group. Thanks to Roger Barga, Jonathan Goldstein, Paul Larson, and David Lomet for being great group members. Thanks to Jayant Madhavan, Peter Mork, and Michalis Petropoulos for being great summer officemates, and the rest of the interns, particularly repeat interns Hongfei Guo and Jingren Zhou (Jingren is now hired permanently) for being great people to learn with and from. Finally, thanks to Sergey Melnik, now in the group, for great conversations, technical and otherwise.

I've been very lucky to have many funding sources. Thanks to Microsoft Research for having me as a fellow for three years, NSF for also providing a three year fellowship, and Lucent Bell Labs for my GRPW grant. Muthu Muthukrishnan was a great mentor for the summer I was at Bell Labs and has continued to provide great advice and mathematical puzzles since. My final fellowship is an Achievement Reward for College Scientists where my fellowship was donated by Dottie and Hunter Simpson, who have both been extremely generous to fund me and great sources of inspiration. Thank you all.

I've always benefited from always having great teachers, in computer science and otherwise. Without their teachings on how to love learning, I would never have gotten a PhD. In particular, thanks to Mr. Gensure for teaching me Pascal in high school (and being so nice when I dropped his class for art). At Duke, Carla Ellis has been a continuing source of technical and moral support in addition to teaching me the invaluable skill of how to read a technical paper. Finally, also at Duke, Owen Astrachan has been instrumental in hooking me on computer science as a career and keeping my interest there. Thank you so much to all of you and all the rest of my teachers.

When I came to graduate school, I had tremendous support from older graduate students who showed me the ropes. A special thanks goes to Mike Ernst for teaching me not only how to be a better researcher, but also how to critique papers and talks and being a fabulous friend as well. Many others helped to show me the ropes, including AJ Bernheim Brush, Craig Kaplan, Tessa Lau, Brian Michalowski, Todd Millstein, Denise Pinnel Salisbury, Sean Sandys, and Doug Zongker. Thank you everyone; I did my best to pass along your good advice!

At the other end of the spectrum are the students who helped me through this difficult last year of finishing up and getting a job. The group of graduating students, particularly AJ Bernheim Brush, Jeff Hightower, Kurt Partridge, Emily Rocke, and Neil Spring have been invaluable; I shall miss the grammar smack downs. My officemates this year have also been great pillars of support; an office won't be an office without Ankur Jain, Jessica Miller, and Marianne Shaw.

In between are all of the other students who have come to practice talks, discussed research, read my papers, and generally been great friends and people such as Miguel Figueroa, Krzysztof Gajos, Maya Rodrig, Vibha Sazawal, Sarah Schwarm, Tammy VanDeGrift, and Ken Yasuhara. Thank you all so much.

The staff of the University of Washington Department of Computer Science and Engineering have always been fantastic both to work with and to come to know as friends. Particular thanks go to Patrick Allen, Frankye Jones, and Lindsay Michimoto for being great friends and helpful well above and beyond the call of duty.

My thanks also go to the faculty of the department. David Notkin has been an unyielding source of support and friendship; I can't imagine how these past few years would have been without him. Richard Anderson and Ed Lazowska have also been fantastic sources of support. My thanks to the three of you for being great friends. Thanks go to them and the rest of the faculty for showing me how to be a good faculty member.

I thank my friends outside the department for reminding me that, all evidence to the contrary, there is life outside of computer science and academia. Kelly Shaw has been a fantastic source of support, both in school and out, for the past 11 years. Wendy Brophy, Jenny Exelbierd, and Robert "Wob" Flowers have all kept me laughing when the laughs were running low.

My parents — Edie Carlson, Rachel Maines, and Garrel Pottinger — helped me to love learning and also helped me realize that sometimes you need to put the book down. Thanks also go to my in-laws, Jay and Shelley Wolfman for proving all those "in-law" jokes wrong; I'm looking forward to having them live closer. My sister, Chloe Meadows, has always been there for me and willing to put up with me. Thank you.

Finally, my husband, Steve Wolfman, has been unbelievably helpful, supportive, and wonderful. I've been very lucky to have him as first a fellow undergraduate student, then a fellow graduate student, and now a fellow faculty member. I wouldn't have gotten here without him.

Dedication

To my husband, Steve Wolfman.

For keeping my head in the air and my feet on the ground.

Chapter 1
Introduction

Initially databases were largely self-contained. Each database had its own locus of expertise, and in that locus of expertise, that database was the sole arbiter of facts and how those facts should be expressed. This meant, among other things, that each database was required to interact with only one *schema*, a representation of the concepts contained in the database. A schema consists of a set of *relations*, each of which is represented as a table. For example, if the database was used to make airplane reservations, the database would store airport data in exactly one representation, and that representation would be described by one or more relations. In this situation, interaction between databases was very uncommon. One reason is that it was extremely manually intensive to set up multiple databases talking to one another. In addition because such care was invested in how to set up the database schema, people were very loath to change it.

However, the falling cost of disks, fast processors, and fast and stable network connections have led to many sources of overlapping information, such as different sources available to find out about airfares. In order to determine what the correct answer to a query – a question over the database – is, the answer must now be asked over multiple sources. Consider Example 1.1:

> **Example 1.1:** Figure 1.1 shows a scenario that might occur if the reader were trying to plan a beach vacation. Three different elements might need to be incorporated (1) where to find a beach (2) where there is good weather and (3) where cheap flights can be found. Each of these pieces of information can in turn be found from a number of different sources. For example, cheap flight information can come from the travel websites AAA, Expedia, Orbitz, or any of a number of other sources. □

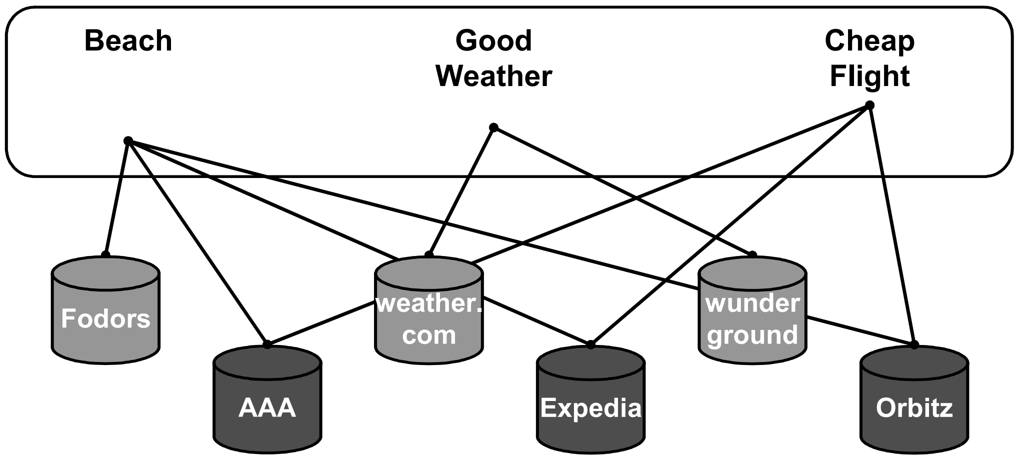


Figure 1.1: A typical data management scenario today often requires incorporating data from multiple sources.

In addition to the existence of many different data sources, there are many more database users with different goals. Because there are so many different queries that users may be interested in, it is not feasible to decide a priori how to partition each user query into queries over the data sources; the user queries must be *reformulated* at query time into queries over each of the different data sources.

> **Example 1.2:** Continuing with Example 1.1, if there were a small number of types of queries that were to be asked, it would be easy to figure out how to reformulate such queries. For example, if the goal of the system was only to find cheap flights, it would be relatively easy to figure out how to translate a query that the user would ask over the system into queries over AAA, Expedia, or Orbitz. However, there are a large number of queries that users may want to ask over the system, so it is impractical to pre-compute how to reformulate all possible user queries. □

This process is similar but distinct from the traditional process of *query optimization*. In query optimization the user's query is asked over a declarative language (such as SQL), and then the query is rewritten in a procedural form. In both query reformulation and query optimization the goal is to create query plans on the fly, hence some of the same techniques that are helpful in query optimization are also useful in query reformulation. Query reformulation, however is a separate process. Even after a data integration query over the mediated schema has been reformulated, it will still need to be optimized for efficient execution.

This scenario where many users are accessing many different databases simultaneously is very common these days, and not just across the World Wide Web. For example, in order to perform their work, bio-medical researchers may need to combine information about genes with information about proteins, each of which can be found in any number of data sources. Similarly, researchers from other fields such as astronomy and literature require information from many different sources. These two examples typify today's data management challenges: in today's world, there is a vast user base wishing to combine data from many overlapping databases without having to know where that data comes from. In this thesis we describe mechanisms to make querying multiple databases simultaneously faster for users and how to improve system construction for administrators.

1.1 Data Warehouses and Data Integration

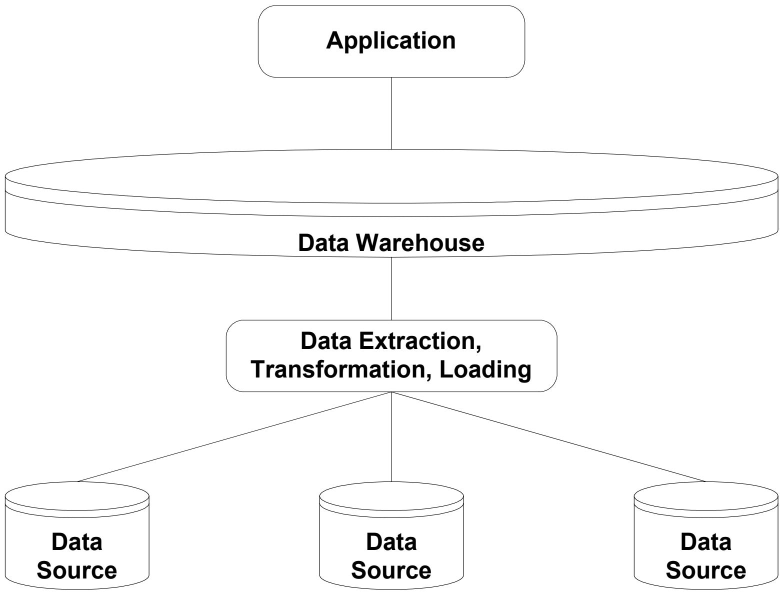


Figure 1.2: Data warehouse architecture

When querying multiple databases simultaneously there are two main data modeling choices; Either (1) the data can be gathered in a central location and then queried over that central repository or (2) the data can be left in the individual sources, queries can be rewritten to be over each of the individual sources, and then the data can be combined. The former approach is taken by *data warehousing systems*, such as the system in Figure 1.2. The latter approach is taken by *data integration systems* such as the system shown in Figure 1.3.

1.1.1 Data Warehouses

The goal of a data warehouse is to gather all of the data relevant to a particular set of queries in the same place. An example data warehouse architecture is shown in Figure 1.2. The application talks to the data warehouse. In order to render the data warehouse usable, all data is processed as a batch through a data extraction, transformation, and loading (ETL) module. This module operates in a batch fashion. At some point when the warehouse is not being queried, the ETL module extracts the data from the base sources through querying the underlying sources, cleans the data such that the data is internally consistent, transforms it to be in the schema of the data warehouse rather than being in the schemas of the individual data sources, and then loads the data into the data warehouse. Queries are asked over the data warehouse's schema and then immediately executed over the warehouse's contents without needing to refer to the original data sources. Because the data has all been stored together and the ETL layer has pre-computed many of the transformations that must occur, data warehouses are particularly good for answering queries that require particularly costly computations that reuse much of the same data. For example, the prototypical data warehouse example is that of seeing what different regions have different characteristics for Wal-Mart sales data. However, because the data has to go through the ETL level before it can be queried, data in data warehouses is often stale. Hence data warehouses are inappropriate for queries that require fresh data.

1.1.2 Data Integration

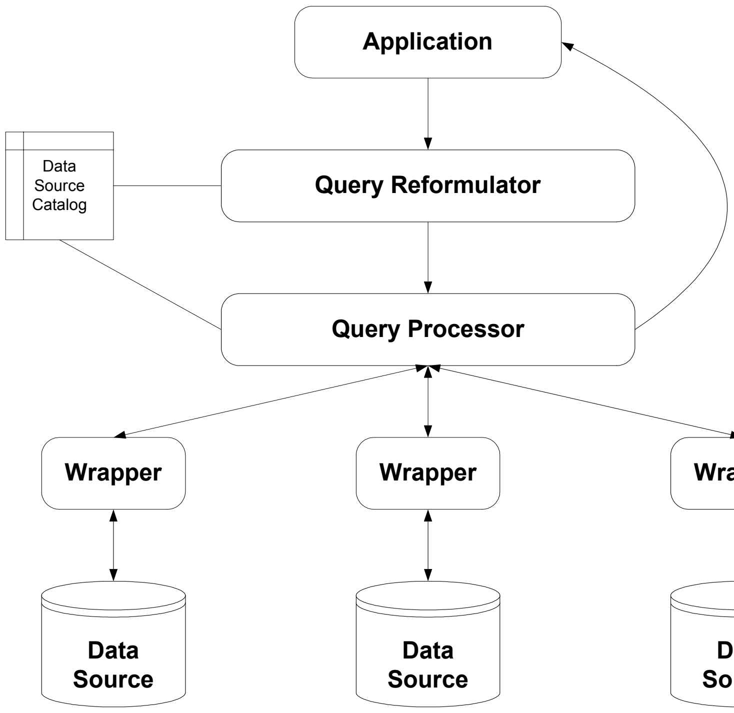


Figure 1.3: Data integration architecture

A data integration system, unlike a data warehouse, leaves the data at the sources. One example of a data integration architecture is shown in Figure 1.3. As such, querying in a data integration system is more complex than querying in data warehousing. As with data warehousing, user queries are asked through some application. This application asks queries over a *mediated schema*, which is a schema that has been designed to represent data from all the data sources in the data integration system, much like the data warehouse's schema.

Since the data has not been pre-computed to be in the mediated schema, the application queries must be *reformulated* into queries over the data sources that the data is stored in. The query reformulator is responsible for looking at the data source catalog to determine what sources are relevant to the query. Following Example 1.1, if a query asked for cheap airfares, the query reformulator would determine from the data source catalog that the AAA, Orbitz, and Expedia sources were relevant, and would translate the query over the mediated schema into a reformulated query over the data sources. This reformulated query would then be given to the query processor which would determine the best way to execute the query (e.g., which data source should be contacted when and how the results should be combined) using statistics from the data source catalog.

Query processing is complicated by the unpredictability of network delays, inaccuracy of statistics, and individual variance from the capabilities of different sources. Work on processing distributed queries has been studied extensively in such projects as eddies and (Avnur et al. 2000) adaptive query processing (Ives 2002) and surveyed, e.g., (Kossmann 2000). When the query processor is ready for data from an individual data source, the query processor queries the data source's wrapper, which translates the data from its native format to the format required by the query processor (Kushmerick et al. 1997). For example, if the data was from the web, the wrapper might change it from HTML to relational data. The query processor then combines the data returned from each data source's wrapper and passes that data back to the application.

Because the data is not cached ahead of time as it is in data warehouses, the data can be fresher than data that is stored in a data warehouse. In addition, there is no requirement to store all the data, which may be very large, or it may even be impossible to warehouse all the data because the data sources do not want you to have their entire database (e.g., the web retailer Amazon does not want anyone to have their entire database, though they are happy to expose parts of it for people to build applications on top of). Hence data integration systems are good for queries where freshness is paramount or the overall amount of data is very large but the amount needed to answer the queries is relatively small. Conversely data integration, like many distributed systems, is not good for applications that require very large quantities of data to be processed or for systems requiring strict performance guarantees.

In this thesis, though we discuss creating the schema for a data warehouse, our primary contributions are making querying data integration systems faster for users and improving system construction for administrators.

1.2 Querying in a Data Integration System

Since queries are asked over the mediated schema but the data is stored in the source schemas, queries are posed over the mediated schema and then translated into queries over the source schemas in which the data is stored. For this translation to occur, we need to know how the source schemas are related to the mediated schema. There are two solutions that are commonly proposed in database literature, both of which use the concept of a *view* – a query that has been named for reuse. In the first solution, Global-As-View (GAV), the mediated schema is formalized as a set of views over the local sources; this approach simply uses traditional views as has been used extensively in database literature. For a recent survey of GAV approaches see (Lenzerini 2002). In the second solution, *Local-As-View* (*LAV*) (Levy et al. 1996), the local sources are formalized as a set of views over the mediated schema.

In GAV, each mediated schema relation is defined in terms of the local sources that correspond to it. For example, continuing with the scenario in Example 1.1, in GAV the data integration system would store that the mediated schema relation of "airfare" can be found by asking a query over AAA, Orbitz and Expedia.

Translating a query in GAV requires only replacing the mediated schema relation in the query with the set of local sources that can provide data for that mediated schema relation. Continuing with the scenario in Example 1.1, translating a query about finding the cheapest airfare to Maui would require only replacing the occurrence of the relation "airfare" with the view (i.e., stored query) defining how to find information about airfares in AAA, Orbitz, and Expedia. Unfortunately, because GAV describes the mediated schema in terms of the local sources, adding new sources is a complicated procedure; the definition for each mediated schema relation may need to be modified to describe how to retrieve data from the new sources. In the continuing example, for instance, if the system were going to add on Air Canada, the view defining airfare would have to change, as would the view defining any other concept to which Air Canada could contribute.

LAV, on the other hand, describes each source as a view over the mediated schema. In our continuing example, there might be a view explaining that AAA could give information about flights that could be bought at the same time as a rental car was rented. Hence, adding a new source to the data integration system requires only adding new queries describing how the local source can answer queries over the mediated schema relations. Again, following our continuing example, if Air Canada were to be added to a LAV system, the information about AAA, Orbitz, and Expedia would all remain the same, and new queries would be added only to define what data Air Canada contains.

Unfortunately answering queries in LAV is much more complex; since the local sources are described in terms of the mediated schema rather than the other way around, translating queries over the mediated schema into queries over the local source schemas is not as simple. In the running example, to find the cheapest airfare to get to a destination with good weather in LAV, we would have to look at the views defining airfare described above and then decide how to combine those sources of information with information about which destinations have good weather. Rewriting a query over the mediated schema into queries over the source schema relies on a technique known as *answering queries using views*; for a survey on answering queries using views see (Halevy 2001).

Answering queries using views is also used in *query optimization* – the process of transforming a declarative query into a procedure for executing the query – in order to reduce execution time for traditional database systems. In query optimization the input is a declarative explanation of what parts of the schema need to be used, and the output is a procedural plan that explains how the query can be executed efficiently. Here views can be used as caches of pre-computed information and can speed up the procedural plan considerably, though the underlying relations are available as well.

In Chapter 3 we present the first contribution of this thesis: the MiniCon Algorithm, a fast algorithm for answering queries using views in the context of data integration and show how it can be extended to the case of query optimization. Prior to the MiniCon Algorithm, many researchers believed that using LAV to describe a data integration architecture would be prohibitively slow since answering queries using views is NP-Complete in the size of the query, even for query languages with limited expressiveness (Levy et al. 1995). Chapter 3 shows not only the MiniCon algorithm but also shows – in the first large-scale experiments on answering queries using views – that MiniCon is practical in many cases and faster than previous algorithms for answering queries using views, sometimes by orders of magnitude. Thus assuming that the mediated schema is related to the source schemas using LAV, answering queries over it is sufficiently fast for users. Next we turn our attention to how to create the mediated schema initially.

1.3 Mediated Schema Creation

Mediated schema creation today is often ad-hoc. Mediated schemas are typically formed by a committee deciding what concepts should be represented in the mediated schema and how to represent those concepts. The designers can guarantee that the mediated schema will meet their requirements. On the other hand, it is problematic for a number of reasons:

- If the mediated schema is related through the mapping languages of GAV or LAV, these languages may overly restrict the way in which the sources can be related to one another.
- Mediated schema creation in this fashion is very expensive. Experts cost money, and bringing them together costs time and money. This means that not only will it take considerable effort to create the mediated schema in the first place, but updating it will be much more expensive in the long run. Because sources change over time, the mediated schema must be flexible enough to add new concepts, which is difficult in this situation. Though using experts to build the mediated schema may be required in some circumstances, in other scenarios such as coordinating disaster relief, the mediated schema may need to be built up much more quickly and automatically.
- There is no guarantee that the mediated schema satisfies any requirements about what should be in the mediated schema.

Mechanisms for creating a mediated schema out of sources have previously been studied in database research; such projects generally focus on how to resolve conflicts such as synonyms and homonyms in relation names or how to ensure that the resulting schema is valid in a particular data model – e.g., SQL. Batini, Lenzerini, and Navathe provide a survey of such methods (Batini et al. 1986). Buneman, Davidson, and Kosky (Buneman et al. 1992) provide a general theory of what it means to merge two source schemas to create a third schema. Others have approached the problem from a more pragmatic point of view, such as the Database Design and Evaluation Workbench (Rosenthal et al. 1994) that allow users to manipulate schemas, including combining multiple views into one schema. Still others have created formal criteria for when two schemas consist of the same information, both for data integration and other applications (Hull 1984; Kalinichenko 1990; Miller et al. 1993). But none of these papers have tackled the following problem: given two schemas, how should we create a mediated schema *and also* the mappings from the mediated schema to the sources.

In Chapter 4 we describe the second contribution of this thesis. We define a set of Mediated Schema Criteria to which any mediated schema must adhere. We then provide the first investigation of creating a data integration system from such criteria. We then explain that creating a mediated schema requires at a minimum two source schemas and information relating the two schemas. The intuition for this is shown in Example 1.3:

> **Example 1.3**: Continuing with Example 1.2, in order to create the mediated schema we must know not only the schemas but where they overlap. For example, in order to create a mediated schema for Expedia and Orbitz we must know that both of them represent airfare information and how that airfare information is stored, and which elements of those representations correspond to each other. □

In Chapter 4 we show that the information relating the local sources drives the choice of mediated schema. Moreover, the language that is used to describe the relationship between the local sources affects the choice of language needed to relate the mediated schema and the source schemas (e.g., GAV or LAV). In particular, we show that even in the case where two schemas are related to one another through a very simple language, neither GAV nor LAV are expressive enough to relate the mediated schema to the source schemas if the Mediated Schema Criteria are satisfied.

1.4 Generically Merging Schemas

While Chapter 4 concentrates only on creating a mediated schema for data integration in the context of the relational data model, merging complex descriptions such as schemas happens in a number of contexts in addition to data integration. Some examples are:

- In data warehouse creation, one can view creating the data warehouse as the problem of merging schemas.

- Ontology merging is increasingly important due to the proliferation of large ontologies through such efforts as medical informatics and the growing interest in the semantic web.
- In software engineering, current trends toward aspect-oriented programming or subject-oriented programming require merging class definitions.

In each of these examples, two or more schemas are merged to create a third schema, and in particular their meta-data — data about how other data is stored, i.e., schema information — is merged. Managing meta-data is integral to creating database applications, yet meta-data functionality is often built from scratch for each new application. Chapter 5 contributes Merge, a generic merge solution that describes how to merge schemas not only across different applications, but also across different meta-models (i.e., XML DTDs, relational schemas, etc.). Merge is designed to be part of a larger system called *Model Management*. The goal of Model Management is to reduce the effort of creating meta-data applications (e.g., building a data warehouse or propagating changes in schema evolution) with three key abstractions: models, mappings, and operators. A *model* is a formal description of a complex application artifact, such as a database schema, an application interface, a UML model, or an ontology. A *mapping* describes the relationship between two models. An *operator* (e.g., Merge, Match, Compose, Diff) manipulates models and mappings as objects. As an example of how Model Management helps with meta-data applications, Model Management can create the mediated schema for data integration by first using Match to create a mapping between two local schemas and then Merge the autonomous local schemas to create the common schema. Similarly, applying more operators can create a common schema if there are more than two local schemas.

Merge combines two input models based on a mapping between them. For instance, suppose the data integration example in Example 1.1 merged the customer account databases for two travel agencies. These models, though potentially very similar, can contain many differences. Consider customers' names. One travel agency might store FirstName and LastName while the other stores simply Name. How should names be represented in the merged model? In addition, suppose that one bank uses a relational database and the other uses XML; not only must the models be merged, the merge must be performed across different types of models. Although this example uses database schemas, Merge is designed for other model types as well, such as merging ontologies and merging classes in object-oriented programming.

Previous merging algorithms either concentrated on resolving data-model-specific conflicts (e.g., a column in a relational model cannot have sub-columns) or used data models that were not rich enough to express many relevant conflicts. In Chapter 5 we describe a Merge that is generic enough to be useful in many different data models yet specific enough to resolve application-specific conflicts. To do so we show three levels of detail. At the most abstract level, we define the input, output, and desired properties of Merge with respect to typical merging scenarios. At a more detailed level we show a representation for models and show that Merge in this representation, when used with other Model Management operators, subsumes many previous merging algorithms. Even though the previous algorithms were designed specifically for the data models from different merging problems (e.g., view integration and ontology merging), Merge can subsume them.

Chapter 6 contributes the most detailed Merge analysis; how Merge interacts with the semantics of the applications in which it will be used. We take the inputs defined in Chapter 4 and show that for most cases Merge as defined in Chapter 5 can be used to build the output defined in Chapter 4, particularly when building the mediated schema is concerned, though building the mappings from the mediated schema to the local sources is more difficult. This first attempt at encoding a semantic operation in Model Management gives us a glimpse into when it succeeds at its goals of allowing users not only to build up the schemas they require but the rest of the application as well, and when human intervention is required.

1.5 Outline of Dissertation

The remainder of this dissertation is structured as follows.

Chapter 2 provides background of terminology that is used throughout the thesis.

Chapter 3 introduces the MiniCon Algorithm, a fast algorithm for answering queries using views in data integration. We also show that the MiniCon Algorithm outperforms previous algorithms for answering queries using views, sometimes by orders of magnitude. In addition, we show how to extend the algorithm to the traditional problem of query optimization.

In Chapter 4 we describe how to create the mediated schema based on a set of Mediated Schema Requirements we have created. We show why traditional GAV or LAV mappings between local sources are insufficient to express the relationship between mediated schemas. We then show the first bottom-up mediated schema creation algorithm, and show how to rewrite queries over that schema.

In Chapter 5 we describe how to extend mediated schema creation to a more general problem: how to merge to schemas. We describe an algorithm for merging schemas that can be in data integration, view integration, ontology merging, and other database and non-database applications. We describe the semantics of Merge in general and for a specific representation, Vanilla, in which Merge subsume previous algorithms. Finally, we merge two large anatomy ontologies to show that Merge scales and is useful in practice.

In Chapter 6 we show how to emulate the mediated schema creation of Chapter 4 using Model Management operators, especially Merge as defined in Chapter 5. We concentrate on showing one method that can be done entirely through encoding the mapping and schemas in multiple Model Management operators.

Chapter 7 concludes.

Chapter 2
Background

In this chapter we present background terminology used throughout this thesis. In much of this thesis we are concerned with relational database schemas:

Definition 2.1: (Relational database schema). A relational database schema is a set of relations. Each relation r consists of an ordered list of attributes $attributes_r$ = [a_{r1}, …, a_{rm}]. We use the term *relation state* when referring to the set of tuples in a relation, i.e., the *database* of the relational database schema □

In the remainder of this chapter we define concepts used throughout the thesis. As in the bulk of the thesis, we describe these concepts within the context of relational database schemas: queries and views (Section 2.1), query containment and equivalence (Section 2.2), answering queries using views (Section 2.3) and query rewriting in data integration (Section 2.4).

2.1 Queries and Views

One of the chief goals of a database system is to answer queries: an application asks a query over a database and then the database system returns values to the application. Queries are expressed in a query language. Throughout this thesis we use *conjunctive queries* (i.e., select-project-join queries without arithmetic comparisons) when a query language is required.

Definition 2.2: (Conjunctive query). A *conjunctive query* Q has the form $q(\overline{X})$:- $e_1(\overline{X_1}),...,e_n(\overline{X_n})$ where q and $e_1, ..., e_n$ are predicate names referring to database relations. The atoms $e_1(\overline{X_1}),...,e_n(\overline{X_n})$ are collectively the body of the query, denoted $body(Q)$ or $Subgoals(Q)$. Each $e_i(\overline{X_i})$ is a *subgoal* or *Extensional Database Predicate (EDB)*. The atom $q(\overline{X})$ is called the *head*. Its predicate name, q, is called the *IDB name* of Q, denoted $IDB(Q)$, which is the name of the answer relation. The tuples $\overline{X}, \overline{X_1}, ..., \overline{X_n}$ contain either variables or constants.

We require that the query Q be *safe*, i.e., that $\overline{X} \subseteq \overline{X_1} \cup ... \cup \overline{X_n}$ (that is, every variable that appears in the head must also appear in the body). The variables in $\overline{X}$ are the ***distinguished*** variables of the query, and all the others are *existential* variables, denoted $distinguished(Q)$ or $existential(Q)$ respectively. Join predicates in this notation are expressed by multiple occurrences of the same variables; i.e., if a variable appears multiple times in the same conjunctive query, we require that the value of the variable be the same each time it appears. For a query Q we denote the variables that have multiple occurrences by $Joined(Q)$.

We denote individual variables by lowercase letters and constants appear in quotation marks. The semantics of conjunctive queries requires that the database values that satisfy the restrictions required by the body are returned in a relation with the name of the head of the query. Unions are expressed by conjunctive queries having the same head predicate. We use $Vars(Q)$ to refer to the set of variables in Q, and $Q(D)$ to refer to the result of evaluating the query Q over the database D. □

Example 2.1: Consider the following schema that we use throughout this chapter and Chapter 3. The relation $cites(p1,p2)$ stores pairs of publication identifiers where $p1$ cites $p2$. The relation $sameTopic$ stores pairs of papers that are on the same topic. The unary relations $inSIGMOD$ and $inVLDB$ store ids of papers published in SIGMOD and VLDB respectively. The following query asks for pairs of papers on the same topic that also cite each other.

$Q(x,y) :- sameTopic(x,y), cites(x,y), cites(y,x)$ □

A *view* is a named query. If the query result is stored, we refer to it as a materialized view, and we refer to the result set as the *extension* of the view. Occasionally we consider queries that contain subgoals with comparison predicates <, ≤, ≠. In this case, we require that if a variable x appears in a subgoal of a comparison predicate, then x must also appear in an ordinary subgoal.

2.2 Query Containment and Equivalence

The concepts of query containment and equivalence enable us to compare the values that two queries will return if they are both asked over the same database instance. In this thesis we primarily use these concepts to determine when a query over a mediated schema is correctly translated into a query over source schemas. We say that a query Q_1 is *contained* in the query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if the answer to Q_1 is a subset of the answer to Q_2 for *any* database instance. We say that Q_1 and Q_2 are *equivalent* if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$, i.e., they produce the same set of tuples for any given database.

Query containment and equivalence have been studied extensively for conjunctive queries and unions of conjunctive queries (Chandra et al. 1977; Sagiv et al. 1981), conjunctive queries with arithmetic comparison predicates (Klug 1988; Kolaitis et al. 1998; Levy et al. 1993; Zhang et al. 1993), and recursive queries (Chaudhuri et al. 1992; Chaudhuri et al. 1994; Levy et al. 1993; Sagiv 1988; Shmueli 1993).

In this thesis we demonstrate containment using containment mappings, which provide a necessary and sufficient condition for testing query containment (Chandra et al. 1977).

Definition 2.3: (Containment mapping). Given a partial mapping τ on the *variables* of a query, we extend τ to apply to *subgoals* of the query as follows: If all the variables of a subgoal g are in the domain of τ and map to a subgoal with the same predicate name and arity, we say that τ maps g. A mapping τ from $Vars(Q_2)$ to $Vars(Q_1)$ is a *containment mapping* if (1) τ maps every subgoal in the body of Q_2 to a subgoal in the body of Q_1, and (2) τ maps the head of Q_2 to the head of Q_1. The query Q_2 contains Q_1 if and only if there is a containment mapping from Q_2 to Q_1 (Chandra et al. 1977). □

2.3 Answering Queries Using Views

Informally speaking, the problem of answering queries using views is the following. Suppose we are given a query Q over a database schema, and a set of view definitions $V_1, ..., V_n$ over the same schema. Is it possible to answer the query Q using *only* the answers to the views $V_1, ..., V_n$, and if so, how? The problem of answering queries using views has recently received significant attention because of its relevance to a wide variety of data management problems (Halevy 2001): query optimization (Chaudhuri et al. 1995; Levy et al. 1995; Zaharioudakis et al. 2000), maintenance of physical data independence (Yang et al. 1987) (Tsatalos et al. 1996) (Popa et al. 2000), data integration (Duschka et al. 1997b; Kwok et al. 1996; Lambrecht et al. 1999; Levy et al. 1996), and data warehouse and web-site design (Harinarayan et al. 1996; Theodoratos et al. 1997). For a recent survey on answering queries using views, see (Halevy 2001). There are two main contexts in which the problem of answering queries using views has been considered. In the first context, where the goal is query optimization or maintenance of physical data independence (Chaudhuri et al. 1995; Tsatalos et al. 1996; Yang et al. 1987), we search for an expression that uses the views and is *equivalent* to the original query as defined in Definition 2.4. The second context is that of data integration, where views describe a set of autonomous heterogeneous data sources; we discuss the goals in this context in Section 2.4

Formally, given a query Q and a set of view definitions $\mathcal{V} = V_1, \ldots, V_m$, a *rewriting* of the query using the views is a query expression Q' whose body predicates are either $V_1, \ldots, V_m$ or comparison predicates.

Definition 2.4: (Equivalent rewriting). Let Q be a query, and $\mathcal{V} = V_1, \ldots, V_n$ be a set of views, both over the same database schema. A query Q' is an equivalent rewriting of Q using $\mathcal{V}$ if for any database D, the result of evaluating Q' over $V_1(D), \ldots, V_n(D)$ is the same as $Q(D)$. □

To tell if two queries are equivalent, we depend on the notion of unfolding or expanding a query as defined in Definition 2.5:

Definition 2.5: (Unfolding or expanding a query). Let Q be a query $q(\overline{X})$:- $e_1(\overline{X_1}),\ldots,e_n(\overline{X_n})$. Q' is an *expansion* or *unfolding* of Q if for each $e_i(\overline{X_i})$ s.t. e_i is defined by view V_i, we replace each occurrence of e_i with the definition of V_i, substituting either the values of $\overline{X_i}$ for distinguished variables in V_i and fresh variable names for the existential variables in V_i. □

Remark 2.1. Since different views may have similar definitions, they may evaluate to the same values even though the view name is different. Thus it is often necessary to unfold a query asked over a view in order to check on whether or not it is equivalent to another query. Example 2.2 gives an example of both the concept of finding an equivalent rewriting and view unfolding. □

Example 2.2: Consider the query from Example 2.1 and the following views. The view $V1$ stores pairs of papers that cite each other, and $V2$ stores pairs of papers on the same topic, each of which cites at least one other paper.

$Q(x,y)$:- $sameTopic(x,y), cites(x,y), cites(y,x)$

$V1(a,b)$:- $cites(a,b), cites(b,a)$

$V2(c,d)$:- $sameTopic(c,d), cites(c,c1), cites(d,d1)$

The following is an equivalent rewriting of Q:

$Q'(x,y)$:- $V1(x,y), V2(x,y)$.

To check that Q' is an equivalent rewriting, we unfold Q' w.r.t. the view definitions to obtain Q'', and show that Q is equivalent to Q'' using a containment mapping (in this case it is the identity on x and y and $x1 \rightarrow y$, $y1 \rightarrow x$).

$Q''(x,y)$:- $cites(x,y), cites(y,x), sameTopic(x,y), cites(x,x1), cites(y,y1)$ □

2.4 *Query Rewriting in Data Integration*

As explained in Chapter 1, one of the main uses of algorithms for answering queries using views is in the context of data integration systems that provide their users with a uniform interface to a multitude of data sources (Friedman et al. 1997; Kwok et al. 1996; Lambrecht et al. 1999; Levy et al. 1996; Ullman 1997). Since users pose queries in terms of a *mediated schema* and the data is stored in the sources, in order to be able to translate users' queries into queries on the data sources, the data integration system needs a description of the contents of the sources. One of the approaches to specifying such descriptions is to describe a data source as a view over the mediated schema, specifying which tuples can be found in the source, an approach known as Local-As-View (LAV) (see Chapter 1). For example, in the domain described in Example 2.1, we may have two data sources, *S1* and *S2*, containing pairs (*p1*, *p2*) of SIGMOD and VLDB papers (respectively) such that *p1* cites *p2* and *p2* cites *p1*. The sources can be described as follows:

S1(a,b) :- *cites(a,b), cites(b,a), inSIGMOD(a), inSIGMOD(b)*

S2(a,b) :- *cites(a,b), cites(b,a) inVLDB(a), inVLDB(b)*

Given a query *Q*, the data integration system first needs to reformulate *Q* to refer to the data sources, i.e., the views. There are two differences between this application of answering queries using views and that considered in the context of query optimization. First, the views in data integration are assumed to adhere to the open world assumption (Definition 2.6):

> **Definition 2.6**: (Open world assumption). A complete view definition is one for which a view provides all tuples that match the view's definition. A sound view definition is any view definition for which the results of the view are those tuples defined by the body of the view; all view definitions are assumed to be sound. A view definition under the open world assumption is assumed to be sound but not complete. □

Due to the open world assumption, views in a LAV data integration system are not assumed to contain *all* the tuples in their definition since the data sources are managed autonomously; i.e., LAV view definitions are not complete. For example, the source *S1* may not contain all the pairs of SIGMOD papers that cite each other. Because of the open world assumption the goal is to find what is known as *certain answers*:

Definition 2.7: (Certain answers). Let V be a view definition over a schema S, I be an instance of the view V and Q a query over S. A tuple t is a *certain answer* to Q under the open world assumption if t is an element of $Q(D)$ for each database D with $I \subseteq V(D)$ (Abiteboul et al. 1998). □

Second, we cannot always find an equivalent rewriting of the query using the views because there may be no data sources that contain all of the information the query needs. Instead, we consider the problem of finding a *maximally-contained rewriting* (Duschka et al. 1997c). Maximally-contained rewritings are defined with respect to a particular query language in which we express rewritings. Intuitively, a maximally-contained rewriting is one that provides all the answers possible from a given set of sources. Formally, it is defined as follows.

Definition 2.8: (Maximally-contained rewriting). The query Q' is a maximally-contained rewriting of a query Q using the views $\mathcal{V} = V_1, \ldots, V_n$ w.r.t. a query language $\mathcal{L}$ if

1. for any database D, and extensions $v_1, \ldots, v_n$ of the views such that $v_i \subseteq V_i(D)$, for $1 \leq i \leq n$, then $Q'(v_1, \ldots, v_n) \subseteq Q(D)$,
2. there is no other query Q_1 in the language $\mathcal{L}$, such that for every database D and extensions $v_1, \ldots, v_n$ as above (a) $Q'(v_1, \ldots, v_n) \subseteq Q_1(v_1, \ldots, v_n)$ and (b) $Q_1(v_1, \ldots, v_n) \subseteq Q(D)$, and (c) there exists at least one database for which 1. is a strict set inclusion.
3. Q_1 and Q' are in $\mathcal{L}$.

Q does not have to be in $\mathcal{L}$. □

Given a conjunctive query Q and a set of conjunctive views $\mathcal{V}$, the maximally-contained rewriting of a conjunctive query may be a union of conjunctive queries (we refer to the individual conjunctive queries as *conjunctive rewritings*). Hence, considering Definition 2.8, if the language $\mathcal{L}$ is less expressive than non-recursive Datalog, there may not be a maximally-contained rewriting of the query. When the queries and the views are conjunctive and do not contain comparison predicates, it follows from (Levy et al. 1995) that we need only consider conjunctive rewritings *Q'* that have at most the number of subgoals in the query *Q*.

> **Example 2.3:** Continuing the example begun in Example 2.1, assuming we have the data sources described by *S1*, *S2* and *V2* and the same query *Q*, the rewriting that will generate the most sound answers given the sources is:
>
> *Q'(x,y)* :- *S1(x,y), V2(x,y)*
>
> *Q'(x,y)* :- *S2(x,y), V2(x,y)* □

The rewriting in Example 2.3 is a union of conjunctive queries, describing multiple ways of obtaining an answer to the query from the available sources. The rewriting is not an equivalent rewriting, since it misses any pair of papers that is not both in SIGMOD or both in VLDB, but we do not have data sources to provide us such pairs. Furthermore, since the sources are not guaranteed to have all the tuples in the definition of the view, our rewritings need to consider different views that may have similar definitions. For example, suppose we have the following source *S3*:

S3(a,b) :- *cites(a,b), cites(b,a), inSIGMOD(a), inSIGMOD(b)*

The definition of *S3* is identical to that of *S1*. However, because of source incompleteness, it may contain different tuples than *S1*. Hence, our rewriting will also have to include the following in addition to the other two rewritings.

Q'(x,y) :- *S3(x,y), V2(x,y)*

The ability to find a maximally-contained rewriting depends in subtle ways on other properties of the problem. It follows from (Abiteboul et al. 1998) that if (1) the query contains comparison subgoals, or (2) the views are assumed to be complete (i.e., the *closed world assumption* holds), then there may not be a maximally-contained rewriting if we consider $\mathcal{L}$ to be the language of unions of conjunctive queries or even if we consider Datalog with recursion. In addition, even if a maximally-contained rewriting exists, a maximally-contained rewriting with respect to $\mathcal{L}$ (Definition 2.8) will only provide all certain answers if $\mathcal{L}$ is monotone (Duschka et al. 1998). A query *Q* over a schema *R* is monotonic if ∀ states σ_i, σ_j of *R*, $\sigma_i \subseteq \sigma_j$

implies that $Q(\sigma_i) \subseteq Q(\sigma_j)$ (Abiteboul et al. 1995). However, given that in all cases that we consider the queries and views are monotone, a maximally-contained rewriting is guaranteed to return all certain answers.

Given these definitions we are now ready to present the rest of the thesis.

Chapter 3
MiniCon

3.1 Introduction

In Chapter 1, we motivated the problem of answering queries using views as a method for reformulating queries asked in a Local-As-View (LAV) data integration system. In this chapter, we assume that a data integration system has been set up using LAV as the mapping language between the mediated schema and the data sources, and focus on how to translate user queries into queries over the data sources using a technique called answering queries using views (a.k.a. rewriting queries using views). We defined answering queries using views in Chapter 2, both for finding equivalent rewritings (Definition 2.4) in query optimization of physical data independence and for data integration where the goal is to find maximally-contained rewritings (Definition 2.8). While the problem is NP-Complete in the number of subgoals of the query, the number of query subgoals is generally quite small. On the other hand, in some data integration applications, the number of data sources may be quite large – for example, data sources may be a set of web sites, a large set of suppliers and consumers in an electronic marketplace, or a set of peers containing fragments of a larger data set in a peer-to-peer environment. Hence, the challenge in this context is to develop an algorithm that scales up in the number of views.

We consider the problem of answering conjunctive queries using a set of conjunctive views in the presence of a large number of views. In general, this problem is NP-Complete because it involves searching through a possibly exponential number of rewritings (Levy et al. 1995). Previous work has mainly considered two algorithms for this purpose. The bucket algorithm, developed as part of the Information Manifold System (Levy et al. 1996), controls its search by first considering each subgoal in the query in isolation, and creating a bucket that contains only the views that are relevant to that subgoal. The algorithm then creates rewritings by combining one view from every bucket. As we show, the combination step has several deficiencies, and does not scale up well. The inverse-rules algorithm, developed in (Duschka et al. 1997a; Qian 1996), is used in the InfoMaster System (Duschka et al. 1997a). The inverse-rules algorithm considers rewritings for each database relation independent of any particular query. Given a user query, these rewritings are combined appropriately. We show that the rewritings produced by the inverse-rules algorithm need to be further processed in order to be appropriate for query evaluation. Unfortunately, in this additional processing step the algorithm must duplicate much of the work done in the second phase of the bucket algorithm.

Based on the insights into the previous algorithms, we introduce the MiniCon algorithm, which addresses their limitations and scales up to a large number of views. The key idea underlying the MiniCon algorithm is a change of perspective: instead of building rewritings by combining rewritings for each query subgoal or database relation, we consider how each of the *variables* in the query can interact with the available views. The result is that the second phase of the MiniCon algorithm needs to consider drastically fewer combinations of views. Hence, as we show experimentally, the MiniCon algorithm scales much better. The specific contributions of this chapter are the following:

- We describe the MiniCon algorithm and its properties.
- We present a detailed experimental evaluation and analysis of algorithms for answering queries using views. The experimental results show (1) the MiniCon algorithm significantly outperforms the bucket and inverse-rules algorithms, and (2) the MiniCon algorithm scales up to hundreds of views, thereby showing for the first time that answering queries using views can be efficient on data integration systems with many sources. We believe that our experimental evaluation in itself is a significant contribution that fills a void in previous work on this topic.
- We describe an extension of the MiniCon algorithm to handle comparison predicates and experimental results on its performance.
- We describe an extension of the MiniCon algorithm to the context of cost-based query optimization, where the goal is to find the single cheapest plan for the query using the views. In doing so we distinguish the role of two sets of views: those that are needed for the logical correctness of the plan, and those that are only needed to reduce the cost of

the plan. We show that different techniques are needed in order to identify each of these sets.

This chapter focuses on answering queries using views for select-project-join queries under set semantics. While such queries are quite common in data integration applications, many applications will need to deal with queries involving grouping and aggregation, semi-structured data, nested structures and integrity constraints. Indeed, the problem of answering queries using views has been considered in these contexts as well (Calvanese et al. 1999; Cohen et al. 1999; Duschka et al. 1997c; Grumbach et al. 1999; Gryz 1998; Gupta et al. 1995; Papakonstantinou et al. 1999; Srivastava et al. 1996). In contrast to the above works, our focus is on obtaining a scalable algorithm for answering queries using views and the experimental evaluation of such algorithms. Hence, we begin with the class of select-project-join queries.

Remark 3.1: It is important to emphasize at this point that this chapter concentrates on ensuring that the rewriting of the query obtains as many answers as possible from a set of views, which is the main concern in the context of data integration. The bulk of this chapter is not concerned with the problem of finding the rewriting that yields the *cheapest* query execution plan over the views, which would be the main concern if our goal was query optimization. In Section 3.6 we present an extension of the MiniCon algorithm to the context of query optimization, and show how the ideas underlying the MiniCon algorithm apply in that context as well. In addition, we do not consider here the issue of ordering the results from the sources. □

The chapter is organized as follows. Section 3.2 discusses the limitations of the previous algorithms. Section 3.3 describes the MiniCon algorithm, and Section 3.4 presents the experimental evaluation. Section 3.5 describes an extension of the MiniCon algorithm to comparison predicates. Section 3.6 describes how to extend the MiniCon algorithm to context of query optimization, and Section 3.7 concludes. The proof of the MiniCon algorithm is described in Appendix A.

3.2 *Previous Algorithms*

The theoretical results on answering queries using views (Levy et al. 1995) showed that when there are no comparison predicates in the query, the search for a maximally-contained rewriting can be confined to a finite space: an algorithm needs to consider every possible conjunction of n or fewer view atoms, where n is the number of subgoals in the query. Two previous algorithms, the bucket algorithm and the inverse-rules algorithm, attempted to find more effective methods to produce rewritings that do not require such exhaustive search. In this section, we briefly describe these algorithms and point out their limitations. In Section 3.4, we compare these algorithms to our MiniCon algorithm and show that the MiniCon algorithm significantly outperforms them. We describe the algorithms for queries and views without comparison subgoals.

3.2.1 The Bucket Algorithm

The bucket algorithm was developed as part of the Information Manifold System (Levy et al. 1996). The key idea underlying the bucket algorithm is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each subgoal in the query in isolation and determine which views may be relevant to a particular subgoal. The bucket algorithm is even more effective in the presence of comparison subgoals because comparison subgoals often enable the bucket algorithm to deem many views as being irrelevant to a query.

We illustrate the bucket algorithm with the following query and views. Note that the query now only asks for a set of papers, rather than pairs of papers.

Q1(x) :− *cites(x,y),cites(y,x),sameTopic(x,y)*

V4(a) :− *cites(a,b), cites(b,a)*

V5(c,d) :− *sameTopic(c,d)*

V6(f,h) :− *cites(f,g),cites(g,h),sameTopic(f,g)*

In the first step, the bucket algorithm creates a bucket for each subgoal in *Q1*. The bucket for a subgoal *g* contains the views that include subgoals to which *g* can be mapped in a rewriting of the query. If a subgoal *g* unifies with more than one subgoal in a view *V*, then the bucket of *g* will contain multiple occurrences of *V*.[1] The bucket algorithm would create the buckets in Table 3.1.

Table 3.1: Example buckets created by Bucket Algorithm

cites(x,y)	*cites(y,x)*	*sameTopic(x,y)*
V4(x)	*V4(x)*	*V5(x,y)*
V6(x,y)	*V6(x,y)*	*V6(x,y)*

Note that it is possible to unify the subgoal *cites(x,y)* in the query with the subgoal *cites(b,a)* in *V4*, with the mapping $x \rightarrow b$, $y \rightarrow a$. However, the algorithm did not include the entry *V4(y)* in the bucket because it requires that every distinguished variable in the query be mapped to a distinguished variable in the view.

In the second step, the algorithm considers conjunctive query rewritings, each consisting of one conjunct from every bucket. Specifically, for each element of the Cartesian product of the buckets, the algorithm constructs a conjunctive rewriting and checks whether it is contained (or can be made to be contained by adding join predicates) in the query. If so, the rewriting is added to the answer. Hence, the result of the bucket algorithm is a union of conjunctive rewritings.

In our example, the algorithm will try to combine *V4* with the other views and fail (as we explain below). Then it will consider the rewritings involving *V6*, and note that by equating the variables in the head of *V6* a contained rewriting is obtained. Finally, the algorithm will also discover that *V6* and *V5* can be combined. Though not originally described as part of the bucket algorithm, it is possible to add an additional simple check that will determine that the resulting rewriting will be redundant (because *V5* can be removed). Hence, the only rewriting in this example (which also turns out to be an equivalent rewriting) is:

[1] If we have knowledge of functional dependencies in the schema, then it is often possible to recover the values of attributes that have been projected away, but we do not consider this case here.

Q1'(x) :- *V6(x,x)*

The main inefficiency of the bucket algorithm is that it misses some important interactions between view subgoals by considering each subgoal in isolation. As a result, the buckets contain irrelevant views, and hence the second step of the algorithm becomes very expensive. We illustrate this point on our example.

Consider the view *V4*, and suppose that we decide to use *V4* in such a way that the subgoal *cites(x,y)* is mapped to the subgoal *cites(a,b)* in the view, as shown below:

Q1(x) :- *cites(x,y), cites(y, x), sameTopic(x,y)*

↓ ↓ ?

V4(a) :- *cites(a,b), cites(b, a)*

We can map *y* to *b* and be able to satisfy both *cites* predicates. However, since *b* does not appear in the head of *V4*, if we use *V4*, then we will not be able to apply the join predicate between *cites(x,y)* and *sameTopic(x,y)* in the query. Therefore, *V4* is not usable for the query, but the bucket algorithm would not discover this.

Furthermore, even if the query did not contain *sameTopic(x,y)*, the bucket algorithm would not realize that if it uses *V4*, then it has to use it for *both* of the query subgoals. Realizing this would save the algorithm exploring useless combinations in the second phase.

As we explain later, the MiniCon algorithm discovers these interactions in the first phase. In this example, MiniCon will determine that *V4* is irrelevant to the query. In the case in which the query does not contain the subgoal *sameTopic(x,y)*, the MiniCon algorithm will discover that the two *cite* subgoals need to be treated atomically.

3.2.2 The Inverse-Rules Algorithm

Like the bucket algorithm, the inverse-rules algorithm (Duschka et al. 1997a; Qian 1996) was also developed in the context of a data integration system. The key idea underlying the algorithm is to construct a set of rules that *invert* the view definitions, i.e., rules that show how to compute tuples for the database relations from tuples of the views. Given the views in the previous example, the algorithm would construct the following inverse rules:

R1: *cites(a, f1(a))* :- *V4(a)*

R2: *cites(f1(a), a)* :- *V4(a)*

R3: *sameTopic(c,d)* :- *V5(c,d)*

R4: *cites(f, f2(f,h))* :- *V6(f,h)*

R5: *cites(f2(f,h), h)* :- *V6(f,h)*

R6: *sameTopic(f, f2(f,h))* :- *V6(f,h)*

Consider the rules R1 and R2; intuitively, their meaning is the following. A tuple of the form *V4(p1)* in the extension of the view *V4* is a witness of two tuples in the relation *cites*. It is a witness in the sense that *V4(p1)* tells that the relation *cites* contains a tuple of the form *(p1, Z)*, for some value of *Z*, and that the relation also contains a tuple of the form *(Z, p1)*, for the *same* value of *Z*.

In order to express the information that the unknown value of *Z* is the same in the two atoms, we refer to it using the functional Skolem term *f1(Z)*. Note that there may be several values of *Z* in the database that cause the tuple *(p1)* to be in the self-join of *cites*, but all that we know is that there exists at least one such value.

The rewriting of a query *Q* using the set of views $\mathcal{V}$ is simply the composition of *Q* and the inverse rules for $\mathcal{V}$. Hence, one of the important advantages of the algorithm is that the inverse rules can be constructed ahead of time in polynomial time, independent of a particular query.

The rewritings produced by the inverse-rules algorithm, as originally described in (Duschka et al. 1997a), are not appropriate for query evaluation for two reasons. First, applying the inverse rules to the extension of the views may invert some of the useful computation done to produce the view. Second, we may end up accessing views that are irrelevant to the query. To illustrate the first point, suppose we use the rewriting produced by the inverse-rules algorithm in the case where the view *V6* has the extension *{(p1, p1), (p2, p2)}*.

First, we would apply the inverse rules to the extensions of the views. Applying R4 would yield *cites(p1, f2(p1,p1)), cites(p2, f2(p2,p2))*, and similarly applying R5 and R6 would yield the following tuples:

cites(p1, f2(p1,p1)),

cites(f2(p1,p1),p1),

cites(f2(p2,p2),p2),

sameTopic(p1,p1),

sameTopic(p2,p2).

Applying the query *Q1* to the tuples computed above obtains the answers *p1* and *p2*. However, this computation is highly inefficient. Instead of directly using the tuples of *V6* for the answer, the inverse-rules algorithm first computed tuples for the relation *cites*, and then had to re-compute the self-join of *cites* that was already computed for *V6*. Furthermore, if the extensions of the views *V4* and *V5* are not empty, then applying the inverse rules would produce useless tuples as explained in Section 3.2.1.

Hence, before we can fairly compare the inverse-rules algorithm to the others, we need to further process the rules. Specifically, we need to expand the query with every possible combination of inverse rules. However, expanding the query with the inverse rules turns out to repeat much of the work done in the second phase of the bucket algorithm. In our example, since we have four rules for *cites* and two rules for *sameTopic*, we may need to consider 32 such expansions in the worst case.

In the experiments described in Section 3.4 we consider an extended version of the inverse-rules algorithm that produces a union of conjunctive queries by expanding the definitions of the inverse rules. We expanded the subgoals of the query one at a time, so we could stop an expansion of the query at the moment when we detect that a unification for a subset of the subgoals will not yield a rewriting (thereby optimizing the performance of the inverse-rules algorithm). We show that the inverse-rules algorithm can perform much better than the bucket algorithm, but the MiniCon algorithm scales up significantly better than either algorithm.

Remark 3.2: It is important to clarify why our study considers the extended version of the inverse-rules algorithm, rather than the original version. It is easy to come up with (real) examples in which the execution of plan generated by the original inverse-rules algorithm would be arbitrarily worse than that of the bucket algorithm or the MiniCon algorithm. Hence, we face the usual tradeoff between spending significant time on optimization, but with much more substantial savings at run-time. An optimizer that would accept the result of the original inverse-rules algorithm would definitely try to optimize the plan by trying to reduce the number of joins it needs to perform. By using the extended version of the inverse-rules algorithm we are putting all three algorithms on equal footing in the sense that one does not need more optimization than the other. Optimizations will still be applied to them, but the same optimizations can be applied to the results of each of the algorithms. □

3.3 The MiniCon Algorithm

The MiniCon algorithm begins like the bucket algorithm, considering which views contain subgoals that correspond to subgoals in the query. However, once the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V, it changes perspective and looks at the variables in the query. The algorithm considers the join predicates in the query (which are specified by multiple occurrences of the same variable) and finds the minimal additional set of subgoals that need to be mapped to subgoals in V, given that g will be mapped to g_1. This set of subgoals and mapping information is called a *MiniCon Description* (MCD), and can be viewed as a generalization of buckets. In the second phase, the algorithm combines the MCDs to produce the rewritings. It is important to note that because of the way we construct the MCDs, the MiniCon algorithm does not require containment checks in the second phase, giving it an additional speedup compared to the bucket algorithm. Section 3.3.1.1 describes the construction of MCDs, and Section 3.3.2 describes the combination step. For ease of exposition we describe the MiniCon algorithm for queries and views without constants. The proof of correctness of the MiniCon algorithm can be found in Appendix A.

3.3.1.1 Forming the MCDs

We begin by introducing a few terms that are used in the description of the algorithm. Given a mapping τ from $Vars(Q)$ to $Vars(V)$, we say that a view subgoal g_1 *covers* a query subgoal g if $\tau(g)=g1$.

A MCD is a mapping from a subset of the variables in the query to variables in one of the views. Intuitively, a MCD represents a fragment of a containment mapping from the query to the rewriting of the query. The way in which we construct the MCDs guarantees that these fragments can later be combined seamlessly.

As seen in our example, we need to consider mappings from the query to specializations of the views, where some of the head variables may have been equated (e.g., $V6(x,x)$ instead of $V6(x,y)$ in our example). Hence, every MCD has an associated *head homomorphism.* A head homomorphism h on a view V is a mapping h from $Vars(V)$ to $Vars(V)$ that is the identity on the existential variables, but may equate distinguished variables, i.e., for every distinguished variable x, $h(x)$ is distinguished, and $h(x)=h(h(x))$. Note that the consideration of the head homomorphisms adds no complexity to the MiniCon algorithm. Since the MiniCon algorithm must check to see if each view subgoal can cover each query subgoal, the least restrictive head homomorphism (which is the one that the MiniCon algorithm uses) follows immediately from looking at which positions in the view the query variables must be assigned to in order for the view to be used in a partial containment mapping.

Formally, we define MCDs as follows.

Definition 3.1: (MCD). A MCD, C for a query Q over a view V is a tuple of the form $(h_C, V(\overline{Y})_C, \phi_C, G_C)$ where:

- h_C is a head homomorphism on V,
- $V(\overline{Y})_C$ is the result of applying h_C to V, i.e., $\overline{Y} = h_C(\overline{A})$, where $\overline{A}$ are the head variables of V,
- ϕ_C is a partial mapping from $Vars(Q)$ to $h_C(Vars(V))$,
- G_C is a subset of the subgoals in Q which are covered by some subgoal in $h_C(V)$ using the mapping ϕ_C (note: not all such subgoals are necessarily included in G_C). □

In words, ϕ_C is a mapping from Q to the specialization of V obtained by the head homomorphism h_C. G_C is a set of subgoals of Q that we cover by the mapping ϕ_C. Property 3.1 below specifies the exact conditions we need to consider when we decide which subgoals to include in G_C. Note that $V(\overline{Y})_C$ is uniquely determined by the other elements of a MCD, but is part of a MCD specification for clarity in our subsequent discussions. Furthermore, the algorithm will not consider all the possible MCDs, but only those in which h_C is the least restrictive head homomorphism necessary in order to unify subgoals of the query with subgoals in a view.

The mapping ϕ_C of a MCD C may map a set of variables in Q to the same variable in $h_C(V)$. In our discussion, we sometimes need to refer to a representative variable of such a set. For each such set of variables in Q we choose a representative variable arbitrarily, except that we choose a distinguished variable whenever possible. For a variable x in Q, $EC_{\phi C}(x)$ denotes the representative variable of the set to which x belongs. $EC_{\phi C}(x)$ is defined to be the identity on any variable that is not in Q.

The construction of the MCDs is based on the following observation on the properties of query rewritings. The proof of this property is a corollary of the correctness proof of the MiniCon algorithm.

> **Property 3.1:** Let C be a MCD for Q over V. Then C can only be used in a non-redundant rewriting of Q if the following conditions hold:
>
> C1. For each head variable x of Q which is in the domain of ϕ_C, $\phi_C(x)$ is a head variable in $\phi_C(V)$.
>
> C2. If $\phi_C(x)$ is an existential variable in $h_C(V)$, then for every g, subgoal of Q, that includes x (1) all the variables in g are in the domain of ϕ_C, and (2) $\phi_C(g) \in h_C(V)$ □

Clause C1 is the same as in the bucket algorithm. Clause C2 captures the intuition we illustrated in our example, where if a variable x is part of a join predicate which is not enforced by the view, then x must be in the head of the view so the join predicate can be applied by another subgoal in the rewriting. In our example, clause C2 would rule out the use of $V4$ for query $Q1$ because the variable b is not in the head of $V4$, but the join predicate with $sameTopic(x,y)$ has not been applied in $V4$.

```
procedure formMCDs($\mathcal{Q},\mathcal{V}$)
/* $Q$ and $\mathcal{V}$ are conjunctive queries. */
    $\mathcal{C}=\varnothing$
 For each subgoal $g \in Q$
For view $V \in \mathcal{V}$ and every subgoal $v \in V$
        Let $h$ be the least restrictive head homomorphism on $V$ such that there exists a mapping $\phi$, s.t.
            $\phi(g)=h(v)$.
        If $h$ and $\phi$ exist, then add to $\mathcal{C}$ any new MCD $C$ that can be constructed where:
            (a) $\phi_C$ respectively $h_C$ is an extension of $\phi$ (respectively $h$),
            (b) $G_C$ is the minimal subset of subgoals of $Q$ such that $G_C$, $\phi_C$ and $h_C$ satisfy Property 3.1
            (c) It is not possible to extend $\phi$ and $h$ to $\phi_C'$ and $h_C'$ s.t. (b) is satisfied and $G_C'$, as defined in
                (b), is a subset of $G_C$.
 Return $\mathcal{C}$
```

Figure 3.1: First phase of the MiniCon algorithm: Forming MCDs.

The algorithm for creating the MCDs is shown in Figure 3.1. Consider the application of the algorithm to our example with the query *Q1* and the views *V4*, *V5*, and *V6*. The MCDs that will be created are shown in Table 3.2.

We first consider the subgoal *cites(x,y)* in the query. As discussed above, the algorithm does not create a MCD for *V4* because clause C2 of Property 3.1 would be violated (the property would require that *V4* also cover the subgoal *sameTopic(x,y)* since *b* is existential in *V4*). For the same reason, no MCD will be created for *V4* even when we consider the other subgoals in the query.

Table 3.2: MCDs formed as part of our example of the MiniCon Algorithm

$V(\overline{Y})$	h	Φ	G
$V5(c,d)$	$c\rightarrow c$, $d\rightarrow d$	$x\rightarrow c$, $y\rightarrow d$	3
$V6(f,f)$	$f\rightarrow f$, $h\rightarrow f$	$x\rightarrow f$, $y\rightarrow g$	1,2,3

In a sense, the MiniCon algorithm shifts some of the work done by the combination step of the bucket algorithm to the phase of creating the MCDs. The bucket algorithm will discover that *V4* is not usable for the query when combining the buckets. However, the bucket algorithm needs to discover this many times (each time it considers *V4* in conjunction with another view), and every time it does so, it uses a containment check, which is much more expensive. Hence, as we show in the next section, with a little more effort spent in the first phase, the overall performance of the MiniCon algorithm outperforms the bucket algorithm and the inverse-rules algorithm.

Another interesting observation is the difference in performance in the presence of repeated occurrences of the same predicate in the views or the query. For the bucket algorithm repeated occurrences lead to larger buckets, and hence more combinations to check in the second phase. For the inverse-rules algorithm, repeated occurrences mean there are more expansions to check in the second phase. In contrast, the MiniCon algorithm can more often rule out the consideration of certain occurrences of a predicate due to violations of Property 3.1.

Remark 3.3: When we construct a MCD *C*, we must determine the set of subgoals of the query G_C that are covered by the MCD. The algorithm includes in G_C only the *minimal* set of subgoals that are necessary in order to satisfy Property 3.1. To see why this is not an obvious choice, suppose we have the following query and views:

Q1'(x) :− *cites(x,y),cites(z,x), inSIGMOD(x)*

V7(a) :− *cites(a,b), inSIGMOD(a)*

V8(c) :− *cites(d,c), inSIGMOD(c)*

One can also consider including the subgoal *inSIGMOD(x)* in the set of covered subgoals for the MCD for both *V7* and *V8*, because *x* is in the domain of their respective variable mappings anyway. However, our algorithm will not include *inSIGMOD(x)*, and will instead create a special MCD for it.

The reason for our choice is that it enables us to focus in the second phase only on rewritings where the MCDs cover *mutually exclusive* sets of subgoals in the query, rather than overlapping subsets. This yields a more efficient second phase. □

3.3.2 Combining the MCDs

Our method for constructing MCDs pays off in the second phase of the algorithm, where we combine MCDs to build the conjunctive rewritings. In this phase we consider combinations of MCDs, and for each valid combination we create a conjunctive rewriting of the query. The final rewriting is a union of conjunctive queries.

The following property states that the MiniCon algorithm need only consider combinations of MCDs that cover pair-wise disjoint subsets of subgoals of the query. The proof of the property follows from the correctness proof of the MiniCon algorithm.

Property 3.2: Given a query Q, a set of views $\mathcal{V}$, and the set of MCDs $\mathcal{C}$ for Q over the views in $\mathcal{V}$, the only combinations of MCDs that can result in non-redundant rewritings of Q are of the form $C_1, \ldots, C_l$, where

D1. $G_{C_1} \cup \ldots \cup G_{C_l} = Subgoals(Q)$,

For every $i \neq j$, $G_{C_i} \cap G_{C_j} = \varnothing$. □

The fact that we only need to consider sets of MCDs that provide partitions of the subgoals in the query drastically reduces the search space of the algorithm. Furthermore, even though we do not discuss it here, the algorithm can also be extended to output the rewriting in a compact encoding that identifies the common sub-expressions of the conjunctive rewritings, and therefore leads to more efficient query evaluation. We note that had we chosen the alternate strategy in Remark 3.3, clause D2 would not hold.

Given a combination of MCDs that satisfies Property 3.2, the actual rewriting is constructed as shown in Figure 3.2.

In the final step of the algorithm we tighten up the rewritings by removing redundant subgoals as follows. Suppose a rewriting Q' includes two atoms A_1 and A_2 of the same view V, whose MCDs were C_1 and C_2, and the following conditions are satisfied: (1) whenever A_1 (respectively A_2) has a variable from Q in position i, then A_2 (respectively A_1) either has the same variable or a variable that does not appear in Q in that position, and (2) the ranges of φ_{C_1} and φ_{C_2} do not overlap on existential variables of V. In this case we can remove one of the two atoms by applying to Q' the homomorphism τ that is (1) the identity on the variables of Q and (2) is the most general unifier of A_1 and A_2. The underlying justification for this optimization is discussed in (Levy et al. 1995), and it can also be applied to the bucket algorithm and the inverse-rules algorithm.

Even after this step, the rewritings may still contain redundant subgoals. However, removing them involves several tests for query containment; both inverse-rules algorithm and the bucket algorithm require these removal steps as well.

```
procedure **combineMCDs**($C$)
/* $C$ are MCDs formed by the first step of the algorithm. */
/* Each MCD has the form ($h_C$, $V(\overline{Y})_C$, $\phi_C$, $G_C$, $EC_C$). */
 Given a set of MCDs, $C_1, \cdots, C_n$, we define the function $EC$ on $Vars(Q)$ as follows:
 If for $i \neq j$, $EC_{\varphi_i}(x) \neq EC_{\varphi_j}(x)$, define $EC_C(x)$ to be one of them arbitrarily but consistently across all
    $y$ for which $EC_{\varphi_i}(y) = EC_{\varphi_i}(x)$
 Let $Answer = \varnothing$
 For every subset $C_1, \cdots, C_n$ of $C$ such that $G_{C_1} \cup G_{C_2} \cup \ldots \cup G_{C_n} = subgoals(Q)$ and for every $i \neq j$,
    $G_{C_i} \cap G_{C_j} = \varnothing$
    Define a mapping $\phi_i$ on the $\overline{Y_i}$ 's as follows:
      If there exists a variable $x \in Q$ such that $\phi_i = y$
        $\Psi_i(y) = x$
      Else
        $\Psi_i$ is a fresh copy of $y$
      Create the conjunctive rewriting
        $Q'(EC(\overline{X})) :- V_{C_1}(EC(\Psi_1(\overline{Y_{C_1}}))), \ldots, V_{C_n}(EC(\Psi_1(\overline{Y_{C_n}})))$
        Add $Q'$ to $Answer$.
 Return $Answer$
```

Figure 3.2: MiniCon second phase: Combining the MCDs.

In our example, the algorithm will consider using $V5$ to cover subgoal 3, but when it realizes that there are no MCDs that cover either subgoal 1 or 2 without covering subgoal 3, it will discard $V5$. Thus the only rewriting that will be considered is

$Q1'(x) :- V6(x,x)$.

Constants in the query and views: When the query or the view include constants, we make the following modifications to the algorithm. First, the domain and range of φ_C in the MCDs may also include constants. Second, a MCD also records a (possibly empty) set of mappings ψ_C from variables in $Vars(Q)$ to constants.

When the query includes constants, we add the following condition to Property 3.1:

- If a is a constant in Q it must be the case that either (1) $\varphi_C(a)$ is a distinguished variable in $h_C(V)$ or (2) $\varphi_C(a)$ is the constant a.

When the views have constants, we modify Property 3.2 as follows:

- We relax clause C1: a variable x that appears in the head of the query must either be mapped to a head variable in the view (as before) or be mapped to a constant a. In the latter case, the mapping $x \rightarrow a$ is added to ψ_C.
- If $\varphi_C(x)$ is a constant a, then we add the mapping $x \rightarrow a$ to ψ_C. (Condition C2 only applies to existential variables, and therefore if $\varphi_C(x)$ is a constant that appears in the body of V but not in the head, a MCD is still created).

Next, we combine MCDs with some extra care. Two MCDs, C_1 and C_2, both of whom have x in their domain, can be combined only if they (1) either both map x to the same constant, or (2) one (e.g., C_1) maps x to a constant and the other (e.g., C_2) maps x to distinguished variable in the view. Note that if C_2 maps x to an existential variable in the view, then the MiniCon algorithm would never consider combining C_1 and C_2 in the first place, because they would have overlapping G_C sets. Finally, we modify the definition of EC, such that whenever possible, it chooses a constant rather than a variable.

The following theorem summarizes the properties of the MiniCon algorithm. Its full proof is given in Appendix A.

> **Theorem 3.1:** Given a conjunctive query Q and conjunctive views $\mathcal{V}$, both without comparison predicates or constants, the MiniCon algorithm produces the union of conjunctive queries that is a maximally-contained rewriting of Q using $\mathcal{V}$. □

It should be noted that the worst-case asymptotic running time of the MiniCon algorithm is the same as that of the bucket algorithm and of the inverse-rules algorithm after the modification described in Section 3.2.2. In all cases, the running time is $O(nmM)^n$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views.

The next section describes experimental results showing the differences between the three algorithms in practice.

3.4 Experimental Results

The goal of our experiments was twofold. First, we wanted to compare the performance of the bucket algorithm, the inverse-rules algorithm, and the MiniCon algorithm in different circumstances. Second, we wanted to validate that MiniCon can scale up to large number of views and large queries. Our experiments considered three classes of queries and views: (1) chain queries, (2) star queries and (3) complete queries, all of which are well known in the literature (Steinbrunn et al. 1997).

To facilitate the experiments, we implemented a random query generator which enables us to control the following parameters (1) the number of subgoals in the queries and views, (2) the number of variables per subgoal, (3) the number of distinguished variables, and (4) the degree to which predicate names are duplicated in the queries and views. The results are averaged over multiple runs generated with the same parameters (at least 40, and usually more than 100). All graphs either contain 95% confidence intervals or the intervals were less than twice as thick as the line in the graph and were thus excluded. An important variable to keep in mind throughout the experiments is the number of rewritings that can actually be obtained.

In most experiments we considered queries and views that had the same query shape and size. Our experiments were all run on a dual Pentium II 450 MHz running Windows NT 4.0 with 512MB RAM. All of the algorithms were implemented in Java and compiled to an executable.

3.4.1 Chain Queries

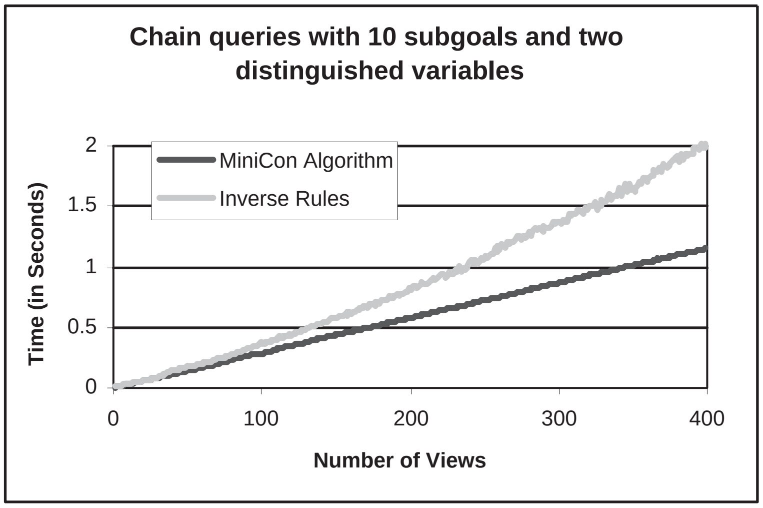


Figure 3.3: Running times for chain queries with two distinguished variables in the views. It shows that the MiniCon algorithm and the inverse-rules algorithms both scale up to hundreds of views. The MiniCon algorithm outperforms the inverse-rules algorithm by a factor of 2.

In the context of chain queries we consider several cases. In the first case, shown in Figure 3.3, only the first and last variables of the query and the view are distinguished. Therefore, in order to be usable, a view has to be identical to the query, and as a result there are very few rewritings. The bucket algorithm performs the worst, because of the number and cost of the query containment checks it needs to perform (it took on the order of 20 seconds for 5 views of size 10 subgoals, and hence we do not even show it on the graph). The inverse-rules algorithm and the MiniCon algorithm scale linearly in the number of views, but the MiniCon algorithm outperforms the inverse-rules algorithm by a factor of about 2 (and this factor is independent of query and view size). In fact, the MiniCon algorithm can handle more than 350 views with 10 subgoals each in less than one second. Since executing a query over 350 sources would likely take considerably more time than one second, this should be sufficiently fast.

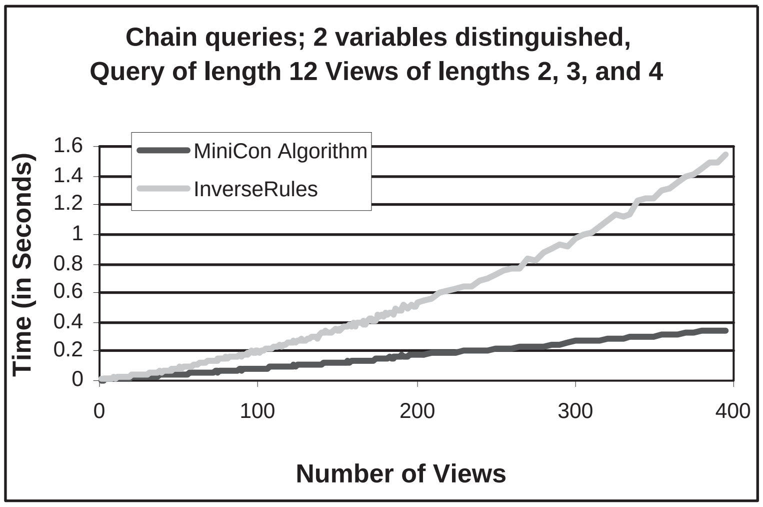


Figure 3.4: Running times for chain queries where the views are of lengths 2, 3 and 4, and the query has 12 subgoals.

The difference in the performance between the inverse-rules algorithm and the MiniCon algorithm in this context and in others is due to the second phases of the algorithms. In this phase, the inverse-rules algorithm is searching for a unification of the subgoals of the query with heads of inverse rules. The MiniCon algorithm is searching for sets of MCDs that cover all the subgoals in the query, but cover pair-wise disjoint subsets. Hence, the MiniCon algorithm is searching a much smaller space, because the number of subgoals is smaller than the number of variables in the query. Moreover the MiniCon algorithm is performing better because in the first phase of the algorithm it already removed from consideration views that may not be usable due to violations of Property 3.1. In contrast, the inverse-rules algorithm must try unifications that include such views and then backtrack. The amount of work that the inverse-rules algorithm will waste depends on the order in which it considers the subgoals in the query when it unifies them with the corresponding inverse rules. If a failure appears late in the ordering, more work is wasted. The important point to note is that the optimal order in which to consider the subgoals depends heavily on the specific views available and is, in general, very hard to find. Hence, it would be hard to extend the inverse-rules algorithm such that its second phase would compare in performance to that of the MiniCon algorithm.

In the second case we consider, shown in Figure 3.4, the views are shorter than the query (of lengths 2, 3 and 4, while the query has 12 subgoals).

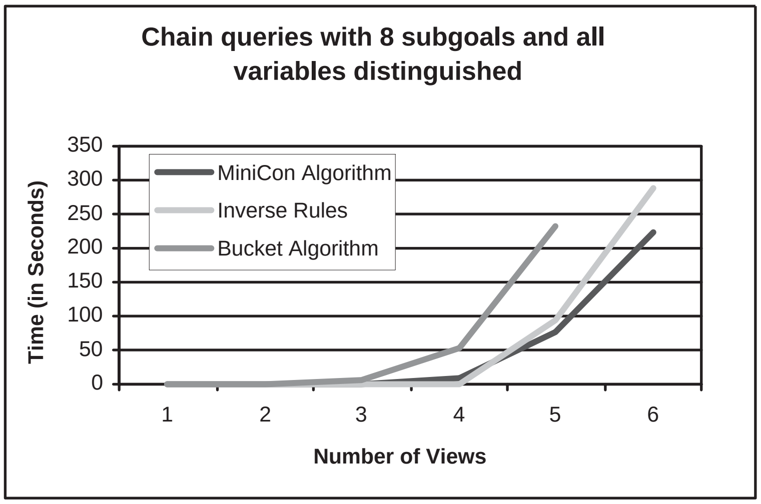


Figure 3.5: Running times for chain queries where all variables in the views are distinguished. The containment check required by the bucket algorithm causes it to be roughly twice as slow as either the MiniCon algorithm or inverse-rules algorithm.

Finally, as shown in Figure 3.5, we also considered another case in which all the variables in the views are distinguished. In this case, there are many rewritings (often more than 1000), and hence the performance of the algorithms is limited because of the sheer number of rewritings. Since virtually all combinations produce contained rewritings, any complete algorithm is forced to form a possibly exponential number of rewritings; for queries and views with 8 subgoals, the algorithms take on the order of 100 seconds for 5 views. The graph in Figure 3.5, shows that on average the MiniCon algorithm performs better than the inverse-rules algorithm by anywhere between 10% and 25%. However, in this case the variance in the results is very high, and hence it is hard to draw any general conclusions. (The confidence intervals cannot be shown in the graph without cluttering it.) The reason for the large variance is that some of the queries in the workload have a huge number of rewritings (and hence take much more time), while others have a very small number of rewritings. Other experiments showed that the savings for the MiniCon algorithm over the inverse-rules algorithm, as expected, grew with the number of views and the number of subgoals in the query; this is because the number of combinations that was considered was much higher and thus the smaller search space that the MiniCon algorithm considered was much more evident.

3.4.2 Star and Complete Queries

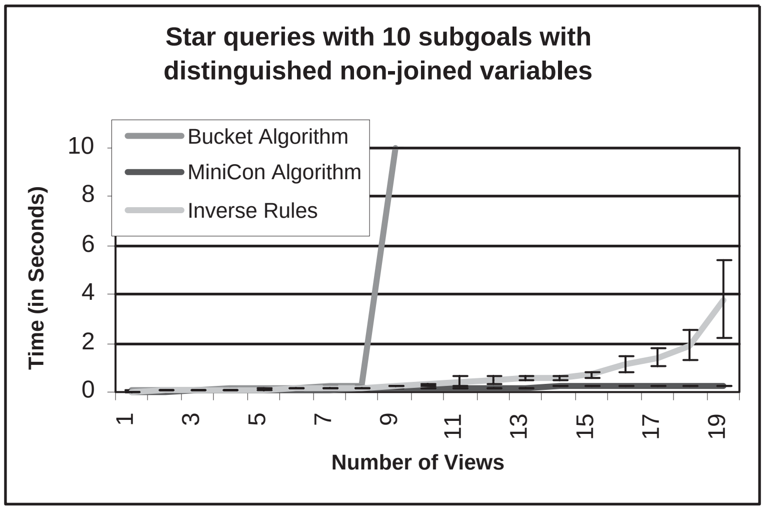


Figure 3.6: Running times for star queries, where the distinguished variables in the views are those not participating in the joins. The MiniCon algorithm significantly outperforms the inverse-rules algorithm.

In star queries, there exists a unique subgoal in the query that is joined with every other subgoal, and there are no joins between the other subgoals. In the cases of two distinguished variables in the views or all view variables being distinguished, the performance of the algorithms mirrors the corresponding cases of chain queries. Hence, we omit the details of these experiments. Figure 3.6 shows the running times of the inverse-rules algorithm and the MiniCon algorithm in the case where the distinguished variables in the views are the ones that do not participate in the joins. In this case, there are relatively few rewritings. We see that the MiniCon algorithm scales up much better than the inverse-rules algorithm. For 20 views with 10 subgoals each, the MiniCon algorithm runs 15 times faster than the inverse-rules algorithm. Here the explanation is that the first phase of the MiniCon algorithm is able to prune many of the irrelevant views, whereas the inverse-rules algorithm discovers that the views are irrelevant only in the second phase, and often it must be discovered multiple times.

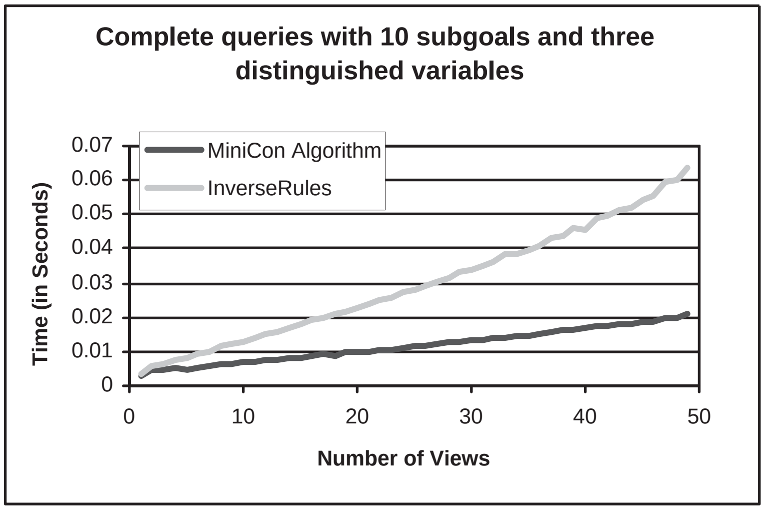


Figure 3.7: Running times for complete queries where three variables are distinguished. As in Figure 3.6, the MiniCon algorithm significantly outperforms the inverse-rules algorithm

An experiment with similar settings but for complete queries is shown in Figure 3.7. In complete queries every subgoal is joined with every other subgoal in the query. As the figure shows, the MiniCon algorithm outperforms the inverse-rules algorithm by a factor of 2.3 for 20 views, and by a factor of 3 for 50 views, which is less of a speedup than with of star queries. The explanation for this is that there are more joins in the query, and thus the inverse-rules algorithm is able to detect useless views earlier in its search because failures to unify occur more frequently. Finally, we also ran some experiments on queries and views that were generated randomly with no specific pattern. The results showed that the MiniCon algorithm still scales up gracefully, but the behavior of the inverse-rules algorithm was too unpredictable (though always worse than the MiniCon algorithm) due to the nature of when the algorithms discover that a rule cannot be unified.

3.4.3 Summary of Experiments

In summary, our experiments showed the following points. First, the MiniCon algorithm scales up to large numbers of views and significantly outperforms the other two algorithms.

This point is emphasized by Table 3.3, where we tried to push the MiniCon algorithm to its limits. The table considers the number of subgoals and number of views that the MiniCon algorithm is able to process given 10 seconds. In some cases, the algorithm can handle thousands of views, which is a magnitude that was clearly out of reach of previous algorithms.

Table 3.3: The number of views that the MiniCon algorithm can process in under 10 seconds in various conditions

Query type	Distinguished	# of subgoals	# of views
Chain	All	3	45
Chain	All	12	3
Chain	Two	5	9225
Chain	Two	99	115
Star	Non Joined	5	12235
Star	Non Joined	99	35
Star	Joined	10	4520
Star	Joined	99	75

Second, the experiments showed that the bucket algorithm performed much worse than the other two algorithms in all cases. More interesting was the comparison between the MiniCon algorithm and the inverse-rules algorithm. In all cases the MiniCon algorithm outperformed the inverse-rules algorithm, though by differing factors. In particular, the performance of the inverse-rules algorithm was very unpredictable. The problem with the inverse-rules algorithm is that it discovers many of the interactions between the views in its second phase, and the performance in that phase is heavily dependent on the order in which it considers the query subgoals. However, since the optimal order depends heavily on the interaction with the views, a general method for ordering the subgoals in the query is hard to find. Finally, all three algorithms are limited in cases where the number of resulting rewritings is especially large since a complete algorithm must produce a possibly exponential number of rewritings.

Although we have shown how to expect MiniCon to behave in a large number of classes of queries and views, in order to draw conclusions for how MiniCon would perform in a real data integration application we would need access both to the query workload and to the data sources. In particular, while we expect that many sources would have existential variables and thus lead to faster query rewriting, we would need to do a thorough investigation.

3.5 Comparison Predicates

The effect of comparison predicates on answering queries using views is quite subtle. If the views contain comparison predicates but the query does not, then the MiniCon algorithm without any changes still yields the maximally-contained query rewriting. On the other hand, if the query contains comparison predicates, then it follows from (Abiteboul et al. 1998) that there can be no algorithm that returns a maximally-contained rewriting, even if we consider rewritings that are recursive Datalog programs (let alone unions of conjunctive queries).

In this section we present an extension to the MiniCon algorithm that would (1) always find only correct rewritings (2) find the maximally-contained rewriting in many of the common cases in which comparison predicates are used, and (3) is guaranteed to produce the maximally-contained rewriting when the query contains only *semi-interval* constraints, i.e., when all the comparison predicates in the query are of the form $x \leq c$ or $x < c$, where x is a variable and c is a constant (or they are all of the form $x \geq c$ or $x > c$). We refer to this algorithm as MiniCon IP. We show experiments demonstrating the scale up of the extended algorithm. Finally, we show an example that provides an intuition for which cases the algorithm will not capture.

In our discussion, we refer to the set of comparison subgoals in a query Q as $I(Q)$. Given a set of variables $\overline{X}$, we denote by $I_{\overline{X}}(Q)$ the subset of the subgoals in $I(Q)$ that includes (1) only variables in $\overline{X}$ or constants and (2) contains at least one existential variable of Q. Intuitively, $I_{\overline{X}}(Q)$ denotes the set of comparison subgoals in the query that *must* be satisfied by the view if

$\overline{X}$ is the domain of a MCD. We assume without loss of generality that $I(Q)$ is logically closed, i.e., that if $I(Q) \models g$, then $g \in I(Q)$. We can always compute the logical closure of $I(Q)$ in time that is quadratic in the size of Q (Ullman 1989).

We make three changes to the MiniCon algorithm to handle comparison predicates. First, we only consider MCDs C that satisfy the following conditions:

1. If $x \in Vars(Q)$, $\phi_C(x)$ is an existential variable in $h_C(V)$ and y appears in the same comparison atom as x, then y must be in the domain of ϕ_C.
2. If $\overline{X}$ is the set of variables in the domain of the mapping ϕ_C, then $I(h_C(V)) \models \phi_C(I_{\overline{X}})$.

The first condition is an extension of Property 3.1, and the second condition guarantees the comparison subgoals in the view logically entail the relevant comparison subgoals in the query. Because of the second condition, the only subgoals in $I_{\overline{X}}(Q)$ that may not be satisfied by V must include only variables that ϕ_C maps to distinguished variables of V. As a result, such a subgoal can simply be added to the rewriting after the MCDs are combined.

The second change is that we disallow all MCDs that constrain variables to be incompatible with the variables they map in the query. For example, if a query has a subgoal $x > 17$ and a MCD maps x to a view variable a, and $a < 5$ is in the view, then we can ignore the MCD.

The third change we make to the MiniCon algorithm is the following: after forming a rewriting Q' by combining a set of MCDs, we add the subgoal $EC(g)$ for any subgoal of $I(Q)$ that is not satisfied by Q'.

Example 3.1: Consider a variation on our running example, where the predicate *year* denotes the year of publication of a paper.

$Q2(x) :- inSIGMOD(x), cites(x,y), year(x,r1), year(y,r2), r1 \geq 1990, r2 \leq 1985$

$V9(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1), year(b,s2), s2 \leq 1983$

$V10(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1), year(b,s2), s2 \leq 1987$

Our algorithm would first consider $V9$ with the mapping $\{x \rightarrow a,\ y \rightarrow b,\ r1 \rightarrow s1,\ r2 \rightarrow s2\}$. In this case, the subgoal $r2 \leq 1985$ is satisfied by the view, but $r1$

≥ 1990 is not. However, since $s1$ is a distinguished variable in $V9$, the algorithm can create the rewriting:

$Q2'(x) :- V9(x,r1),\ r1 \geq 1990$

When the algorithm considers a similar variable mapping to $V10$, it will notice that the constraint on $r2$ is not satisfied, and since it is mapped to an existential variable in $V10$, no MCD is created. □

Example 3.2: The following example provides an intuition for which rewritings our extended algorithm will not discover. Consider the following query and view:

$Q3(u) :- e(u,v),\ u \leq v$

$V11(a) :- e(a,b),\ e(b,a)$

The algorithm will not create any MCD because the subgoal $u \leq v$ in the query is not implied by the view. However, the following is a contained rewriting of $Q3$.

$Q3'(u) :- V11(u)$

In general, in order to find a containment mapping in the presence of comparison predicates, (Klug 1988) shows that we must find a mapping for every ordering of the variables. For example, we must consider two different containment mappings, depending on whether $a \leq b$ or $a > b$. In each of these mappings, the subgoal $e(u,v)$ may be mapped to a different subgoal. Our algorithm will only find rewritings in which the target of the mapping for a subgoal in the query is the same for any possible order on the variables. □

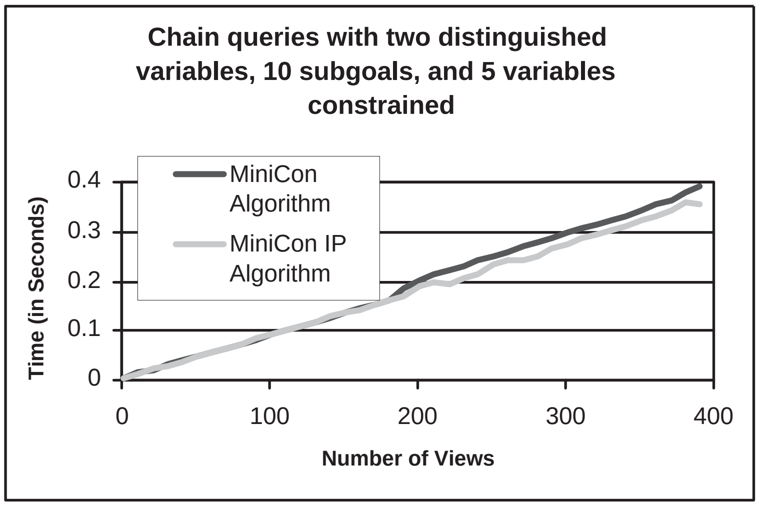


Figure 3.8: Experiments with the MiniCon algorithm and comparison predicates. The query and view shapes are the same as in Figure 3.3. The graph shows that adding comparison predicates does not appreciably slow the MiniCon algorithm, and the additional views that can be pruned cause the algorithm to speed up overall.

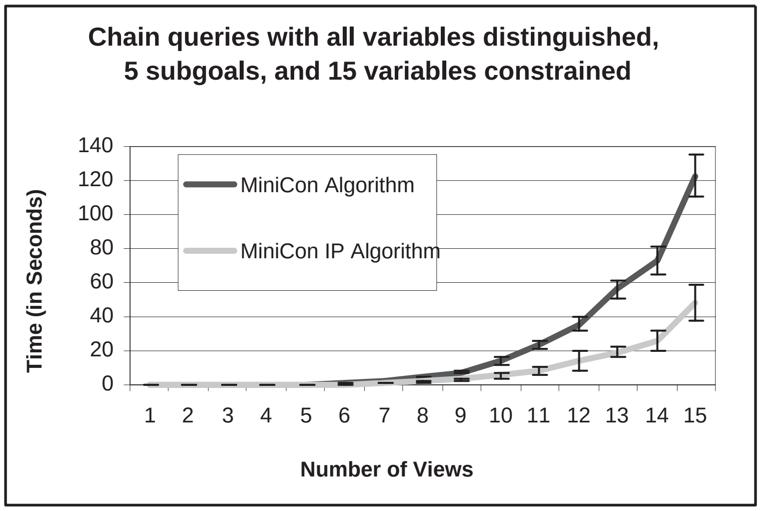


Figure 3.9: Running times for the MiniCon algorithm and comparison predicates when all of the variables in the views are distinguished

Figure 3.8 and Figure 3.9 show sample experiments that we ran on the extended algorithm in the case of chain queries. In the experiments, we took the identical queries and views and added a number of comparison subgoals of the form $x < c$ or $x > c$ to the queries under consideration by MiniCon IP.

The experiments show that the same trends we saw without comparison predicates appear here as well. In general, the addition of comparison predicates reduces the number of rewritings because more views can be deemed irrelevant. This is illustrated in Figure 3.9 where all of the variables in the views are distinguished and therefore without comparison predicates there would be many more rewritings. However, since the comparison predicates reduce the number of relevant views, the algorithm with comparison predicates scales up to a larger number of views. In Figure 3.8, the number of rewritings is very small, but the addition of the overhead to deal with comparison predicates does not appreciably slow the MiniCon algorithm.

3.6 Cost-Based Query Rewriting

The previous sections considered the problem of answering queries using views for the context of data integration, where the incompleteness of the data sources required that we consider the union of all possible rewritings of the query. In this section we show how the principles underlying the MiniCon Algorithm can also be used for answering queries using views in the context of query optimization (as in (Tsatalos et al. 1996) (Chaudhuri et al. 1995)), and in the process, shed some light on query optimization with views. The fundamental difference in this context is that we want the *cheapest rewriting* of the query using the views. Since the views are assumed to be complete (i.e., include all the tuples satisfying their definition) and since we are looking for an equivalent rewriting, we can limit ourselves to a single rewriting.

The following example shows how considering cost affects the result of a rewriting algorithm.

Example 3.3: Suppose we have the following query and views:

$Q4(x,y) :- e1(x,y), e2(y,z), e3(z,x)$

$V12(a,b,c) :- e1(a,b), e2(b,c)$

$V13(d,e,f) :- e2(d,e), e3(e,f)$

$V14(g) :- e1(g,h), e3(i, g)$

If the join of *e1* and *e3* is very selective, the cheapest rewriting of the query may be the following (assuming the subgoals are joined from left to right):

Q4'(x,y) :- *V14(x), V12(x,y,z), V13(y,z,x)*

Here, the view *V14* does not contribute to the *logical* correctness of the query, but only to reducing the cost of the query plan. The MiniCon Algorithm would not consider *V14(x)* because it would not create a MCD for *V14*, since Property 3.1 would not be satisfied. □

In general, the problem of answering queries using views in the context of query optimization requires that we consider views for two different roles: the logical correctness of the query, and the reduction in the cost of the rewriting. In fact, it is shown in (Chirkova et al. 2001) that the optimal query execution plan may include an exponential (in the size of the query and schema) number of views in the second role, while it follows from (Levy et al. 1995) that the number of views in the first role is bounded by the number of subgoals in the query.

We proceed in two steps. In Section 3.6.1 we show how the information captured in MCDs can be used to improve the bottom-up dynamic-programming algorithm used in (Tsatalos et al. 1996) for query optimization using materialized views. However, the algorithm we describe in Section 3.6.1 only considers views that contribute to the logical correctness of the rewriting, and therefore may not produce the optimal rewriting. In Section 3.6.2 we show how we can augment the resulting rewriting with cost-reducing views. Note that the approach we describe in Section 3.6.2 is inherently heuristic, and its goal is to avoid the exhaustive enumeration whose cost (according to (Chirkova et al. 2001)) would be prohibitive.

3.6.1 Modifying GMAP to Consider MCDs

In the context of query optimization, we may have access to the database relations in addition to the views. In order to uniformly treat database relations and views, we assume that for every database relation E we define a view of the form $V_E(\overline{X})$:- $E(\overline{X})$, where $\overline{X}$ is a tuple of distinct variables. In our running example we will assume that we do not have access to the database relations.

We first briefly recall the principles underlying the GMAP algorithm (Tsatalos et al. 1996), and then describe how we modify it to exploit MCDs. The GMAP algorithm is a modification of System-R style bottom-up dynamic programming, except that the optimizer builds query execution plans by accessing a set of views, rather than a set of database relations. Hence, in addition to the meta-data that the query optimizer has about the materialized views (e.g., statistics, indexes) the optimizer is also given as input the query expressions defining the views.

The GMAP algorithm begins by considering only views that can be used in a rewriting of the query (e.g., pruning views that refer to relations not mentioned in the query or do not apply necessary join predicates). The algorithm distinguishes between *partial query execution plans* of the query and *complete execution plans*, that provide an equivalent rewriting of the query using the views. The enumeration of the possible join trees terminates when there are no more unexplored partial plans.

The GMAP algorithm grows the plans by combining a partial plan (using all join methods) with an additional view. A partial plan P is pruned from further consideration if there is another plan P' such that (1) P' is cheaper than P, and (2) P' contributes the same or more to the query than P. Informally, a plan P' contributes more to the query than the plan P if it covers more of the relations in the query and selects more of the attributes that are needed further up the query tree.

Our algorithm precedes the join enumeration phase by the creation of MCDs, but it considers only a subset of the views that were considered in the data integration context.

In our discussion, we use the following notation to make use of the variable mappings used in procedure **combineMCDs** (Figure 3.2). Given a set of MCDs, $C = C_1, \ldots, C_l$, we denote by $VtoQ_C$, the set of atoms $V_{C_1}(EC(\Psi_1(\overline{Y_{C_1}}))), \ldots, V_{C_l}(EC(\Psi_l(\overline{Y_{C_l}})))$, as defined in procedure **combineMCDs**. $VtoQ_C$ effectively creates a set of atoms of the heads of the views in C, such that the atoms use the variables of Q whenever possible. Hence, $VtoQ_C$ makes explicit exactly which join predicates need to be applied between view atoms in the rewriting. So, in our example, if C_1 denotes the set of MCDs created for the views in the rewriting $Q4'$, then $VtoQ_{C_1}$ is $V12(x,y,z)$, $V13(y,z,x)$, $V14(x)$.

Given a query Q and a set of views, $V_1, \ldots, V_n$, our algorithm proceeds as follows:

1. We prune from further consideration any view V for which there does not exist a variable mapping Ψ from the variables of V to the variables of Q, such that for every subgoal $g \in V$, $\Psi(g)$ is a subgoal in Q. (This condition is similar to that of a containment mapping (Chandra et al. 1977), except that we do not require that Ψ map the head of V to the head of Q.) Views that do not satisfy this condition cannot be part of an equivalent rewriting of Q using the views. In our example, if we also had a view defined as:

   $V15(m,n)$:- $e1(m,n), e4(m)$

   then we would prune $V15$ because it cannot be part of an equivalent rewriting of Q (the subgoal $e4$ cannot be mapped to Q).
2. With the views selected in the first step, we construct the MCDs as described in Section 3.3.1.1. In our example we would create MCDs for $V12$ and $V13$, but we do not create a MCD for $V14$ because it does not satisfy Property 3.1.
3. We now begin the bottom-up construction of *candidate solutions*. A candidate solution is a query execution plan over the views, which may either be a partial or complete plan for the query.[2]

   a. For the base case, we start with plans that access a single view. Specifically, for every MCD C, we create the atom $VtoQ_{\{C\}}$. We then select the best access path to (the single atom in) $VtoQ_{\{C\}}$. In our example, we create the atoms $V12(x,y,z)$ and $V13(y,z,x)$.

   b. With every candidate solution P, we associate a subset of the subgoals of the query, denoted by P_G. Intuitively, this set specifies which subgoals in the query are covered by the solution P, and this information is gleaned from the MCDs. In the base case, the set P_G associated with the candidate solution constructed for the view in MCD C is G_C.

[2] We describe the algorithm for the case where we construct only left-linear trees, but the generalization to arbitrary bushy trees is straightforward.

We combine a candidate solution P with a candidate solution (of size 1) P' only if the union of P_G and P'_G contains strictly more subgoals than either P_G or P'_G. Using the information in P_G and P'_G enables us to significantly prune the number of candidate solutions we consider compared to the GMAP algorithm. For example, suppose our example also included the view: $V16(k,l) :- e1(k,l)$ and we had a partial solution, P, that included the single atom $V12(x,y,z)$. Then, we would not combine P with $V16(x,y)$ since $V16$ does not cover any more subgoals than $V12$. On the other hand, we would consider adding $V13(y,z,x)$ to P since $V13$ covers $e3$, which is not covered by P, and P covers $e1$ which is not covered by $V13$.

Given the views in P (whose corresponding MCDs are $\mathcal{C}$) and the view V in P', whose MCD is C_V, we compute $VtoQ_{\{\mathcal{C} \cup \{C_V\}\}}$. This tells us exactly which join predicates need to be applied between P and P' (specifically, whenever P and P' share a variable, a join predicate needs to be applied). We will try combining P and P' using every possible join method for every join predicate that needs to be applied.

c. As in the GMAP algorithm, we distinguish complete solutions, which correspond to equivalent rewritings of the query using the views, and partial solutions which can possibly be extended to complete solutions. Furthermore, as in GMAP, we compare every pair of candidate solutions P and P'. If P is both cheaper than P' and contributes as much or more to the query, then we prune P'. For example, if we had two candidate solutions $P1$, which consists of $V12(x,y,z)$ and the candidate solution $P2$ which consists of $V16(x,y)$, if $P1$ is cheaper than $P2$ we would prune $P1$ because $P1$ is both cheaper than $P2$ and contributes more than $P1$. However, if $P2$ is cheaper than $P1$, we would prune neither candidate solution because $P1$ contributes more than $P2$.

d. We terminate when there are no new combinations of partial solutions to be explored.

3.6.2 Adding Cost-Reducing Views

As stated earlier, the algorithm in Section 3.3 may not produce the cheapest plan because it only considers views that are needed for the logical correctness of the plan, and not cost-reducing views. (Note, however, that the algorithm will always find a plan if one exists even when we do not have access to the database relations). In this section we describe a heuristic approach to augmenting the plan produced in the previous section with cost-reducing views. Informally, we consider each cost-reducing view in turn, and try to place it in the places in the plan where it may have an effect. For example, consider the view $V14(x)$ in our example. This view can only be useful if it is placed before the atom $V12(x,y,z)$ (in order to reduce the number of values of x) or after the atom $V12(x,y,z)$ (to reduce the size of the join with $V13(y,z,x)$. However, $V14(x)$ is useless if placed after $V13(y,z,x)$.

We denote the plan produced by the algorithm in the previous section by P_{mg}. Recall that we are considering left-linear plans in our description. We create *cost-reducing view atoms* as follows:

1. As in the previous section, we consider only views that can be part of an equivalent rewriting of the query using the views.

2. We create MCDs for these views, except that we do *not* require the MCDs to satisfy Property 3.1. Denote the resulting MCDs by $C_1, \ldots, C_k$. In our example we would create MCDs for $V12(x,y,z)$, $V13(y,z,x)$, $V14(x)$.

3. Let the set of MCDs corresponding to the views in the plan P_{mg} be $\mathcal{C}_{mg}$. For every MCD C_j, $1 \leq j \leq k$, we compute $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$, and we denote by U_j the atom corresponding to C_j in $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$ (recall that $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$ computes an atom for every MCD). We will now try to insert the atoms $U_1, \ldots, U_k$ in the plan P_{mg}.

4. With every join operation in P_{mg} we can associate a set of variables, specifically, the variables that occur in the sub-tree of the join operator. The positions in P_{mg} that are *relevant* to the atom U_j are the join operators beginning with the first operator whose

variable set includes any of the variables in U_j, and ending with the first join operator that includes all the variables in U_j.

For every j, $1 \leq j \leq k$, we proceed as follows. We consider the cheapest plan P'_{mg}, that results from inserting U_j in one of the positions relevant to U_j. If a variable in U_j appears in the left-most leaf of the join tree, then we also consider the plan in which U_j is the left child of the first join operator in the plan. If P_{mg} is cheaper than P_{mg}, we replace P_{mg} by the plan P'_{mg}.[3]

5. We continue iterating through the cost-reducing view atoms until no change is made to the resulting plan.

In our example, we would consider placing the atom $V14(x)$ as the first or second left-most leaf of the tree (i.e., either before $V12(x,y,z)$ or immediately after it).

It is important to note that our algorithm may still not obtain the cheapest plan. The main reason is that we are beginning from the plan P_{mg}, and only modifying it locally, while the cheapest plan may actually be an augmentation of a plan that was found to be more expensive than P_{mg} in the cost-based join enumeration. It is possible to consider applying our algorithm to several plans from the cost-based join enumeration, rather than only to the cheapest one. However, in general, obtaining the cheapest plan may involve a prohibitively expensive search.

3.7 Related Work

Algorithms for rewriting queries using views are surveyed in (Halevy 2001). Most of the previous work on the problem focused on developing algorithms for the problem, rather that on studying their performance. In addition to the algorithms mentioned previously, algorithms have been developed for conjunctive queries with comparison predicates (Yang et al. 1987), queries and views with grouping and aggregation (Cohen et al. 1999; Grumbach et al. 1999; Gupta et al.

---

[3] For ease of exposition, we chose to describe a relatively conservative condition on the positions in which we can insert a cost-reducing view atom. Several further optimizations are possible the most obvious of which is that we would not insert a cost-reducing view atom in a plan *after* all the joins performed in the view have already been performed in the plan.

1995; Srivastava et al. 1996), queries over semi-structured data (Calvanese et al. 1999; Papakonstantinou et al. 1999), and OQL queries (Florescu et al. 1996). The problem of answering queries using views has been considered for schemas with functional and inclusion dependencies (Duschka et al. 1997c; Gryz 1998), languages that query both data and schema (Miller 1998), and disjunctive views (Afrati et al. 1999). Clearly, each of the above extensions to the basic problem represents an opportunity for a possible extension of the MiniCon algorithm. Two works (Abiteboul et al. 1998; Grahne et al. 1999) examine the complexity of finding all the possible answers from a set of view extensions. They show that if the views are assumed to be complete, then finding the maximal set of answers is NP-hard in the size of the data. Hence, finding a maximally-contained rewriting may not be possible if we consider query languages with polynomial data complexity. Mitra (Mitra 1999) developed a rewriting algorithm that also captures the intuition of Property 3.1, and thus would likely lead to better performance than the bucket algorithm and the inverse-rules algorithm. He also considered an optimization similar to our method for removing redundant view subgoals.

Several works discussed extensions to query optimizers that try to make use of materialized views in query processing (Afrati et al. 2001; Chaudhuri et al. 1995; Tsatalos et al. 1996) (Bello et al. 1998; Popa et al. 2000; Zaharioudakis et al. 2000). In some cases, they modified the System-R style join enumeration component (Chaudhuri et al. 1995; Tsatalos et al. 1996), and in others they incorporated view rewritings into the rewrite phase of the optimizer (Popa et al. 2000; Zaharioudakis et al. 2000). These works showed that considering the presence of materialized views did not negatively impact the performance of the optimizer. However, in these works the number of views tended to be relatively small. In (Afrati et al. 2001) the authors consider the problem of finding the most efficient rewriting of the query using a set of views, in the context of query optimization. The paper considers three specific cost models, and for each describes an algorithm that produces the cheapest plan. The algorithm we describe in Section 3.6 is independent of a particular cost model, and can incorporate the models described in (Afrati et al. 2001). In addition, our algorithm can also handle cost models that consider relation sizes, special orders and specific join implementations, as done in traditional query optimizers.

In (Popa et al. 2000), the authors consider a more general setting where they use a constraint language to describe views, physical structures and standard types of constraints.

A commercial implementation of answering queries using views is described for Oracle 8i in (Bello et al. 1998). Their algorithm works in two phases. In the first phase, the algorithm applies a set of rewrite rules that attempt to replace parts of the query with references to existing materialized views. The result of the rewrite phase is a query that refers to the views. In the second phase, the algorithm compares the estimated cost of two plans: the cost of the result of the first phase, and the cost of the best plan found by the optimizer that does *not* consider the use of materialized views. The optimizer chooses to execute the cheaper of these two plans. The main advantage of this approach is its ease of implementation, since the capability of using views is added to the optimizer without changing the join enumeration module. On the other hand, the algorithm considers the cost of only one possible rewriting of the query using the views, and hence may miss the cheapest use of the materialized views.

3.8 Conclusions

This chapter makes two important contributions. First, we present a new algorithm for answering queries using views, and second, we present the first experimental evaluation of such algorithms. We began by analyzing the two existing algorithms, the bucket algorithm and the inverse-rules algorithm, and found that they have significant limitations. We developed the MiniCon algorithm, a novel algorithm for answering queries using views, and showed that it scales gracefully and outperforms both existing algorithms. As a result of our work, we have established that answering queries using views can be done efficiently for large-scale problems. Finally, we described an extension of our algorithm to handle comparison predicates, and showed that the techniques underlying the MiniCon algorithm are also useful for the context of cost-based query optimization using views.

We close by briefly discussing another important extension of the MiniCon algorithm. In data integration applications, where views represent data sources, we often have limited access patterns to the data. For example, if Amazon.com has a relation *Book(title, price)*, we cannot ask for all tuples in the relation. Instead, we need to provide a value for the title in order to get a price. The problem of answering queries using views in this context has been considered in

(Duschka et al. 1997c; Kwok et al. 1996; Lambrecht et al. 1999; Rajaraman et al. 1995). In (Rajaraman et al. 1995) it is shown that when we consider equivalent rewritings, the rewriting may be *longer* than the query. In (Duschka et al. 1997c) it is shown that if we are looking for the maximally-contained rewriting, it may have to be a recursive Datalog program over the views.

The MiniCon algorithm can be adapted in a straightforward fashion to the presence of binding patterns. Specifically, we can follow the same strategy of (Duschka et al. 1997c), where inverse rules were augmented by *domain rules*. In our case, we produce the rewriting by the MiniCon algorithm by first ignoring the binding pattern limitations. Then we add domain rules, and augment the rewriting by adding domain subgoals where necessary.

Hence, given that the mediated schema is related to the source schema through LAV mappings, MiniCon can provide a fast method of rewriting the queries. In Chapter 4, Chapter 5, and Chapter 6 we consider how mediated schemas are created.

Chapter 4
Creating a Mediated Schema

4.1 Introduction

In Chapter 3 we discussed how to answer queries when the mediated schema is related to the source schemas using Local-As-View (LAV) mappings. We assumed that the mediated schema had been created in some other process. In this chapter, we consider how to create the mediated schema and the mappings to the source schemas for data integration.

Mechanisms for creating a mediated schema out of sources have previously been studied. Such projects generally focus on how to resolve conflicts such as synonyms and homonyms in relation names or how to ensure that the resulting schema is valid in a particular data model – e.g., SQL. Batini, Lenzerini, and Navathe provide a survey of such methods (Batini et al. 1986). Buneman, Davidson, and Kosky (Buneman et al. 1992) provide a general theory of what it means to merge two source schemas to create a third schema. Others have approached the problem from a more pragmatic point of view, such as the Database Design and Evaluation Workbench (Rosenthal et al. 1994) that allow users to manipulate schemas, including combining multiple views into one schema. Still others have created formal criteria for when two schemas consist of the same information, both for data integration and other applications (Hull 1984; Kalinichenko 1990; Miller et al. 1993). But none of these papers have tackled the problem we describe in this chapter: given two schemas, how should we create a mediated schema *and also* the mappings from the mediated schema to the sources.

Most research on querying data integration systems, such MiniCon (see Chapter 3), assumes that the mediated schema has been created elsewhere and the mappings from the mediated schema to the source schemas can be expressed in a particular language such as LAV or GAV. However, systems constructed by a priori choosing GAV or LAV as the mapping language restrict how the sources can be related to the mediated schema. This means that the mediated schema may be less than ideal, since a mediated schema can only include information that can be mapped to the data sources. For example, it might mean that the mediated schema has information duplicated because there is no single representation of that information in the mediated schema that can be mapped to the two representations in the two overlapping sources.

Others have previously noted that LAV and GAV have different expressive power, and indeed are incomparable if no constraints are present in the mediated schema (Calì et al. 2002). As well, additional, richer mapping languages such as GLAV (Friedman et al. 1999) and BAV (McBrien et al. 2003) have been created. However, previous work has not examined how the expressive power of relationships between elements in the source schemas affects the expressive power required in the mediated schema to source schema mappings.

Regardless of how the mediated schema is created, the main goal of data integration is to have users be able to query multiple databases without knowing where the data comes from. Hence it is imperative that there be a common representation in the mediated schema of concepts that come from different data sources and overlap. Thus, building the mediated schema requires knowing how the source schemas are related to one another.

> **Example 4.1:** In a data integration system helping passengers make airline reservations, the mediated schema should include a unified representation of airfares, regardless of whether the airfare came from the travel agent websites Travelocity, or Expedia, or from an airline website, or from some other source; the user just wants to find the cheapest fare, regardless of its source. In addition, the mediated schema may contain concepts having no direct correlation in any other local source. For example, if Travelocity gives the gate information about a specific flight and Expedia does not, we may still wish to include the gate information in the airfare representation. □

We propose that to create a mediated schema and the mapping from mediated schema to source schemas, we must first understand the relationships between the sources and how this drives the mediated schema creation. Our goal in this chapter is to study mediated schema creation based on the above observations. We describe a simple version of the problem: Given two schemas, E and F, and a mapping Map_{E_F} that specifies how E and F are related to one another, create a mediated schema G and mappings Map_{G_E} and Map_{G_F} where the mappings can be used to translate queries over G into queries over E and F. How to find how two schemas are is another problem, called schema matching; it is a major topic of ongoing research and is not covered in this thesis; see (Rahm et al. 2001) for a recent survey and (He et al. 2003), (Kang et al. 2003) and (Dhamankar et al. 2004) for examples of work since then. In this chapter we concentrate only on data integration, though this work could be extended to other data management problems.

Section 4.2 describes mediated schema creation for data integration of relational schemas. Section 4.3 refines the problem for the specific case where Map_{E_F} is a *conjunctive mapping*. Section 4.4 describes alternate definitions and metrics. Section 4.5 discusses extensions to the problem for relational schemas, including Section 4.5.1 which focuses on creating a mediated schema for more than two sources. Section 4.6 discusses LAV and GAV as choices for data integration systems in light of this work. Section 4.7 concludes.

4.2 Generic Mediated Schema Creation

In this section, we describe relational mediated schema creation for data integration without pinning down the language for the mapping. That is, given two relational source schemas, E and F, what other inputs are needed to create the mediated schema, and what kinds of properties would we like the mediated schema to have? Allowing overlapping concepts in the mediated schema to be accessed in the same fashion is a matter of convenience – if the goal were simply to preserve all the information from the input sources, then the mediated schema could be simply a union of the source schemas. Because of this, traditional metrics for schema design – such as ensuring that the schema is in a normal form – are orthogonal to this issue, so we must create a new set of criteria. Because we want these correctness criteria to be as independent of the type of mapping between sources as possible, we define the correctness criteria first for a very general class of mappings. In later sections, we build on this definition for the specific case of a *conjunctive mapping* relating E and F to allow us to see what the semantics of a specific mapping imply for the mediated schema and the mappings from the mediated schema to the source schemas.

Recall the definition of a relational schema from Definition 2.1. We define a function $Order$ that describes the position of each attribute in its relation, that is, given a relational schema Σ

$Order(a_{rj}) = j$ for all $r \in \Sigma$ and $a_{rj} \in attributes_r$. We also define a function n that maps each element in Σ (e.g., relation or attribute) into a name. We occasionally abuse notation by equating a schema object with its name.

Notice that by "relation," we mean a schema for a relation. We use the word "state" to refer to a set of tuples that conforms to a relation (i.e. its schema). Thus, we define the *state of a relation* r (or *relation state*) to be a set or bag of m-tuples, where r has m attributes. The *state of a schema* is a set of states of relations, one relation state for each relation in the schema. For a relational schema Σ, $I(\Sigma)$ is the set of all states of Σ. Thus, each $\sigma_\Sigma \in I(\Sigma)$ denotes a state of Σ. For each relation $r \in \Sigma$, we use $\pi_r(\sigma)$ to denote the state of r in schema state σ (i.e., the projection of σ on r).

Given that we do not consider integrity constraints, states of relational schemas are closed under union. Formally, given schema Σ, if $\sigma_{\Sigma 1} \in I(\Sigma)$ and $\sigma_{\Sigma 2} \in I(\Sigma)$, then $\sigma_{\Sigma 1} \cup \sigma_{\Sigma 2} \in I(\Sigma)$. Since a schema is a set of relations, taking the union of two states $\sigma_{\Sigma 1}$ and $\sigma_{\Sigma 2}$ of a schema Σ amounts to taking the union of the states of the relations in $\sigma_{\Sigma 1}$ and $\sigma_{\Sigma 2}$. That is, for each relation r in Σ, $\pi_r(\sigma_{\Sigma 1} \cup \sigma_{\Sigma 2}) = \pi_r(\sigma_{\Sigma 1}) \cup \pi_r(\sigma_{\Sigma 2})$.

Without loss of generality, we assume that the relation names in different schemas are disjoint. That is, given relations $r1$ and $r2$ of schemas $\Sigma 1$ and $\Sigma 2$ respectively, we assume that $r1$ and $r2$ have different names. This assumption can be enforced simply by assigning a unique name to each schema and appending the unique schema name to the name of each relation in that schema. It follows that if $\Sigma 1$ and $\Sigma 2$ are schemas, then $\Sigma 1 \cup \Sigma 2$ is a (well-formed) schema that consists of the union of the relations of $\Sigma 1$ and $\Sigma 2$. Since every two schemas are disjoint, we can represent the state of the union of two schemas by a pair of states, one for each schema in the union. That is, $I(\Sigma 1 \cup \Sigma 2) = I(\Sigma 1) \times I(\Sigma 2)$, so $\forall \sigma_{\Sigma 1} \in I(\Sigma 1)\ \forall \sigma_{\Sigma 2} \in I(\Sigma 2)$, $(\sigma_{\Sigma 1}, \sigma_{\Sigma 2}) \in I(\Sigma 1 \cup \Sigma 2)$.

We are now ready to define the information provided as input to the process of mediated schema creation. Given two relational source schemas, E and F, the goal is to create a mediated schema G over E and F. In addition to E and F, two other inputs are required: (1) how concepts in E and F overlap and (2) how concepts independently of interest in E and F should be expressed in the mediated schema along with the overlapping information. We express (1) as a query over E and F that returns the same concept from E or F. We call such a query an *intersection,* which we define in Definition 4.2. We express (2) as queries over E and F. Each such query is called a *component* and is defined in Definition 4.3. Although intersections and components are both expressed as queries, we give them different names to clarify their different roles. We do not want to restrict our definitions of correctness to a particular query language. Instead we define the notion of a generic query which is an arbitrary function over a given relational schema, and therefore can be specialized to any concrete query language.

Definition 4.1: (Generic Query). A *generic query* Q over a relational schema Σ is a function $I(\Sigma_Q) \rightarrow I(T)$ where Σ_Q is the set relations in Q and a subset of the relations in Σ and T is a relation (which may not be in Σ), called the *target relation* for Q. Given a state $\sigma' \in I(\Sigma_Q)$, *the value returned by generic query* Q on σ' is $Q(\sigma')$. If $\sigma \in I(\Sigma)$, then we define $Q(\sigma)$ to be equal to $Q(\pi_{\Sigma_Q}(\sigma))$. □

For example, in a conjunctive query, the head of the query is the target, T, the relations in the body of the query are Σ_Q, and the query's conjunctive formulas define the function $I(\Sigma_Q) \rightarrow I(T)$.

Definition 4.2: (Intersection). Let schema $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$. An *intersection* is a generic query $Q_N = qn_1 \cup \ldots \cup qn_m$ s.t. $m > 1$ and each generic query qn_i is over some schema Σ_j, $1 \leq j \leq n$ and the target relations of all qn_i are the same. Given a state $\sigma \in I(\Sigma)$, the state of an intersection Q_N over Σ is $Q_N(\sigma)$. □

Example 4.2: an intersection for *Undergrad* may be described as follows, where *UBCUgrad* and *UWUgrad* are in different schemas:

Undergrad(FirstName,LastName) :– *UBCUgrad(FirstName,LastName,year), year < 5*

Undergrad(FirstName, LastName) :– *UWUgrad(FirstName,LastName, major)* □

Definition 4.3: (Component). A *component* of a schema Σ is a generic query Q_C over Σ. Given a state $\sigma \in I(\Sigma)$, the *state of a component* Q_C of is $Q_C(\sigma)$. □

Example 4.3: a component may be used to express that the concept "seniors at UBC" should be represented separately in the meditated schema even though that information is not explicitly represented in the input schemas:

UBCSenior(FirstName,LastName):-UBCUgrad(FirstName,LastName,year), year=4□

Intersections and components may overlap in the relations they contain; for example, a relation may appear in the domain of both an intersection and a component. For example, *Undergrad* and *UBCSenior* in Example 4.2 and Example 4.3 both use the relation *UBCUgrad*. When this occurs, we want to eliminate the redundancy of the overlap to produce a minimal output schema, G. Therefore we require a formal way to describe that overlap, which we do using the concept of subsumption (Definition 4.4) to compare the states and schema definitions of intersections and components.

Definition 4.4: (Generic query subsumption). Let component Q_C and intersection Q_N be defined over the same schema Σ and have target relations name T_C and T_N respectively. Suppose T_C and T_N have the same name, and attributes$_{TC} \subseteq$ attributes$_{TN}$. Then Q_C is ***subsumed*** by Q_N if for all $\sigma \in I(\Sigma)$, $Q_C(\sigma) \subseteq \pi(Q_N(\sigma))$. □

Note that the definition of subsumption differs from the notion of containment because we allow for $Q_C(\sigma)$ to be a projection of $Q_N(\sigma)$.

Example 4.4: The component:

Grad(FirstName,LastName) :-UBCGrad(FirstName, LastName)

is subsumed by the intersection below:

Grad(FirstName,LastName):-UWGrad(FirstName,LastName, Office)

Grad(FirstName,LastName):-UBCGrad(FirstName,LastName) □

At this point we have all of the formalism required to describe the input. However, the input alone is not enough to solve the problem; we must define a set of criteria telling what the output should be. The output is the mediated schema G and how G is related to each of the source schemas. To relate G to the source schemas, we rely on the definition of a mapping in Definition 4.5.

> **Definition 4.5:** (Mapping). A mapping $Map_{\Sigma1\_\Sigma2}$ over schemas $\Sigma1$ and $\Sigma2$ is an expression that defines a subset of $I(\Sigma1) \times I(\Sigma2)$.[4] □

Definition 4.5 is broader than some of the common definitions of mappings. Hull (Hull 1986) would define a mapping $Map_{\Sigma1\_\Sigma2}$ over schemas $\Sigma1$ and $\Sigma2$ as a transformation from $I(\Sigma1) \rightarrow I(\Sigma2)$. In many cases a Hull mapping suffices. However, as we show later sometimes we require the additional expressiveness in Definition 4.5.

Consider a relation in the mediated schema that pulls information from more than one source relation. The reason for combining those source relations into one mediated schema relation is because we have identified an intersection or a component, Q_C, over the source relations that characterize information that should be put together. The mediated schema relation may contain more attributes than those required by that intersection or component over the source relations because there may be other intersections or components over other sets of source relations that overlap Q_C. We are interested in easily accessing each intersection or component. Thus, given a component or intersection defined by a generic query Q_C, the mediated schema should have a relation over which one can write a generic query Q that can be rewritten as an equivalent generic query Q' over the local sources. We call this a *canonical query for* Q_C. We express this formally in Definition 4.6. Note that we do not require that the canonical query be unique. Canonical queries are defined for any generic query over $E \cup F$, not just components or intersections.

[4] $Map_{\Sigma1\_\Sigma2}$ is a *function* if for every state $\sigma_1 \in I(\Sigma1)$ there is at most one state $\sigma_2 \in I(\Sigma2)$ s.t. $\langle\sigma_1, \sigma_2\rangle \in I(Map_{\Sigma1\_\Sigma2})$. If $Map_{\Sigma1\_\Sigma2}$ and its inverse are functions, then $Map_{\Sigma1\_\Sigma2}$ is *injective*. $Map_{\Sigma1\_\Sigma2}$ is *surjective* if $\forall \sigma_2 \in I(\Sigma2)$, $\exists\ \sigma_1 \in I(\Sigma1)$ s.t. $\langle\sigma_1, \sigma_2\rangle \in I(Map_{\Sigma1\_\Sigma2})$. $Map_{\Sigma1\_\Sigma2}$ is *total* if $\forall\ \sigma_1 \in I(\Sigma1)$, $\exists\ \sigma_2 \in I(\Sigma2)$ s.t. $\langle\sigma_1, \sigma_2\rangle \in I(Map_{\Sigma1\_\Sigma2})$. Examples of the definitions are included in Appendix B.

This situation is somewhat backwards from the normal situation. We are trying to ensure that we can capture all values for $E \cup F$ in G rather than that queries over G will get all values from $E \cup F$. Hence we restate certain answers (Definition 2.7) for our situation to avoid confusion: Given an instance $\sigma_{EF} \in$ I(Σ_{EF}), a tuple t is a certain answer for a generic query Q over G w.r.t. Map_{G_EF} if t is an element of $Q(\sigma_G)$ for every σ_G s.t. σ_{EF} s.t. $Map_{G\_EF}(\sigma_G, \sigma_{EF})$.

Definition 4.6: (Canonical queries). Let Q_N be a generic query over schema $\Sigma 2$ with target T; Q_N may be a component, an intersection, or any other query over $\Sigma 2$. Let $Map_{\Sigma 1\_\Sigma 2}$ be a mapping over schemas $\Sigma 1$ and $\Sigma 2$. We say that Q is a *canonical query* for generic query Q_N over schema $\Sigma 1$ and mapping $Map_{\Sigma 1\_\Sigma 2}$ if (1) the body of Q contains exactly one relation and (2) $\forall\ \sigma_{\Sigma 1} \in I(\Sigma 1)$, $\forall\ \sigma_{\Sigma 2} \in I(\Sigma 2)$ such that $Map_{\Sigma 1\_\Sigma 2}(\sigma_{\Sigma 1}, \sigma_{\Sigma 2})$, $Q(\sigma_{\Sigma 1})$ computes the certain answers to $Q_N(\sigma_{\Sigma 2})$. □

Example 4.5: Assume the intersection defining *Undergrad* in Example 4.2. If *MergedUndergrad* is the name of a relation in the mediated schema, and querying *MergedUndergrad* using the query Q yields all of the certain answers for *Undergrad*, then Q is a canonical query for *Undergrad*. □

We only want to allow relations in the input to be represented by the same relation in the mediated schema if they are related through the mapping. To define this formally we rely on the concept of *connected relations* in Definition 4.7. In Definition 4.8 we define a *partial order* on schemas to allow us to compare two schemas:

Definition 4.7: (Connected relations). Two relations r_1 and r_2 are *connected* in a set of generic queries M if

- $r_1 \in \Sigma_Q$ and $r_2 \in \Sigma_Q$ for some generic query $Q \in M$ over Σ_Q, or
- $r_1, r_3 \in \Sigma_{Q1}$ and $r_2, r_4 \in \Sigma_{Q2}$ for some generic queries $Q_1, Q_2 \in M$ and r_3 and r_4 are connected. □

Example 4.6: Suppose we had the intersection for *Undergrad* in Example 4.2 and the following component for *UWStudent*:

UWStudent(FirstName,LastName,Major) :– *UWUgrad(FirstName,LastName,major)*

UWStudent(FirstName,LastName,Major) :– *UWgrad(FirstName,LastName, major)*

UWGrad is connected to *UBCUgrad* because *UWGrad* appears with *UWUgrad* in the component *UWStudent* and *UWUgrad* appears with *UBCUgrad* in the intersection *Undergrad*. □

Definition 4.8: (Partial order of schemas). We define a partial order << over schemas as follows: Given two schemas $\Sigma 1$ and $\Sigma 2$, we define that $\Sigma 1 << \Sigma 2$ if the relations of $\Sigma 1$ are a subset of the relations in $\Sigma 2$ and for each relation $r1 \in \Sigma 1$ and $r2 \in \Sigma 2$ s.t. $n_{s1} = n_{s2}$, $attributes_{s1} \subseteq attributes_{s2}$. (Buneman et al. 1992) □

Given these definitions, we define the *mediated schema creation problem* as follows:

Definition 4.9: (Mediated schema creation problem). Given E, F, INT and $COMP$, where INT is a set of intersections over E and F and $COMP$ is a set of components over E and F, produce G, Map_{G_E}, and Map_{G_F} satisfying Mediated Schema Criteria (MSCs) 1-5 (below) where G is a schema, Map_{G_E} is a mapping over G and E, and Map_{G_F} is a mapping over G and F. □

To help us define MSC 1-5, we define a mapping Map_{G_EF} that represents the union of Map_{G_E} and Map_{G_F}. Let schema $EF = E \cup F$. Recall that a state $\sigma_{EF} \in I(EF)$ is a pair (σ_E, σ_F), where $\sigma_E \in I(E)$, $\sigma_F \in I(F)$. We define $Map_{G\_EF} = \{(\sigma_G, (\sigma_E, \sigma_F)) \in I(G) \times (I(E) \times I(F)) \mid (\sigma_G, \sigma_E) \in Map_{G\_E}$ and $(\sigma_G, \sigma_F) \in Map_{G\_F}\}$; in some sense, Map_{G_EF} represents the union of Map_{G_E} and Map_{G_F}. We are now ready to define the Mediated Schema Criteria (MSCs) as follows. We first define each one briefly and then examine each one in more detail:

MSC 1: **Completeness:** G is complete. That is, there are total, injective, functional mappings from $I(E)$ to $I(G)$ and from $I(F)$ to $I(G)$.

MSC 2: **Intersection accessibility:** For every intersection Q_N in INT, there exists a canonical query for Q_N over G and Map_{G_EF}.

MSC 3: **Component accessibility:** For every component Q_C in $COMP$ not subsumed by any intersection in INT, there exists a canonical query for Q_C over G and Map_{G_EF}.

MSC 4: **Connectivity required:** For every generic query $Q{:}I(\Sigma_Q) \rightarrow I(T)$ over EF where the relations in $Q = EF' \subseteq EF$ and $|EF'| > 1$, if there is a canonical query for Q over G and Map_{G_EF} then every pair of relations in Σ_Q is connected in Map_{E_F}.

MSC 5: **Minimality:** G is a minimal schema that satisfies MSCs 1-4. That is, there exists no G' satisfying MSCs 1-4 such that

1. $G' << G$ or $|G'| < G$ (i.e., G' contains fewer relations than G)
2. $\forall$ relations $g \in G \nexists$ sets of attributes $p_1 \subseteq attributes_g$, $p_2 \subseteq attributes_g$ s.t. $p_1 \cap p_2 = \emptyset$ and $\forall$ $\sigma_G \in I(G)$ $\pi p_1(\pi_g(\sigma_G)) \subseteq \pi p_2(\pi_g(\sigma_G))$

We now describe each MSC in more detail. We begin the discussion of each MSC by providing its label and then quoting the full text of the MSC.

MSC 1: Completeness: "That is, there are total, injective, functional mappings from $I(E)$ to $I(G)$ and from $I(F)$ to $I(G)$." *Information capacity* is a metric describing when information can be preserved in applications that span multiple schemas (Hull 1986). Miller, Ioannidis, and Ramakrishnan (Miller et al. 1993) show that data integration requires (1) the ability to query the source schemas E and F from the mediated schema G and (2) the ability to view through G all tuples stored under E and F. They show that these goals in turn require that G *dominates* EF – that there exists an information capacity preserving mapping from EF to G. An information capacity preserving mapping $M{:}\ I(EF) \rightarrow I(G)$ must be total, injective, and functional. Since E and F are disjoint schemas, there is a total, injective function from $E \cup F$ into G if Map_{E_G} and Map_{F_G} are total, injective functions. Hence completeness is equivalent to ensuring that the information capacity required by data integration – G dominates EF – is preserved. The notion of completeness in creating a merged or mediated schema is common, not just for information capacity but in other generic algorithms such as the specification by Buneman, Davidson, and Kosky (Buneman et al. 1992).

MSC 2: Intersection accessibility: "For every intersection Q_N in INT, there exists a canonical query for Q_N over G and Map_{G_EF}. MSC 2 ensures that overlapping information can be accessed uniformly." For example, in Example 4.1, we would define an intersection of Travelocity and Expedia that includes airfare information. In this case, MSC2 would require that there is a canonical query able to provide information about airfares, regardless of whether they come from Travelocity or Expedia.

MSC 3: Component accessibility: "For every component Q_C in $COMP$ not subsumed by any intersection in INT, there exists a canonical query for Q_C over G and Map_{G_EF}." MSC 3 ensures that additional non-overlapping information described by a component is expressed in a single relation. For example, in representing a common concept of "student" in the mediated schema when the source schemas only consist of relations for "grad student" and "undergrad student", the user may wish to include the office number of the graduate students in the student representation, even though there is no direct correlation between the grad and undergrad students and undergrad students do not have office numbers. A component corresponds to the notion of being a semantically necessary relationship (Rosenthal et al. 1994) – a concept that the user would like to have easy access to, but does not necessarily alter the information content of the schema. Because intersections and components may overlap, they may be defined in the same relation in G. Since an intersection is required to return information from all relevant sources, if there is an intersection that subsumes a component, the component may not be able to be retrieved separately from the intersection. Hence MSC 3 applies only to components that are not subsumed by an intersection.

MSC 4: Connectivity required: "For every generic query $Q{:}I(\Sigma_Q) \rightarrow I(T)$ over $\Sigma_Q \subseteq EF$ where $|\Sigma_Q| > 1$, if there is a canonical query Q' for Q over $\Sigma_{Q'} \subseteq G$ and Map_{G_EF} then every pair of relations in $\Sigma_{Q'}$ is connected in Map_{E_F}." MSCs 2 and 3 ensure that concepts defined in Map_{E_F} are not broken apart in G. By contrast, MSC 4 ensures that attributes in E and F should be represented by the same relation of G only if the input requires it. MSC 4 ensures this by requiring that the relations in EF be connected, which allows relations in related components or intersections to be included in the same relation in G. To determine what concepts in EF are being related to one another in G, we see what queries over EF we can answer by using a single relation in G – a canonical query.

The condition works as follows. We want to allow easy querying in G to each relation in EF, particularly since MSC 1 requires completeness. Therefore we restrict our consideration of what queries over EF should *not* be easily accessible in G to those queries that are over more than one relation in EF. The intuition for the rest of the statement is that there is a canonical query for a generic query Q only if Q is either a projection of or contained in an intersection or component, i.e., there is a reason for them to be related in G. MSC 4 specifies this intuition more generically and succinctly without referring to either projections or containment of queries by relying on the definition of connected relations (Definition 4.7).

MSC 5: Minimality: "G is a minimal schema that satisfies MSCs 1-4. That is, there exists no G' satisfying MSCs 1-4 such that (1) $G' << G$ or $|G'| < G$ (i.e., G' contains fewer relations than G), and (2) $\forall$ relations $g \in G \nexists$ sets of attributes $p_1 \subseteq attributes_g$, $p_2 \subseteq attributes_g$ s.t. $p_1 \cap p_2 = \varnothing$ and $\forall\ \sigma_G \in I(G)\ \pi p_1(\pi_g(\sigma_G)) \subseteq \pi p_2(\pi_g(\sigma_G))$. " We begin by considering the first requirement. In addition to requiring that there be no schema $G' << G$ satisfying the MSCs, we require that there is no schema G' satisfying the MSCs with fewer relations than G because we wish to ensure that concepts are split into as few relations as possible. The partial order (<<) alone does not guarantee this. The notion that a mediated schema must not include extraneous information is also common in mediated schema creation algorithms such as those surveyed in (Batini et al. 1986) and the generic merged schema creation algorithm of (Buneman et al. 1992). The second bullet ensures that no relation in E or F is represented in two different ways in the same relation in G.

4.3 *Creating a Relational Mediated from Conjunctive Mappings*

Section 4.2 gives correctness criteria for creating a mediated schema in data integration with respect to an arbitrary query language that satisfies the definition of a generic query (Definition 4.1). We now consider the more specific case of *conjunctive mappings*. We refine the case to conjunctive mappings to allow a thorough analysis of what the semantics of a particular type of mappings means for both the mediated schema and the mappings from the mediated schema to the source schemas. Section 4.3.1 defines conjunctive mappings. Section 4.3.2 refines the MSCs for conjunctive mappings. Section 4.3.3 defines G, Map_{G_E} and Map_{G_F} that satisfy the requirements in Section 4.3.2, and Section 4.3.4 proves that the requirements are indeed satisfied.

4.3.1 Conjunctive Mappings

We assume there are two input schemas, $E=\{e_1, \ldots, e_n\}$ and $F=\{f_1, \ldots, f_m\}$, and Map_{E_F}, a *mapping* that relates E and F. Map_{E_F} is a conjunctive mapping if it consists of a set of *conjunctive queries* (Definition 2.2). We consider conjunctive queries that contain only variables; i.e., the queries can contain no constants and hence no selection predicates.

In many scenarios, such as transforming data from a source schema to a target schema, the mapping Map_{E_F} would be used to constrain the set of states that can be valid simultaneously in E and F.

Example 4.7: Assume we are given two input schemas E and F, and Map_{E_F} was used to populate E with tuples from F. Map_{E_F} might be:

$e_1(x,y) :- f_1(x,y)$

$e_1(x,y) :- f_2(x,y)$

In this case, Map_{E_F} is an exact mapping; if there exists a tuple $f_1(1,2)$ or $f_2(1,2)$, there must exist a tuple $e_1(1,2)$, and there are no other tuples in e_1. This mapping constrains the valid states of E with respect to the states of F. □

GAV and LAV mappings are examples of mappings types that adhere to the open world assumption (Definition 2.6) on the instances of the schemas they relate; a tuple in the mediated schema state must appear in a tuple or set of tuples in one or more local sources. However, when creating the mediated schema for data integration, a direct mapping between E and F is inappropriate since the states of E and F are not required to be the same or even overlapping. Thus, the role of Map_{E_F} between a pair of source schemas is not to constrain the states of either E or F. Rather, Map_{E_F} describes how E and F are related to a helper schema, D:

Definition 4.10: (Helper schema). Given two schemas E and F, D is a helper schema for Map_{E_F} if Map_{E_F} constrains the states of E and F with respect to the states of D. □

4.3.1.1 Syntax of Conjunctive Mappings

A *conjunctive mapping* is a set of conjunctive queries restricted as follows. We require that each conjunctive query $Q \in Map_{E\_F}$ in a conjunctive mapping Map_{E_F} is over exactly one of E or F. We refer to each $Q \in Map_{E\_F}$ as a *mapping statement*.

We make some additional restrictions for ease of exposition. These three rules are not required in order for the mapping to be conjunctive, nor do they affect the expressive power of the queries allowed. The goal of the rules is to make it easier to name the attributes of the relations in G. Any set of conjunctive queries can be made to satisfy the restrictions simply by renaming variables. The restrictions are as follows:

1. If an IDB appears more than once in a mapping, then for each variable position of the IDB, the same variable name must be used in all appearances. Hence, definitions $q_5(x,y) :- e_1(x,y)$, and $q_5(z,y) :- f_2(x,y)$ cannot be in the same mapping since the variable in the first position of q_5 is named x in the first definition and z in the second. This restriction is made so that it is clear what the corresponding attribute should be named in G.
2. An existential variable name may appear in at most one mapping statement for a given IDB name. For example, $q_{15}(x) :- e_1(x,y)$ and $q_{15}(x) :- f_1(x,y)$ cannot exist in the same mapping since y is existential in both. This restriction is made so that it is clear what the corresponding attributes should be named in G.
3. Each relation name appears in at most one of E, F, and the IDB names of Map_{E_F}. This can be accomplished by doing a predicate re-name if necessary.

There are two additional restrictions that make mappings less powerful than the full range of conjunctive queries, but that we use to simplify the problem:

1. A relation can appear at most once within a mapping. For example, $q_{17}(x) :- e_1(x,y)$ and $q_{18}(w) :- e_1(w,u)$ cannot appear in the same mapping because they both map e_1. Similarly $q_{19}(x) :- e_1(x,y),\ e_1(y,z)$ cannot appear in a mapping because e_1 is again mapped twice.

2. An EDB must not repeat variables. For instance $q_7(x)$:− $e1(x,x)$ cannot be in a mapping because x appears twice in e_1.

To refer to the IDBs in a mapping we define the concept formally in Definition 4.11:

Definition 4.11: (IDBs in a mapping). We define $IDB(Map_{E\_F}) = \{idb \mid \exists\ ms \in Map_{E\_F}$ s.t. $IDB(ms) = idb\}$ □

4.3.1.2 Semantics of Conjunctive Mappings

Map_{E_F} defines the valid states of the helper schema, D. Specifically, we consider the schema D to be created by reifying the IDBs of Map_{E_F} into relations. We define the tuples of D as a function of the tuples in E and F, as follows.

Let D be the schema consisting of one relation for each IDB $j \in$ IDB(Map_{E_F}). That is, for each IDB $j \in$ IDB(Map_{E_F}), there exists a relation $r \in D$ s.t.

1. $n_r = n_j$
2. $attributes_r = Vars(j)$

By definition of conjunctive mapping, each mapping statement in Map_{E_F} consists of EDBs from exactly one of E or F. Thus, we can partition Map_{E_F} into two sub-mappings: Map_{E_D} and Map_{F_D}, where Map_{E_D} consists of all mapping statements that contain EDBs from E, and Map_{F_D} consists of all mapping statements that contain EDBs from F. So $Map_{E\_D} \cup Map_{F\_D} = Map_{E\_F}$.

Given states $\sigma_E \in I(E)$ and $\sigma_F \in I(F)$, we define $\sigma_D = Map_{E\_D}(\sigma_E) \cup Map_{F\_D}(\sigma_F)$. That is, the union of the queries in Map_{E_F} applied to σ_E and σ_F yield the tuples in σ_D.

4.3.2 Refining the MSCs to the Conjunctive Mapping Case

The mediated schema creation problem requires components and intersections as input. Since these are not explicitly defined in a conjunctive mapping Map_{E_F}, as defined in Section 4.3.1.1, we must determine how to tease intersections and components from conjunctive mapping Map_{E_F}.

The intersections are easy to determine: each IDB name induces an intersection which is comprised of all the statements with that IDB name; Example 4.8 declares an intersection on the first attributes of e_1 and f_1.

Example 4.8:

$q_2(x) :- e_1(x,y)$

$q_2(x) :- f_1(x,z)$ □

The mapping in Example 4.8 is just one of many possible between these two relations. Example 4.9 shows another possibility:

Example 4.9:

$q_2(x,y) :- e_1(x,y)$

$q_2(x,y) :- f_1(x,y)$ □

Example 4.9 declares an intersection on both the first and second attributes of e_1 and f_1. Note that because we have artificially restricted the conjunctive queries allowed in Map_{E_F} the variable y is now used for the second variable in both appearances of q_2, though this could be easily achieved through a variable renaming.

Because we need to ensure canonical queries for intersections, we must return to the notion of a canonical query. However, now that we have specified Map_{E_F} to be a conjunctive mapping, we can be more specific in the conjunctive statement of the MSCs. To simplify what follows, Definition 4.12 defines a canonical query for an IDB:

Definition 4.12: (Canonical query for an IDB). Given IDB i in $IDB(Map_{E\_F})$, let $MS_{idb} = \{ms_j \in Map_{E\_F} \mid \text{IDB}(ms_j) = idb\}$. If there exists a query Q over G such that Q is a canonical query for MS_{idb} w.r.t. and Map_{G_EF}, then we say that Q is a canonical query for idb. □

The components are slightly more difficult to determine because conjunctive queries must be safe; the variables that appear in the head can only include attributes that are in all of the statements with that IDB name. For example, in the definition of q_2 in Example 4.8, we cannot express that y should be present in a component by having it appear in the head of the first mapping statement, because it would then make the second mapping statement unsafe.

However, because we assume that the mediated schema is complete and we are not "breaking apart" relations from EF in G, we know that there exists some way to access the values of the existential variables in Example 4.8 via a canonical query, so there is no problem in accessing the component $e_1(x,y)$.

Now let us extend Example 4.8 slightly. Suppose E includes the relation $e_2(y,z)$ and we want the input to the mediated schema creation problem to include a component that includes the join of e_1 and e_2 which says that their attributes appear in one relation of the mediated schema. Suppose we also want to relate the tuples in the join of e_1 and e_2 to the tuples in f_1. We would want to create the mapping in Example 4.10:

Example 4.10:

$q_{17}(x,y,z) :- e_1(x,y), e_2(y,z)$

$q_{17}(x,z) :- f_1(x,z)$ □

However, this mapping gives q_{17} both an arity of two and an arity of three, which is counter to the definition of conjunctive queries. Instead we are forced to make the mapping as in Example 4.11 and simply require that all of the attributes of each mapping statement appear in the mediated schema in a single relation.

Example 4.11:

$q_3(x,z) :- e_1(x,y), e_2(y,z)$

$q_3(x,z) :- f_1(x,z)$ □

In contrast to the example with the existential y value in Example 4.8, in Example 4.11, we are not assured to have all the values for $e_1(x,y), e_2(y,z)$ accessible via a canonical query. Therefore we require that there be a canonical query for each projection-free component:

Definition 4.13: (Projection-free components). For each mapping statement ms let q equal $IDB(ms)$. We say that the *projection-free component* of ms, denoted pf_{ms}, is the query $q(Vars(ms)) :- body(ms)$, where $Vars(ms)$ and $body(ms)$ are the variables and body of ms, respectively. □

Notice that pf_{ms} is identical to ms except that all variables of pf_{ms} are distinguished. In addition, as shown in Section 4.4.2, the assumption that each mapping statement defines projection-free components allows G to contain fewer relations than it would otherwise. We discuss alternate semantics where we assume that mapping statements do not define projection-free components in Section 4.4.2.

Given the definitions of inputs, we can now refine the MSCs for the conjunctive mapping case as follows:

CreateMediatedSchema(E, F, Map_{E_F}**)** $\rightarrow$ G, Map_{G_EF} is said to satisfy the Conjunctive Mediated Schema Criteria (CMSCs) if:

CMSC 1: **Conjunctive completeness:** G is complete. There are total, injective, functional mappings from E to G and from F to G.

CMSC 2: **Conjunctive intersection accessibility:** For each IDB idb in $IDB(Map_{E\_F})$, let $MS_{idb} = \{ms_j \in Map_{E\_F} \mid \text{IDB}(ms_j) = idb\}$. Then there exists a query Q over G and Map_{G_EF} such that Q is a canonical query for MS_{idb}.

CMSC 3: **Conjunctive component accessibility:** For every mapping statement ms in Map_{E_F}, let $Q = q(Vars(ms))$:- $body(ms)$. If ms has an existential variable, then there exists a canonical query for Q over G and Map_{G_EF}.

CMSC 4: **Conjunctive connectivity required:** every query $Q{:}I(\Sigma_Q) \rightarrow I(T)$ over $\Sigma_Q \subseteq EF$ where $|\Sigma_Q| > 1$, if there is a canonical query Q' for Q over $\Sigma_{Q'} \subseteq G$ and Map_{G_EF} then for every pair of relations $r1$, $r2 \in EF'$ there exist mapping statements $ms1 \in Map_{E\_F}$ and $ms2 \in Map_{E\_F}$ such that $IDB(ms1) = IDB(ms2)$, $r1 \in body(ms1)$, and $r2 \in body(ms2)$.

CMSC 5: **Conjunctive minimality:** G is a minimal schema that satisfies CMSCs 1-4. That is, there exists no G' satisfying CMSCs 1-4 such that

1. $G' << G$ or $|G'| < G$ (i.e., G' contains fewer relations than G)
2. $\forall$ relations $g \in G$ $\nexists$ sets of attributes $p_1 \subseteq attributes_g$, $p_2 \subseteq attributes_g$ s.t. $p_1 \cap p_2 = \emptyset$ and $\forall$ $\sigma_G \in I(G)$ $\pi p_1(\pi_g(\sigma_G)) \subseteq \pi p_2(\pi_g(\sigma_G))$.

4.3.3 Definitions of G, Map_{G_E}, and Map_{G_F}

A solution to the mediated schema creation problem consists of a mediated schema G and mappings Map_{G_E}, and Map_{G_F} that together satisfy CMSC 1-5. In this section, we define such a G, Map_{G_E}, and Map_{G_F}. In Section 4.3.4 we prove that they do indeed satisfy CMSC 1-5.

4.3.3.1 Definition of G

The CMSCs essentially say that there are two sets of relations that must exist in G: (1) relations required to answer canonical queries (i.e., CMSCs 2 and 3) and (2) relations required to ensure completeness (i.e., CMSC 1). CMSC 5 requires that G contain the minimum number of relations. In essence, CMSC 5 works in opposition to the other CMSCs; it is trying to decrease the number of relations in G while the others are trying to add relations to G. We can determine the number of relations in G by analyzing which relations are required and how the number of relations can be minimized. Because some of the relations created by CMSCs 2 and 3 may be able to resolve some of CMSC 1's completeness requirements, we begin by considering CMSCs 2 and 3.

Let $G_{IDB} \subseteq G$ be a set of relations needed to satisfy CMSCs 2 and 3. From the definition of canonical queries (Definition 4.6), we know CMSCs 2 and 3 can be satisfied by creating G_{IDB} such that every mapping statement and every IDB has its own relation in G_{IDB}. In order to satisfy CMSC 5, however, we must see if a smaller number of relations will suffice. The only way that we can combine two relations, g_1 and g_2, required by CMSC 2 or 3 is to satisfy CMSC 4; g_1 and g_2 must be used in mapping statements for the same IDB. Since the input restrictions allow each relation in EF to appear in only one mapping statement, each relation in EF may be used in the definition of only one IDB. Similarly, by definition each mapping statement is used in defining exactly one IDB. Hence to satisfy CMSC 2 G_{IDB} must contain at least one relation per IDB in $IDB(Map_{E\_F})$.

To minimize G_{IDB}, if it is possible to answer the queries required by CMSC 3 using the relations required by CMSC 2 (which we show is possible in Section 4.3.3.2), then to satisfy CMSC 5 (conjunctive minimality) there must be no additional relations created to satisfy CMSC 3. Therefore G_{IDB} must contain exactly one relation per IDB in $IDB(Map_{E\_F})$ and no other relations. In addition, to satisfy CMSC 3 (Conjunctive component accessibility) each relation corresponding to an IDB idb in $IDB(Map_{E\_F})$ must also contain enough information to answer the canonical query on the projection-free component of the mapping statements defining idb.

Now we turn to the attributes of the relations in G_{IDB}. Let $MS_q = \{ms_j \in Map_{E\_F} \mid \text{IDB}(ms_j) = q\}$ and let $Vars(MS_q) = Vars(q) \cup (\bigcup_i existential(ms_i))$ where $ms_i \in MS_q$. In order to allow canonical queries over both the intersections and components, we require that a relation $gidb \in G_{IDB}$ retains an attribute for each variable used in the mapping statement used to create it. Formally, $\forall$ relations $g_{IDB} \in G_{IDB}$ if $n(g_{IDB}) = q$, where $q \in \text{IDB}(Map_{E\_F})$ we require that $attributes_{gIDB} = Vars(MS_q)$.

We are now ready to consider which relations in G_{IDB} can also be used to ensure CMSC 1 (Conjunctive completeness). The answer has to be considered for each relation $g \in G_{IDB}$ and hinges on the shape of the mapping statements used to create g. While this relies on Map_{EF_G} as defined in Section 4.3.3.2, Example 4.12 gives an intuition for which relations in E and F the relations in G_{IDB} can ensure completeness, and which relations in E and F must appear separately in G:

Example 4.12: Consider the following mapping:

$q_1(w) :- e_1(w)$

$q_1(w) :- f_1(w,u)$

$q_2(x) :- e_2(x,y)$

$q_2(x) :- f_2(x,z), f_3(x,v)$

Assume that e_1, e_2, f_1, f_2 and f_3 are the only relations in E and F. By the remarks above, G_{IDB} contains $q_1(w,u)$ and $q_2(x,y,z,v)$. Can the relations q_1 and q_2 be used to ensure the completeness of e_1, e_2, f_1, f_2, and f_3? Consider relation q_1. It defines the intersection $(q_1(w) :- e_1(w)) \cup (q_1(w) :- f_1(w,u))$. Since q_1 retrieves values of w from both e_1 and f_1, it is impossible to retrieve only the w values of e_1 from the relation in G_{IDB} that corresponds to q_1. Hence for completeness with respect to e_1, e_1 must be retained separately in G. f_1 is a little different since u is existential. In Section 4.3.3.2 we show how to write a query over q_1 to retrieve exactly the values needed in f_1; in essence the query requires a value for both w and u; so only values from f_1 are retained. Hence a relation corresponding to f_1 does not have to appear separately in G to maintain CMSC 1 (Completeness) since f_1 has an existential variable.

Examining q_2 we see by an argument similar to that for f_1 above it may possible to ask a query that returns the value of all tuples in e_2. However, relations corresponding to f_2 and f_3 must appear separately in G since the canonical query for the component defined by $q_2(x) :- f_2(x,z), f_3(x,v)$ only guarantees that tuples of f_2 and f_3 that have the same first value will be accessible. So these tuples cannot be retrieved separately from relation in G_{IDB} that corresponds to q_2. We formalize this notion of relations in EF whose completeness can be guaranteed through the relations in G_{IDB} in Definition 4.14 – mapping-included relations. □

Definition 4.14: (mapping-included relations). For relation $r \in EF$, if there exists a mapping statement $ms \in Map_{E\_F}$ s.t. ms contains an existential variable and $body(ms) = r$, then r is *mapping-included* in Map_{E_F}. □

In the definitions below, which explain the output of CreateMediatedSchema, we are going to explicitly document the output. We define a correspondence relation between the inputs and outputs of CreateMediatedSchema as follows:

Definition 4.15: (ξ). We define the correspondence relation ξ to $\subseteq \{EF \cup \text{IDB}(Map_{E\_F})\} \times G$. That is, $\xi(o,g)$ means that o is either a relation name in EF or an IDB name in Map_{E_F}. The intuition is that g is produced as output partially from o as input. □

Putting it all together we arrive at the following definition of well-formed mediated schema, which, as we show in Section 4.3.4, satisfies the CMSCs.

Definition 4.16: (Well-formed Mediated Schema). Given relational schemas E and F and mapping Map_{E_F}, a mediated schema G created by CreateMediatedSchema(E, F, Map_{E_F}) is *well-formed* if:

1. $\forall$ relations $e \in E$ s.t. e is not mapping-included, $\exists\ g \in G$ s.t. $n_e = n_g$, $attributes_e = attributes_g$, and $\xi(e,g)$. Similarly for all $f \in F$.
2. $\forall$ IDB names $q \in$ IDB(Map_{E_F}), $\exists\ g \in G$ s.t. $q = n_g$, $attributes_g = Vars(MS_q)$, and $\xi(q,g)$.
3. G contains no additional relations. □

Example 4.13:

$E = \{e_1(a,b), e_2(c,d)\}$

$F = \{f_1(e,f)\}$

$Map_{E\_F} = \{$

$q_1(x) :- e_1(x,y)$

$q_1(x) :- f_1(x,z)$

$\}$

$Vars(MS_{q1}) = \{x, y, z\}$. Since f_1 is mapping-included (z is existential and f_1 appears alone in the body of its mapping statement) f_1 is not included in G, and $G = \{e_1(a,b), e_2(c,d), q_1(x,y,z)\}$ □

4.3.3.2 Properties and Definition of Map_{G_E} and Map_{G_F}

We now must define Map_{G_E} and Map_{G_F} so that they retain enough information to create all of the canonical queries required in the CMSCs and satisfy the remaining CMSCs. We begin by considering CMSC1, the only CMSC that places a restriction on Map_{G_E} and Map_{G_F} other than the canonical queries. CMSC 1 requires a total, injective, functional mapping from E to G and F to G. Since E and F are disjoint schemas there is a total, injective function from $E \cup F$ into G if Map_{E_G} and Map_{F_G} are total, injective functions. None of the other CMSCs place any restrictions on Map_{E_G} or Map_{F_G}, so we turn to creating the canonical queries.

We express Map_{G_E} and Map_{G_F} as a subset of GLAV mappings (Friedman et al. 1999), as follows. We relate G and E through an intermediate schema, C, which represents the coverage of each mediated schema relation available from a particular source. Each mapping statement ms s.t. $body(ms)$ is over EF creates two views in Map_{G_EF}. The first view, lv_{ms}, is called a *local view definition* for ms. It is a conjunctive query from G to the intermediate schema C, and is included in the set of *local view definitions for* G, LV_G. The second view, gv_{ms}, is called a *global view definition* for ms. gv_{ms} is a function from E to C, and is included in the *global view definitions for* G, GV_G. Note that the views in Map_{G_EF} are sound but not complete; we illustrate why in Example 4.15 after we formally define LV_G and GV_G.

How can we translate queries using Map_{G_EF}? We give the intuition here, and we show formally why this is so in Theorem 4.2 after formally defining LV_G and GV_G. Our primary goal in showing the query rewriting scheme is to ensure that G and Map_{G_EF} satisfy the CMSCs. To show that, we must ensure that canonical queries for all intersections and projection-free components exist. As shown later by Lemma 4.5 and Lemma 4.6, the canonical queries required are conjunctive queries, so we consider only how to answer conjunctive queries. We answer a query Q over G by first using the local view definitions LV_G and then using the global view definitions, GV_G. Recall from Section 2.4 that a LAV view definition is a sound but incomplete view from the mediated schema to the source schemas, and answering queries using views can be used to rewrite a query from the mediated schema to source schemas in LAV. Hence we can use answering queries using views to rewrite a query Q over G into a query Q' over the intermediate schema C[5].

The global view definitions in GV_G are much like view definitions in GAV. Hence, query unfolding (Definition 2.5) can rewrite Q' over C into a query Q'' over EF. Putting this together with the rewriting step from the local view definitions, to rewrite queries over G into queries

[5] Because the queries in LV_G are significantly less expressive than conjunctive queries and the only queries that we are rewriting over are LV_G conjunctive queries, a simpler specialized algorithm could be created. However answering queries using views is guaranteed to be sound and complete in finding certain answers (Definition 2.7) since Map_{G_EF} is sound but not complete and LV_G and the queries over it are only conjunctive queries.

over EF answering queries using views is first applied to the local views and then view unfolding is applied to the global views. Now that we have an intuition for Map_{G_EF}, we are ready to define it formally. Recall that Definition 4.16 (well-formed mediated schema) guarantees that for each mapping statement ms with IDB name q, there exists some relation $rq \in G$ with name q and attributes = $Vars(MS_q)$.

> **Definition 4.17: (**Well-formed Mediated Schema Mapping). Let Map_{G_EF} be a mediated schema mapping created by CreateMediatedSchema(E, F, Map_{E_F}), where E and F are relational schemas and Map_{E_F} is a conjunctive mapping between E and F adhering to the requirements in Section 4.3.1.1. Map_{G_EF} is said to be *well-formed* if it consists of two sets of view definitions, LV_G and GV_G where LV_G provides a mapping from G to an intermediate schema C and GV_G provides a mapping from EF to C, s.t.:
>
> 1. $\forall$ relations $g \in G$ s.t. $\xi(e,g)$ for some $e \in EF$, let q_j be a fresh IDB name.
>
> $lv_g = q_j(attributes_g) :- g(attributes_g)$
>
> $gv_g = q_j(attributes_g) :- e(attributes_e)$ [6]
>
> $lv_g \in LV_G$
>
> $gv_g \in GV_G$
>
> 2. $\forall$ mapping statements $ms \in Map_{G\_EF}$ with IDB name q, let $rq \in G$ be a relation with name q and $\xi(q,rq)$. Let q_j be a fresh IDB name (i.e., q_j is an IDB name that appears in no other mapping statements in Map_{E_F} or in any other local view definitions or global view definitions in Map_{G_EF}).

[6] The relations g and e in the definition of lv_g and gv_g respectively are the same, from the definition of a well-formed mediated schema Definition 4.16. Similarly, $attributes_g = attributes_e$.

$lv_{ms} = q_j(Vars(ms))$:- rq

$gv_{ms} = q_j(Vars(ms))$:- $body(ms)$

$lv_{ms} \in LV_G$

$gv_{ms} \in GV_G$

3. LV_G and GV_G contain no views other than those required above.

Note that the cases here are the same cases as in the definition of the well-formed mediated schema (Definition 4.16); i.e., the first bullet here creates the mapping statements for the relations created in the first bullet, etc. □

Definition 4.18: ($ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G, Map_{G\_EF}$). The conjunctive mediated schema creation problem takes as input schemas E and F, and conjunctive mapping Map_{E_F}, and produces a schema G and mapping Map_{G_EF} that satisfy Definition 4.16 and Definition 4.17 respectively. □

An example of how to create LV_G and GV_G and then rewrite a query is shown in Example 4.14.

Example 4.14: Recall that in both LV_G and GV_G the left hand sides of the mappings consist of fresh IDB names – the sole point of the names is to match the concepts in LV_G and GV_G. As well, in the right hand side of GV_G the EDB names are the names of relations in the source schemas. In the right hand side of LV_G the EDB names are the names of relations in the mediated schema, which by construction of G are either the same as the names of the relations in E and F or the IDBs in $IDB(Map_{E\_F})$. Assume the schemas and mappings from Example 4.13. By Definition 4.17 LV_G and GV_G are as follows. Note that because relation names in G also appear both as relation names in E and F and in IDB names in $MapE\_F$, we prepend each relation name with the schema to which it belongs.

$LV_G = \{$

$q_2(x,y)$:- $G.q_1(x,y,z)$ // step 1 of Definition 4.17 for first m.s. in Map_{E_F}

$q_3(x,z)$:- $G.q_1(x,y,z)$ // step 1 of Definition 4.17 for second m.s. in Map_{E_F}

$q_4(a,b)$:- $G.e_1(a,b)$ // step 2 of Definition 4.17 for the first relation in E

$q_5(c,d)$:- $G.e_2(c,d)$ // step 2 of Definition 4.17 for the second relation in E

$q_6(e,f)$:- $G.f_1(e,f)$ // step 2 of Definition 4.17 for the sole relation in F

}

GV_G = {

$q_2(x,y)$:- $E.e_1(x,y)$ // step 1 of Definition 4.17 for first m.s. in Map_{E_F}

$q_3(x,z)$:- $F.f_1(x,z)$ // step 1 of Definition 4.17 for second m.s. in Map_{E_F}

$q_4(a,b)$:- $E.e_1(a,b)$ // step 2 of Definition 4.17 for the first relation in E

$q_5(c,d)$:- $E.e_2(c,d)$ // step 2 of Definition 4.17 for the second relation in E

$q_6(e,f)$:- $F.f_1(e,f)$ // step 2 of Definition 4.17 for the sole relation in F

}

Suppose that given these mappings and relations the query Q = $qnew(r)$:- $q1(r,s,t)$ was asked. The process to answer the query would be as follows:

1. Answering queries using views would be used over the view definitions in LV_G to find a maximally-contained rewriting for $qnew$ expressed over the intermediate schema. In this case, Q' = $qnew'(r)$:- $q2(r,u)$ ∪ $qnew'(r)$:- $q3(r,v)$.
2. The view definitions in GV_G expanded (see Definition 2.5) to rewrite Q' into a query Q'' over EF. Q'' = $qnew''(r)$:- $e1(r,u)$ ∪ $qnew''(r)$:- $f1(r,v)$.□

Example 4.15: In point 1 of Definition 4.17 it only makes sense to search for a maximally-contained rewriting if Map_{G_EF} is sound but not complete (e.g., the view definitions provide descriptions of what tuples they produce, but not every tuple that satisfies that description is produced by the view). In this example we show why Map_{G_EF} is sound but not complete. Consider the following mapping:

$q_{21}(x,y)$:- $e_1(x,y)$

$q_{21}(x,y)$:- $f_1(x,y)$

and assume that $e_1(a,b)$ and $f_1(c,d)$ are the only relations in E and F. By Definition 4.16 $G = \{q_{21}(x,y), e_1(a,b), f_1(c,d)\}$. By Definition 4.17 LV_G and GV_G are as follows. Note that because relation names in G also appear both as relation names in E and F and as IDB names in $MapE\_F$, we prepend each relation name with the schema to which it belongs.

$LV_G = \{$

$q_{22}(x,y)$:- $G.q_{21}(x,y)$

$q_{23}(x,y)$:- $G.q_{21}(x,y)$

$q_{24}(a,b)$:- $G.e_1(a,b)$

$q_{25}(c,d)$:- $G.f_1(c,d)$

$\}$

$GV_G = \{$

$q_{22}(x,y)$:- $E.e_1(x,y)$

$q_{23}(x,y)$:- $F.f_1(x,y)$

$q_{24}(a,b)$:- $E.e_1(a,b)$

$q_{25}(c,d)$:- $F.f_1(c,d)$

$\}$

q_{22} and q_{23} clearly illustrate why the views in Map_{G_EF} are not complete. If E contains the tuple $e_1(<1,2>)$, then clearly the G should contain the tuple $g_{21}(<1,2>)$. However, rewriting q_{21} using only f_1 should not yield $f_1(<1,2>)$. Therefore q_{25} – and the views in Map_{G_EF} in general – cannot be complete. □

We're now ready to show that the rewriting procedure explained informally above correctly rewrites queries over G into queries over EF. To do so we rely on Theorem 4.2 from (Abiteboul et al. 1998) which we renumber here for simplicity:

Theorem 4.1: For view definition $\mathcal{V} \subseteq \mathcal{CQ}$ (the set of conjunctive queries), $\mathcal{Q} \in \mathcal{D}atalog$ (the set of all Datalog queries), and query plan $\mathcal{P}$ that is maximally-contained in $\mathcal{Q}$ with respect to $\mathcal{V}$, $\mathcal{P}$ computes exactly the certain answers of $\mathcal{Q}$ under the open world assumption for each view instance $\sigma_V \in \mathcal{I}(\mathcal{V})$

Proof: This is Theorem 4.2 of (Abiteboul et al. 1998); we refer you to their proof. □

Theorem 4.2 provides a formal description of an algorithm for rewriting queries over G into queries over EF.

Theorem 4.2: Given input schemas E, F, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G,\ Map_{G\_EF}$, let query Q'' over EF be created by (1) creating a maximally-contained rewriting Q' over the intermediate schema C using LV_G and (2) expanding Q' using GV_G to form query Q'' over EF. Then $\forall$ instances $\sigma_E \in I(EF)$, and $\sigma_G \in I(G)$ s.t. $Map_{G\_EF}(\sigma_G, \sigma_{EF})$, $Q''(\sigma_{EF})$ finds exactly the certain answers of $Q(\sigma_G)$.

Proof: As demonstrated in Example 4.15, LV_G adheres to the open world assumption: it is sound but not complete, and $LV_G \subseteq \mathcal{CQ}$. Hence by Theorem 4.1, Q' computes exactly the certain answers of Q in C.

From the definition of a view, we know that rewriting Q' by unfolding the definitions in GV_G will create a query over EF that will retrieve exactly all tuples that match that view definition. Therefore $Q''(\sigma_{EF})$ finds exactly the certain answers of $Q(\sigma_G)$. □

4.3.4 Proof that G and Map_{G_EF} Satisfy the CMSCs

Given that we have now defined a well-formed mediated schema (Definition 4.16) and well-formed mediated schema mapping (Definition 4.17), we must show that if G and Map_{G_EF} satisfy these definitions, they satisfy the CMSCs as stated in Theorem 4.3.

> **Theorem 4.3**: Given input schemas E, F, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G,\ Map_{G\_EF}$, then G and Map_{G_EF} satisfy the CMSCs. □

We prove Theorem 4.3 by proving that well-formed mediated schemas and mappings adhere to each of the CMSCs in Section 4.3.4.1 through Section 4.3.4.5 in Lemma 4.1 through Lemma 4.8 respectively.

4.3.4.1 CMSC 1: Conjunctive Completeness

Recall that CMSC 1 is "G is complete. There are total, injective, functional mappings from E to G and from F to G." In order to ensure this, we create $CertainG$, a subset of G which contains only those tuples that are mapped to G from EF:

> **Definition 4.19:** ($CertainG$, $Map_{EF\_CertainG}$). Given input schemas E, F, and mapping Map_{E_F} between them, let $ConjunctiveMediatedSchemaCreation(E, F, Map_{EF}) \rightarrow G,\ Map_{G\_EF}$.
>
> - Define $Certain_G = G$.[7]
> - Define $Map_{EF\_CertainG}$ as follows: $\forall\ \sigma_{EF} \in I(EF)$, $\forall\ \sigma_{certainG} \in I(CertainG)$ $Map_{EF\_CertainG}(\sigma_{EF},\ \sigma_{CertainG})$ if $\forall$ relations $r \in CertainG$, $\pi_r(\sigma_G) = Q''(\sigma_{EF})$, where query Q'' is defined as follows:
>   i. Query $Q = q(attributes_r) :- r(attributes_r)$ where q is some fresh IDB name.
>   ii. Query Q' is a maximally-contained rewriting over the intermediate schema C using LV_G.
>   iii. Query Q'' is the result of expanding Q' using GV_G to form a query over EF.
> - There are no other tuples in $\sigma_{CertainG}$. □

[7] Recall that this denotes that the *schemas* are equal, not the instances.

We are now ready to show that CMSC 1 holds if CreateMediatedSchema(E, F, Map_{E_F})→ G, Map_{G_EF}:

Lemma 4.1: Given input schemas E, F, and mapping Map_{E_F} between them, if *ConjunctiveMediatedSchemaCreation*$(E,F,Map_{EF})\rightarrow G$, Map_{G_EF}, then G and Map_{G_EF} satisfy CMSC 1.

Proof Sketch: We show: (1) Lemma 4.3: $Map_{EF\_CertainG}$ is functional and total and (2) Lemma 4.4: $Map_{EF\_CertainG}$ is injective. Together these show that $Map_{EF\_CertainG}$ is a total, injective, functional mapping from E to $CertainG$ and from F to $CertainG$. Together Lemma 4.3 and Lemma 4.4 imply that G and Map_{G_EF} satisfy CMSC 1. □

To prove Lemma 4.3, we require that the certain answers to a query are unique, which we prove in Lemma 4.2:

Lemma 4.2: Let V be a view definition over a schema S, I be an instance of the view V and Q a query over S. The set of all certain answers to Q is unique.

Proof: A tuple t is a certain answer to Q under the open world assumption if t is an element of $Q(D)$ for each database D with $I \subseteq V(D)$ (Definition 2.7). Since the set of all D with $I \subseteq V(D)$ is uniquely defined and Q is a function, the result follows immediately. □

Lemma 4.3: Given input schemas E, F, and mapping Map_{E_F} between them, suppose *ConjunctiveMediatedSchemaCreation*$(E,F,Map_{EF})\rightarrow G$, Map_{G_EF}. Let $CertainG$ and $Map_{EF\_CertainG}$ be as in Definition 4.19. Then $\forall\ \sigma_{EF} \in I(EF)\ \exists$! state $\sigma_{CertainG} \in I(CertainG)$ such that $Map_{EF\_CertainG}(\sigma_{EF}, \sigma_{CertainG})$. That is, $Map_{EF\_CertainG}$ is a total function.

Proof: For some relation $r \in CertainG$ let Q be as in Definition 4.19(i). By Lemma 4.2, the set of certain answers for $Q(\sigma_{CetrainG})$ is unique. By Theorem 4.2,

$Q''(\sigma_{EF})$ computes the certain answers for $Q(\sigma_{CertainG})$. By Definition 4.19, for each relation $r \in CertainG$, $\pi_r(\sigma_{CertainG}) = Q''(\sigma_{EF})$. Therefore $\sigma_{certainG}$ is unique. □

Lemma 4.4: Given input schemas E, F, and mapping Map_{E_F} between them, suppose $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G, \ Map_{G\_EF}$. Let $CertainG$ and $Map_{EF\_CertainG}$ be as in Definition 4.19. Then $Map_{EF\_CertainG}$ is injective. That is, $\forall$ σ_{EF1}, σ_{EF2}, $\nexists$ $\sigma_{CertainG}$ s.t. $Map_{EF\_CertainG}(\sigma_{EF1}, \sigma_{CertainG})$ and $Map_{EF\_CertainG}(\sigma_{EF2}, \sigma_{CertainG})$.

Proof: Since $\sigma_{EF1} \neq \sigma_{EF2}$, there must be some relation $r \in EF$ s.t. $\pi_r(\sigma_{EF1}) \neq \pi_r(\sigma_{EF2})$. There are two cases: either (1) r is not mapping-included (Definition 4.14) or (2) r is mapping-included.

(1) Assume r is not mapping-included. Then since G is a well-formed mediated schema $\exists$! relation $g \in G$ s.t. $\xi(r,g)$ and $name_g = name_r$ and $attributes_g = attributes_r$ (Definition 4.16 bullet 1). By definition of $CertainG$ (Definition 4.19), $CertainG = G$ (i.e., the schemas are equal). Hence $\exists$! relation $certain_g \in CertainG$ s.t. $certain_g = g$. Let $qfresh$ be a fresh IDB name. Let $Q = qfresh(attributes_{certaing}) :- certain_g$.[8]

Since Map_{G_EF} is well-formed, LV_G is a set of conjunctive queries adhering to the open world assumption (from the definition of GV_G and LV_G (Definition 4.17)). By construction, Q is conjunctive query. Thus, from Theorem 4.1 there is a maximally-contained rewriting Q' over the intermediate schema C using LV_G that finds all certain answers for Q. Let Q'' be the result of expanding Q' using GV_G to form a query over EF. From the definition of a view and the definition of Q', $Q''(\sigma_{EF})$ contains all certain answers of $Q(\sigma_{CertainG})$.

From the definition of GV_G and LV_G (Definition 4.17 bullet 1) there exists exactly one view $V_g \in LV_G$ that can be used to rewrite Q, and the rewriting is $qfresh(attributes_{certaing}) :- V_g$. Since GV_G is well-formed, there exists one view

[8] For succinctness, in what follows we occasionally abbreviate $r(attributes_r)$ by r.

$V_{idb} \in GV_G$ s.t. $IDB(V_{idb}) = IDB(V_g)$, and the body of V_{idb} contains exactly r. Hence $Q'' = qfresh(attributes_{certaing}) :- r$. Therefore, if $Map_{EF\_CertainG}(\sigma_{EF1}, \sigma_{CertainG})$, then $\pi_r(\sigma_{CertainG}) = \pi_r(\sigma_{EF1})$, and if $Map_{EF\_CertainG}(\sigma_{EF2}, \sigma_{CertainG})$, then $\pi_r(\sigma_{CertainG}) = \pi_r(\sigma_{EF2})$. Since $\sigma_{EF1} \neq \sigma_{EF2}$, if r is not mapping-included, $\nexists$ $\sigma_{CertainG}$ s.t. $Map_{EF\_CertainG}(\sigma_{EF1}, \sigma_{CertainG})$ and $Map_{EF\_CertainG}(\sigma_{EF2}, \sigma_{CertainG})$.

(2) Now assume that r is mapping-included. Since G is a well-formed mediated schema $\exists$! relation $g \in G_{IDB}$ s.t. $\xi(r,g)$, $name_g = name_r$ and $attributes_g \supseteq attributes_r$ (Definition 4.16 bullet 2). By definition of $CertainG$ (Definition 4.19), $CertainG = G$ (i.e., the schemas are equal). Hence $\exists$! relation $certaing \in CertainG$ s.t. $certaing = g$. Let $qfresh$ be a fresh IDB name. Let $Q = qfresh(attributes_{certaing}) :- certaing$. Let $Q = qfresh(attributes_r) :- certaing$.

Since Map_{G_EF} is well-formed, LV_G is a set of conjunctive queries adhering to the open world assumption (from the definition of GV_G and LV_G (Definition 4.17)). By construction, Q is conjunctive query. Thus, from Theorem 4.1 there is a maximally-contained rewriting Q' over the intermediate schema C using LV_G that finds all certain answers for Q. Let Q'' be the result of expanding Q' using GV_G to form a query over EF. From the definition of a view and the definition of Q', $Q''(\sigma_{EF})$ contains all certain answers of $Q(\sigma_{CertainG})$.

Since Map_{G_EF} is well-formed, from the definition of GV_G and LV_G (Definition 4.17 bullet 2) there exists exactly one view, $V_g \in LV_G$ that can be used to rewrite Q, and the rewriting is $qfresh(attributes_r) :- V_g$. Since GV_G is well-formed, there exists one view $V_{idb} \in GV_G$ s.t. $IDB(V_{idb}) = IDB(V_g)$, and the body of V_{idb} contains exactly r. Hence $Q'' = qfresh(attributes_r) :- r$. Therefore, if $Map_{EF\_CertainG}(\sigma_{EF1}, \sigma_{CertainG})$, $\pi_r(\sigma_{CertainG}) = \pi_r(\sigma_{EF1})$, and if $Map_{EF\_CertainG}(\sigma_{EF2}, \sigma_{CertainG})$, then $\pi_r(\sigma_{CertainG}) = \pi_r(\sigma_{EF2})$. Since $\sigma_{EF1} \neq \sigma_{EF2}$, if r is mapping-included, $\nexists$ $\sigma_{CertainG}$ s.t. $Map_{EF\_CertainG}(\sigma_{EF1}, \sigma_{CertainG})$ and $Map_{EF\_CertainG}(\sigma_{EF2}, \sigma_{CertainG})$.

Since two states of EF map to different states of $CertainG$ if r is or is not mapping-included, which are the only two cases, two states of EF always map to different states of $CertainG$, and $Map_{EF\_CertainG}$ is injective. □

Lemma 4.3 and Lemma 4.4 prove Lemma 4.1, that Map_{EF_G} is total, injective, and functional.

4.3.4.2 CMSC 2: Conjunctive Intersection Accessibility

Recall that CMSC 2 is "For each IDB idb in $IDB(Map_{E\_F})$, let $MS_{idb} = \{ms_j \in Map_{E\_F} \mid$ IDB(ms_j) $= idb\}$. Then there exists a query Q over G and Map_{G_EF} such that Q is a canonical query for MS_{idb}."

Lemma 4.5: Given input schemas E, F, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G,\ Map_{G\_EF}$, then G and Map_{G_EF} satisfy CMSC 2 where the required canonical query is a conjunctive query.

Proof Sketch: We prove Lemma 4.5 by construction of the canonical queries required. For each IDB idb in $IDB(Map_{E\_F})$, let $MS_{idb} = \{ms_j \in Map_{E\_F} \mid$ IDB(ms_j) $= idb\}$. We construct a query Q_{EF} over EF s.t. $Q_{EF} = MS_{idb}$. We need a canonical query for Q_{EF}. So we construct a conjunctive query Q_G over one relation in G and show that rewriting Q_G using Map_{G_EF} produces a query Q_G' over EF that returns the certain answers to Q_{EF}. Therefore Q_G is a canonical query for Q_{EF}. We show an example of creating Q_G, Q_G'', Q_{EF}, and Q_{EF}' in Example 4.16.

Proof: By Definition 4.16 of G $\forall$ IDBs $idb \in$ IDB(Map_{E_F}), $\exists$ relation $g \in G$ s.t. the name of idb is n_g.

Let q be a fresh IDB name

Let Q_{EF} be the query $q(Vars(idb)) :- idb$

Let Q_{EF}' be the expansion of Q_{EF} using Map_{E_F}.[9]

Let Q_G be the query $q(Vars(idb)) :- g$

Let Q_G'' be the rewriting of Q_G using Map_{G_EF}.

[9] See Definition 2.5 for the definition of expansion.

Let $MS_{IDB} = \{ms_{idb} \mid IDB(ms_{idb}) = idb\}$

We now show that $Q_{EF}' \sqsubseteq Q_G''$ and $Q_G'' \sqsubseteq Q_{EF}'$.

By Theorem 4.2, $\forall\ \sigma_{EF} \in I(EF)$, $\forall\ \sigma_G \in I(G)$, s.t. $Map_{G\_EF}(\sigma_{EF}, \sigma_G)$, rewriting $Q_G(\sigma_G)$ to be over EF yields Q_G'', where $Q_G''(\sigma_{EF})$ finds exactly the certain answers of $Q_G(\sigma_G)$. By Theorem 4.2 obtaining Q_G'' requires first answering Q_G using the local views, LV_G. Q_G only asks for elements in the head of idb. Since Map_{G_EF} is a well-formed mediated schema mapping (Definition 4.17), $\forall\ ms \in MS_{IDB}$, LV_G will contain one view $V_{ms} = qfresh(Vars(ms)) :- g$ where $qfresh$ is an IDB name used nowhere else in LV_G. Since each mapping statement must be safe, $Distinguished(ms) \subseteq Vars(ms)$. LV_G and Q_G are both conjunctive queries, LV_G uses the open world assumption, and $Map_{\ G\_EF}$ is a well-formed mediated schema mapping (Definition 4.17), therefore answering queries using views can be used to find a maximally-contained rewriting $QG' = \bigcup_{ms_{idb} \in MS_{IDB}} q(Vars(idb))$ $:- qfresh_{msidb}(Vars(ms_{idb}))$.

By Theorem 4.2, Q_G' is then expanded into a query Q_G'' using GV_G. From the definition of GV_G, expanding each conjunctive query $cq_i \in QG'$ yields $q(Vars(idb)) :- Body(ms_i)$, and $QG'' = \bigcup_i cq_i$.

By the definition of expansion of a view (Definition 2.5), $Q_{EF}' = \bigcup_{ms_{idb} \in MS_{IDB}}$ $q(Vars(idb)) :- Body(ms_{idb})$.

Hence, Q_G'' and Q_{EF}' are equivalent. Since Q_G is conjunctive and defined over a single relation of EF, Q_G is a conjunctive, canonical query for idb as defined in Definition 4.12. □

Example 4.16: The queries constructed in Lemma 4.5 for the schemas and mappings of Example 4.13 as follows:

$Q_G = q(x) :- q_1(x)$

$Q_G'' = q(x) :- e_1(x,y) \cup q(x) :- f_1(x,z)$

$Q_{EF} = q(x) :- q_1(x,y,z)$

$Q_{EF}' = q(x) :- e_1(x,y) \cup q(x) :- f_1(x,z)$ □

4.3.4.3 CMSC 3: Conjunctive component accessibility

Recall CMSC 3 is "For every mapping statement *ms* in Map_{E_F}, let $Q = q(Vars(ms)) :- body)$. If there are any existential variables in *ms*, then there exists a canonical query for *Q* over *G* and Map_{G_EF}."

Lemma 4.6: Given input schemas *E*, *F*, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G$, Map_{G_EF}, then *G* and Map_{G_EF} satisfy CMSC 3 using only conjunctive queries for canonical queries.

Proof sketch: We prove Lemma 4.6 by construction of the canonical queries required. For each mapping statement *ms* we construct a query Q_{EF} s.t. $Q_{EF} = ms$. We need a canonical query for Q_{EF}. So we construct a conjunctive query Q_G which is over one relation in *G* and show that rewriting Q_G using Map_{G_EF} produces a query Q_G'' over *EF* that is equivalent to Q_{EF}. Therefore Q_G is a conjunctive, canonical query for Q_{EF}. We show an example of creating Q_G, Q_G'', and Q_{EF} in Example 4.17.

Proof: By definition of *G* and Map_{G_EF}, $\forall$ mapping statements $ms \in Map_{E\_F}$ s.t. there is no intersection that subsumes *ms*, $\exists$ relation $g \in G$ s.t. $IDB(ms) = n_g$.

Let *q* be a fresh IDB name.

Let Q_{EF} be the query $q(Vars(ms)) :- body(ms)$

Let Q_G be the query $q(Vars(ms)) :- g$

Let Q_G'' be the rewriting of Q_G using Map_{G_EF}

We now show that $Q_{EF} \sqsubseteq Q_G''$ and $Q_G'' \sqsubseteq Q_{EF}$.

By Theorem 4.2, $\forall\ \sigma_{EF} \in I(EF)$, $\forall\ \sigma_G \in I(G)$, s.t. $Map_{G\_EF}(\sigma_{EF}, \sigma_G)$, rewriting $Q_G(\sigma_G)$ to be over *EF* yields Q_G'', where $Q_G''(\sigma_{EF})$ finds exactly the certain answers of $Q_G(\sigma_G)$. By Theorem 4.2 obtaining Q_G'' requires first answering Q_G using the local views, LV_G. Since Map_{G_EF} is a well-formed mediated schema mapping (Definition 4.17), LV_G will contain exactly one local view V_{ms} = $qfresh(Vars(ms))$:− g where $qfresh$ is an IDB name used nowhere else in LV_G. Since Map_{G_EF} is a well-formed mediated schema mapping (Definition 4.17), V_{ms} is the only way of accessing the attributes corresponding to existential variables in ms required by Q_G. LV_G and Q_G are both conjunctive queries, LV_G uses the open world assumption, and $Map\ _{G\_EF}$ is a well-formed mediated schema mapping (Definition 4.17). Therefore, answering queries using views can be used to find a maximally-contained rewriting $QG' = q(Vars(ms))$:− $qfresh(Vars(ms))$.

By Theorem 4.2, Q_G' is then expanded into a query Q_G'' using GV_G. From the definition of GV_G, expanding QG' yields $QG'' = q(Vars(ms))$:− $Body(ms)$. This is exactly the definition of Q_{EF}.

Hence, Q_G'' and Q_{EF} are equivalent. Since Q_G is conjunctive and defined over a single relation of EF, Q_G is a conjunctive, canonical query for ms as defined in Definition 4.12. □

Example 4.17: Assume the schemas and mappings from Example 4.13.

$Q_{EF} = q(x,y)$:− $e_1(x,y)$

$Q_G = q(x,y)$:− $q_1(x,y,z)$

$Q_G'' = q(x,y)$:− $e_1(x,y)$ □

4.3.4.4 CMSC 4: Conjunctive Connectivity Required

Recall CMSC 4 is "For every query $Q{:}I(\Sigma_Q) \rightarrow I(T)$ over $\Sigma_Q \subseteq EF$ where $|\Sigma_Q| > 1$, if there is a canonical query Q' for Q over $\Sigma_{Q'} \subseteq G$ and Map_{G_EF} then for every pair of relations $r1$, $r2 \in EF'$ there exists some mapping statements $ms1 \in Map_{E\_F}$ and $ms2 \in Map_{E\_F}$ such that $IDB(ms1) = IDB(ms2)$, $r1 \in body(ms1)$, and $r2 \in body(ms2)$."

Lemma 4.7: Given input schemas E, F, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G, Map_{G\_EF}$, then G and Map_{G_EF} satisfy CMSC 4.

Proof: A query over G can only be rewritten into a query over EF as explained in Theorem 4.2. Hence, since a canonical query must be over only one relation in G, it is enough to show that any query over a single relation in G satisfies the conditions in CMSC 4. Consider query Q_g over some relation $g \in G$. By Theorem 4.1 and the definition of rewriting queries using views, we know local view definition $lv_i \in LV_G$ can only be used to rewrite Q_g to produce query Q_g' if the body of lv_i contains g. By Definition 4.17 (well-formed mediated schema mapping) we know that given two local views lv_i and lv_j, the bodies of lv_i and lv_j will only contain the same relation name, g, if $\xi(r, ms_i)$, $\xi(r, ms_j)$ where ms_i and ms_j are the mapping statements used to create lv_i and lv_j. By the definition of well-formed mediated schema mappings (Definition 4.17) we know that this requires that ms_i and ms_j have the same IDB name.

By Theorem 4.2 we know that to produce a query Q_g'' over EF, query Q_g' must be expanded using the global view definitions in GV_G. By Definition 4.16 (well-formed mediated schema) we know that the only global view definitions that can be used to expand lv_i and lv_j must have bodies equal to the bodies of ms_i and ms_j. Hence any query Q written over a single relation $g \in G$ will satisfy the requirements in CMSC 4. □

4.3.4.5 CMSC 5: Minimality

Recall CMSC 5 is "G is a minimal schema that satisfies CMSCs 1-4. That is, there exists no G' satisfying CMSCs 1-4 such that (1) $G' << G$ or $|G'| < G$ (i.e., G' contains fewer relations than G). and (2) $\forall$ relations $g \in G$ $\nexists$ sets of attributes $p_1 \subseteq attributes_g$, $p_2 \subseteq attributes_g$ s.t. $p_1 \cap p_2 = \emptyset$ and $\forall$ $\sigma_G \in I(G)$ $\pi p_1(\pi_g(\sigma_G)) \subseteq \pi p_2(\pi_g(\sigma_G))$."

Lemma 4.8: Given input schemas E, F, and mapping Map_{E_F} between them, if $ConjunctiveMediatedSchemaCreation(E,F,Map_{EF}) \rightarrow G, Map_{G\_EF}$, then G and Map_{G_EF} satisfy CMSC 5.

Proof: By definition of well-formed mediated schema (Definition 4.16), $G = G_{IDB} \cup G_{EF}$ where $G_{IDB} = \{g_{IDB} \in G \mid \exists\ idb \in IDB(Map_{E\_F})$ and $\xi(idb, g_{IDB})\}$ and $G_{EF} = \{g_{ef} \in G \mid \exists\,!\ ef \in EF$ s.t. $\xi(ef, g_{ef})$ and ef is not mapping-included in $Map_{E\_F}\}$. We show that G_{IDB} and G_{EF} are minimal, including why relations from the other set cannot be used to minimize them further.

G_{IDB}**:** In order to satisfy CMSC 2 we must have a relation $g \in G$ for each $idb \in IDB(Map_{E\_F})$ s.t. $\xi(idb, g)$. By CMSC 4 we cannot combine relations in G_{IDB} because they are not connected (Definition 4.7). In order to allow canonical queries for idb and each projection-free mapping statement pf_{ms} where $IDB(ms) = idb$ (CMSC 2 and CMSC 3), $attributes_g$ must equal $Vars(MS_{idb})$[10], which is exactly as defined in Definition 4.16. Hence G_{IDB} is minimal given CMSC 1-4.

G_{EF}**:** $\forall$ relations $g_{ef} \in G_{EF}$, by construction of G_{EF} $\exists\,!\ ef \in EF$ s.t. $\xi(ef, g_{ef})$. By the definition of Map_{E_F} it is either the case that (1) $\exists$ mapping statement $ms \in Map_{E\_F}$ s.t. $ef \in body(ms)$ or (2) $\nexists\ ms \in Map_{E\_F}$ s.t. $ef \in body(ms)$. Since these two cases span the entire space of possibilities, considering them both proves that G_{EF} is minimal. We consider these cases separately.

Case (1), $\exists$ mapping statement $ms \in Map_{E\_F}$ s.t. $ef \in body(ms)$. From the definition of a well-formed mediated schema (Definition 4.16) and G_{IDB}, $\exists!$ relation $g_{idb} \in G_{IDB}$ s.t. $\exists$ mapping statement ms where $ef \in Body(ms)$. Let $IDB(ms) = idb$. In order to satisfy CMSCs 2 and 3, there needs to be a canonical query over g_{idb} for idb. Hence any attribute $a \in attributes_g$ corresponding to a variable in the head of ms must be represented such that any query over G retrieves the values from all mapping statements $ms_{idb} \in MS_{idb}$ s.t. $IDB(MS_{idb}) =$

[10] Recall that MS_{idb} = the union of *all* mapping statements ms such that $IDB(ms) = idb$.

idb. Hence those attributes cannot be used to satisfy CMSC 1: Completeness. In order to satisfy the second clause in CMSC 5, no attribute of a relation in *EF* can be represented in a relation in *G* in two different fashions. Therefore g_{idb} cannot have extra attributes to satisfy CMSC 1, and there must exist some relation $g_{ef1} \in G$ s.t. $g_{ef1} \neq g_{idb}$. The minimal relation that can ensure completeness for *ef* is if $name_{gef1} = name_{ef}$ and $attributes_{gef1} = attributes_{ef}$ which is exactly what g_{ef} is from the definition of a well-formed mediated schema (Definition 4.16 bullet 2).

Case (2): $\nexists$ $ms \in Map_{E\_F}$ s.t. $ef \in body(ms)$. In this case there are no relations $ef_1 \in EF$ s.t. *ef* is connected to ef_1 (Definition 4.7). Thus by CMSC 4, $\forall$ $ef_1 \in EF$ s.t. $ef \neq ef$, there can be no relations $g \in G$ s.t. $\xi(ef,g)$ and $\xi(ef_1,g)$. Hence there must exist some separate relation $g_{ef1} \in$ G s.t. $\xi(ef, g_{ef1})$. The minimal relation that can ensure completeness for *ef* is if $g_{ef1} = ef$, which is exactly what g_{ef} is from the definition of a well-formed mediated schema (Definition 4.16 bullet 2).

Putting together the two cases for G_{EF}, G_{EF} is minimal given CMSC 1-4. Since G_{IDB} is also minimal given CMSC 1-4, *G* is minimal given CMSC 1-4. □

4.4 Alternate Definitions of *G*

While we have described in depth a well-formed mediated schema (Definition 4.16), there may be other possible criteria that users may desire to create a mediated schema. Our goal in creating the well-formed mediated schema was to make it as general as possible, so adapting to these alternate definitions is quite easy. In this section we discuss other desired criteria.

4.4.1 Relaxing the Mediated Schema Criteria

In this section we consider relaxing the *generic* Mediated Schema Criteria. Section 4.4.1.1 considers relaxing MSC 1 (Completeness). We do *not* consider relaxing MSCs 2 or 3, because the intersections and components expressed in the generic case identify relations that are defined to be relevant to the person creating the mapping. Similarly, we do not consider relaxing MSC 4. Section 4.4.1.2 considers relaxing MSC 5 (Minimality).

4.4.1.1 Relaxing MSC 1: Completeness

If MSC 1 is relaxed, then there is no need to include relations in G to satisfy completeness.

Example 4.18: Assume $E = \{e_1(a,b), e_3(c,d)\}$, $F = \{f_1(e,f)\}$ and Map_{E_F} is:

$q_1(x) :- e_1(x,y), e_3(x,t)$

$q_1(x) :- f_1(x,z)$ □

Consider Example 4.18. If the well-formed mediated schema definition (Definition 4.16) is used, G contains relations $q_1(x,y,z,t)$, $e_1(a,b)$, $e_3(c,d)$. The semantics of the well-formed mediated schema is that Map_{E_F} is being used to express how overlapping information is accessed in the mediated schema, but all tuples should be represented somehow in the mediated schema. Because only relations that are mapping-included are excluded, all instances from E and F are reachable from the mediated schema.

However, the mediated schema creator may wish to use Map_{E_F} to express exactly which tuples are accessible in the mediated schema. In Example 4.18 this implies that the mediated schema contains only the relation $q_1(x,y,z,t)$. Changing G is simple; bullet 1 of Definition 4.16 is ignored. Changing Map_{G_EF} to satisfy this new definition is also simple; the mapping statements created in the second bullet of Definition 4.17 are simply excluded.

4.4.1.2 Relaxing MSC 5: Minimality

One could argue that rather than having just the relations needed for G to be complete, G should include *all* relations in E and F. In Example 4.18 this implies that the mediated schema has relations $q_1(x,y,z,t)$, $e_1(a,b)$, $e_3(c,d)$, and $f_1(e,f)$. This semantics is needed if each relation in E and F corresponds to a different potentially-interesting set of instances that is not recoverable without all relations. After modifying G to include all relations in EF, Map_{G_EF} does not need to be changed at all.

4.4.2 Modifying the Interpretation of Conjunctive Mappings

One could make the case that a conjunctive mapping should not include all projection-free components (Definition 4.13). We consider here G', a mediated schema that does not contain all projection-free components. Given a mapping statement $ms \in Map_{E\_F}$ and a relation $g \in G'$, s.t. $\xi(ms,g)$, it would be unnecessary for g to contain the attributes corresponding to existential variables in Map_{E_F} for non-mapping-included relations in ms. However, in this interpretation of conjunctive mappings, g would still be required to retain the attributes of mapping-included relations.

Example 4.19. Assume the schemas and mappings in Example 4.13. Recall that f_1 is not represented separately in G because f_1 is mapping-included – it is already included in G due to the relations created as a result of Map_{E_F}. Ignoring MSC 5 (Minimality), whereas G contains $q_1(x,y,z)$, our initial attempt at creating G' might be $\{e_1(a,b), e_2(c,d), q_1(x), f_1(e,f)\}$. Because all values of f_1 are not represented in G' since $q_1(x,y,z)$, is not included in G', we cannot use q_1 to provide completeness for f_1, so we now must include f_1 in G' as a separate relation. However, when we consider MSC 5, regardless of the fact that we no longer consider Map_{E_F} to define projection-free components, q_1 *must* be able to represent all values of f_1, or G' is not a minimal schema. Hence $G' = \{e_1(a,b), e_2(c,d), q_1(x,z)\}$. □

Hence, even though G' does not contain any relations for projection-free components, in the case of mapping-included relations G' is the same as G. Otherwise, the attributes for projection-free components are not included and Definition 4.17 is changed so that the EDBs in LV_G only include the attributes that actually occur in G'.

It is worth noting that combining the change in G' with either of the two changes in Section 4.4.1 can allow a significant simplification of $Map_{G'\_EF}$: if none of the relations in G that correspond to IDB names in $Map_{G'\_EF}$ include attributes corresponding to existential variables in the mapping statements used to create them, $Map_{G'\_EF}$ can be a simple GAV mapping. We expound further on the reasons for this in Section 4.6.

4.5 Extensions

4.5.1 Merging More Than Two Local Sources

In this chapter we have considered creating a mediated schema for two source schemas. However, most mediated schemas are created over more than two input schemas. There are two main choices for extending this work to merging more than two source schemas: (1) the source schemas can be merged all at once or (2) the schemas can be merged pair-wise. Some examples are shown in Batini, Lenzerini and Navathe (Batini et al. 1986).

If the schemas are merged all at once, extending the definitions of a well-formed mediated schema (Definition 4.16) and mapping (Definition 4.17) is simple; replace every occurrence of *EF* in the definitions by all of the schemas over which the mediated schema is being created.

Merging the schemas pair-wise allows for greater flexibility. However, it is more complicated as show in Example 4.20:

Example 4.20. To pair-wise merge schemas *E*, *F*, and *H*, begin by merging *E* and *F* to form a new mediated schema *M*. *M* can now be merged with another schema *H* to form a new schema *N*. We begin by merging *E* and *F* using the following mappings

m1(x,y) :- *e1(x,y,z)*

m1(x,y) :- *f1(w,x,y)*

This would result, as shown before, in the mediated schema relation *m1(w, x, y, z)* in schema *M*. IDB *m1* could be used in the definition of a mapping statement, *n1*, that describes *M*'s intersection with *H*:

n1(w,x,y) :- *m1(w,x,y,z)*

n1(w,x,y) :- *h1(v,w,x,y)*

to form the final mediated schema relation *n1(v, w, x, y, z)*. □

Rewriting queries in this situation becomes more difficult since the definition essentially requires composing GLAV mappings. For example, rewriting a query over *N* to a query over *E* in Example 4.20 would require first rewriting a query over the GLAV mapping Map_{N_MH} and then composing that result with a query rewritten over the GLAV mapping Map_{M_EF}. Composing GLAV mappings in general is very complicated and beyond the scope of this thesis. Recent results on this topic can be found in (Madhavan et al. 2003) and (Fagin et al. 2004).

Depending on the characteristics desired and the mappings provided, pair-wise merging of three or more schemas can be done either by providing a mapping between the intermediate schemas, such as Map_{M_G} above, or by creating mappings to whichever source is logically closest and then composing the mappings. E.g., in Example 4.20 instead of having the input be a direct mapping Map_{M_H}, relating M and H, the input could be a mapping Map_{E_H}. Merging M and H requires a mapping Map_{M_H}, which could be formed by taking the composition of Map_{E_H} and Map_{E_M}. This pair-wise method of mediated schema creation is likely to extend better to situations where the sources involved are more fluid, such as peer data management (Aberer et al. 2002; Arenas et al. 2003; Bernstein et al. 2002; Halevy et al. 2003; Ooi et al. 2003).

4.5.2 Expanding Conjunctive Mappings to Consider More Complicated Queries

4.5.2.1 Allowing Relations to be in Multiple Mapping Statements

We now consider expanding our mappings to allow the same relations to be present in more than one mapping statement. Suppose, for example, we had a mapping:

$q_1(x,y,z) :- e_1(x,y), e_2(y,z)$

$q_1(x,y,z) :- f_1(x,y,z)$

$q_2(y,z,u) :- e_2(y,z), e_3(z,u)$

$q_2(y,z,u) :- f_2(y,z,u)$

Note that e_2 appears in the definition of both q_1 and q_2. What should the mediated schema be?

One choice is that each IDB represents a separate relation of interest and that the correct semantics is to have *both* q_1 and q_2 as relations in the mediated schema.

Another choice is to assume that equality in the mapping is transitive; e.g., the mapping specifies both that querying tuples of e_2 should cause tuples of f_1 to be queried and that querying tuples of e_2 should cause tuples of f_2 to be queried. However, since conjunctive queries do not disambiguate between the two choices, we do not consider them.

4.5.3 Limitations of Conjunctive Mappings

Thus far we have been concentrating on the ambiguities of conjunctive mappings: that is, where the full extent of conjunctive mappings cannot be used because the semantics of the problem is not rich enough to know what the merged result should be. On the flip side, the use of conjunctive queries is *overly* restricts the relationships that can be expressed between elements in E and F. There are a large number of things that cannot be expressed, even when we restrict ourselves only to the domain of relational schemas:

- Relationships that express anything other than equality between attributes. For example, SchemaSQL (Lakshmanan et al. 1996) allows attributes of one relation to be equated with data values in another. Conjunctive queries are not rich enough to handle this kind of a mapping.
- Conjunctive mappings are incapable of expressing foreign keys or other such constraints. While it is true that we have artificially restricted our definition of relational schemas to exclude such constraints, this was partially done because the conjunctive mappings did not lead to considering them.
- In addition, an orthogonal issue is whether or not to include a separate attribute expressing the lineage of the tuple. For example, using the inputs in Example 4.18, we could express the lineage of q_1 by having the mediated schema contain $q_1(x,y,z,t,n)$ where n is the name of the relation that the tuple originates from.

4.6 Global-As-View and Local-As-View

We chose to use GLAV rather than either GAV or LAV mappings since neither is adequate to express the relationships required between the mediated schema and the source schemas. In this section we explore when the conjunctive mappings become complex enough that GAV and LAV are not rich enough in Sections 4.6.1and 4.6.2 respectively.

4.6.1 Global-As-View (GAV)

In Global-As-View (GAV) each mediated relation is defined as a set of views over the database sources (for a recent survey of GAV approaches see (Lenzerini 2002)). So, for example, g_1 might be defined as:

$g_1(x, y) :- e_1(x, y)$

$g_1(x, y) :- f_2(x, z), f_3(z, y)$

To get all of the tuples for g_1 it is necessary to take the join of f_2 and f_3 and union those results with the tuples in e_1. GAV breaks down as a possible mapping language for our case when there are existential variables in any of the mapping statements.

For example, take the mapping:

$q_{19}(x) :- e_1(x, z)$

$q_{19}(x) :- f_1(x, y)$

The mapping states that the concept of x is common but e_1 contains additional information about some attribute z and f_1 contains additional information about some attribute y.

According to the definition of G in Section 4.3.3.1 the corresponding mediated schema relation in G for q_{19} should be $q_{19}(x, y, z)$. A GAV definition must provide a value for each attribute in the mediated schema relation it is defining, such as one for q_{19}. However, there is no conjunctive query that can define all of the attributes for q_{19} since the mapping indicates that the result should be the *union* of the tuples, not the intersection or join. Hence GAV cannot provide a valid view definition for this situation. While we could extend the language of GAV to be ILOG (Hull et al. 1990) instead of Datalog, GLAV mappings work as well.

4.6.2 Local-As-View (LAV)

In LAV each source relation is defined as a conjunctive view over the mediated schema. As an example, a source relation, e_1, may be defined as a join over two mediated schema relations, g_1 and g_2, as follows: $e_1(x) :- g_1(x, z), g_2(z, y)$. A full algorithm for how to answer queries in LAV is given in Chapter 3. Here we provide a brief synopsis to understand what is relevant for the purposes of this section.

LAV may not contain enough sources to give an equivalent answer to the user's query. For example, given only the mapping above ($e_1(x) :- g_1(x, z), g_2(z, y)$) it is not possible to find a source to give all of these answers for the query $q_8(x, y) :- g_1(x, y)$; the mapping gives us only g_1 joined with g_2, e.g., $g_1(x, z), g_2(z, y)$. Hence in LAV queries are answered through a *maximally-contained rewriting*; that is, a rewriting that gives all possible sound answers for the query using the sources provided but the reformulated query is not guaranteed to be equivalent to the query over the mediated schema.

As an additional consideration, data integration often operates with the open world assumption (Definition 2.6): That is, each data source is assumed to be sound but incomplete. In the example above, this means that e_1 contains only valid tuples in g_1 join g_2, but it may not contain all such tuples.

It can be shown that the maximally-contained rewriting of a conjunctive query, Q, over conjunctive views, V, using the open world assumption can be expressed as a set of conjunctive queries over the view. As an example, using e_1 as defined above and the additional sources

$e_2(x, y) :- g_3(x, y)$

$f_3(x, y) :- g_3(x, y)$

a query $q_9(a, b) :- g_1(a, b), g_3(a, c)$ can be rewritten:

$q_9'(a, b) :- e_1(a), e_2(a, c)$

$q_9'(a, b) :- e_1(a), f_3(a, c)$

Note that (1) even though e_2 and f_3 have the same definition both must appear in the maximally-contained rewriting since under the open world assumption they may contain different tuples and (2) the rewriting is not equivalent to q_9.

LAV breaks down in the context of conjunctive mappings between source schemas when the mapping statements consist of more than one subgoal. An example of this is:

$q_{13}(x, y, z) :- e_4(x, z), e_5(z, y)$

$q_{13}(x, y, z) :- f_4(x, y, z)$

The desired result again mimics the Datalog; answers to queries over q_{13} should consider tuples from the join of e_4 and e_5 with the union of tuples from f_4.

Classic LAV cannot map this relationship completely. In LAV each source relation must be described as a query over the mediated schema. Hence some of the details of this relationship could be lost by declaring the mapping to be:

$e_4(x, z) :- q_{13}(x, y, z)$

$e_5(z, y) :- q_{13}(x, y, z)$

$f_4(x, y, z) :- q_{13}(x, y, z)$

However, while this would allow e_4 and e_5 to answer some queries, it would not allow combining them in order to answer queries about the entire mediated schema relationship. For example, they could not answer the query $qm_{13}(x, y, z) :- q_{13}(x, y, z)$.

While one can disagree over the meaning of having two mapping statements for the same IDB predicate (union vs. join) here the mapping clearly states that the result should be the join.

4.7 Conclusions

In this chapter we described a set of criteria for creating a relational mediated schema based on concepts from the literature and traditional metrics such as information capacity. We then showed how to translate these criteria to a conjunctive mapping and alternative semantics that may be desired.

The problem of merging two schemas is not limited to the creation of a relational mediated schema. In Chapter 5 we describe how this problem can be treated more generically, both across data models and across applications. In Chapter 6 we revisit the solution presented here and show how to encode it in the generic solution presented in Chapter 5.

Chapter 5
Generic Merge

5.1 Introduction

In Chapter 4 we discussed how to create a mediated schema based on a mapping comprised of queries. However, this is not the only situation in which two schemas must be combined in order to create a third schema. The problem of merging schemas lies at the core of many meta-data applications, such as view integration, mediated schema creation for data integration, and ontology merging. In each case, two given schemas need to be combined into one. In this chapter, we consider this merging problem for more than just relational schemas; we consider the merging of *models*. A *model* is a formal description of a complex application artifact, such as a database schema, an application interface, a UML model, an ontology, or a message format. Because there are many different kinds of models and applications, this problem has been tackled independently in specific domains many times. This chapter provides a generic framework that can be used to merge models in all these contexts.

Combining two models requires first determining how the two models are related and then merging the models based on those relationships. These relationships may take the form of the conjunctive mappings in Chapter 4, or they may be given in some other format. Finding relationships between schemas is called schema matching; it is a major topic of ongoing research and is not covered in this thesis; see (Rahm et al. 2001) for a recent survey and (He et al. 2003), (Kang et al. 2003) and (Dhamankar et al. 2004) for examples of work since then. Rather, this chapter focuses on combining the models after the relationships between schemas are established. We encapsulate the problem in an operator, Merge, which takes as input two models, A and B, and a mapping Map_{A_B} between them that embodies the given inter-schema relationships. It returns a third model that is the "duplicate-free union" of A and B with respect to Map_{A_B} – A and B have been unioned together, but duplicates have been removed. This is not as simple as set union because the models have structure, so the semantics of "duplicates" and duplicate removal may be complex. In addition, the result of the union can manifest constraint violations, called *conflicts*, that Merge must repair.

An example of the problems addressed by Merge can be seen in Figure 5.1. It shows two representations of Actor, each of which could be a class, concept, table, etc. Models A and B are to be merged. $\text{Map}_{\text{A\_B}}$ is the mapping between the two; relationships relating the models are shown by dashed lines. In this case, it seems clear that Merge is meant to collapse A.Actor and B.Actor into a single element, and similarly for Bio. Clearly, A.ActID should be merged with B.ActorID, but what should the resulting element be called? What about the actor's name? Should the merged model represent the actor's name as one element (ActorName), two elements (FirstName and LastName), three elements (ActorName with FirstName and LastName as children), or some other way?

These cases of differing representations between input models are called *conflicts*. For the most part, conflict resolution is independent of the representation of A and B. Yet most work on merging schemas is data-model-specific, revisiting the same problems for ER variations (Spaccapietra et al. 1994), XML (Beeri et al. 1999), data warehouses (Calvanese et al. 1998), semi-structured data (Bergamaschi et al. 1999), and relational and object-oriented databases (Buneman et al. 1992). These works, like ours, consider merging only the models, not the instances of the models. Some models, such as ontologies and ER diagrams, have no instance data, and merging the models is a necessary precursor to merging those models with instance data.

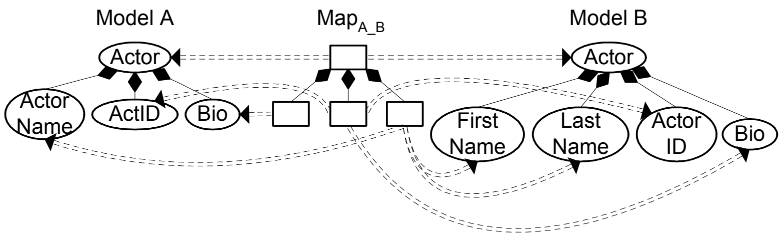


Figure 5.1: Examples of models to be merged

The similarities among these solutions offer an opportunity for abstraction. One important step in this direction was an algorithm for schema merging and conflict resolution of models by Buneman, Davidson, and Kosky (hereafter *BDK*) (Buneman et al. 1992). Given a set of pair-wise correspondences between two models that have Is-a and Has-a relationships, BDK give a formal definition of merge and show how to resolve a certain kind of conflict to produce a unique result. We use their theoretical algorithm as a base, and expand the range of correspondences, model representations, conflict categories, and applications, yielding a robust and practical solution.

The main contribution of this chapter is the design of a practical generic merge operator. It includes the following specific contributions:

- Generic requirements for merge that every design should satisfy. These requirements differ from the ones in Chapter 4 by (1) being for *any* application in which merge is used, not just creating a mediated schema for data integration (2) being for any data model, not just relational and (3) using a more general mapping language.
- The use of an input mapping that is a first-class model, enabling us to express richer correspondences than previous approaches.
- A characterization of when Merge can be automatic.
- A taxonomy of the conflicts that can occur and a definition of conflict resolution strategies using the mapping's richer correspondences.
- Experimental evaluation showing that our approach scales to a large real world application.
- An analysis that shows our approach subsumes previous merge work.

Merge is one of the operators proposed in (Bernstein 2003) as part of *Model Management*, a framework that consists of operators for manipulating models and mappings. Other Model Management operators include: Match, which returns a mapping between two given models; Apply, which applies a given function to all the elements of a model; and Diff, which, given two models and a mapping, returns a model consisting of all items in the first model that are not in the second model (Bernstein 2003). In our analysis of previous work, we sometimes refer to other Model Management operators to show that our approach subsumes the previous work.

The chapter is structured as follows: Section 5.2 gives a precise definition of Merge. Section 5.3 describes our categorization of conflicts that arise from combining two models. Section 5.4

describes how to resolve conflicts in Merge, often automatically. Section 5.5 defines our merge algorithm. Section 5.6 discusses the associativity and commutativity of Merge. Section 5.7 discusses alternate merge definitions and how to simulate one of them using Merge and other Model Management operators. Section 5.8 evaluates Merge experimentally by merging two large anatomy databases and conceptually by showing how our approach subsumes previous work. Section 5.9 is the conclusion.

5.2 Problem Definition

5.2.1 Representation of Models

Defining a representation for models requires (at least) three meta-levels. Using conventional meta-data terminology, we can have: a *model*, such as the database schema for a billing application; a *meta-model*, which consists of the type definitions for the objects of models, such as a meta-model that says a relational database schema consists of table definitions, column definitions, etc.; and a *meta-meta-model*, which is the representation language in which models and meta-models are expressed, for example a generic meta-meta-model may say that a schema could consist of objects, where an object could be a table, XML element, or a class definition.

The goal of our merge operator, Merge, is to merge two models based on a mapping between them. For now, we discuss Merge using a small meta-meta-model (which we extend in Section 5.4.1). It consists of the following:

1. *Elements* with semi-structured properties (i.e., for an element X, there may exist 0, 1, or many p properties). Elements are the first class objects in a model. Three properties are required: Name, ID, and History. Name is self-explanatory. ID is the element's unique identifier, used only by the Model Management system. History describes the last operator that acted on the element.

2. Binary, directed, kinded[11] *relationships* with cardinality constraints. A relationship is a connection between two elements. We enumerate relationship kinds in Section 5.4.1. Relationships can be either explicitly present in the model or *implied* according to the meta-meta-model's rules. Such a rule might say that "a is a b" and "b is a c" implies that "a is a c." Relationship cardinalities are omitted from the figures for ease of exposition.

These models are much more general than the relational models we have been using before now. In Chapter 6 we show how to encode relational schemas in this representation. Figure 5.1 shows an example model in this small meta-meta-model; elements are shown as nodes, the value of the Name property is the node's label, mapping relationships are edges with arrowheads, and sub-element relationships are diamond-headed edges.

5.2.2 Merge Inputs

The inputs to Merge are the following:

1. Two models: A and B.
2. A mapping, Map_{A_B}, which is a model that defines how A and B are related.
3. An optional designation that one of A or B is the *preferred model*. When Merge faces a choice that is not specified in the mapping, it chooses the option from the preferred model, if there is one.
4. Optional overrides for default Merge behavior (explained further below).

The input mapping is more expressive than just simple equalities; it is a first-class model consisting of elements and relationships. Some of its elements are *mapping elements*. A mapping element is like any other element except it also is the origin of a *mapping relationship*, M(x, y), which specifies that the origin element, x, *represents* the destination element, y. So a

[11] We use the word "kinded" to denote that the relationships are of different named kinds. We use "kinded" rather than "typed" to avoid confusion with the fact that there is a kind of relationship called "Type-of".

given mapping element, x, represents all elements y such that M(x, y). All elements of Map_{A_B} in Figure 5.1 are mapping elements. In Map_{A_B} in Figure 5.3 AllBios is not a mapping element.

There are two kinds of mapping relationships: equality and similarity. An *equality mapping relationship* Me asserts that for all y1, y2 ∈ Y such that Me(x, y1) and Me(x, y2), y1=y2. All elements represented by the same equality mapping relationship are said to *correspond* to one another. A *similarity mapping relationship* Ms asserts that the set of all y ∈ Y such that Ms(x, y) are related through a complex expression that is not interpreted by Merge. This expression is the value of x's Expression property, which is a property of all mapping elements that are the origin of mapping similarity relationships. Equality mapping relationships are represented by double-dashed-lines (=); similarity mapping relationships are represented by double-wavy-lines (≈). Figure 5.2 shows a mapping that mostly consists of mapping equality relationships but also has mapping similarity relationships originating from element m_2.

Whereas the conjunctive mappings we used in Chapter 4 have instance-level semantics, these mappings are meant to be more generic, and thus they are uninterpreted. In Chapter 6 we show how to encode conjunctive mappings in this representation.

Note that in (Pottinger et al. 2003) instead of having mapping equality relationships and mapping similarity relationships, we distinguished between equality vs. similarity of elements x ∈ Map_{E_F} and y ∈ E or F by (1) a relationship M(x, y) indicating that there was a mapping relationship that originated at x and ended at y and (2) the "how related" property of x denoting whether the elements are related by equality or similarity.

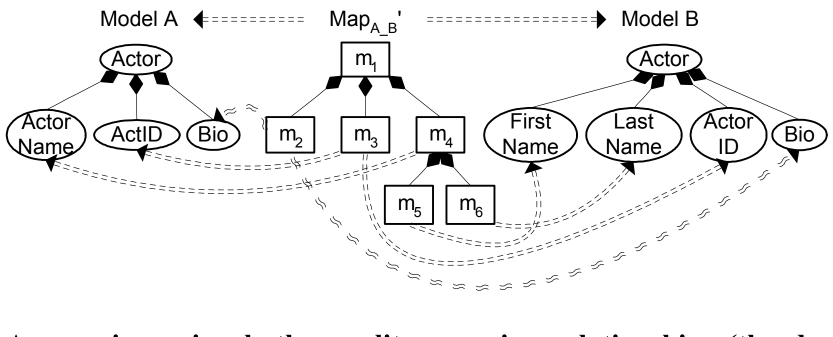


Figure 5.2: A mapping using both equality mapping relationships (the double-dashed-lines) and similarity mapping relationships (the double-wavy lines)

Given this rich mapping structure, complex relationships can be defined between elements in A and B, not just simple correspondences. For example, the mapping in Figure 5.3 (which is between the same models in Figure 5.1) shows that the FirstName and LastName of model B should be sub-elements of the ActorName element of model A; this is expressed by element m_4, which represents ActorName in A and contains elements m_5 and m_6 which represent FirstName and LastName respectively in B.

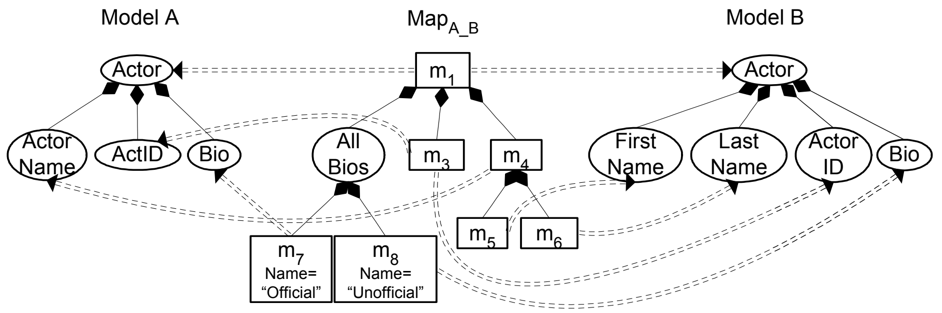


Figure 5.3: A more complicated mapping between the models in Figure 5.1

A mapping can also contain non-mapping elements that do not represent elements in either A or B but help describe how elements in A and B are related, such as AllBios in Figure 5.3. The mapping Map_{A_B} in Figure 5.3 indicates that A.Bio should be renamed "Official," B.Bio should be renamed "Unofficial," and both are contained in a new element, AllBios, that appears only in Map_{A_B}.

A mapping can express *similarity* between elements in A and B. For example, if A.Bio is a French translation of B.Bio and this needs to be reflected explicitly in the merged model, they could be connected by similarity mapping relationship to a mapping element with an Expression property "A.Bio = English2French(B.Bio)" not shown in Figure 5.3.

Prior algorithms, whose mappings are not first-class models, cannot express these relationships. Often, they require user intervention during Merge to incorporate relationships that are more complicated than simply equating two elements. Merge can encode simple correspondences in a mapping, so it can function even if a first-class mapping is unavailable.

5.2.3 Merge Semantics

The output of Merge is a model that retains all non-duplicated information in A, B, and Map_{A_B}; it collapses information that Map_{A_B} declares redundant. If we consider the mapping to be a third model, this definition corresponds to the least-upper-bound defined in BDK (Buneman et al. 1992), "a schema that presents all the information of the schemas being merged, but no additional information." We require Merge to be generic in the sense that it does not require its inputs or outputs to satisfy any given meta-model. We consider another merge definition in Section 5.7. In Chapter 6, we show how Merge compares with the conjunctive mediated schema creation in Chapter 4.

We now define the semantics of Merge more precisely. The function "Merge(A, Map_{A_B}, B) $\rightarrow$ G" merges two models A and B based on a mapping Map_{A_B}, which describes how A and B are related, producing a new model G that satisfies the following Generic Merge Requirements (GMRs). [12]

1. **Element preservation:** Each element in the input has a corresponding element in G. Formally: each element $e \in A \cup B \cup \text{Map}_{A\_B}$ corresponds to exactly one element $e' \in G$. We define this correspondence as $\chi(e, e')$; informally $\chi(e, e')$ represents that e', is derived in part from e.
2. **Equality preservation:** Input elements are mapped to the same element in G if and only if they are equal in the mapping, where equality in the mapping is transitive. Formally: two elements $s, t \in A \cup B$ are said to be *equal* in Map_{A_B} if there is an element $v \in A \cup B$ and an equality mapping element x such that Me(x, s) and Me(x, v), where either $v = t$ or v is equal to t in Map_{A_B}. If two elements $s, t \in A \cup B$ are equal in Map_{A_B}, then there exists a unique element $e \in G$ such that $\chi(s, e)$ and $\chi(t, e)$. If s and t are not equal in Map_{A_B}, then there is no such e, so s and t correspond to different elements in G.

[12] Whereas the Mediated Schema Criteria in Chapter 4 were interpreted with respect to the semantics that they are used in, the GMRs are left uninterpreted so that they are more generic.

3. **Relationship preservation:** Each input relationship is explicitly in or implied by G. Formally: for each relationship $R(s, t) \in A \cup B \cup Map_{A\_B}$ where $s, t \in A \cup B \cup Map_{A\_B}$ and R is not a mapping relationship Me(s, t) or Ms(s, t) with $s \in Map_{A\_B}$, if $\chi(s, s')$ and $\chi(t, t')$, then either $s' = t'$, $R(s', t') \in G$, or $R(s', t')$ is implied in G.
4. **Similarity preservation:** Elements that are declared to be similar (but not equal) to one another in Map_{A_B} retain their separate identity in G and are related to each other by some relationship. More formally, for each pair of elements $s, t \in A \cup B$, where s and t are the destination of similarity mapping relationships originating at a mapping element, x, in Map_{A_B} and s and t are not equal, there exist elements $e, s', t' \in G$ and a meta-model specific non-mapping relationship R such that $\chi(s, s')$, $\chi(t, t')$, $R(e, s')$, $R(e, t')$, $\chi(x, e)$, and e includes an expression relating s and t.
5. **Meta-meta-model constraint satisfaction:** G satisfies all constraints of the meta-meta-model. G may include elements and relationships in addition to those specified above that help it satisfy these constraints. Note that we do not require G to conform to any meta-model.
6. **Extraneous item prohibition:** Other than the elements and relationships specified above, no additional elements or relationships exist in G.
7. **Property preservation:** For each element $e \in G$, e has property p if and only if $\exists\, t \in A \cup B \cup Map_{A\_B}$ s.t. $\chi(t, e)$ and t has property p.
8. **Value preference:** The value, v, of a property p, for an element e is denoted $p(e) = v$. For each $e \in G$, p(e) is chosen from mapping elements corresponding to e if possible, else from the preferred model if possible, else from any element that corresponds to e. More formally:
   - $T = \{t \mid \chi(t, e)\}$
   - $J = \{j \in (T \cap Map_{A\_B}) \mid p(j) \text{ is defined}\}$
   - $K = \{k \in (T \cap \text{the preferred model}) \mid p(k) \text{ is defined}\}$
   - $N = \{n \in T \mid p(n) \text{ is defined}\}$
     - If $J \neq \emptyset$ then $p(e) = p(j)$ for some $j \in J$

- Else if $K \neq \emptyset$, then $p(e) = p(k)$ for some $k \in K$
- Else $p(e) = p(n)$ for some $n \in N$

GMR 8 illustrates our overall conflict resolution strategy: give preference first to the option specified in the mapping (i.e., the explicit user input), then to the preferred model, else choose a value from one of the input elements. The ID and History properties are determined differently as discussed in Section 5.5.

For example, the result of merging the models in Figure 5.3 is shown in Figure 5.4. Note that the relationships Actor-FirstName and Actor-LastName in model B and the Actor-Bio relationships in both models are implied by transitivity in Figure 5.4, so GMR 3 is satisfied.

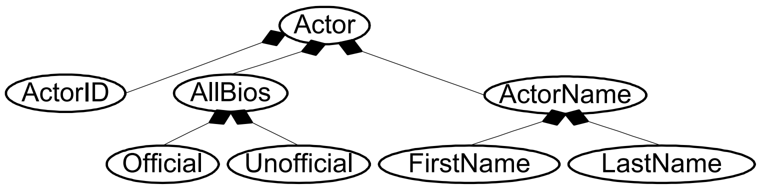


Figure 5.4: The result of performing the merge in Figure 5.3

The GMRs are not always satisfiable. For example, if there are constraints on the cardinality of relationships that are incident to an element, then there may be no way to preserve all relationships. Depending on the relationships and meta-meta-model constraints, there may be an automatic resolution, manual resolution or no possible resolution that satisfies the GMRs. In Section 5.4 we present conflict resolutions for a set of common constraints and discuss when such resolution can be automatic. We also specify default resolution strategies for each category of constraint and note when resolution can be made to satisfy the GMRs outlined above.

5.3 Conflict Resolution

Determining the merged model requires resolving conflicts in the input. We categorize conflicts based on the meta-level at which they occur:

- **Representation conflicts** (Section 5.3.1) are caused by conflicting representations of the same real world concept – a conflict at the *model level*. For example, one representation conflict is that in Figure 5.1 model A represents Name by one element, ActorName, while model B represents it by two elements, FirstName and LastName. Resolving these conflicts requires manual user intervention. Such conflict resolution is necessary for many uses of mappings – not just Merge. Hence we isolate it from Merge by requiring it to be captured in the input mapping.

- **Meta-model conflicts** (Section 5.3.2) are caused by the constraints in the *meta-model* (e.g., SQL DDL). For example, suppose that in Figure 5.3 Actor is a SQL table in model A, an XML database in model B, and a SQL table in the merged model. If the mapping in Figure 5.3 is used, there will be a meta-model conflict in the merge result because SQL DDL has no concept of sub-column. This does not violate any principle about the *generic* merged outcome. Rather, it is meta-model-specific. Enforcing such constraints is inherently non-generic, so we resolve them using a separate operator after Merge.
- **Fundamental conflicts** (Section 5.3.3) are caused by constraints in the meta-meta-model. These conflicts must be resolved to ensure that the merge result conforms to the meta-meta-model. For example, if a model had an element with two types, this would be a conflict in many meta-models, not just in one. Unlike representation conflicts, fundamental conflicts must be resolved by Merge since subsequent operators count on the fact that the Merge result is a well-formed model.

5.3.1 Representation Conflicts

A representation conflict arises when two models describe the same concept in different ways. For example, in Figure 5.1 model A represents Name by one element, ActorName, while model B represents it by two elements, FirstName and LastName. After merging the two models, should Name be represented by one, two or three elements? The decision is application dependent.

Merge resolves representation conflicts using the input mapping. Having a mapping that is a model allows us to specify that elements in models A and B are either:

- The same, by being the destination of equality mapping relationships that originate at the same mapping element. Merge can collapse these elements into one element that includes all relationships incident to the elements in the conflicting representations.

- Related by relationships and elements in our meta-meta-model. E.g., we can model FirstName and LastName in A as sub-elements of ActorName in B by the mapping shown in Figure 5.3.
- Related in some more complex fashion that we cannot represent using our meta-meta-model's relationship kinds. E.g., we can represent that ActorName equals the concatenation of FirstName and LastName by a mapping element that has similarity mapping relationships incident to all three and an Expression property describing the concatenation. Resolution can be done by a later operator that understands the semantics of Expression.

The mapping can also specify property values. For example, in Figure 5.3 $\text{Map}_{\text{A\_B}}$ specifies that the elements contained by AllBios should be named Official and Unofficial.

Solving representation conflicts has been a focus of the ontology merging literature (Noy et al. 1999; Noy et al. 2000) and of database schema merging (Batini et al. 1986; Spaccapietra et al. 1994).

5.3.2 Meta-model Conflicts

A meta-model conflict occurs when the merge result violates a meta-model-specific (e.g., SQL DDL) constraint. For example, suppose that in Figure 5.3 Actor is a SQL table in model A, an XML database in model B, and a SQL table in the merged model. If the mapping in Figure 5.3 is used, there will be a meta-model conflict in the merge result because SQL DDL has no concept of sub-column. This does not violate any principle about the *generic* merged outcome. Rather, it is meta-model-specific. Traditionally, merge results are required to conform to a given meta-model during the merge. However, since Merge is meta-model independent, we do not resolve this category of conflict in Merge. Instead, we break out coercion as a separate step, so that Merge remains generic and the coercion step can be used independently of Merge. We therefore introduce an operator, EnforceContraints, that coerces a model to obey a set of constraints. This operator is necessarily meta-model specific. However, it may be possible to implement it in a generic way, driven by a declarative specification of each meta-model's constraints. EnforceContraints would enforce other constraints, such as integrity constraints, as well. Preliminary work suggests that some of the work created for the purpose of translating between data models can be leveraged in order to create this operator. In particular some of the work by Atzeni and Torlone in viewing meta-models as consisting of different patterns and changing a schema from one meta-model to another (Atzeni et al. 1996) seems promising, as does some of the similar work in M(DM) (Barsalou et al. 1992).

5.3.3 Fundamental Conflicts

The third and final category of conflict is called a fundamental conflict. It occurs above the meta-model level at the meta-meta-model level, the representation to which all models must adhere. A fundamental conflict occurs when the result of Merge would not be a model due to violations of the meta-meta-model. This is unacceptable because later operators would be unable to manipulate it.

One possible meta-meta-model constraint is that an element has at most one type. We call this the *one-type restriction*. Given this constraint, an element with two types manifests a fundamental conflict. For example in the model fragments in Figure 5.5(a) ZipCode has two types: Integer and String. In the merge result in Figure 5.5(b), the two ZipCode elements are collapsed into one element. But the type elements remain separate, so ZipCode is the origin of two type relationships.

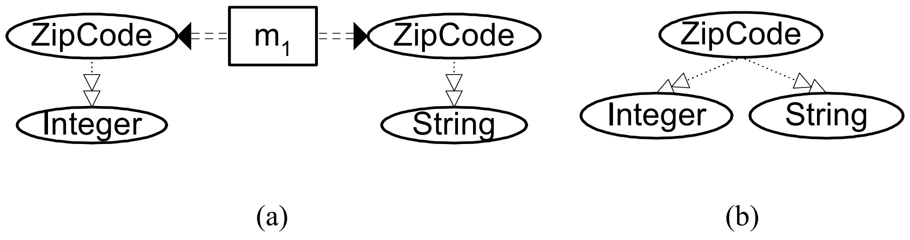


Figure 5.5: A merge that violates the one-type restriction

Since Merge must return a well-formed instance of the meta-meta-model, it must resolve fundamental conflicts. Resolution rules for some fundamental conflicts have been proposed, such as (Buneman et al. 1992) for the one-type restriction. We have identified other kinds of fundamental conflicts and resolution rules for them which we describe in Section 5.4 and incorporate into our generic Merge.

Alternate merge semantics might ignore GMR 1 (Element preservation) and resolve fundamental conflicts by requiring the mapping to include preferences to resolve the conflict. For example, the mapping in Figure 5.5 could specify that the type of ZipCode is String, and Merge should ignore the conflicting information that ZipCode is of type Integer. In general we do not recommend this strategy because it loses information (e.g., that ZipCode is of type Integer). However, since Merge allows users to specify alternate resolutions, as discussed in Section 5.4, this strategy can be easily incorporated by specifying it as the resolution strategy for one-type conflicts.

The choice of meta-meta-model, particularly the constraints on the relationships, is therefore integrally related to Merge. However, since we are skeptical that there is a meta-meta-model capable of solving all meta-data management problems, we chose the following approach: We define the properties of Merge using very few assumptions about the meta-meta-model — only that it consists of elements and relationships. We then define fundamental conflict resolution for a meta-meta-model that includes many of the popular semantic modeling constructs. Finally we describe other typical meta-meta-model conflicts and provide conflict resolution strategies for them.

5.4 Resolving Fundamental Conflicts

The meta-meta-models we consider are refinements of the one described in Section 5.2.1. Section 5.4.1 describes *Vanilla*, an extended entity-relationship-style meta-meta-model that includes many popular semantic modeling constructs. Section 5.4.2 describes our merging strategy, both for Vanilla and for relationship constraints that may be used in other meta-meta-models.

5.4.1 The Vanilla Meta-Meta-Model

Elements are first class objects with semi-structured properties (i.e., for an element X, there may exist 0, 1, or many p properties). Name, ID, and History are the only required properties. These are properties of the element viewed as an instance, not as a template for instances. For example, suppose an element e represents a class definition, such as Person. Viewing e as an instance, it has a Name property whose value is “Person,” and might have properties

CreatedBy, LastModifiedBy, Comments, and IsInstantiable. To enable instances of Person to have a property called Name (thereby viewing e as a template for an instance), we create a relationship from e to another element, a, where Name(a) = "Name."

Relationships are binary, directed, kinded, and have an optional cardinality constraint. They are also ordered, as in XML, but the order can be ignored in meta-models that do not use it. A relationship kind is one of "Associates", "Contains", "Has-a", "Is-a", and "Type-of" (described below). Reflexive relationships are disallowed. We assume that between any two elements there is at most one relationship of a given kind and cardinality pairing.

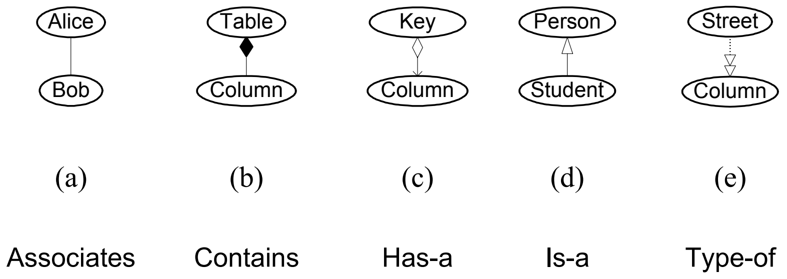


Figure 5.6: Different relationship kinds in Vanilla

There are cases where the previous restriction is inconvenient. For example, one might want two kinds of Has-a relationships between "Movie" and "Person", namely "director" and "actor". This can be handled either by specializing Person into two sub-elements, or by reifying the director and actor Has-a relationships (i.e., turn the relationships into objects), which is the choice used in Vanilla. We disallow multiple named relationships of the same cardinality and kind between two elements because it leads to a need for correspondences between named relationships of different models. E.g., if the director and actor relationships are called "réalisatuer" and "acteur" in another model, we need a relationship between director and réalisatuer and between actor and acteur. These correspondences would complicate the meta-meta-model. The same expressiveness is gained by reifying relationships, thereby avoiding this complexity. Merge does not treat these reified relationships specially; since the GMRs require all elements and relationships to appear in the merged model, they will appear in the merged model as well.

A relationship R(x, y) between elements x and y may be a mapping similarity or equality relationship, Me(x, y) or Ms(x, y), described earlier, or one of the following:

- Associates - A(x, y) means x is associated with y. This is the weakest relationship that can be expressed. It has no constraints or special semantics. Figure 5(a) says that Alice is Associated with Bob.
- Contains - C(x, y) means *container* x contains *containee* y. Intuitively, a containee cannot exist on its own; it is a part of its container element. Operationally, this means that if all of the containers of an element, y, are deleted, then y must be deleted. Contains is a transitive relationship and must be acyclic. If C(x, y) and x is in a model M, then y is in M as well. Figure 5.6(b) says that Table Contains Column.
- Has-a - H(x, y) means x has a sub-component y (sometimes called "weak aggregation"). Has-a is weaker than Contains in that it does not propagate delete and can be cyclic. Figure 5.6(c) says that Key Has-a Column.
- Is-a - I(x, y) means x is a specialization of y. Like Contains, Is-a is transitive, acyclic, and implies model membership. Figure 5.6(d) says that Student Is-a Person.
- Type-of - T(x, y) means x is of type y. Each element can be the origin of at most one Type-of relationship (the one-type restriction described in Section 5.3.3). Figure 5.6(e) says that the Type-of Street is Column.

Vanilla has the following *cross-kind-relationship implications* that imply relationships based on explicit ones:

- If T(q, r) and I(r, s) then T(q, s)
- If I(p, q) and H(q, r) then H(p, r)
- If I(p, q) and C(q, r) then C(p, r)
- If C(p, q) and I(q, r) then C(p, r)
- If H(p, q) and I(q, r) then H(p, r)

A model L is a triple (E_L, Root(L), Re_L) where E_L is the set of elements in L, Root(L) $\in E_L$ is the root of L, and Re_L is the set of relationships in L. Given a set of elements E and relationships Re (which may include mapping relationships), membership in L is determined by applying the following rules to Root(L) $\in$ E, adding existing model elements and relationships until a fix point is reached (i.e., until applying each rule results in no new relationships):

- I(x, y), x $\in$ E_L $\rightarrow$ y $\in$ E_L; if an element x is in the model, then its generalization y is in the model
- C(x, y), x $\in$ E_L $\rightarrow$ y $\in$ E_L; if a container x is in the model, then its containee y is in the model
- T(x, y), x $\in$ E_L $\rightarrow$ y $\in$ E_L; if an element x is in the model, then its type y is in the model
- R(x, y), x $\in$ E_L, y $\in$ E_L $\rightarrow$ R(x, y) $\in$ Re_L
- Me(x, y), x $\in$ E_L $\rightarrow$ Me(x, y) $\in$ Re_L
- Ms(x, y), x $\in$ E_L $\rightarrow$ Ms(x, y) $\in$ Re_L

Since a mapping is a model its elements must be connected by relationships indicating model containment (Contains, Is-a, or Type-of). However, since these relationships obfuscate the mapping, we often omit them from figures when they do not affect Merge's behavior.

In what follows, when we say relationships are "implied", we mean "implied by transitivity and cross-kind-relationship implication."

We define two models to be *equivalent* if they are identical after all implied relationships are added to each of them until fixpoint is reached (i.e., applying each rule results in no new relationships).

A *minimal covering* of a model is an equivalent model that has no edge that is implied by the union of the others. A model can have more than one minimal covering. For example, the model in Figure 5.7(a) is a minimal covering of the model in Figure 5.7(b).

To ensure that the merge result G is a model, we require that Root(Map_{A_B}) is a mapping element with Me(Root(Map_{A_B}), Root(A)) and Me(Root(Map_{A_B}), Root(B)), and that Root(Map_{A_B}) is the origin of no other mapping relationships.

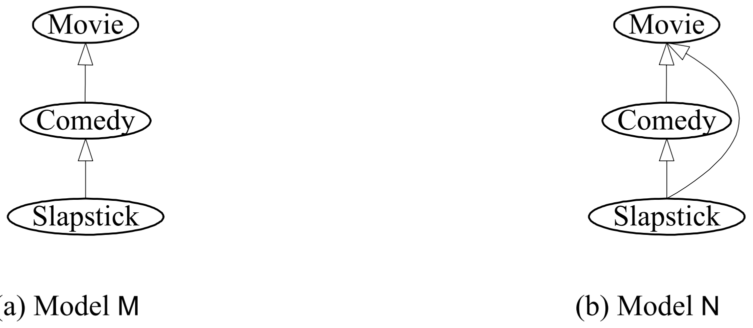


Figure 5.7: Model M is a minimal covering of model N

5.4.2 Meta-Meta-Model Relationship Characteristics and Conflict Resolution

This section explores resolution of fundamental conflicts in Merge with respect to both Vanilla and other meta-meta-models: what features lead to an automatic Merge, when manual intervention is required, and default resolutions. The resolution strategies proposed here are incorporated in the Merge algorithm in Section 5.5. Since the default resolution may be inadequate due to application specific requirements, Merge allows the user to either (1) specify an alternative function to apply for each conflict resolution category or (2) resolve the conflict manually.

Vanilla has only two fundamental constraints (i.e., that can lead to fundamental conflicts): (1), the Is-a and Contains relationships must be acyclic and (2) the one-type restriction. These fundamental conflicts can be resolved fully automatically in Vanilla.

5.4.2.1 Relationship-Element Cardinality Constraints

Many meta-meta-models restrict some kinds of relationships to a maximum or minimum number of occurrences incident to a given element. For example, the one-type restriction says that no element can be the origin of more than one Type-of relationship. Such restrictions can specify minima and/or maxima on origins or destinations of a relationship of a given kind.

Cardinality Constraints in Vanilla - Merge resolves one-type conflicts using a customization of the BDK algorithm (Buneman et al. 1992) for Vanilla; a discussion of the change can be found in Appendix C. Recall Figure 5.5 where the merged ZipCode element has both Integer and

String types. The BDK resolution creates a new type that inherits from both Integer and String and replaces the two Type-of relationships from ZipCode by one Type-of relationship to the new type, as shown in Figure 5.8. Note that both of the original relationships (ZipCode is of type Integer and String) are implied.

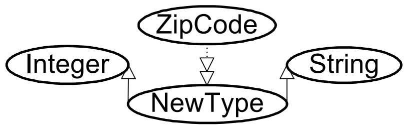


Figure 5.8: Resolving the one-type conflict of Figure 5.5

This creates a new element, NewType in Figure 5.8, whose Name, ID, and History properties must be determined. The ID property is assigned an unused ID value, and Name is set to be the names of the elements it inherits from, delineated by a slash; e.g., NewType in Figure 5.8 is named "Integer/String." The History property records why the element came into existence, in this case, that Merge created it from the elements Integer and String. As with any other conflict resolution, this behavior can be overridden.

This approach to resolving one-type conflicts is an example of a more general approach, which is the one we use as a default: to resolve a conflict, alter explicit relationships so that they are still implied and the GMRs are still satisfied. Thus, the more implication rules in the meta-meta-model, the easier conflict resolution is.

Requiring that G, the output of Merge, is a model is a form of a minimum element-relationship cardinality; by Vanilla's definition, a model G *satisfies model membership* if all elements of G are reachable from G's root by following containment relationships: Is-a, Contains, and Type-of. Hence, each element must be the origin or destination of at least one such relationship (depending on the relationship containment semantics). Ignoring conflict resolution, we know that G satisfies this constraint:

- χ(Root(A), Root(G)), χ(Root(B), Root(G)), χ(Root(Map_{A_B}), Root(G)) from the input and GMR 2 (Equality preservation).

- Root(G) is not the destination of any relationships (and hence is a candidate to be root) because of GMR 6 (Extraneous item prohibition) and because it only corresponds to Root(A), Root(B), and Root($\text{Map}_{\text{A\_B}}$) which likewise are roots.
- Each element $g \in G$ can be determined to be a member of the model with root Root(G): Each element e such that $\chi(e, g)$ must be a member of A, B, or $\text{Map}_{\text{A\_B}}$. Assume without loss of generality that $e \in A$. Then there must be a path P of elements and relationships from Root(A) to e that determines that e is in A. By GMR 1 (Element preservation) and GMR 3 (Relationship preservation), a corresponding path P′ must exist in G, and hence g is a member of the model with root Root(G).

Hence, conflict resolution notwithstanding, G is guaranteed to satisfy model membership. After conflict resolution for Vanilla, G still satisfies model membership; the BDK solution to the one-type restriction only adds relationships and elements that satisfy model containment. As shown in Section 5.4.2.2, the acyclic resolution only collapses a cycle, which cannot disturb the model membership of the remaining element.

Cardinality Constraints in General - There are two kinds of relationship-element cardinality constraints: for some *n*: (1) at least *n* relationships of a given kind must exist (*minimality constraints*) and (2) at most *n* relationships of a given kind may exist (*maximality constraints*).

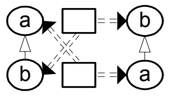
Since Merge (excluding conflict resolution) preserves all relationships specified in the input, the merged model is guaranteed to preserve minimality constraints. For example, one potential minimality constraint is that each element must be the origin of one Type-of relationship. If this were the case, then each of the input models, A, B, and $\text{Map}_{\text{A\_B}}$ would have to obey the constraint. Hence each element in A, B, and $\text{Map}_{\text{A\_B}}$ would be the origin of at least one Type-of relationship. Since Merge preserves the relationships incident to each element, each element in G is also the origin of at least one Type-of relationship. Conflict resolution may break this property, so conflict resolution strategies must consider these kinds of constraints.

More care is required for a maximality constraint, such as the one-type restriction. If it occurs in a meta-meta-model, the generic merge attempts resolution by removing redundant relationships. Next, the default Merge resolution will look for a cross-kind implication rule that can resolve the conflict (i.e., apply the default resolution strategy). If no such rule exists, then we know of no way to resolve the conflict while still adhering to the GMRs. To continue using the one-type restriction as an example, first we calculate a minimal covering of the merged model and see if it still has a one-type restriction conflict. If so, then we apply a cross-kind implication rule (if T(q, r) and I(r, s) then T(q, s)) which allows us to resolve the conflict and still satisfy the GMRs.

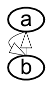
5.4.2.2 Acyclicity

Many meta-meta-models require some relationship kinds to be acyclic. In Vanilla, Is-a and Contains must be acyclic. In this section, we first consider acyclic constraints in Vanilla and then acyclicity constraints in general.

Acyclicity in Vanilla - Merging the example in Figure 5.9 (a) would result in Figure 5.9 (b) which has a cycle between elements a and b. Since Is-a is transitive, a cycle of Is-a relationships implies equality of all of the elements in the cycle. Thus Merge's default solution is to collapse the cycle into a single element. As with all conflicts, users can override with a function or manual resolution. To satisfy GMR 7 (Property preservation), the resulting merged element contains the union of all properties from the combined elements. GMR 8 (Value preference) dictates the value of the merged element's properties.



(a)



(b)

Figure 5.9: Merging the models in (a) causes the cycle in (b)

Acyclicity Constraints in General - If the constrained relationship kind is not transitive, collapsing the cycle would not retain the desired semantics in general (although it does work for cycles of length two). The default resolution is to see if any cross-kind-relationship implications allow all relationships to exist implicitly without violating the acyclicity constraint. If so, the conflict can be resolved automatically. Without such a relationship implication it is impossible to merge the two models while retaining all of the relationships; either some default resolution strategy must be applied that does not retain all relationships, or human intervention is required.

5.4.2.3 Other Relationship Conflicts

The following are conflicts that may occur in meta-meta-models other than Vanilla:

- Certain relationship kinds many not be allowed to span meta-levels or Is-a-levels. For example, an Is-a hierarchy may not cross meta-levels, or a Type-of relationship may not cross Is-a levels.
- If a meta-meta-model allows only one relationship of a given kind between a pair of elements, the cardinality of the relationship must be resolved if there is a conflict. For example, in Figure 5.10 what should be the cardinality of the Contains relationship between Actor and ActID? 1:n? m:1? m:n? One could argue that it should be m:n because this is the most general, however this may not be the desired semantics. Any resolution of this conflict is going to lose information and therefore will not satisfy GMR 3 (Relationship preservation), so no generic resolution can satisfy the GMRs.

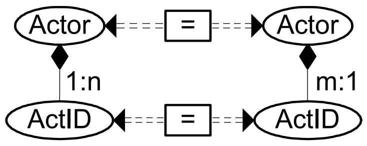


Figure 5.10: Merging multiple cardinalities

- If only one set of specializations of an element may be declared disjoint, merging two orthogonal such sets requires conflict resolution, e.g., if actors are specialized as living/dead in one model and male/female in another.

5.5 The Merge Algorithm

This section describes an algorithm for Merge that satisfies the GMRs; an implementation of this algorithm is discussed in Section 5.8.1.

Definition 5.1: (Merge).

1. **Initialize** the merge result G to ∅.
2. **Elements:** Induce an equivalence relation by grouping the elements of A, B, and Map_{A_B}. Initially each element is in its own group. Then:

a. If a relationship $Me(d, e)$ exists between an element $e \in (A \cup B)$ and a mapping element $d \in Map_{A\_B}$, then combine the groups containing d and e.

b. After iterating (a) to a fix point, create a new element in G for each group.

3. **Element Properties:** Let e be a merged element in G corresponding to a group I. The value v of property p of e, $p(e) = v$, is defined as follows:

a. The properties of e are the union of the properties of the elements of I. Merge determines the values of properties of e other than History and ID as follows:

$J = \{j \in (I \cap Map_{A\_B}) \mid p(j) \text{ is defined}\}$

$K = \{k \in (I \cap \text{the preferred model}) \mid p(k) \text{ is defined}\}$

$N = \{n \in I \mid p(n) \text{ is defined}\}$

i. If $J \neq \emptyset$, then $p(e) = p(j)$ for some $j \in J$

ii. Else if $K \neq \emptyset$, then $p(e) = p(k)$ for some $k \in K$

iii. Else $p(e) = p(n)$ for some $n \in N$

By definition of N, some value for each property of e must exist. In (i) – (iii) if more than one value is possible, then one is chosen arbitrarily.

b. Property $ID(e)$ is set to an unused ID value. Property $History(e)$ describes the last action on e. It contains the operator used (in this case, Merge) and the ID of each element in I. This implicitly connects the Merge result to the input models and mapping without the existence of an explicit mapping between them.

4. **Relationships:** For every two elements e′ and f′ in G that correspond to distinct groups E and F, where E and F do not contain elements that are the origin of similarity mapping relationships, if there exists $e \in E$ and $f \in f$ such that R(e, f) is of kind t and has cardinality c, then create a (single) relationship R(e′, f′) of kind t and cardinality c. Reflexive mapping relationships (i.e., mapping relationships between elements that have been collapsed) are excluded since they no longer serve a purpose. For example, without this exception, after the Merge in Figure 5.3 is performed, the mapping relationship between elements ActorName and m_4 would be represented by a reflexive mapping relationship with both relationship ends on ActorName. However, this relationship is redundant, so we eliminate it from G.
    a. Replace each similarity mapping relationship, Ms, whose origin is m by a Has-a relationship whose origin is e and whose destination is the element of G that corresponds to Ms's destination's group. For example, if the two Bio elements in Figure 5.1 were connected by similarity mapping relationships instead of equality mapping relationships, the result would be as in Figure 5.11.
    b. Relationships originating from an element are ordered as follows:
        i. First those corresponding to relationships in Map_{A_B},
        ii. Then those corresponding to relationships in the preferred model but not in Map_{A_B},
        iii. Then all other relationships. Within each of the above categories, relationships appear in the order they appear in the input. Finally, Merge removes implied relationships from G until a minimal covering remains.
5. **Fundamental conflict resolution:** After steps (1) – (4) above, G is a duplicate-free union of A, B, and Map_{A_B}, but it may have fundamental

conflicts (i.e., it may not satisfy meta-meta-model constraints). For each fundamental conflict, if a special resolution strategy has been defined, then apply it. If not, apply the default resolution strategy described in Section 5.4.2 □

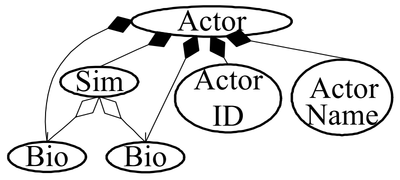


Figure 5.11: Results of Merge on Figure 5.1 if the Bio elements were connected by similarity mapping relationships

Resolving one conflict may interfere with another, or even create another. This does not occur in Vanilla; resolving a one-type conflict does create two Is-a relationships, but they cannot be cyclic since their origin is new and thus cannot be the destination of another Is-a relationship. However, if interference between conflict resolution steps is a concern in another meta-meta-model, then Merge can apply a priority scheme based on an ordered list of conflict resolutions. The conflict resolutions are then applied until reaching fixpoint. Since resolving one-type conflicts cannot create cycles in Vanilla, conflict resolution in Vanilla is guaranteed to terminate. However, conflict resolution rules in other meta-meta-models must be examined to avoid infinite loops.

The algorithm described above satisfies the GMRs in Section 5.2.3. We can see this as follows:

- Step 1 (Initialization) initializes G to the empty set.
- Step 2 (Elements) enforces GMR 1 (Element preservation). It also enforces the first direction of GMR 2 (Equality preservation); elements equated by Map_{A_B} are equated in G. No other work is performed in step 2.

- Step 3 (Element properties) performs exactly the work in GMR 7 (Property preservation) and GMR 8 (Value preference) with the exceptions of the refinements in steps 3b and 3c for the ID and History properties. No other work is performed in step 3.
- In step 4 (Relationships), step 4a enforces GMR 3 (Relationship preservation) and step 4b enforces that a relationship exists between elements mapped as similar, as required in GMR 4 (Similarity preservation). Step 4d removes only relationships that are considered redundant by the meta-meta-model. Step 4c (relationship ordering) is the only step not explicitly covered by a GMR, and it does not interfere with any other GMRs.
- Step 5 (Fundamental conflict resolution) enforces GMR 5 (Meta-meta-model constraint satisfaction) and performs no other work.

If special resolution strategies in step 5 do nothing to violate any GMR or equate any elements not already equated, GMRs 2 (Equality preservation), 4 (Similarity preservation) and 6 (Extraneous item prohibition) are satisfied, and all GMRs are satisfied. Other than special properties (ID and History) and the ordering of relationships, no additional work is performed beyond what is needed to satisfy the GMRs.

5.6 Algebraic Properties of Merge

Since meta-data operations seldom occur in isolation, the properties of sequences of merges must be examined, namely associativity and commutativity. Section 5.6.1 examines the commutativity of Merge. Section 5.6.2 examines the associativity of Merge. Section 5.6.3 discusses commutativity and associativity when the order of merges affects the choice of mappings that are used to drive each merge. For ease of exposition we only consider cases where the outcome is uniquely specified by the inputs (e.g., exactly one correct choice of value exists for each property). To fully explore these properties we rely on the definition of two other Model Management operators: Match and Compose. Describing them in detail is beyond the scope of this thesis. The Match used here is simple since it is based on only the ID and History properties of elements; any Match algorithm (e.g., (Madhavan et al. 2001)) should be able to create the required mappings. We describe a basic Compose operator in Appendix D.

5.6.1 Commutativity

We say that Merge is commutative if, for any pair of models M and N and any mapping Map_{M_N} between them, Merge(M, Map_{M_N}, N) = Merge(N, Map_{M_N}, M) = G. We assume that (1) the same model is the preferred model in each Merge and (2) if there are unspecified choices to be made (e.g., choosing a property value from among several possibilities, each of which is allowed by Merge) the same choice is made in both Merges. We begin by showing that commutativity holds for Merge as specified by the GMRs and then show that it holds for the Merge algorithm specified in Section 5.5.

The commutativity of Merge as specified by the GMRs in Section 5.2.3 follows directly from their definition, since they are symmetric: Rules 1-4 and 6-7 are inherently symmetric. Rule 8 (Value preference) is symmetric as long as the preferred model is the same in both Merges and unspecified choices are the same in both Merges, as stipulated in (2) above. Rule 5 is the resolution of fundamental conflicts. In Vanilla this is symmetric since collapsing all cycles and resolving one type violations using the BDK algorithm are both symmetric[13]. Hence in Vanilla the GMRs are symmetric and thus commutative. However, Merge in other meta-meta-models may not be commutative, depending on their conflicts resolution rules.

The algorithm specification in Section 5.5 is commutative as well; again we show this from the algorithm's symmetry. Steps 1 (Initialize) and 2 (Elements) are symmetric. Steps 3 (Element properties) and 4 (Relationships) are symmetric as long as the preferred model is the same in both merges and arbitrary choices are the same, as stipulated in (2) above. Step 5 (Fundamental conflict resolution) is symmetric if the conflict resolutions are symmetric. As argued above, this holds for conflict resolution in Vanilla, and hence the Merge algorithm is symmetric and thus commutative in Vanilla.

5.6.2 Associativity

We say that two models are *isomorphic* if there is a 1:1 onto correspondence between their elements, and they have the same relationships and properties (but the values of their properties

[13] Appendix C gives the details of our modifications to the BDK algorithm.

and the ordering of their relationships may differ). Merge is associative if, for any three models M, N, and O, and any two mappings Map_{M_N} (between M and N) and Map_{N_O} (between N and O), R is isomorphic to S where:

P = Merge(M, Map_{M_N}, N)

Q = Merge(N, Map_{N_O}, O)

Map_{P_N} = Match(P, N)

Map_{N_Q} = Match(N, Q)

R = Merge(P, Compose(Map_{P_N}, Map_{N_O}), O)

S = Merge(M, Compose(Map_{M_N}, Map_{N_Q}), Q)

Match(P, N) and Match(N, Q) are meant to compute χ as defined in the GMRs by matching the IDs of N with the IDs in the History property of P and Q respectively. N, P, Q, Match(P, N), and Match(N, Q) are shown in Figure 5.12.

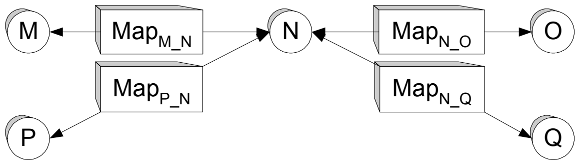


Figure 5.12: Showing associativity requires intermediate mappings

The Compose operator takes a mapping between models A and B and a mapping between models B and C and returns the composed mapping between A and C. Consider Compose(Map_{P_N}, Map_{N_O}). Intuitively it must transfer each mapping relationship of Map_{N_O} having a destination in N to a relationship having a destination in P. Since Map_{P_N} maps each element in N to exactly one element in P, any Compose operator will provide this functionality (such as the one described in Appendix D). Compose(Map_{M_N}, Map_{N_Q}) operates similarly.

A *morphism* is a set of directed *morphism relationships* from elements of one model to elements of another. To show that the two final merged models R and S are isomorphic, we define a morphism $\varphi(R \rightarrow S)$ and show that (i) φ is 1:1 and onto, (ii) $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$, and (iii) x has property p if and only if $\varphi(x)$ has property p. We initially consider the result of Merge ignoring the fundamental conflict resolution. We phrase the argument in terms of the GMRs. We do not repeat the argument for the algorithm, for the following reason: The end of Section 5.5 shows that the algorithm maintains all of the GMRs and that the only additional work done by the merge algorithm beyond the GMRs is (1) to order the relationships and (2) set the value of the ID property. The latter two additions do not affect the isomorphism, so we do not repeat the associativity argument for the algorithm.

We create φ as follows. First we create the morphisms shown as arrows in Figure 5.13 by using Match and Compose the same way they were used to create R and S. We refer to the morphisms in Figure 5.13 that start at R as $Morph_R$ and the morphisms that end at S as $Morph_S$. Next we create five morphisms from R to S by composing $Morph_R$ with $Morph_S$. φ is the duplicate-free union of the five morphisms from the previous step.

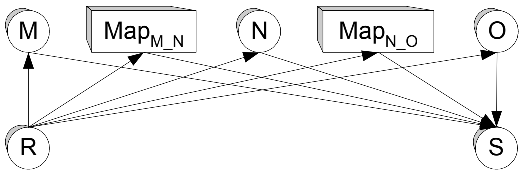


Figure 5.13: Initial morphisms created to show associativity

We want to show that φ is an isomorphism from R to S; this will show that R and S are isomorphic to one another and hence that Merge is associative. We first show that φ is onto (i.e., for all $y \in S$, there exists $x \in R$ such that $\varphi(x) = y$):

1. Let T be the set of elements in M, Map_{M_N}, N, Map_{N_O}, and O.
2. From GMRs 1 (Element preservation) and 2 (Equality preservation) and the definitions of Match and Compose, we know that each element in T is the destination of exactly one morphism relationship in $Morph_R$. I.e., each element in M, Map_{M_N}, N, Map_{N_O}, and O corresponds to exactly one element in the merged model. From GMR 6 (Extraneous item prohibition) and the definitions of Match and Compose we know that each element in R is the origin of at least one morphism relationship to T. I.e., each element in R must correspond to some element in M, Map_{M_N}, N, Map_{N_O}, or O. Recall that we are not considering conflict resolution, so there will be no elements introduced due to GMR 5 (Meta-meta-model constraint satisfaction).

3. Similarly each element of T is the origin of exactly one morphism relationship in Morph_S and each element in S is the destination of at least one Morph_S morphism relationship from T.
4. Hence from steps 2 and 3 and the definitions of Match and Compose, φ is onto.

   Next we show that φ is 1:1(i.e., for all $x_1, x_2 \in R$, $\varphi(x_1) = \varphi(x_2) \rightarrow x_1 = x_2$).
5. From the definition of φ, the only way for φ to not be 1:1 is if:
   a. Some element $r \in R$ is the origin of more than one morphism relationship in Morph_R or
   b. Some element $s \in S$ is the destination of more than one morphism relationship in Morph_S.
6. If statement 5a is true, then from GMR 2 (Equality preservation) and the definitions of Match and Compose, r must be the result of merging some elements from T that were equal in some mapping. Similarly, if statement 5b is true, then from GMR 2 (Equality preservation) and the definitions of Match and Compose, s must be the result of merging some elements from T that were equal in some mapping. We now must show that the equating of the elements is associative and hence for element r in step 5a, each morphism relationship in φ that begins with r will end at the same element in S, thus providing a duplicate morphism relationship and not one that contradicts that φ is 1:1.
7. The equating of elements is associative; this follows directly from the grouping strategy in Merge step 2 (Element properties).
   a. If elements are not equated by a mapping, then they will not be merged into the same object.
   b. Hence the only interesting case is when elements from three different models are mapped to one another; take the example of elements $r \in M$, $t \in N$, $v \in O$ as shown in Figure 5.14. Given the mapping elements m_1 and m_3, r, t, and v are merged into one element, regardless of which order the models are combined. If, however, both relationships implied similarity, then all three elements will exist. If one relationship implied similarity (say r similar to t) and the other equality (say t equals v), then the

resulting elements are the same regardless of order; elements representing r and t are combined, and two elements representing t and v still exist separately.

8. Hence the equating of the elements is associative and φ is 1:1.

Now that we have shown that φ is 1:1, we have one step in showing φ is an isomorphism from R to S. Next we must show that $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$. That is, a relationship exists in R if and only if a corresponding relationship exists in S. GMR 3 (Relationship preservation) guarantees that each relationship R input to Merge has a corresponding relationship R′ in the merged model unless R's origin and destinations have been collapsed into a single element. Similarly the Match and Compose definitions preserve the elements, and a relationship $R(x, y) \in R_R$ if and only if $R(\varphi(x), \varphi(y)) \in R_S$

The last step to show that φ is an isomorphism is to show that each element $r \in R$ has property p if and only if $\varphi(r)$ has property p. GMR 7 (Property preservation) implies that each element in the merged model has a property p if and only if some input element that it corresponds to has property p. From the argument showing that φ is 1:1, we know that the equating of elements is associative, and hence $r \in R$ has property p if and only if $\varphi(r)$ has property p. Hence φ is an isomorphism from R to S and Merge is associative.

Merge is not associative with respect to the *values* of properties. Their value is determined as explained in GMR 8 (Value preference). After a sequence of Merges, the final value of a property may depend on the order in which the Merges are executed. This occurs because the value assigned by the last Merge in the sequence can overwrite the values of any Merges that preceded it. For example, in Figure 5.14 the mapping element m_1 in Map_{M_N} specifies the value a for property Bio. In addition, the Merge definition specifies that O is the preferred model for the merge of N and O. If the sequence of operators is:

Merge(M, Map_{M_N}, N) $\rightarrow$ P

Merge(P, Compose(Match(P, N), Map_{N_O}), O) $\rightarrow$ R

Then in model P the Bio property as a result of merging r and t will have the Bio value a since it is specified in m_1. In the second Merge, model O will be the preferred model, and the value of the Bio property of the resulting element will be c.

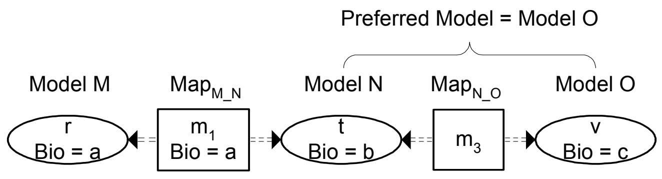


Figure 5.14: A series of mappings and models

However, if the sequence of operators is:

Merge(N, Map$_{N\_O}$, O) → Q

Merge(M, Compose(Map$_{M\_N}$, Match(N, Q)), Q) → S

Then the Bio property of the element that corresponds to t and v will have the value c since model O is the preferred model. Since the value of Bio in mapping element m$_1$ is a, the final result will have a as the value of Bio instead of c as in the first example.

Unless Merge can express a total preference between models – which is impractical – it will not be associative with respect to the final values of properties.

Hence, ignoring conflict resolution, Merge is associative. Since all of the fundamental conflict resolution in Vanilla is associative, Merge is associative for Vanilla as well (see (Buneman et al. 1992) for references on the associativity of the BDK).

5.6.3 Mapping-independent Commutativity and Associativity

We say that Merge is *mapping-independent commutative* (respectively *associative*) if it is commutative (respectively associative) even when the order of Merge operations affects the choice of mapping that is used in each Merge. For example, consider the models and mappings in Figure 5.15. In (a), Map$_{M\_N}$ is the only mapping that equates elements s and u. When Map$_{M\_N}$ is used, as in (b), elements s and u are combined. However, when Map$_{M\_N}$ is not used, as in (c), s and u remain as separate elements.

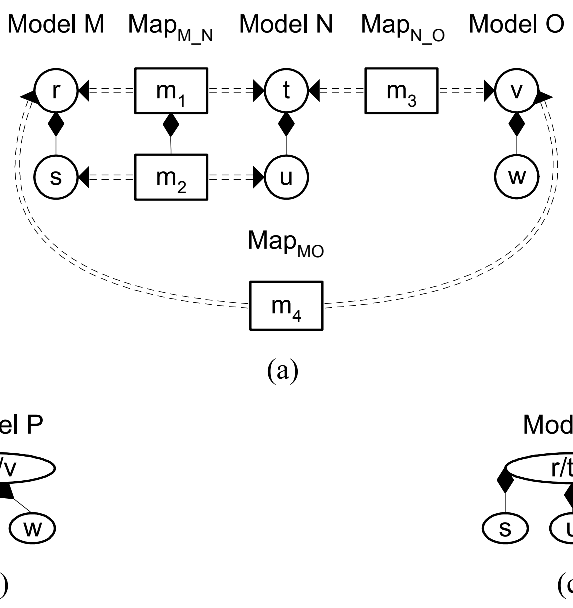


Figure 5.15: A series of merges (a) A set of models and mappings. (b) the result of merging the models using Map_{M_N} and Map_{N_O}. (c) the results of merging the models using Map_{N_O} and Map_{M_O}.

When is Merge guaranteed to be mapping-independent associative and commutative? Ignoring meta-meta-model constraint satisfaction, given a set of models, S (e.g., {M, N, O} in Figure 5.15), and two sets of mappings $Mappings_A$ (e.g., {Map_{M_N}, Map_{N_O}}) and $Mappings_B$ (e.g., {Map_{N_O}, Map_{M_O}}) over S, in order for Merge to produce isomorphic results it must be the case that:

- Elements r and v are equated to one another either directly or transitively in $Mappings_A$ if and only if they are equated to one another directly or transitively in $Mappings_B$; r can be declared equal to t and t equal to v in one set of mappings and in another set of mappings r can be declared equal to v and v equal to t.
- Elements r and v are declared to be "similar to" another element in $Mappings_A$ if and only if they are declared to be "similar to" the same element in $Mappings_B$.
- Additional elements and relationships are introduced in $Mappings_A$ if and only if corresponding elements and relationships are introduced in $Mappings_B$.

Informally, we know that these are the only requirements because:

- Merge is associative and commutative if the mappings are the same, as shown above.
- Mappings have three roles with respect to Merge; they can (1) declare elements to be equal (2) declare elements to be similar or (3) add in additional elements and relationships. We address each of these three roles below:

  Equality: Because equality is transitive, we only need to enforce that elements that are equated in one set of mappings are also equated in the other.

  Similarity: Mappings_A and Mappings_B must both declare that the same elements are similar if they are to be isomorphic to each other. However, since similarity is not transitive, if similarity is used then there is an implicit restriction on the sets of mappings; if Mappings_A declares an element in model S_1 to be similar to an element in model S_2, then Mappings_B must contain a mapping between S_1 and S_2 in order for the similarity relationship to be expressed. We do not need to consider the more complicated case when one mapping declares two elements to be similar through a mapping element, s, and then another mapping element, t, declares s to be similar to some other element because by our problem definition the set of mappings cannot map results of previous Merges.

  Additional elements and relationships: Finally, because mappings can also add elements and relationships, if Mappings_A adds an element or a relationship, then Mappings_B must add a corresponding element or relationship as well. However, as with similarity, there may be an implicit restriction on the set of mappings; if Mappings_A declares an element in model S_1 to contain an element in model S_2, then Mappings_B must contain a mapping between S_1 and S_2 in order for the Contains relationship to be expressed. Again, because the set of mappings cannot map results of previous merges, we need not consider more complicated cases.

5.7 *Alternate Merge Definitions*

Many alternate merge definitions can be implemented using our Merge operator in combination with other Model Management operators. In this section we consider three-way merge, a common merging problem that occurs in file versioning and computer supported collaborative work (Balasubramaniam et al. 1998). Given a model and two different modified versions of it, the goal is to merge the modified versions into one model.

5.7.1 Merge Only Elements Specifically Mentioned in the Mapping

Some applications want to merge only elements specifically mentioned in the mapping. One such example is Subject Oriented Programming (Ossher et al. 1996), which uses Merge to combine classes. After classes are combined, some variables should not be merged because they are private. This formulation of Merge can be implemented by:

DeepCopy(Map$_{A\_B}$) → Map$_{A\_B'}$, A′, B′

Apply(A′, a function to delete elements not mapped by Map$_{A\_B'}$)

Apply(B′, a function to delete elements not mapped by Map$_{A\_B'}$)

Merge(A′, Map$_{A\_B'}$, B′) → G

where DeepCopy is a variant of the Copy operator that copies the mapping as well as the models that it connects (Bernstein et al. 2000).

5.7.2 Three-Way Merge

Three-way merge is a common merging problem that occurs in file versioning and computer supported collaborative work (Balasubramaniam et al. 1998). Given a model and two different modified versions of it, the goal is to merge the modified versions into one model.

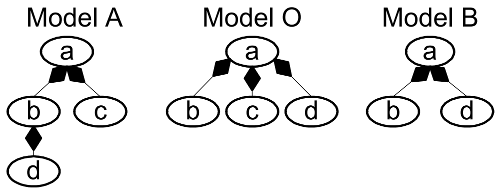


Figure 5.16: A three-way merge assuming name equality. Model O is the common ancestor of models A and B.

For example, consider Figure 5.16 where model O has been modified in two different ways to create both models A and B. Suppose there are mappings between O and A and between O and B based on element name equivalence. Notice that in A, element d has been moved to be a child of element b, and in B the element c has been deleted.

There are several variations of three-way merge which arise due to different treatments of an element that was modified in one model and deleted or modified in the other. One variation assumes that elements deleted in one model but modified in the other should be included in the merged model. More precisely it assumes that the merged model L should have the following properties:

- If an element e was added in A or B, then e is in L.
- If an element e is present and unmodified in A, B, and O, then e is in L.
- If an element e was deleted in A or B and unmodified or deleted in the other, then e is not in L.
- If an element e was deleted in A or B and modified in the other, then e is in L (because by modifying e the model designer has shown that e is still of interest).
- If an element e was modified in A or B and unmodified in the other, then the modified version of e is in L.
- If an element e was modified in both A and B, then conflict resolution is needed to determine what is in L.

This 3-way merge can be implemented as follows. We determine equality for elements in A and B based on the History property.

1. Create a mapping Map_{A_B} between A and B such that:
    a. If $a \in A$ and $b \in B$ are equal, a mapping element expressing equality between a and b is added to Map_{A_B}.
    b. If an element e exists in each of O, A, and B, and a property of e has been changed in exactly one of A or B, then Map_{A_B} has the changed property value in the mapping element corresponding to e.

2. Create model D such that if an element or relationship has been deleted in one of A or B and is unmodified in the other, it is included in D.
3. G = Merge(A, Map_{A_B}, B).
4. Map_{G_D} = Match(G, D) – based on the History property
5. Return Diff(G, D, Map_{G_D}).

This three-way merge definition does not handle a new element x created independently in both A and B. To allow this, a new mapping could be created to relate A.x and B.x.

Creating the information contained in Map_{A_B} and D can be done using a sequence of Model Management operators. Appendix E shows this in detail.

Most algorithms for three-way merge have (1) a "preferred" model that breaks ties and (2) a method for resolving conflicts such as when an element is deleted in one descendent model and modified in the other. We support the former with Merge's preferred model the latter by applying the Model Management Apply operator.

5.8 Evaluation

Our evaluation has two main goals: Section 5.8.1 shows that Merge can be applied to a real world application where it scales to large models and discovers relevant conflicts and Section 5.8.2 shows that our Merge definition subsumes previous work.

5.8.1 Applying Merge to Large Ontologies

We tested Merge on a large bioinformatics application to show that Merge scales to large models and uncovers real conflicts caused by merging such large models. The problem is to merge two models of human anatomy: the Foundational Model of Anatomy (FMA) (Rosse et al. 1998), which is designed to model anatomy in great detail, and the GALEN Common Reference Model (Rector et al. 1994), which is designed to aid clinical applications. These are very large models. As expressed in a variant of Vanilla, FMA contains 895,307 elements and 2,032,020 relationships, and GALEN contains 155,307 elements and 569,384 relationships; both of the models were larger in the Vanilla variant than in their "native" format since many of their relationships required reification. The two models have significant structural differences (e.g., some concepts expressed in FMA by three elements are expressed in GALEN by four elements), so merging the two is challenging. Note that there is no additional instance information for either model. Merge was implemented generically in approximately 7,500 lines of C# with SQL Server as a permanent store.

A database researcher familiar with FMA, GALEN, and Model Management took 13 weeks to import the models into a variant of Vanilla and create a mapping consisting of 6265 correspondences. The mapping is small relative to the model sizes since the models have different goals and thus different contents. It contains only 1-to-1 correspondences, so we were unable to test our hypothesis that having the mapping as a first class model enables more accurate merging. Hence we concentrated on three other issues: (1) few changes to Vanilla and Merge would be needed to merge the models, even though Merge was not tailored for this domain, (2) Merge would function on models this large, and (3) the merged result would not be simply read from the mapping (i.e., the conflicts that we anticipated would occur).

For the first issue, the researcher needed to add to Vanilla two relationship kinds: Contains-t(x, y), which says that x *can contain* instances of y, and Has-t(x, y), which says that x *can have* instances of y. Neither relationship kind led to new fundamental conflicts Also, the one-type restriction was not relevant to the anatomists. The only change to Merge's default behavior was to list the two new relationship kinds and ignore the one-type restriction.

Merging these models took approximately 20 hours on a Pentium III 866 with 1 GB of RAM. This is an acceptable amount of time since Merge would only be run occasionally in a relatively long project (13 weeks in our case). The merge result before fundamental conflict resolution had 1,045,411 elements and 2,590,969 relationships. 9,096 relationships were duplicates, and 1,339 had origins and destinations that had been equated.

Since the input mapping only uses 1-to-1 correspondences, we would expect most elements in the merged model to correspond to exactly two elements: one in FMA and one in GALEN. However, 2344 merged elements correspond to exactly three elements in FMA and GALEN, and 623 correspond to more than 3 elements. One merged element corresponds to 1215 elements of GALEN and FMA.

The anatomists verified that the specialization hierarchy should be acyclic, as it was in both inputs. However, before conflict resolution the merge result contained 338 cycles in the specialization hierarchy, most of length 2. One was of length 18.

The anatomists agreed that the result of the merge was useful both as a final result, assuming that the input mapping was perfect, and as a tool for determining possible flaws in the input mapping. Exploring the former is a largely manual process and is the subject of ongoing medical informatics research (Mork et al. 2004).

5.8.2 Comparison to Previous Approaches

There has been considerable work on merge in other contexts and applications. An important result of our work is that it subsumes previous literature on merge. In this section we show how Merge, assisted by other Model Management operators, can implement previous approaches to generic merging (Section 5.8.2.1), view integration (Section 5.8.2.2), and ontology merging (Section 5.8.2.3) even though it is not tailored to their meta-models.

5.8.2.1 Generic Merging Algorithms

BDK provides the basis for our work: their algorithm creates the duplicate free union of two models based on name equality of the models' elements. Their meta-meta-model contains elements with a name property and two relationship kinds, Is-A and Has-a, where Has-a must obey the one-type restriction.

Essentially Merge encompasses all of the BDK work by taking the duplicate free union of two models and then applying the one-type conflict resolution. Their work considers no other meta-meta-model conflicts, and no other resolutions when their solution to the one-type conflict is inappropriate. In addition, BDK cannot resolve representation conflicts because it lacks the explicit mapping that allows it to do so. Further details of how Merge corresponds to the BDK algorithm can be found in Appendix C.

Rondo (Melnik et al. 2003) is a Model Management system prototype that includes an alternate Merge definition based entirely on equality mappings. Two elements can be declared to be equal, and each 1-1 mapping relationship can specify a preference for one element over another. Like our Merge and BDK's, Rondo essentially creates the duplicate-free union of the elements and relationships involved. Some conflicts require removing elements or relationships from the merged model (e.g., if a SQL column is in two tables in a merge result, it must be deleted from one of them). Just as our Merge resolves such meta-model conflicts later, Rondo does such resolutions in a separate operator.

Our Merge is richer than Rondo's in several respects:

1. It can resolve representation conflicts more precisely, since the input mapping structure can relate elements in some fashion other than equivalence.
2. It can resolve conflicts that require the creation of additional elements and relationships rather than pushing the work to a subsequent manual step.
3. By specifying that a choice is first taken from the mapping, then the preferred model, and then any model, it allows for some preferences to be made once per Merge in addition to those made at each mapping element

5.8.2.2 View Integration

View integration is the problem of combining multiple user views into a unified schema (Batini et al. 1986). This problem has been studied in many contexts (Beeri et al. 1999; Bergamaschi et al. 1999; Biskup et al. 1986; Calvanese et al. 1998; Larson et al. 1989; Shu et al. 1975; Song et al. 1996). View integration algorithms (1) ensure the merged model contains all of the objects in the two original models, (2) reconcile representation conflicts in the views (e.g., if a table in one view is matched with a column in another), and (3) require user input to guide the merge. Batini, Lenzerini, and Navathe (Batini et al. 1986) also survey algorithms for creating mediated schemas for data integration, which require the same processes as those for view integration.

Spaccapietra and Parent have a well known algorithm (Spaccapietra et al. 1994) that consists of a set of rules and a prescribed order in which to apply them. Their meta-meta-model, ERC+, has three different object types: attributes, entities, and relations. An entity is an object that is of interest on its own. An attribute describes data that is only of interest while the object it characterizes exists. A relation describes how objects in the model interact. ERC+ has three kinds of relationships: Is-a, Has-a, and May-be-a, which means that an object may be of that type.

Vanilla can encode ERC+ by representing attributes, entities and relations as elements. ERC+ Is-a relationships are encoded as Vanilla Is-a relationships. ERC+ Has-a relationships are encoded as Vanilla Contains relationships (the semantics are the same). To encode in Vanilla the May-be-a relationships originating at an element e, we create a new type t such that Type-of(e, t) and for all f such that e May-be-a f, Is-a(f, t).

The Spaccapietra and Parent algorithm for merging models can be implemented using Model Management by encoding their conflict resolution rules either directly into Merge or in mappings.

Below, we summarize each of their rules and how it is covered by GMRs to merge two ERC+ diagrams A and B to create a new diagram, G. Again we use $\chi(e, e')$ to say that $e \in A \cup B$ corresponds to an element $e' \in G$.

1. Objects integration – If $a \in A$, $b \in B$, $a = b$, and both a and b are not attributes, then add one object g to G such that $\chi(a, g)$ and $\chi(b, g)$. Also, if a and b are of differing types, then g should be an entity. This corresponds to GMR 1 (Element preservation) plus an application of the EnforceConstraints operator to coerce the type of objects of uncertain type into entities.

2. Links integration – If there exist relationships R(p, c) and R(p′, c′), where $p, c \in A$, $p', c' \in B$, $p = p'$, $c = c'$, $\chi(p, g)$, $\chi(p', g)$, $\chi(c, t)$, and $\chi(c', t)$ (i.e., two parent-child pairs are mapped to one another), where neither g nor t are attributes, then R(g, t) is added to G. This is covered by GMR 3 (Relationship preservation).

3. Paths integration rule - Exclude implied relationships from the merged model. This is covered by GMR 3 (Relationship preservation) and Merge algorithm step 4d (Relationships: removing implied relationships). If the user indicates other (non-implied) redundant relationships, they must be either removed outside Merge to avoid violating GMR 3 (Relationship preservation) or expressed by an element representing an integrity constraint in the mapping and hence in the merge result.

4. Integration of attributes of corresponding objects – If there exist relationships R(p, c) and R(p′, c′) where $p, c \in A$, $p', c' \in B$, $p = p'$, $c = c'$, $\chi(p, g)$, $\chi(p', g)$ (i.e., two parent-

child pairs are mapped to one another), and c and c' are attributes, then add an attribute t to G such that $\chi(c, t)$, $\chi(c', t)$ and $R(g, t)$. This is covered by GMRs 2 and 3 (Equality and Relationship preservation).

5. Attributes with path integration – if for some attributes $c \in A$ and $c' \in B$, $c = c'$, there is no relationship R such that $R(p, c)$ and $R(p', c')$ where $p = p'$ (i.e., c and c' have different parents), add an element g to G such that $\chi(c, g)$, $\chi(c', g)$, and add all relationships necessary to attach g to the merged model. If one of the relationship paths is implied and the other is not, add only the non-implied path. This is covered by GMRs 1 and 3 (Element and Relationship preservation).
6. Add objects and links without correspondent – All objects and relationships that do not correspond to anything else are added without a correspondent. This is covered by GMR 1 (Element preservation) and 3 (Relationship preservation).

5.8.2.3 Ontology Merging

The merging of ontologies is another model merging scenario. An ontology is a domain theory that specifies a domain-specific vocabulary of objects and a set of relationships that hold among the items in the vocabulary (Fikes 1996). In general, an ontology can be viewed as a graph of hierarchical objects that have specific attributes and constraints on those attributes and objects. A frame-based ontology specifies a domain-specific vocabulary of objects and a set of relationships among them; the objects may have properties and relationships with other objects. The two relationships are Has-a and Is-a. Ontologies include constraints (called *facets*), but they were ignored by all algorithms that we studied. We describe here PROMPT (Noy et al. 2000), a.k.a. SMART (Noy et al. 1999), which combines ontology matching and merging.

PROMPT focuses on driving the match, since once the match has been found, their merge is straightforward. As in Merge, their merging and matching begin by including all objects and relationships from both models. As the match proceeds, objects that are matched to one another are collapsed into a single object. Then PROMPT suggests that objects, properties, and relationships that are related to the merged objects may match (e.g., if two objects each with a “color” property have been merged, it suggests matching those “color” properties).

Like many merge algorithms, PROMPT includes the notion of a preferred model. It also has two modes, override and merge. In override mode PROMPT chooses the option from the preferred model, while in merge mode the user is prompted for input.

PROMPT tracks its state with two lists: *Conflicts* and *ToDo*. *Conflicts* lists the current model's conflicts with the meta-model. *ToDo* keeps track of suggested matches. A more complete description of the PROMPT algorithm is shown in Appendix F.

Our algorithm allows us to provide as much merging support as PROMPT. In the merge of two models, A and B, to create a new model G, PROMPT has the following merge functionality, which we relate to our GMRs. We consider PROMPT's match functionality to be outside Merge's scope.

1. Each set of objects $O \in A \cup B$ whose objects have been matched to each other correspond to one object in G. This is covered by GMR 2 (Equality preservation).
2. Each object $o \in A \cup B$ that has not been matched to some other object corresponds to its own object in G. This is covered by GMR 2 (Equality preservation).
3. An object $g \in G$ consists of all of the properties of the objects in A or B that correspond to it. This is covered by GMR 7 (Property preservation).
4. If a conflict exists on some property's name or value, it is resolved either (1) by the user, corresponding to the user input in Merge's mapping or (2) by choosing from the "preferred" model. This is covered by GMR 8 (Value preference).

Hence, given the input mapping, our algorithm provides a superset of PROMPT's merge functionality.

A similar tool is provided by the Chimæra system (McGuinness et al. 2000), part the Ontolingua Server (Farquhar et al. 1996) from Stanford's Knowledge Systems Laboratory. However, their tool concentrates almost entirely on Match rather than the problem of Merge. Specifically, their goal was to build a tool that "focuses the attention of the editor on particular portions of the ontology that are semantically interconnected and in need of repair or further merging." After discovering parts of an ontology (model) that need further merging, their algorithm operates much like the first step of ours; the two objects that have been equated are now combined into one object and their properties and relationships with other objects must be combined.

FCA (Stumme et al. 2001) from Stumme and Maedche at the Institute AIFB University of Karlsruhe merges ontologies based on a lattice approach; they perform a match (in the terms of Model Management) or an alignment (in the terms of Noy and Musen's description of ontology operations) automatically. They examine a slightly easier problem because in addition to not considering facets (as is the case with SMART and Chimæra), they also do not consider slots. The lattice describes both the structure of the merged document and which elements in the ontology match according to the classes' semantic content. The created lattice may contain both nodes that are labeled with more than one class (indicating that merging may be required) and nodes with no corresponding class in the original ontology (suggesting that the user may want to insert a new class). The lattices are found automatically (Stumme et al. 2000), but the merging is largely manual. A similar lattice approach is taken in comparing user viewpoints in building a system in (Sabetzadeh et al. 2003).

5.8.2.4 Object-Oriented Programming Languages

There are two aspects of programming languages that can benefit from a merge operator: multiple inheritance and paradigms where classes are combined to provide more information about a topic, such as subject-oriented programming. In multiple inheritance, when a class C inherits from more than one class, the resulting members of C are essentially the union of the members of its parents. A conflict occurs if C inherits from classes A and B and both A and B have a member x with different definitions. In this case, the classes are said to be incompatible and an error is returned (Bracha et al. 1992). This corresponds in Model Management to the user being unable to find an acceptable mapping.

Subject-oriented programming (Harrison et al. 1993; Ossher et al. 1992; Ossher et al. 1996) is a programming paradigm that focuses on *subjects* rather than *objects*. A subject is a description of a number of objects and operations from one point of view. To determine how the subjects interact, their components are merged. A similar notion is aspect-oriented programming (Kiczales et al. 1997).

The key to subject-oriented programming is that different users need different information and functionality for the same subject. For example, a car rental agency requires very different information than a department of motor vehicles; a car rental agency needs to know when the car is rented, and a department of motor vehicles needs to know who licenses the car. Neither of these is inherent in the notion of a car. Thus Ossher and Harrison introduce the notion of *subjects*. A subject is “a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool” (Harrison et al. 1993).

Each subject is composed of a number of the following types of objects:

- **Classes:** The classes that are defined or used by the subject. Classes contain both member variables and member functions. Some member variables may be private to a subject, that is they should not be merged if the subject is merged.
- **Operations:** The signatures (i.e., function name, parameters, and return type) of the functions used by the subject.
- **Mapping:** A list of how to map the (operation, class) pairs to the functions that need to be executed when a subject’s operations are called; more than one function may need to be computed in order to provide one operation for a subject.

Figure 5.17 presents a series of subjects modeling cars and drivers and the composition of those subjects from (Ossher et al. 1996). In the combined subject, CarRenting, the class of Renter has been augmented by the variable “license” since Renter in the subject Renting matched the class Driver in the subject DMV. Also, the mapping for the operation (Check, Renter), which initially only called Renter.Check() now calls both Renter.Check() and Driver.GoodDriver(); information on whether a renter is a good prospect can be gained from both the information stored by the rental agency and in the information stored by the department of motor vehicles.

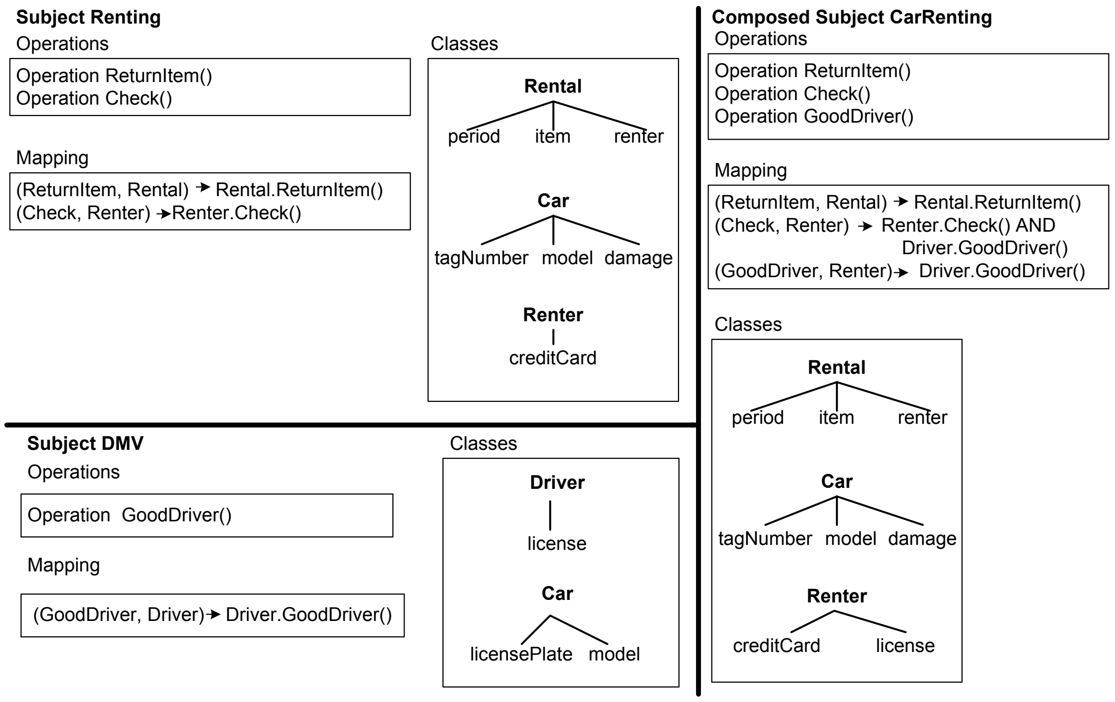


Figure 5.17: Subjects being composed (Ossher et al. 1996). Only member variables are shown in the class diagrams. The Renter class in the Renting subject is merged with the Driver class in the DMV subject to form the Renter class in the composed subject CarRenting. The return type of the operations is not shown.

Subjects are specified in a normal object-oriented programming language, such as C++, and compiled to binaries. A *compositor* composes the subjects into the executables; thus the compositor is most similar to merging in Model Management. Like all other forms of Merge that we have explored, the compositor requires input to tell whether the objects are related to one another.

Merging subjects requires merging classes, operations and mappings. We show below how each can be encoded in Vanilla and how Model Management operators can provide the same functionality as the compositor.

Classes, their member variables and member functions can all be encoded in Vanilla as elements. Vanilla encodes that a class has certain member variables or functions by saying that a Contains relationship exists from the class element to the member variable/function. For example, we would model the Rental class from the Renting subject shown in Figure 5.17 in

Vanilla as shown in Figure 5.18. Since some variables are private to a subject (i.e., they should not be merged if the subject is merged), we use the Merge variant in Section 5.7.1 (merge only elements specifically mentioned in the mapping) to perform the merge.

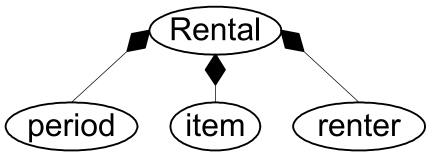


Figure 5.18: Modeling in Vanilla the Rental class from the Renter subject in Figure 5.17

Merging operations and mappings is very similar to merging classes, but resolving conflicts is different. Each operation is encoded in Vanilla as an element that has sub-elements that specify (1) the parameters, and (2) the return type. Each mapping is encoded as an element that has a sub-element for (1) the subject – class pairs to be executed when an operation is called, and (2) the class and operation to which it belongs. Each of these sub-elements may be broken into other sub-elements if necessary. In most merge algorithms, the conflicting information results in taking the values of one of the elements and ignoring the other. Merging the mappings and operations in subject-oriented programming requires the user to specify the appropriate value, but it is quite often the combination of some, if not all, of the values of the properties. Ossher and Harrison provide a number of different resolutions for this. This decision must be specified by the match (since it requires user intervention to discover the appropriate resolution) and hence would be encoded in the input mapping to Merge.

5.8.2.5 Computer Supported Collaborative Work

Both computer supported collaborative work (CSCW) (Berger et al. 1998; Berlage et al. 1993; Munson et al. 1994) and file merging (Balasubramaniam et al. 1998) generally involve three-way merge as described in Section 5.7.2. In these contexts there is a common ancestor and the two later models must be merged together based on the relationship between them and their common ancestor. With the added information from the common ancestor, the initial matching is much simpler, but the merging can be much more difficult.

The work in this area that is the most flexible and automatic is that by Munson and Dewan (Munson et al. 1994). They propose a system that, depending on the parameters that are specified, can be either manual, semi-automatic, or fully automatic. Users who choose a more automatic system are likely to receive a merged result that does not correspond to exactly what they want, but they will not have to manually guide the system. The model that they look at, while applicable to files, considers the general case of objects by encapsulating the differences in the object types. It is their automatic algorithm from which we take the correctness criteria for three-way merge in Section 5.7.2.

5.9 Conclusions

In this chapter we defined the Merge operator for model merging, both generically and for a specific meta-meta-model, Vanilla. We defined and classified the conflicts that arise in combining two models and described when conflicts from different classes must be resolved. For conflicts that must be resolved in Merge, we gave resolution strategies, both for Vanilla and in general. We evaluated Merge by showing how Merge in Vanilla can be used to subsume some previous merging algorithms and by testing Merge on two large real-world ontologies.

In the next chapter, Chapter 6, we show how the Merge result, when applied to models and mappings that are templates for instances, has an appropriate interpretation on instances by using Merge and other Model Management operators to implement the mediated schema creation for conjunctive mappings as defined in Chapter 4. This will demonstrate the usefulness of Merge in specific applications such as data integration.

Chapter 6
Using Merge to Create a Mediated Schema

6.1 Using Merge with Conjunctive Mappings

In Chapter 5 we described a generic merge algorithm, Merge (Definition 5.1), that could be used to merge schemas in a number of different circumstances, such as data integration, data warehousing, ontology merging, view integration, and so on. We showed that Merge could be used with other Model Management operators to subsume other algorithms made specifically for those domains. We now investigate how to encode conjunctive mediated schema creation (Definition 4.18) with the generic Merge and other Model Management operators described in Chapter 5. Showing how Merge can be used with such semantically rich mappings demonstrates how Model Management can help build the entire application, rather than just the schema.

In order to use Merge (Definition 5.1) to perform conjunctive mediated schema creation (Definition 4.18) the inputs to the conjunctive mediated schema creation, E, F, and Map_{E_F}, must be encoded into Vanilla (the representation that Merge uses), and then the results must be exported back into the relational model. In this chapter we discuss the algorithms necessary to complete that procedure. The remainder of this section shows how to encode relational schemas and conjunctive mappings in the Vanilla meta-meta-model described in Section 5.4.1.

Section 6.2 describes the algorithms MergeConjunctiveMediatedSchemaCreation (Definition 6.10) and $CreateMap_{G\_EF}$ (Definition 6.12) which produce a well-formed mediated schema (Definition 4.16) and well-formed mediated schema mapping (Definition 4.17) respectively using the Merge described in Definition 5.1. Section 6.3 proves that MergeConjunctiveMediatedSchemaCreation (Definition 6.10) and $CreateMap_{G\_EF}$ (Definition 6.12) create the output required in conjunctive mediated schema creation (Definition 4.18). In Section 6.4 we discuss using richer mappings than conjunctive mappings. Section 6.5 considers how to use Merge to perform some of the alternatives mentioned in both Section 4.4 and in Section 6.1.2. Section 6.6 concludes. To make the difference between relational and Vanilla objects immediately obvious, in the remainder of this thesis, we continue the usage in other chapters and use Arial font to denote objects in Vanilla and *Batang* font to represent relational objects.

6.1.1 Encoding Relational Schemas

The definition of the Vanilla meta-meta-model in Section 5.4.1 is by intent very general. In order to encode the conjunctive mediated schema creation (Definition 4.18), we provide a representation for relational models encoded in Vanilla.

A relational model encoded in Vanilla consists of a single root element which contains an element for each relation. Each relation contains an element representing each of its attributes. We encode a relational schema in Vanilla as shown in Figure 6.1.

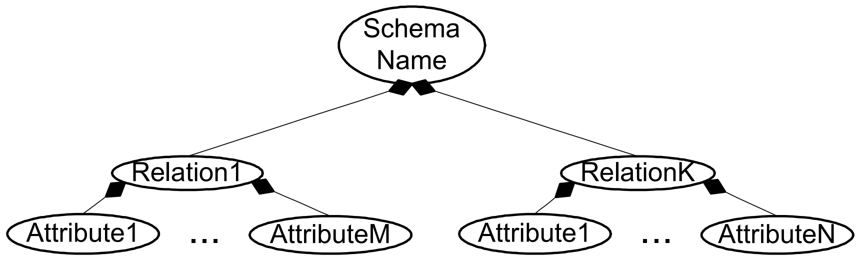


Figure 6.1: How to encode relevant relational schema information in Vanilla

We define a function μ: *R*→ M to describe the correspondence between a relational schema *R* and a Vanilla model M. In the Vanilla notation described in Section 5.4.1, a model M is a set of elements and relationships between elements, and the root of M is denoted Root(M). Vanilla consists of five different kinds of relationships in addition to the two kinds of mapping relationships. Encoding a relational schema uses only one kind of relationship: Contains. Contains is denoted by C(x, y), which means *container* x Contains *containee* y. Figure 5.6(b) says that Table Contains Column. Each element in Vanilla consists of semi-structured properties. In encoding a relational schema we only use the Name property where the name of an element e is denoted by Name(e).We define how to encode a relational schema in Vanilla in Definition 6.1.

Definition 6.1: (Vanilla encoding of a Relational Schema).

Relational schema *R* is encoded in a Vanilla model M as follows:

1. $\mu(R)$ = Root(M)
2. $\forall$ relations $r \in R$, $\exists$! element e $\in$ M s. t. $\mu(r)$= e and C(M, e) $\in$ M[14]
3. $\forall$ attributes $a \in r$, $\exists$! element a $\in$ M s.t.
   a. $\mu(a)$ = a
   b. C(r, a) $\in$ M where $\mu(r)$= r
4. Order(a) = *Order*(a)
5. If $\mu(x)$ = y, then name(y) = $n_R(x)$
6. No other elements or relationships exist in M □

μ is one-to-one and total with respect to relational definitions. That is, for each relation or attribute $ar \in R$ there exists exactly one element e$\in$ M such that $\mu(ar)$ = e.

Definition 6.2: (Relational encoded Vanilla model). There are two restrictions that a Vanilla model must obey to satisfy the relational meta-model (i.e., to be a representation of a relational schema): (1) the model must have three layers of elements: the schema name is the root, which Contains elements corresponding to relations, and each element corresponding to an attribute is contained by exactly one element corresponding to a relation and (2) the model must have no other elements or relationships other than Has-a relationships. □

The Has-a relationships in Definition 6.2 correspond to constraints (e.g., foreign key constraints); we do not explore these more fully in this chapter. We use an additional construct not found in the description of Vanilla in Section 5.4.1: if C(x, y), then Order(y) = i defines that y is the i'th child of x. In the relational meta-model representation in Vanilla, each element is contained by at most one other element (e.g., each attribute is contained by a relation, and each

[14] "$\exists$! e" means "there exists a unique e."

relation is contained by a database schema). Thus, in this case, since each element y is guaranteed to be contained by at most one x, we do not need to represent x in the Order notation. While database relations are considered to be unordered, we define an order on the relations to ensure that the ordering of variables in Map_{G_EF} is consistent.

Definition 6.3: (ImportRelationalSchema). ImportRelationalSchema(R, M) is a Model Management operator that takes a relational schema R and creates a Vanilla model M and function μ that satisfy Definition 6.1. □

6.1.2 Encoding Conjunctive Mappings

A conjunctive mapping is a set of Datalog formulas, as explained in Section 4.3.1. We do, however, place an additional restriction on the input mapping:

Remark 6.1: Rather than being a full conjunctive mapping we only consider those mappings where an IDB name appears in a mapping statement at most twice in Map_{E_F}, once in a mapping statement over E and once in a mapping statement over F. Although this is more restrictive than a full conjunctive mapping, it is as expressive a mapping as we can simulate using Merge and still easily create the view definitions needed to translate queries over the mediated schema into queries over the source schemas. In Section 6.4 we show why we require this restriction. □

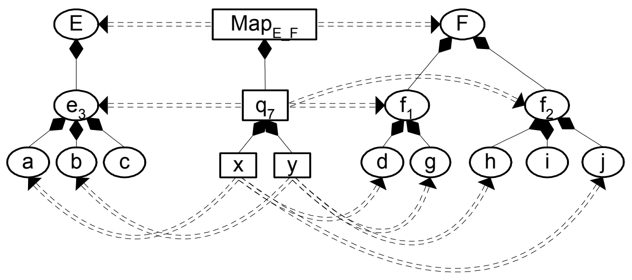


Figure 6.2: A Vanilla representation of a conjunctive mapping; all mapping relationships shown are mapping equality relationships

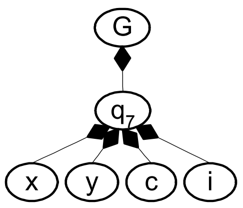


Figure 6.3: Result of Merge of Figure 6.2

A Vanilla mapping is as explained in Chapter 5. The goal of this chapter is to encode a conjunctive mapping in a Vanilla mapping. In Section 6.1.2.1 we discuss how to encode Map_{E_F} if the definition of G, Map_{G_E}, and Map_{G_F} are as in Section 4.3.3. To make this encoding easier to follow, Definition 6.4 provides notation to describe a variable being mapped to an attribute:

Definition 6.4: (VM – Variable Mapping). For a given schema relation $r_i(a_{i,1}, \ldots, a_{i,n})$, a variable $x_{i,k}$ *maps* attribute $a_{i,k}$, denoted $VM(x_{i,k}, a_{i,k})$, if $x_{i,k}$ appears in the k^{th} position of r_i. □

For example, if there exists a relation $r_1(a,b,c)$ and query $q(x) :- r_1(x,y,z)$, then $VM(x,a)$, $VM(b,y)$, and $VM(z,c)$.

6.1.2.1 Encoding Conjunctive Mappings in Generic Merge

To prepare for both the definition of well-formed relational mediated schemas and mediated schema mappings in Section 4.3.3 and the alternatives in Section 4.4 we assume that retention of an attribute or relation has been ascertained by some function $Keep$:

Definition 6.5: ($Keep$). For all $x \in E \cup F$, $Keep(x) = true$ if the concept x is retained separately in G, otherwise $Keep(x) = false$. For the base case described in Section 4.3.3, $Keep(r) = false$ only for any relation r that is mapping-included (Definition 4.14) and its attributes, $attributes_r$. □

Other cases of $Keep$ are in Definition 6.6 which defines how to encode a conjunctive mapping in Vanilla.

While we define the encoding of conjunctive mappings such that most of the alternatives in Section 4.4 can be encoded, we only consider the case of conjunctive mappings where each IDB name is defined at most once over *E* and once over *F*. While we define this more formally in Definition 6.6, we describe here a brief intuition of our encoding: we encode each IDB name *idb* as an element s in Map_{E_F}. Because each IDB can be defined only once over *E* and once over *F*, given an IDB name and schema it is over (*E* or *F*), we can tell to which mapping statement a mapping relationship corresponds. We create similarity mapping relationships from s to elements representing relations in *E* and *F* that we wish to include separately in *G*. We create mapping equality relationships from s to elements representing relations in *E* and *F* that we do not want to include separately in *G*. Each of the variables that are either joined-on or distinguished in defining *idb* are represented by elements d that are contained in s. We must represent the joined-on variables in Map_{E_F} to ensure that they are represented by the same attribute in *G* and similarly we represent the distinguished variables in Map_{E_F} to ensure that attributes that are represented by the same position in an IDB are represented by the same attribute in *G*. Again if we want to keep the relation separately in *G*, d is the origin of a mapping similarity relationship and if we do not, d is the origin of a mapping equality relationship. Because mapping equality relationships that are incident to relations will pull in the attributes of relations that are not mapped, in order to show that the mediated schema is isomorphic to a well-formed mediated schema, we do not have to include attributes that are not joined-on. Because relations created by mapping similarity relationships will not pull in attributes of the relations in *E* and *F* to which they correspond, we create a sub-element for all variables in a mapping statement. We now formally describe this process by extending μ so that it also encodes a mapping of the formulas of the semantic mapping into a Vanilla mapping.

Definition 6.6: (Vanilla Encoding of a Conjunctive Mapping). In Definition 6.1 μ is defined over *E* and *F*: ($\mu: E \cup F \rightarrow \mathsf{E} \cup \mathsf{F}$). We extend μ s.t. $\mu: (E \cup F \cup Map_{E\_F}) \rightarrow (\mathsf{E} \cup \mathsf{F} \cup \mathsf{Map}_{E\_F})$, as follows:

1. ***Map_{E_F}* corresponds to the root of $\mathsf{Map_{E\_F}}$**. Formally: $\mu(Map_{E\_F}) = \mathsf{Root(Map_{E\_F})}$
2. **The root of $\mathsf{Map_{E\_F}}$ is connected to the roots of E and F via equality mapping relationships**. Formally: $\mathsf{Me(Root(Map_{E\_F}), Root(E))} \in \mathsf{Map_{E\_F}}$ and $\mathsf{Me(Root(Map_{E\_F}), Root(F))} \in \mathsf{Map_{E\_F}}$ where $\mu(E) = \mathsf{Root(E)}$ and $\mu(F) = \mathsf{Root(F)}$.
3. **Each IDB and its mapping statements correspond to exactly one element in $\mathsf{Map_{E\_F}}$**. Formally: $\forall$ IDB names $idb \in \text{IDB}(Map_{E\_F})$, $\exists$ $\mathsf{m} \in \mathsf{Map_{E\_F}}$ s.t.
   a. $\forall$ $s \in Map_{E\_F}$, $\mu(s) = \mathsf{m}$ iff $\text{IDB}(s) = idb$ [15]
   b. $\mathsf{name(m)} = idb$
4. **Each mapping statement has its structure represented in $\mathsf{Map_{E\_F}}$**. Formally: $\forall$ pairs $<s, \mathsf{m}>$, $s \in Map_{E\_F}$, $\mathsf{m} \in \mathsf{Map_{E\_F}}$ s.t. $\mu(s) = \mathsf{m}$
   a. $\mathsf{C(Root(Map_{E\_F}), m)} \in \mathsf{Map_{E\_F}}$
   b. For each relation r in $body(s)$, where $\mu(r) = \mathsf{r}$, if $Keep(r) = true$, then $\mathsf{Me(m, r)}$ else $\mathsf{Ms(m, r)}$
   c. $\forall$ variables v in s s.t. $v \in distinguished(s)$ or $v \in Joined(s)$
      i. $\exists$ an element $\mathsf{v} \in \mathsf{Map_{E\_F}}$ s.t. $\mu(v) = \mathsf{v}$ and $\mathsf{C}(\mu(s), \mathsf{v}) \in \mathsf{Map_{E\_F}}$
      ii. $\forall$ attributes $a \in E \cup F$ s.t. $VM(v, a)$, $\exists$ an element $\mathsf{a} \in \mathsf{Map_{E\_F}}$ s.t.
         1. $\mu(a) = \mathsf{a}$

[15] Recall that by Remark 6.1 we restricted Map_{E_F} so that there are at most two mapping statements such that this is true, and at most one can be over E or over F.

2. If $Keep(v) = true$, then Me(v, a) else Ms(v, a)
3. $Order(v)$ = Order(v)

d. $\forall$ variables v in s s.t. not [$v \in distinguished(s)$ or $v \in Joined(s)$] and $Keep(v)$

i. $\exists$ an element v $\in$ $\text{Map}_{\text{E\_F}}$ s.t. $\mu(v)$= v and C($\mu(s)$, v) $\in$ $\text{Map}_{\text{E\_F}}$

ii. For each attribute $a \in E \cup F$ s.t. $VM(v, a)$, $\exists$ an element a $\in$ $\text{Map}_{\text{E\_F}}$ s.t.

4. $\mu(a)$= a
5. Ms(v, a)
6. $Order(v)$ = Order(v)

5. **No other elements or relationships exist in $\text{Map}_{\text{E\_F}}$** □

Example 6.1: If the schema E is:

$e_3(a,b,c)$

and F is:

$f_1(d,g)$

$f_2(h,i,j)$

And Map_{E_F} consists of the mapping statements:

$q_7(x,y) :- e_3(x,y,w)$

$q_7(x,y) :- f_1(x,y), f_2(y,z,x)$

Then $\text{Map}_{\text{E\_F}}$ is the mapping in Figure 6.2; the result of Merge is shown in Figure 6.3. Recall that each IDB name can only be defined at most twice, once over E and once over F, so there can be no additional mapping statements defining q_7. □

Note that the names of the existential variables that are not joined-on are not retained. This could be easily expanded to adhere precisely to the requirement that a well-formed mediated schema (Definition 4.16) retain the names of all variables used in the mapping, but for the sake of brevity and a cleaner mapping, we leave it out.

Definition 6.7: (ImportConjunctiveMapping). ImportConjunctiveMapping (Map_{E_F}, Map_{E_F}) is a Model Management operator that takes a conjunctive mapping Map_{E_F} and a function μ: $E \cup F \rightarrow \mathsf{E} \cup \mathsf{F}$ and creates a Vanilla mapping Map_{E_F} and a function μ: $R \cup \mathit{Map}_{E\_F} \rightarrow \mathsf{M} \cup \mathsf{Map}_{E\_F}$ that satisfy Definition 6.6. □

Remark 6.2: The Vanilla encoding of a conjunctive mapping (Definition 6.6) *only* creates Map_{E_F}; it does *not* change E or F. This can be seen by inspection of each bullet in Definition 6.6. □

6.1.3 Exporting to the Relational Model

To translate from the generic Model Management representation back to the relational model, we essentially reverse the process in Definition 6.1. We define here a function γ: $\mathsf{M} \rightarrow R$ which describes the correspondence between a Vanilla model M and a relational schema R. If $\gamma(\mathsf{x}) = y$, then $\mathsf{name}(\mathsf{x}) = n_R(y)$.We define the correct encoding in Definition 6.8.

Definition 6.8: (Decoding a Vanilla Model into a Relational Schema).
Let M be a Vanilla model that corresponds to a relational schema R as defined in Definition 6.2.

1. $\gamma(n_R(R)) = \mathsf{Root}(\mathsf{M})$
2. $\forall$ non-leaf non-root elements $\mathsf{e} \in \mathsf{M}$, $\exists$! relation $r \in R$ s.t. $\gamma(\mathsf{e}) = r$.
3. $\forall$ leaf elements $\mathsf{e} \in \mathsf{M}$, $\forall$ $\mathsf{p} \in \mathsf{M}$ where $\mathsf{C}(\mathsf{p}, \mathsf{e})$ and $\gamma(\mathsf{p}) = r$, $\exists$! attribute $a \in A_{r,R}$ s.t.
   a. $\gamma(\mathsf{e}) = a$,
   b. $\mathsf{Order}(\mathsf{e}) = \mathit{Order}(a)$

4. There are no other objects in $\mathcal{G}$. □

Definition 6.9: (ExportRelationalSchema) ExportRelationalSchema(M, $\mathcal{R}$) is a Model Management operator that creates a Vanilla model M and a function. γ: M → $\mathcal{R}$ that satisfy Definition 6.8. Has-a relationships are ignored since they correspond to constraints that we do not represent in the relational model. □

6.2 *Using Syntactic Merge with Conjunctive Mappings*

Conjunctive mediated schema creation (Definition 4.18) can be encoded in syntactic Merge as described Definition 5.1. To do this the input mappings are transformed into Vanilla mappings, Merge is performed, and the result is exported back to the relational model. Formally:

Definition 6.10: (MergeConjunctiveMediatedSchemaCreation). Given two relational schemas $\mathcal{E}$ and $\mathcal{F}$ and a conjunctive mapping $\mathcal{M}ap_{\mathcal{E}\_\mathcal{F}}$ between them, the algorithm MergeConjunctiveMediatedSchemaCreation operates as follows:

1. ImportRelationalSchema($\mathcal{E}$, E) (Definition 6.3)
2. ImportRelationalSchema($\mathcal{F}$, F) (Definition 6.3)
3. ImportConjunctiveMapping($\mathcal{M}ap_{\mathcal{E}\_\mathcal{F}}$, $\text{Map}_{\text{E\_F}}$) (Definition 6.7)
4. Vanilla Merge(E, F, $\text{Map}_{\text{E\_F}}$) → G (Definition 5.1)
5. ExportRelationalSchema(G, $\mathcal{G}$) (Definition 6.9) □

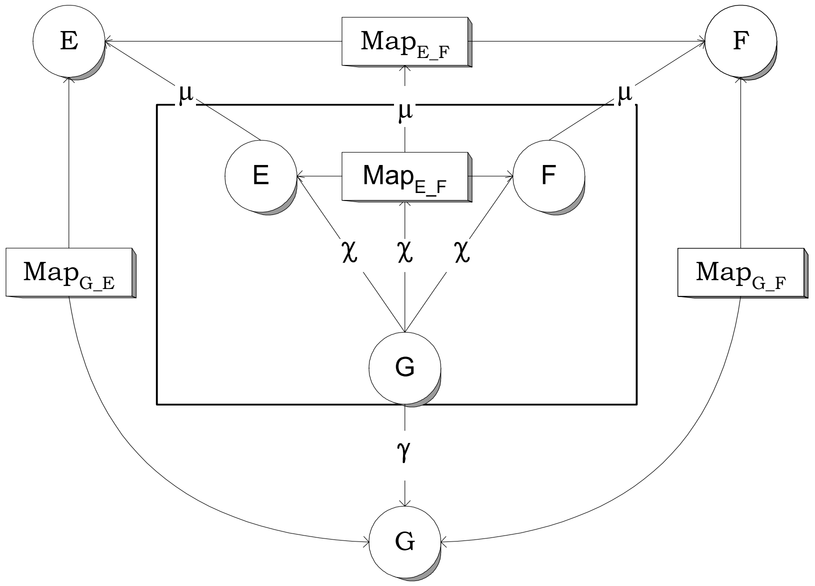

E
$\text{Map}_{\text{E\_F}}$
F
μ
μ
μ
E
$\text{Map}_{\text{E\_F}}$
F
χ
χ
χ
$\text{Map}_{\text{G\_E}}$
$\text{Map}_{\text{G\_F}}$
G
γ
G

Figure 6.4: Relationships between the various schemas and models in MergeConjunctiveMediatedSchemaCreation. Everything inside the box is a Vanilla model. Everything outside the box is a relational schema or conjunctive mapping. The edges are labeled by the correspondence relations.

The relationships between these models can be seen in Figure 6.4. Step 5 requires that G satisfies the relational meta-model. The merged model will obey the restrictions for a relational encoded Vanilla model (Definition 6.1) because the mappings are conjunctive queries, because of the way we have translated the queries into syntactic mappings, and because of the semantics of Merge. The details are shown in Theorem 6.1:

Theorem 6.1: For *G* created by MergeConjunctiveMediatedSchemaCreation (Definition 6.10), *G* obey the restrictions for a relational encoded Vanilla model (Definition 6.1).

Proof: We break the proof down into three pieces as follows, one for each bullet in Definition 6.1:

1. **The model must have three layers of elements: the schema name is the root, which Contains elements corresponding to relations, and each element corresponding to an attribute is contained by exactly**

one element corresponding to a relation. Due to the syntax of conjunctive queries, Map_{E_F} only maps relations to relations and only maps attributes to attributes. Hence Map_{E_F} only maps second-level elements to second-level elements and third-level elements to third-level elements. In addition the roots of the schemas map to one another. Thus, the merged model will have three levels.

In a conjunctive mapping, relations are mapped along with their attributes. From the definition of intersections (Definition 4.2) and projection-free components (Definition 4.13) as used in the definition of conjunctive mediated schema creation (Definition 4.18), attributes will be mapped to one another only if their parent relations are also mapped to one another. Hence from the definition of μ we know that two elements corresponding to attributes will only be mapped to each other if their parents (i.e., elements corresponding to relations) are mapped to each other. Therefore each syntactic element in the merged model that corresponds to a relational attribute will have only one parent relation. In addition since the Merge definition requires that the roots of the models map to one another, each relation will have only one root which is the only parent of all second-level elements.

2. **The model must have no other elements or relationships other than Has-a relationships:** E and F will have only contains relationships because they are Vanilla encodings of relational schemas (Definition 6.1) – this can also be seen from the definition of ImportRelationalSchema (Definition 6.3). By the definition of μ, Map_{E_F} will only have Contains relationships and mapping relationships to elements in E and F. Merge produces a model that has only the relationships in the input mapping, plus Has-a relationships (used to record relationships denoted by Mapping Similarity Relationships), minus mapping relationships from Map_{E_F} to E and F (Definition 5.1). Hence, the only relationships in G will be Contains

relationships or Has-a relationships. As described above, the Contains relationships satisfy Definition 6.1, so the only remaining relationships are Has-a relationships.

Hence G satisfies the relational data model. □

MergeConjunctiveMediatedSchemaCreation creates the mediated schema *G*, but we still lack the GLAV mappings Map_{G_E} and Map_{G_F}. We now describe how to create Map_{G_E} and Map_{G_F}.

6.2.1 Creating Map_{G_E}

Merge(E, F, Map_{E_F}) → G creates an implicit morphism from G to E in the History property. For this rest of this section we concentrate on expanding this morphism into Map_{G_E}; Map_{G_F} can be created mutatis mutandis[16]. We begin with some notation:

The relation χ: (E $\cup$ F $\cup$ Map_{E_F}) → G describes the relationship provided by the History property: $\chi(x) = y$ IFF History(y) includes x; this is the same χ function defined in the GMRs as in Section 5.2.3.

Definition 6.11: ($Mapped_e$). We define the ordered list $Mapped_e$ to be the elements in G corresponding to sub-elements of a given second-level element (i.e., a relation) e in E or F. Formally, for each $e \in X$, $X \in \{E, F\}$ s.t. C(Root(X),e), let ordered list $Mapped_e = \{,<v_i, a_j> \mid \exists\, a_j \in X$ s.t. $C(e, a_j)$ and $\chi(a_j) = v_j \in G$, and $Order(v_j) = j$ [17]$\}$. □

Figure 6.5 is a pictorial representation of the elements and relationships used in this definition.

[16] "Mutatis mutandis" means repeating the same argument only the necessary changes have been made by substituting in new terms.

[17] The GMRs for Merge imply that every input element corresponds to exactly one output element, so this is well defined.

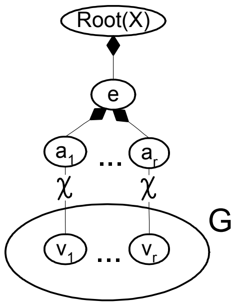


Figure 6.5: Pictorial representation of relationships used in creating Mapped$_e$

In Definition 6.12, we show how to create Map$_{E\_F}$ given *Map$_{E\_F}$*. For the algorithm below, Figure 6.6 and Figure 6.7 show the variables used in Definition 6.12 in steps (2) and (3) respectively. Steps (1) – (3) are illustrated in Example 6.2 - Example 6.4.

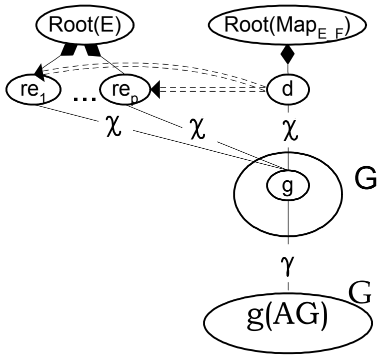


Figure 6.6: Variables used in creating mapping views for relations created as a result of mapping equality relationships in Step (2) of Definition 6.12.

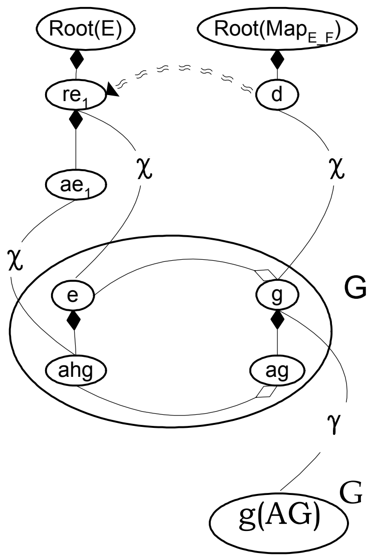


Figure 6.7: Variables used in creating mapping views for relations created as a result of mapping similarity relationships in Step (3) of Definition 6.12.

Definition 6.12: (*CreateMap$_{G\_EF}$*). We show how to create *Map$_{G\_E}$* in the steps below. *Map$_{G\_F}$* can be created by replacing each occurrence of the model E with the model F and each occurrence of *Map$_{G\_E}$* with *Map$_{G\_F}$*. *Map$_{G\_EF}$* is the union of *Map$_{G\_E}$* with *Map$_{G\_F}$*. Throughout this definition we will scope variables at each step. However, to make following the definitions easier we provide here a list of the commonly used variables and the reasoning for their names:

- *g* = the relation being created
- d = elements corresponding to relations created by *Map$_{E\_F}$*, named for the helper schema in Definition 4.10.
- *re* = relations in *E*
- hg = elements corresponding to relations in *G* that are the destination of Has-a relations
- ae = elements corresponding to attributes in *E*
- ag = elements corresponding to attributes in *G*

- ahg = elements corresponding to attributes in *G* that are the destination of Has-a relationships in G.

Step (1) **Create the views in Map_{G_E} for relations in *G* not represented in Map_{E_F}:** For each relation $g \in G$ with attributes *AG* where $\gamma(g) = g$[18] if $\exists$ no d $\in$ Map_{E_F} s.t. $\chi(d) = g$ where $g \in G$ and $\exists$ re $\in$ E s.t. $\chi(er) = g$ then do the following:

Let *q* be a fresh IDB name

Add to GV_G: $q(AG) :- g(AG)$

Add to LV_G: $q(AG) :- g(AG)$

Step (2) **Create the mapping views for relations in *G* created as a result of mapping equality relationships in Map_{E_F}.** Formally: $\forall$ relations $g \in G$ with attributes *AG* where $\gamma(g) = g$ if $\exists$ d $\in$ Map_{E_F} s.t. $\chi(d) = g$ where $g \in G$, then do the following.

Let RE be the ordered list of elements

$[re_1, \ldots, re_p \mid re_j \in E, Me(d, re_j), \chi(re_j) = g$ and $\forall i\ (1 \leq i < j)$ $Order(re_i) < Order(re_j)]$

Let $Mapped_{RE}$ = the concatenation of $Mapped_{re1}$ through $Mapped_{rep}$, where only the first occurrence of each element is included if there are duplicates.

Let *q* be a fresh IDB name

Add to LV_G: $q(Mapped_{RE}) :- g(AG)$

Add to GV_G: $q(Mapped_{RE}) :- re_1(Mapped_{re1}), \ldots, re_p(Mapped_{rep})$

Step (3) **Create the mapping views for relations in *G* created as a result of mapping similarity relationships in Map_{E_F}.** Formally: $\forall$ relations $g \in G$ with attributes *AG* where $\gamma(g) = g$, H(g, eg), $\chi(re) =$

[18] The definition of ExportRelationalSchema implies there must exist such a *g*.

```
eg, re $\in$ E (i.e., re is the destination of a similarity mapping relationship), and $\chi(d) = g$ where g $\in$ G and d $\in$ $Map_{E\_F}$.
Let RE be the ordered list of elements in E s.t. the corresponding
    element in G is the destination of a Has-a relationship from g.
    Formally: let RE be the ordered list of elements
    $[re_1, \ldots, re_p \mid re_j \in E, Ms(d,re_i), \text{ and } \chi(d) = g \ \forall\ i, 1 \leq i < j \rightarrow$
        $Order(re_i) < Order(re_j)]$
Let $\psi_g$ describe the mapping of the attributes. Formally: Let $\psi_g$ =
    $\{<ag, ae> \mid ag \in G, C(g, ag), H(ag, ahg), re \in E, C(re, ae),$
    $\chi(ae, ahg)\}$.
Let $AM_g(ae)$[19] = $\{ ae \mid re \in RE, C(re, ae)$, and if $\psi_g (ag, ae)$ then
    $AM_g(ae) = name(ag)$, else $AM_g(ae) = name(ae)\}$
Let ordered list VarsMapped = $[ag_i \mid \psi_g(ag_i, ae_i)$ and $\forall$ j, $1 \leq j < i \rightarrow$
    $Order(ae_j) < Order(ae_i)]$.
$\forall$ re $\in$ RE
        Let ordered list $AE_{re} = [ae_i \mid C(re, ae_i)$ and $\forall$ j, $1 \leq j < i \rightarrow$
            $Order(ae_j) < Order(ae_i)]$.
        Let $m = |AE_{re}|$
        Let $name_{re} = name(re)$
        Let $AMrel_g(re) = name_{re}(AM_g(ae_1), \ldots, AM_g(ae_m))$.
Let $q$ be a fresh IDB name
    Add to $LV_G = q(VarsMapped) :- g(AG)$
    Add to $GV_G = q(VarsMapped) :- AMrel_g(re_1), \ldots, AMrel_g(re_p)$
```

Similarly for Map_{G_F}. □

We now provide an example of each step in Definition 6.12.

[19] AM is short for attribute mapping

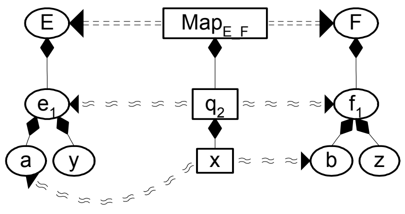


Figure 6.8: A mapping that allows the attributes not to be kept

Example 6.2: For step (1) in Definition 6.12, suppose we are merging the schemas in Figure 6.8 and creating Map_{G_E}:

$g = e_1$

$AG = [a,y]$

Add to GV_G: $qfresh_1(a,y)$:− $e_1(a,y)$

Add to LV_G: $qfresh_1(a,y)$:− $e_1(a,y)$. □

Example 6.3: For step 2 in Definition 6.12, assume Figure 6.2 and Figure 6.3 as input, and that we are creating Map_{G_F}[20]

$g = q_7$ (from Figure 6.3)

$AG = [x,y,c,i]$

d = q_7 (q_7 in Figure 6.2)

RE = [f_1, f_2]

Mapped$_{RE}$ = [x,y,i]

Mapped$_{re1}$= [x,y]

Mapped$_{re2}$ = [x,y,i]

we add to LV_G:

$qfresh_2(x,y,i)$:− $q_7(x,y,c,i)$

We add to GV_G:

$qfresh_2(x,y,i)$:− $f_1(x,y)$, $f_2(x,y,i)$. □

[20] Because we have defined the variable names to be simple to understand w.r.t. defining Map_{G_E}, we use RE to refer to elements corresponding to relations in F.

Example 6.4: For step 3 in Definition 6.12, suppose that we are merging the schemas in Figure 6.8 and creating Map_{G_E}.

$g = q_2$

$AG = [x]$

d = q_2 (in Map_{E_F})

RE = $[e_1]$

AE_{re1} = [a,y]

ψ_g(x, a)

VarsMapped = $[x]$

$AMrel_q(re_1) = e_1(x,y)$

Add to LV_G: $qfresh_4(x)$:− $q_2(x)$

Add to GV_G: $qfresh_4(x)$:− $e_1(x,y)$ □

Not all alternate semantics, such as those defined in Section 4.4 or alternate CMSCs, can be described using only Import, Merge, and Export; we describe in Section 6.5 when difficulties are encountered.

6.3 Correctness of MergeConjunctiveMediatedSchemaCreation

We now must prove that MergeConjunctiveMediatedSchemaCreation (Definition 6.10) and $CreateMap_{G\_EF}$ (Definition 6.12) correctly create G and Map_{G_EF}, as specified in the correctness criteria for a well-formed mediated schema (Definition 4.16) and well-formed mediated schema mappings (Definition 4.17) in Section 4.3.3.2.

Theorem 6.2: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) and $CreateMap_{G\_EF}$ (Definition 6.12) generate mediated schema G, and mapping Map_{G_EF} s.t. G and Map_{G_EF} are well-formed. □

To prove Theorem 6.2 we break it into two lemmas, Lemma 6.1 which shows that G is well-formed and Lemma 6.2 which shows that Map_{G_EF} is well-formed:

Lemma 6.1: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates mediated schema G s.t. G is well-formed. □

Lemma 6.2: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) and $CreateMap_{G\_EF}$ (Definition 6.12) generate mediated schema G, and mapping Map_{G_EF} s.t. Map_{G_EF} is well-formed. □

We prove Lemma 6.1 in Section 6.3.1and Lemma 6.2 in Section 6.3.2. Together, Lemma 6.1 and Lemma 6.2 prove Theorem 6.2.

6.3.1 Proof Correctness of G

In this section we prove Lemma 6.1: G created by MergeConjunctiveMediated-SchemaCreation (Definition 6.10) is a well-formed schema as required by Definition 4.16. Recall that Definition 4.16 defines a mediated schema G to be well-formed if:

1. $\forall$ relations $e \in E$ s.t. e is not mapping-included, $\exists$ $g \in G$ s.t. $n_e = n_g$ and $attributes_e = attributes_g$. Similarly for all $f \in F$.
2. $\forall$ IDB names $q \in$ IDB(Map_{E_F}), $\exists$ $g \in G$ s.t. $q = n_g$ and $attributes_g = Vars(MS_q)$
3. G contains no additional relations.

We prove Lemma 6.1 by proving Lemma 6.3, Lemma 6.4, and Lemma 6.5, each of which proves one of the bullets in Definition 4.16:

Lemma 6.3: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates mediated schema G, s.t. $\forall$ relations $e \in E$ where e is not mapping-included, $\exists$ $g \in G$ s.t. $n_e = n_g$ and $attributes_e = attributes_g$. Similarly for all relations $f \in F$. □

Lemma 6.4: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates mediated schema G, s.t. $\forall$ IDB names $q \in$ IDB(Map_{E_F}), $\exists\ g \in G$ s.t. $q = n_g$ and $attributes_g = Vars(MS_q)$. □

Lemma 6.5: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates mediated schema G, s.t. G contains only those relations needed to satisfy Lemma 6.3 and Lemma 6.4. □

We prove Lemma 6.3 in 6.3.1.1, Lemma 6.4 in Section 6.3.1.2, and Lemma 6.5 in Section 6.3.1.3. To prove the above lemmas, we require Lemma 6.6, Lemma 6.7, Lemma 6.8, Lemma 6.9, Lemma 6.10, and Lemma 6.11:

Lemma 6.6: $\forall$ relations $r \in EF$, $\exists!$ e $\in$ EF s.t. $\mu(r,e)$. $\forall$ attributes $a \in attributes_r$ $\exists!$ $a_1 \in$ EF s.t. $\mu(a,a_1)$ and C(e,a1).

Proof: This follows directly from the definition of μ. □

Lemma 6.7: $\forall$ g $\in$ G s.t. C(Root(G), g), $\exists!$ relation $r \in G$ s.t. $\gamma(g, r)$. $\forall$ elements g_1 s.t. C(g,g_1), $\exists!$ $a \in attributes_r$ s.t. $\gamma(g_1,a)$.

Proof: This follows directly from the definition of γ. □

Lemma 6.8: Each element ef $\in$ EF is the destination of at most one mapping relation in Map_{E_F}. Therefore $\forall$ elements ef_1, $ef_2 \in$ EF, $\chi(ef_1,g)$ and $\chi(ef_2,g)$ iff $\exists$ some d $\in Map_{E\_F}$ s.t. Me(d,ef_1), Me(d,ef_2); by definition of Merge (Definition 5.1 bullet 2) i.e., there is no transitivity of equality in Map_{E_F}.

Proof sketch: We have constrained conjunctive mappings to disallow mappings that would contain transitivity of equality in Map_{E_F} by disallowing relations to appear in more than one mapping statement (Section 4.3.1.1)

Proof: From the definition of Merge (Definition 5.1) elements d_1, $d_2 \in Map_{E\_F}$, $d_1 \neq d_2$, will only correspond to the same element $g \in G$ (i.e., $\chi(d_1,g)$, $\chi(d_2,g)$) if $\exists$ $e \in E$ or F s.t. $Me(e, d_3)$, $Me(e,d_4)$, and $d_3 \neq d_4$.

By the definition of Vanilla encoding of a conjunctive mapping (Definition 6.6 bullet 2), the roots of E and F are only the destination of a mapping equality relationship from the root of Map_{E_F}.

Since a relation may appear in at most one mapping statement (Section 4.3.1.1), from the definition of Vanilla encoding of a conjunctive mapping (Definition 6.6 bullet 4b), $\forall$ elements e_1 s.t. $\mu(r,e_1)$ where relation $r \in EF$, e_1 is the destination of at most one mapping relationship.

The argument for attributes is similar: A relation may appear in at most one mapping statement (Section 4.3.1.1). From the definition of conjunctive queries an attribute is only mapped if its relation is mapped. From the definition of attributes of relations, each attribute only appears once in a relation. Therefore, from the definition of Vanilla encoding of a conjunctive mapping (Definition 6.6 bullet 4.c and 4.d), $\forall$ elements e_2 s.t. $\mu(a, e_2)$ where attribute $a \in attributes_{r2}$ for some relation $r2 \in EF$, e_2 is the destination of at most one mapping relationship. Therefore $\forall$ elements ef_1, $ef_2 \in EF$ $\chi(ef_1,g)$ and $\chi(ef_2,g)$ iff $\exists$ some $d \in Map_{E\_F}$ s.t. $Me(d,ef_1)$, $Me(d,ef_2)$; i.e., there is no transitivity of equality in Map_{E_F} □

Lemma 6.9: In Merge(E, F, Map_{E_F}) $\rightarrow$ G (Definition 5.1) in MergeConjunctiveMediatedSchemaCreation (Definition 6.10), no fundamental conflict resolution is required, nor are there any implied relationships removed.[21]

Proof: By ImportRelationalSchema (Definition 6.3), E and F will both have three levels and obey the restrictions for a relational encoded Vanilla model (Definition 6.1). By Definition 6.3 E and F will only have Contains relationships. Due to the syntax of conjunctive queries, Map_{E_F} only maps relations to relations and attributes to attributes. Hence $\text{Map}_{\text{E\_F}}$ only maps second-level elements to second-level element and third-level elements to third-level elements. More formally: by ImportConjunctiveMapping (Definition 6.7) we know that $\text{Map}_{\text{E\_F}}$ consists of three levels, and if C(Root(E), e), M(s,e)[22] where $\exists$ relation $r \in EF$ s.t. $\mu(r,\text{e})$, then $\text{C(Root(Map}_{\text{E\_F}}\text{),s)}$. Similarly, if C(Root(E), e), $\text{C(e,e}_1\text{)}$, and $\text{M(d}_1\text{,e}_1\text{)}$ where $\exists$ attribute a and relation r s.t. $a \in attributes_r$ and $\mu(a,\text{e}_1)$ then $\exists$ s $\in \text{Map}_{\text{E\_F}}$ s.t. $\text{C(Root(Map}_{\text{E\_F}}\text{),s)}$, and $\text{C(s,d}_1\text{)}$. The only fundamental conflict in Vanilla that uses Contains or Has-a relationships is that the containment hierarchy must be acyclic (5.4.2.2). Because the roots of E, F, and $\text{Map}_{\text{E\_F}}$ are only mapped to each other, the second-level elements are mapped only to second-level elements, and the third-level elements are mapped only to third-level elements, there are no cycles in the containment hierarchy. So no fundamental conflict resolution is required.

Since G can only consist of Has-a or Contains relationships, the only relevant relationship implication rule (Section 5.4.1) is that containment is transitive. However, because the roots of E, F, and $\text{Map}_{\text{E\_F}}$ are only mapped to each other, the second-level elements are mapped only to second-level elements, and the third-level elements are mapped only to third-level elements,

[21] Note that this proof and the beginning of the proof of Theorem 6.1 share many of the same constructs.

[22] Recall that M denotes either a Mapping Similarity Relationship or a Mapping Equality Relationship.

there is no transitive containment in G. So no relationship implication rules are used. Thus Lemma 6.9 holds. □

Lemma 6.10: ∀ relations $g \in G$ s.t. G was created by an IDB *idb*, n_g = *idb*.

Proof: From the definition of Vanilla encoding of a conjunctive mapping (Definition 6.6) bullet 3, ∃! element e ∈ Map_{E_F} s.t. μ(*idb*,e) and name(e) = *idb*. Since e ∈ Map_{E_F} and thus for χ(e,g) the value of the Name property of g is taken from e, (Definition 5.1 bullet 3.a.i) there are no other elements in Map_{E_F} that will be equated with e (Lemma 6.8) that could override e's values, and there are no fundamental conflicts or implied constraints in the creation of G to override e's values (Lemma 6.9), from Merge (Definition 5.1) ∃! element g ∈ G s.t. name(g) = name(e). From the definition of ExportRelationalSchema (Definition 6.9) ∃! relation g_1 ∈ G s.t. n_{g1} = name(g). Hence Lemma 6.10 holds.□

Lemma 6.11: ∀ attributes $ag \in G$ s.t. G was created by a variable v in IDB *idb*, n_{ag} = name of v.

Proof: From the definition of Vanilla encoding of a conjunctive mapping (Definition 6.6) bullet 4, ∃ ! element e ∈ Map_{E_F} s.t. μ(v,e) and name(e) = v. Since e ∈ Map_{E_F}, and thus for χ(e,g) the value of the Name property of g is taken from e, (Definition 5.1 bullet 3.a.i) there are no other elements in Map_{E_F} that will be equated with e (Lemma 6.8) that could override e's values, and there are no fundamental conflicts or implied constraints in the creation of G to override e's values (Lemma 6.9), from Merge (Definition 5.1) ∃! element g ∈ G s.t. name(g) = name(e). From the definition of ExportRelationalSchema (Definition 6.9) ∃! attribute g_1 ∈ G s.t. n_{g1} = name(g). Hence Lemma 6.10 holds. □

6.3.1.1 Proof that Non-Mapping-Included Relations are in *G*

We are now ready to prove Lemma 6.3: non-mapping-included relations are represented in *G*. There are two different cases to consider (1) non-mapping-included relations that do not appear in Map_{E_F} (Lemma 6.12) and (2) non-mapping-included relations that appear in Map_{E_F} (Lemma 6.13).

Lemma 6.12: $\forall$ relations $e \in EF$ where e is not mapping-included and $\nexists$ $ms \in Map_{E\_F}$ s.t. $e \in body(ms)$, $\exists$ $g \in G$ s.t. $n_e = n_g$ and $attributes_e = attributes_g$.

Proof sketch: Since e and its attributes are not mentioned in Map_{E_F}, no elements corresponding to them will be mapped in Map_{E_F}. Since Merge preserves all elements that are not mapped, and ExportRelationalSchema will correctly create a relation for all elements corresponding to relations in G, e will appear in G. In a bit more depth: the elements corresponding to neither e nor any of e's attributes appear as the destination of any mapping relationship in Map_{E_F}. From the definition of Merge, we know that if an element e_1 appears in E or F but not in Map_{E_F}, an element identical to e_1 appears in G, and if e_1 and e_2 appear in E or F and there exists a relationship $C(e_1,e_2)$, if there are separate elements e_3 and e_4 corresponding to e_1 and e_2 respectively, then $C(e_3, e_4)$ unless fundamental conflict resolution has occurred or $C(e_3, e_4)$ is implied by other relationships. Since there is no need for fundamental conflict resolution in G and the relationship between e and its children is not implied by any other relationship (Lemma 6.9), and there is no transitivity of equality in Map_{E_F} (Lemma 6.8), e and e's attributes appear in G as children of G's root. By the definition of ExportRelationalSchema, a relation identical to e, including all of e's attributes appears in G.

Proof: Let $r \in EF$ be a relation that is not mapping-included in Map_{E_F}. By ImportRelationalSchema (Definition 6.3), $\exists!$ e $\in$ EF s.t. $\mu(r,e)$. ImportConjunctiveMapping (Definition 6.7) requires that μ satisfies the Vanilla Encoding of a Conjunctive mapping (Definition 6.6). And by Remark 6.2, Definition 6.6 does not change E or F. By bullet 5 of Definition 6.6, there are no relationships in Map_{E_F} other than the relationships required in bullets 1-4. Mapping relationships are only created in bullets 2 and 4. Bullet 2 only creates mapping relationships with destination on the schema names. Bullet 4 only creates mapping relationships with destination elements e_5 s.t. $\mu(r_5,e_5)$, and $r_5 \in body(ms)$ for some $ms \in Map_{E\_F}$ or $\mu(a_5,e_5)$ and $VM(v_5,a_5)$ for some variable $v_5 \in Map_{E\_F}$. Hence, since r does not appear in the body of any mapping statement, e does not appear as the destination of any mapping relationships.

From the definition of μ, $\forall$ attributes $a \in attributes_e$, $\exists!$ a_1 s.t $\mu(a, a_1)$ and $C(e,a_1)$. As in the previous paragraph, by definition of μ (Definition 6.6), the only relevant creation of mapping relationships occurs in Bullet 4. Bullet 4 of Definition 6.6 only creates mapping relationships with destination elements e_5 s.t. $\mu(r_5,e_5)$, and $r_5 \in body(ms)$ for some $ms \in Map_{E\_F}$ or $\mu(a_5,e_5)$ and $VM(v_5,a_5)$ for some variable $v_5 \in Map_{E\_F}$. So $\forall$ a_1 s.t. $\mu(a, a_1)$ and $C(e,a_1)$ if e does not appear as the destination of any mapping relationship, a_1 does not appear as the destination of any mapping relationships.

From the definition of Merge (Definition 5.1) and given that Lemma 6.9 tells us that there are no implied relationships or fundamental conflicts in creating G, and that Lemma 6.8 tells us there is no transitivity of equality in Map_{E_F}, we know that if an element e_1 appears in E or F and $\nexists$ element $e_6 \in Map_{E\_F}$ s.t. $Ms(e_6,e_1)$ or $Me(e_6,e_1)$, then $\exists!$ element e3 $\in$ G s.t. $\chi(e_1) = e_3$. Similarly if an element $e_2 \in$ E or F and $\nexists$ element $e_7 \in Map_{E\_F}$ s.t. $Ms(e_7,e_2)$ or $Me(e_7,e_2)$, then $\exists!$ element $e_4 \in$ G s.t. $\chi(e_2) = e_4$. As well, since Lemma 6.9 tells us that there are no implied relationships or fundamental conflicts in creating G, if $\exists$ a relationship $C(e_1,e_2) \in$ E or F and $\chi(e_1) = e_3$ and $\chi(e_2) = e_4$, then $\exists$ a relationship $C(e_3,e_4) \in$ G.

By Theorem 6.1, G satisfies the relational meta-model (Definition 6.2). Since ExportRelationalSchema (Definition 6.9) requires that $\forall$ elements e_8, e_9 s.t. $C(Root(G),e_8)$ and $C(e_8,e_9)$, then $\exists$ a relation g s.t. $\gamma(e_8,g)$ and $ga \in attributes_g$ where $\gamma(e_9,ga)$. Hence it follows that each relation in EF that does not appear in Map_{E_F} appears in G. □

Lemma 6.13: $\forall$ relations $r \in E$ where r is not mapping-included and $\exists$ $ms \in Map_{E\_F}$ s.t. $r \in body(ms)$, $\exists$ $g \in G$ s.t. $n_e = n_g$ and $attributes_e = attributes_g$. Similarly for all relations $f \in F$.

Proof sketch: From Lemma 6.6, it follows that each relation $r \in EF$ corresponds to a unique element e $\in$ EF s.t. $\forall$ attributes of r, e contains elements corresponding to those attributes. It follows from ImportConjunctiveMapping that each element corresponding to a non-mapping included relation that is represented in Map_{E_F} and its attributes is only the destination of mapping similarity relationships. From the definition of Vanilla Merge, and the fact that no fundamental conflict resolution occurs and that no relationship implication rules are applied (Lemma 6.9), and that there is no equality of transitivity in $\text{Map}_{\text{E\_F}}$ (Lemma 6.8), it follows that there exists a unique element g s.t. e corresponds to g, and g contains elements corresponding to all of the elements contained by e. By Lemma 6.7, it follows there exists a relation $g_1 \in G$ s.t. g_1 corresponds to g and g_1 contains attributes for each element contained by g. Hence each non-mapping included relation appearing in Map_{E_F} appears in G.

Proof: From Lemma 6.6 it follows that for all relations $r \in EF$, $\exists!$ e $\in$ EF s.t. $\mu(r,\text{e})$. $\forall$ attributes $a \in attributes_r$ $\exists!$ $\text{a}_1 \in$ EF s.t. $\mu(a,\text{a}_1)$ and C(e,a1).

From the definition of $Keep$ (Definition 6.5), it follows that if a relation r is non-mapping-included, $Keep(r) = true$, and $\forall$ attributes $a \in attributes_r$, $Keep(a) = true$.

From ImportConjunctiveMapping (Definition 6.7) it follows that $\forall$ elements e s.t. $\mu(ms,\text{e})$ where $ms \in Map_{E\_F}$, since $Keep(r) = true$, $\exists!$ d $\in$ $\text{Map}_{\text{E\_F}}$ s.t. Ms(d,e) where $\mu(r,\text{e})$. Similarly $\forall$ e_3 s.t. C(e, e_3), $\exists!$ $\text{d}_2 \in \text{Map}_{\text{E\_F}}$ s.t. Ms(d_2,e_3).

From the definition of Merge, and the fact that no fundamental conflict resolution occurs and that no relationship implication rules are applied (Lemma 6.9) and that there is no transitivity of equality in $\text{Map}_{\text{E\_F}}$ (Lemma 6.8), it follows that there exists a unique element g s.t. χ(e, g), and $\forall$ e_3 s.t. C(e,e_3), $\exists$ element g1 $\in$ G s.t. χ(e_3,g_1) and C(g,g_1).

By Lemma 6.7, it follows $\forall$ g $\in$ G s.t. C(Root(G), g), $\exists!$ relation $r \in G$ s.t. γ(g, r). $\forall$ elements g_1 s.t. C(g,g_1), $\exists!$ a $\in$ $attributes_r$ s.t. γ(g_1,a). Hence Lemma 6.13 holds. □

Hence Lemma 6.3 holds.

6.3.1.2 Proof that Relations Generated from Map_{E_F} are in G

To prove Lemma 6.4 (relations generated from MapE_F are in G) we split it up into two lemmas: one for relations created by mapping similarity relationships (Lemma 6.14) and one for relations created by mapping equality relationships (Lemma 6.15). These are the only two cases because of the following: $\forall$ IDBs $idb \in IDB(Map_{E\_F})$, by ImportConjunctiveMapping,(Definition 6.7), there exists a μ that satisfies Definition 6.6. By the third bullet of Definition 6.6 $\exists!$ element s $\in$ Map_{E_F} s.t. μ(idb,s). Bullet 4b of Definition 6.6 implies that s is the origin of exactly two mapping relationships, mr_1 and mr_2. mr_1 and mr_2 may be either similarity or equality mapping relationships. Hence these are the only two cases.

Lemma 6.14: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates a mediated schema G, s.t. $\forall$ IDB names $idb \in$ IDB(Map_{E_F}), $\forall$ MS_q s.t. $IDB(MS_q) = idb$, $\exists$ s $\in$ Map_{E_F} s.t. μ(idb,s) and s is the origin of a mapping similarity relationship, $\exists$ $g \in G$ s.t. $idb = n_g$ and $attributes_g = Vars(MS_q)$.

Proof sketch: Assuming that we are considering only IDB names that are represented using mapping similarity relationships, from the definition of μ we know that $\forall$ IDBs idb, idb will be represented by one element d $\in$ Map_{E_F}. From μ we also know that $\forall$ mapping statements ms s.t. $IDB(ms) = idb$, $\forall$ relations $r \in body(ms)$, $\exists$ a mapping similarity relationship from s, the element representing idb, to e, the element representing r. From the definition of μ we know that there exists no other mapping relationship with destination at e.

Similarly for all variables $v \in ms$ there exists a unique element d_1 in Map_{E_F} s.t. d_1 corresponds to v and d_1 is contained by d, and v is the destination of no mapping equality relationships. Hence by the definition of Merge and the fact that there is no fundamental conflict resolution or relationship implication in the creation of G (Lemma 6.9) and there is no transitivity of equality in Map_{E_F} (Lemma 6.8), there exists exactly one element g in G s.t. g corresponds to d and g contains one element for each element corresponding to a variable in *idb*. Hence by the definition of γ, there exists one relation g in G s.t. g has as attributes the variables of all mapping statements *ms* s.t. $IDB(ms) = idb$; Lemma 6.11 ensures that the names of the attributes of g are correct, and Lemma 6.10 ensures that $idb = n_g$.

Proof: Assuming that we are considering only IDB names that are represented using mapping similarity relationships, from the definition of the Vanilla encoding of a conjunctive mapping (Definition 6.6) we know that $\forall$ IDBs *idb*, $\exists$! element $s \in Map_{E\_F}$.s.t. $\mu(idb,s)$ (bullet 3) and $C(Root(Map_{E\_F}),s)$ (bullet 4a). From bullet 4 of Definition 6.6 we also know that $\forall$ mapping statements *ms* s.t. $IDB(ms) = idb$, $\forall$ relations $r \in body(ms)$, $\exists$! $e \in E$ and $d \in Map_{E\_F}$ s.t. $Ms(d,e)$. From the definition of μ we know that there exists no other mapping relationship with destination at e. Similarly for all variables $v \in ms$ $\exists!$ element $d_1 \in Map_{E\_F}$ s.t. $\mu(v,d_1)$, and $C(d,d_1)$, and $\nexists$ $d_2 \in Map_{E\_F}$ s.t. $Me(d_2, d_1)$. Hence by the definition of Merge (Definition 5.1) and that there is no fundamental conflict resolution or relationship implication in the creation of G (Lemma 6.9) and there is no transitivity of equality in Map_{E_F} (Lemma 6.8), $\exists!$ element $g \in G$ s.t. $\chi(d,g)$ and $\forall$ d_1 s.t. $v \in Vars(ms)$ where $IDB(ms) = idb$ and $\mu(v,d_1)$, $C(g,g_1)$. Hence by the definition of γ, there exists one relation $g \in G$ s.t. g has as attributes the variables of all mapping statements *ms* s.t. $IDB(ms) = idb$. Finally, Lemma 6.11 ensures that the names of the attributes of g are correct, and Lemma 6.10 ensures that $idb = n_g$, so Lemma 6.14 holds. □

Lemma 6.15: Given relational schemas E and F and a conjunctive mapping Map_{E_F}, MergeConjunctiveMediatedSchemaCreation (Definition 6.10) generates mediated schema G, s.t. $\forall$ IDB names $idb \in$ IDB(Map_{E_F}), $\forall$ MS_q s.t. $IDB(MS_q) = idb$, $\exists$ s $\in$ Map_{E_F} s.t. $\mu(idb, s)$ and s is the origin of a mapping equality relationship, $\exists$ $g \in G$ s.t. $q = n_g$ and $attributes_g = Vars(MS_q)$.

Proof sketch: Assuming that we are considering only IDBs that are represented using mapping equality relationships, all relations r that appear in a mapping statement for an IDB $idb \in IDB(Map_{E\_F})$ will be equated to the same element in Map_{E_F} by the definition of ImportConjunctiveMapping, and similarly for the variables in idb's definition. Similarly, each attribute $a \in attributes_r$ will be mapped iff a is joined-on in a mapping statement or a is mapped to a distinguished variable. The only time that attributes will be mapped by the same element in Map_{E_F} is if they are mapped by the same variable. From Merge and the fact that there is no fundamental conflict resolution or relationship implication in the creation of G (Lemma 6.9) and there is no transitivity of equality in Map_{E_F} (Lemma 6.8), we know that this will result in an element g $\in$ G that contains one element for each attribute not joined-on, and one element for each joined-on variable. From ExportRelationalSchema we know that there exists a relation $g_1 \in G$ that contains one attribute for each element contained in g. Finally, Lemma 6.11 ensures that the names of the attributes of g are correct, and Lemma 6.10 that $idb = n_g$.

Proof: $\forall$ IDBs $idb \in IDB(Map_{E\_F})$, $\forall$ ms s.t. $IDB(ms) = idb$, $\exists$! s $\in$ Map_{E_F} s.t. $\mu(ms, s)$ iff $IDB(ms) = idb$ by μ (Definition 6.6) bullet 3. We know from Definition 6.6 bullet 4b that $\forall$ relations $r \in body(ms)$, $\exists!$ e $\in$ E s.t. $\mu(r, e)$ and Me(s,e). From Definition 6.6 bullets 4.c.ii.2, $\forall$ attributes $a \in attributes_r$ s.t. $VM(v,a)$ and $v \in distinguished(ms)$ or $v \in join(ms)$, $\exists$ unique elements $e_1$and s_2 in Map_{E_F} s.t. $\mu(a, e_1)$, $\mu(v, s_2)$, and C(e,e_1), C(s, s_2), Me(s_2, e_1).

From the definition of conjunctive queries (Definition 2.2), an attribute is only mapped if its relation is mapped. From the definition of conjunctive mappings (Section 4.3.1.1), a relation is only mapped once. From Lemma 6.8 there is no transitivity of equality in Map_{E_F}. By the definition of Merge (Definition 5.1) and the fact that there are no fundamental conflicts or implied relationships in creating G (Lemma 6.9), $\exists\,!\ g \in G$ s.t. $\chi(s, g)$. In addition g contains exactly one element g_1 per element d_1 s.t. $C(s,d_1)$, and one element per element $e_1 \in E$ s.t. $Me(e,e_2)$, $C(e_2,e_1)$. By Lemma 6.7, there will be exactly relation $rg \in G$ s.t. $\gamma(g,rg)$, and if $C(g,g_1)$ then $\exists!\ ag$ s.t. $\gamma(g_1,ag)$ and $ag \in attributes_{rg}$. Finally, Lemma 6.11 ensures that the names of the attributes of g are correct, and Lemma 6.10 that $idb = n_g$. Hence Lemma 6.15 holds. □

Since Lemma 6.14 and Lemma 6.15 hold, Lemma 6.4 holds.

6.3.1.3 Proof that No Additional Relations Exist in *G*

From the definitions of ImportRelationalSchema and ImportConjunctiveMapping, it follows that no other elements exist in Map_{E_F} besides those elements required above. Hence, from the definition of Merge, no other relations exist in G and Lemma 6.5 holds.

Hence with the proofs of Lemma 6.3 and Lemma 6.4 above, Lemma 6.1 holds.

6.3.2 Proof of Correctness of Map_{G_E} and Map_{G_F}

We now show Lemma 6.2: LV_G and GV_G are well-formed mediated schema mappings as defined in Section 4.3.3.2. Recall that Definition 4.17 requires that a well-formed relational mediated schema mapping obeys the following lemmas, one for each of the numbered bullets in Definition 4.17:

Lemma 6.16: $\forall$ mapping statements ms with IDB name q, let $rq \in G$ be a relation with name q and attributes = $Vars(MS_q)$ and $\xi(q,rq)$. Let q_j be a fresh IDB name (i.e., an IDB name that appears in no other mapping statements in Map_{E_F} or in any other local view definitions or global view definitions in

Map_{G_EF}).

$lv_{ms} = q_j(Vars(ms)) :- rq$

$gv_{ms} = q_j(Vars(ms)) :- body(rq)$

$lv_{ms} \in LV_G$

$gv_{ms} \in GV_G$ □

Lemma 6.17: relations $g \in G$ s.t. $\xi(e,g)$ and $\xi(e_1,g)$ implies $e_1 = e$ (i.e., directly corresponding to a relation in E or F), let q_j be a fresh IDB name (i.e., q_j is an IDB name that appears in no other mapping statements in Map_{E_F} or in any other local view definitions or global view definitions in Map_{G_EF}).

$lv_g = q_j(attributes_g) :- g(attributes_g)$

$gv_g = q_j(attributes_g) :- g(attributes_g)$

$lv_g \in LV_G$

$gv_g \in GV_G$ □

Lemma 6.18: LV_G and GV_G contain no views other than those required above. □

We prove Lemma 6.16, Lemma 6.17, and Lemma 6.18 in Sections 6.3.2.1, 6.3.2.2, and 6.3.2.3 respectively.

6.3.2.1 Proof that Mapping Views for Relations Corresponding to IDBs are in Map_{G_EF}

We now prove Lemma 6.16 – mapping view for relations corresponding to IDBs are in Map_{G_EF}. There are two cases: A mapping statement is represented by mapping equality relationships (Lemma 6.21) or mapping similarity relationships (Lemma 6.24).

6.3.2.1.1 Proof that Mapping Views for Created by Mapping Equality Relationships are in G

Assume that we are creating Map_{G_E} (Map_{G_F} can be shown mutatis mutandis). Let ms be a mapping statement with IDB name idb and $\forall$ relations $r \in body(ms)$, $Keep(r) = false$ (i.e., ms is going to be expressed using mapping equality relationships). We show that mapping views created for mapping equality relationships in Lemma 6.21. To prove Lemma 6.21 we require Lemma 6.19 and Lemma 6.20. Lemma 6.19 shows that the relations in *body(ms)* in the first bullet of Definition 4.17 are the same as $re_1, \ldots, re_p$ in Step (2) in Definition 6.12 and Lemma 6.20 shows that the variables used in Lemma 6.19 are isomorphic to one another. Both Lemma 6.19 and Lemma 6.20 use the definitions of *ms*, *idb*, and *r* as above:

Lemma 6.19: $\forall$ mapping statements $ms \in Map_{E\_F}$, $re_1, \ldots, re_p$ in Step (2) in Definition 6.12 equals the relations in *body(ms)* the first bullet of Definition 4.17.

Proof: Since we know that each IDB only appears in at most one mapping statement over E (Remark 6.1), from the definition of μ, a relation $r \in body(ms)$ IFF $\exists$ e $\in$ E such that $\mu(r, e)$ and either Me(d, e), or Ms(d, e) for some d $\in$ Map_{E_F}. Since $Keep(r) = false$, it follows from bullet 4.b of Definition 6.6 that Me(d,e). From the definition of Merge (Definition 5.1), Lemma 6.9 (there are no fundamental conflicts or implication rules used in the creation of G), and Lemma 6.8 (there is no transitivity of equality in Map_{E_F}), it follows that there exists some g s.t. χ(e,g) and e $\in$ E IFF Me(d,e). Hence $re_1, \ldots, re_p$ in Step (2) of Definition 6.12 is a set of elements corresponding to those relations that are in the body of *ms*. By Lemma 6.10 the names of $re_1, \ldots, re_p$ equal the names of the relations in the body of *ms*, and thus $re_1, \ldots, re_p$ equals the relations in *body(ms)* in Definition 4.17. □

Lemma 6.20: $\forall$ mapping statements $ms \in Map_{E\_F}$, the elements in $Mapped_{rei}$ $\forall$ i, $1 \leq i \leq p$, where $p = |Mapped_{RE}|$ in Step (2) of Definition 6.12 are isomorphic to the variables of *r* in *body(ms)* in the second part of Definition 4.17.

Proof: $\forall$ *r* in *body(ms)*, by the definition of μ:

- $\exists$ e $\in$E s.t. C(Root(E),e) and μ(r,e)
- $\exists$ d $\in Map_{E\_F}$ s.t. Me(d,r).

- $\forall$ attributes $a \in attributes_r$, $\exists$ element $e_1 \in E$ s.t. $C(e,e_1)$ and $\mu(a,e_1)$.
- $\exists\ d \in Map_{E\_F}$ s.t. $\mu(ms) = d$.
- $\forall$ variables v s.t. $v \in distinguished(ms)$ or $v \in Joined(ms)$, there exists a unique $e_2 \in E$ s.t. $\mu(v,e_2)$ and $d_1 \in Map_{E\_F}$ s.t. $C(d, d_1)$, $Me(d_1,e_2)$, and $C(d,d_1)$.

We know that there will be no more elements $e_3 \in E$ and $d_3 \in Map_{E\_F}$ s.t. $Me(d_3, e_3)$ and $C(d,d_3)$ since each IDB only appears in at most one mapping statement over E (Remark 6.1). From these facts and the definition of Merge (which requires a value to be taken from the mapping before from either of the other schemas), it follows that the elements in $Mapped_r$ in the second bullet of 6.2.1 are isomorphic to the attributes of r in $body(ms)$ in the second part of Definition 4.17[23]. □

We are now ready to prove that LV_G and GV_G are correct for relations created as a result of mapping equality relationships:

Lemma 6.21: $\forall$ relations created in G as a result of mapping equality relationships as in the second bullet of 6.2.1, LV_G and GV_G contain the required mappings from Definition 4.17.

Proof: We show that: 1. the left hand sides of both lv_{ms} and gv_{ms} (which are identical) are created correctly, 2. the right hand side of lv_{ms} is created correctly, and 3. the right hand side of gv_{ms} is created correctly. For each we first show the definition in Section 6.2.1 followed by the definition in Definition 4.17:

1. [The left hand side of lv_{ms} and gv_{ms}] ($Mapped_{RE}$) = $q_j(Vars(ms))$. q and q_j are both fresh IDB names, and since the name does not matter, only that it is unused elsewhere, we may safely assume that they are equal.

[23] Note that they differ only on the existential variables that are not joined-on (see Section 6.1.2.1 for how we could make this identical)

From Lemma 6.19 the elements of RE = *body(ms)*. From that and Lemma 6.20, it follows that $\text{Mapped}_{\text{RE}}$ = *Vars(ms)*. Hence $q(\text{Mapped}_{\text{RE}})$ = $q_j(Vars(ms))$.

2. [The right hand side of lv_{ms}] $g(AG) = rq$. Given Lemma 6.1, we know that there exists a unique relation $rq \in G$ s.t. $n_{rq} = idb$ and $attributes_r = Vars(MS_q)$. Thus it follows that $g(AG) = rq$.
3. [The right hand side of gv_{ms}] $\text{re}_1(\text{Mapped}_{\text{re1}})$, …, $\text{re}_p(\text{Mapped}_{\text{rep}})$ = *body(ms)* – This follows directly from Lemma 6.19 and Lemma 6.20.

Hence ∀ relations created in G as a result of mapping equality relationships as in the second bullet of 6.2.1, LV_G and GV_G contain the required mappings from Definition 4.17, and Lemma 6.21 holds. □

6.3.2.1.2 *Proof that Mapping Views Created by Mapping Similarity Relationships are in G*

We proceed here much as in the previous section. Assume that we are creating Map_{G_E} (Map_{G_F} can shown mutatis mutandis). Let *ms* be a mapping statement with IDB name *idb* and ∀ $r \in body(ms)$, $Keep(r) = true$. (i.e., *ms* is going to correspond to mapping similarity relationships). We require Lemma 6.22 and Lemma 6.23 each of which use the definitions of *ms*, *idb*, and *r* as above and only refer to the case where mapping similarity relationships are under consideration (i.e., excluding mapping equality relationships):

Lemma 6.22: ∀ mapping statements **ms** ∈ $\text{Map}_{\text{E\_F}}$, re_1, …, re_p = relations in the body of ms.

Proof:

- By the definition μ: a relation $r \in body(ms)$, IFF μ(*r*, e) and either $\text{Me}(\text{e}_1, \text{e})$, or $\text{Ms}(\text{e}_1, \text{e})$ for some $\text{e}_1 \in \text{Map}_{\text{E\_F}}$.
- Since $Keep(r) = true$, $\text{Ms}(\text{e}_1,\text{e})$.
- From μ: ∃ element $\text{e}_1 \in \text{E}$ s.t. μ(*ms*,e), μ(*r*,e_1), $\text{Ms}(\text{e},\text{e}_1)$.

- From the definition of Merge and hence χ and the fact that we have constrained the input to disallow any mappings that would contain transitivity of equality in Map_{E_F}: $\exists$ some $g_1 \in G$ s.t. $\chi(e,g)$, $\chi(e_1,g_1)$, $H(g,g_1)$.
- Hence RE consists of all of the relations in *ms*.
- We know that there will be no more elements $e_3 \in E$ and $d_3 \in \mathrm{Map}_{E\_F}$ s.t. $Me(e_1, e_3)$ and $C(e_1,d_3)$ since we know that each IDB only appears in at most one mapping statement over E (Remark 6.1). □

Lemma 6.23: $\forall$ mapping statements $ms \in Map_{E\_F}$, $\forall$ attributes $a \in attributes_r$, $r \in body(ms)$, $VM(v,a)$[24] IFF $\exists$ $e_1 \in E$, $e_2 \in \mathrm{Map}_{E\_F}$, $g_1 \in G$ s.t. $\mu(a,e_1)$, $\mu(v, e_2)$, $\psi_g(g_1,e_1)$, $\mathrm{name}(g_1) = \mathrm{name}(e_2)$. **Proof:**

- From μ: $\forall$ attributes $a \in attributes_r$, $r \in body(ms)$, if $VM(v,a)$ then $\exists$ a_E, $r_E \in E$, v_{mape_f}, $ms_{mape\_f} \in \mathrm{Map}_{E\_F}$ s.t. $\mu(a,a_E)$, $\mu(v, vmap_{e\_f})$, $\mu(r,r_E)$, $\mu(ms, ms_{mape\_f})$, $C(r_E,a_E)$, $C(ms_{mape\_f}, v_{mape\_f})$, $Ms(v_{mape\_f},a_E)$, and $Ms(ms_{mape\_f},r_E)$.
- We know that there will be no more elements $e_3 \in E$ and $d_3 \in \mathrm{Map}_{E\_F}$ s.t. $Me(d_3, e_3)$ and $C(map_{e\_f},d_3)$ since we know that each IDB only appears in at most one mapping statement over E (Remark 6.1).
- From the definition of Merge and hence χ: $\exists$ r_g, v_g, $a_g \in G$ s.t. $\chi(r_E,r_g)$, $\chi(a_E,a_g)$, $\chi(ms_{mape\_f},g)$ $C(g,g_v)$, $H(g_v,a_g)$, $C(r_g,a_E)$.
- Hence Lemma 6.23 holds. □

We are now ready to prove Lemma 6.24.

[24] Recall that VM is defined to be the mapping of a variable in Definition 6.4 in Section 6.3.2.

Lemma 6.24: $\forall$ relations created in G as a result of mapping similarity relationships as in the third bullet of 6.2.1, LV_G and GV_G contains the required mappings from Definition 4.17.

Proof: We break this down into three parts: 1. showing that the left hand sides of both lv_{ms} and gv_{ms} (which are identical) are created correctly, 2. showing that the right hand side of lv_{ms} is created correctly, and 3. showing that the right hand side of gv_{ms} is created correctly.

1. [The left hand sides of both lv_{ms} and gv_{ms}] $q(\text{VarsMapped}) = q_j(Vars(ms))$. From Lemma 6.23 and the definition of ψ_g we know that ψ_g is defined on exactly the variables in ms. Hence $\text{VarsMapped} = Vars(ms)$. Since $ms = ms$ and q and q_j are both fresh IDB names, $q(\text{VarsMapped})$ is functionally equivalent to $q_j(Vars(ms))$.
2. [The right hand side of lv_{ms}] $g(AG) = rq$. Given the proofs in Section 6.3.1, we know that there exists a unique relation $rq \in G$ where $n_{rq} = idb$ and $attributes_r = Vars(MS_q)$. Thus $g(AG) = rq$.
3. [The right hand side of gv_{ms}] $\text{AMrel}_g(\text{re}_1), \ldots, \text{AMrel}_g(\text{re}_p) = body(ms)$. Lemma 6.22 proves that ER = the relations in $body(ms)$. From Lemma 6.23 we know that each $\text{AMrel}_g(\text{re}_j)$, $\text{re}_j \in \text{RE}$, contains the same attributes as its corresponding relation in $body(ms)$. Hence $\text{AMrel}_g(\text{re}_1), \ldots, \text{AMrel}_g(\text{re}_p) = body(ms)$.

Hence, $\forall$ relations created in G as a result of mapping similarity relationships as in the third bullet of 6.2.1, LV_G and GV_G contains the required mappings from Definition 4.17, and Lemma 6.24 holds. □

Since all relations corresponding to IDBs are created by either a result of mapping equality relationships or mapping similarity relationships, since we have now proved Lemma 6.21 and Lemma 6.24, Lemma 6.16 holds.

6.3.2.2 Proof that Map_{G_EF} is Correct for Relations in G Corresponding to Relations in E or F

We are now ready to prove Lemma 6.17: Map_{G_EF} is correct for relations in G corresponding to relations in E or F. By the proof in Section 6.3.1.1 we know that all non-mapping included relations will exist in G with the proper attributes. Section 4.3.3.2 requires that GV_G and LV_G must each include $q(attributes_g)$:- $g(attributes_g)$, where q is a fresh IDB name for each $g \in G$ s.t. g corresponds to a non-mapping included relation in E or F. The requirements in the first bullet in Section 6.2.1 will add exactly those mapping views. Hence Lemma 6.17 holds.

6.3.2.3 Proof that No Other Mapping Views Exist in Map_{G_EF}

We are now ready to prove Lemma 6.18: no other mapping views exist in Map_{G_EF}. Since there exist no more relations in G than those required for Lemma 6.16 and Lemma 6.17 and the relationships in G are also only those required to satisfy Lemma 6.16 and Lemma 6.17, there are no other mapping views in GV_G and LV_G, Lemma 6.18 holds.

Since Lemma 6.16, Lemma 6.17, and Lemma 6.18 holds, Lemma 6.2 holds.

Since Lemma 6.1 holds (Section 6.3.1) and Lemma 6.2 as shown here, Theorem 6.2 also holds.

6.4 Allowing More than Two Mapping Statements per IDB

In Section 6.1 we restrict the input to consider only IDB names that appear in two mapping statements. We make this restriction not for the sake of having a correct mediated schema, but to ensure the correctness of Map_{G_EF}. The reason is as follows. Consider the case when we allow IDBs to appear in more than two mapping statements; unless the mapping statements are mapped in pairs, there is no method for differentiating between joins (i.e., the relationships between relations within a mapping statement) and unions (i.e., the relationship between relations in separate mapping statements with the same IDB name) in Map_{E_F}. As an example, consider the mapping in Example 6.5:

Example 6.5:

$q_7(x,y)$:- $e_3(x,y,w)$

$q_7(x,y)$:− $f_1(x,y)$

$q_7(x,y)$:− $f_2(y,z,x)$ □

If only one element in $\text{Map}_{\text{E\_F}}$ corresponded to the concept of q_7, the result would be the mapping in Figure 6.2. While the correct model G is, in fact, the model in Figure 6.3, Map_{G_EF} could not be created correctly without referring to Map_{E_F}. In the current creation of Map_{G_EF} given $\text{Map}_{\text{G\_EF}}$, we assume that all relations in F that are the destination of a mapping relationship from the same mapping element are related in Map_{G_EF} by a join. However, now the relations in Map_{G_EF} could either be related through a join *or* through a union; $\text{Map}_{\text{G\_EF}}$ does not encode enough information for us to be able to differentiate between the two. Figure 6.9 shows how duplicating elements in $\text{Map}_{\text{E\_F}}$ can represent the difference between a join and a union in Map_{E_F}. Hence Map_{G_E} and Map_{G_F} can be created from E, F, and $\text{Map}_{\text{E\_F}}$ rather than needing to refer all the way back to Map_{E_F}. This mapping results in the same model G as shown in Figure 6.3

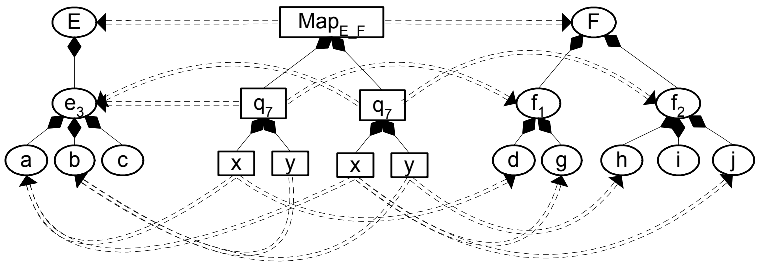


Figure 6.9: A Vanilla mapping for the conjunctive mapping in Example 6.5 **that retains enough information to create** Map_{G_E} **and** Map_{G_F}**.**

However, this strategy relies on the transitivity of mapping equality relationships, and hence only works in the case where mapping equality relationships are used. If mapping similarity relationships are used, then the strategy does not work since those relationships are not transitive. For example, if all of the mapping equality relationships in Figure 6.9 were replaced with mapping similarity relationships, there would be two relations corresponding to q_7 in G.

6.5 *Encoding Alternative Conjunctive Definitions in* **Merge**

A key idea of generic Merge is that every schema element in E, F, or Map_{E_F} should be present in some form in G. Hence, though the encoding in Section 6.1.2.1 can encode the majority of the alternate scenarios in Section 4.4, there are some situations described in Section 4.4 that cannot be encoded using generic Merge alone. As one would expect, both require discarding MSC 1: Completeness as in Section 4.4.1.1. In Section 6.5.1 we discuss when Map_{E_F} is assumed to contain no components (Section 4.4.2) and minimality is also discarded as discussed in Section 4.4.1.2. In Section 6.5.2 we discuss when relations in E and F not referenced in Map_{E_F} are not retained.

6.5.1 Relaxing MSC 3: Not Keeping All Attributes of Relations Corresponding to Relations in D

6.5.1.1 Problems with G

Recall the third Mediated Schema Criterion: "For every component Q_C in Map_{E_F} not subsumed by any intersection in Map_{E_F}, there exists a canonical query for Q_C over G and Map_{G_EF}." If it is relaxed, it is unclear whether to retain attributes represented by existential variables in Map_{E_F}. Using only Encode, Decode, and Merge, it is impossible to encode the case when attributes represented by existential variables in Map_{E_F} in non-mapping-included relations are excluded from G unless relations that appear in Map_{E_F} are retained. Consider the mapping from Example 4.13:

$q_2(x) :- e_1(x,y)$

$q_2(x) :- f_1(x,z)$

This mapping can be encoded in a Vanilla mapping using either mapping equality relationships or mapping similarity relationships. If we connect q_2 to e_1 and f_1 with similarity mapping relationships, then q_2 can be created without representing either y or z as in Figure 6.8. The concepts of y and z are still retained in e_1 and f_1, so the Generic Merge Requirements for Merge are satisfied.

However, if q_2 is connected to e_1 and f_1 with equality mapping relationships, as shown in Figure 6.10, then the representation of q_2 will contain both y and z since Merge retains all relationships in the input, and y, z, and the Contains relationships connecting them must be represented in G to agree with the GMRs. Hence, because of the GMRs because e_1 and f_1 are only represented by q_2, it is impossible to build a mediated schema where q_2 must include y and z.

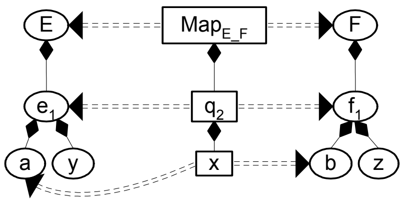


Figure 6.10: A mapping that does not allow the attributes to be kept.

The use of the equality mapping corresponds to encoding the case where the only relations related by Map_{E_F} that are kept are the relations that appear in D; in this case G would not contain either f_1 or e_1. Unfortunately, this is the most natural situation in which the user likely would not want to have y or z in the mediated schema either. This case could, however, be handled by a combination of Model Management operators.

6.5.1.2 Problems with Map_{G_EF}

Another problem can exist if all attributes of a relation r in G created from a mapping statement are not kept. This time it is not a problem of building G but a problem of creating Map_{G_EF}. In this case, the problem occurs if r is created from mapping similarity relationships and all attributes that are joined-on are not retained. In this case, because those attributes are not to be kept in the mediated schema, they do not appear in Map_{E_F}, so there is no way to tell that those attributes are to be joined-on. It is difficult, if not impossible, to see how this could be handled sensibly by any combination of Model Management operators.

6.5.2 Relaxing MSC 1 (G is complete): Discarding Relations not in Map_{E_F}

A key idea of Merge is that every element in E, F, and Map_{E_F} should be represented in some fashion in the merged model. Hence, as in the previous case, if there are relations in E and F that are not to be mentioned at all in G, a separate differencing operator would need to remove them from the relations that were not needed in G.

6.6 Conclusions

In Chapter 4 we considered the problem of using a conjunctive mapping to create a mediated schema. In Chapter 5 we introduced a generic Merge operator for use in creating mediated schemas and other applications. In this chapter we showed how Merge could be used to encode the algorithm in Chapter 4. We now consider the lessons that we have learned from this exercise.

6.6.1 Creating Mediated Schema Easy using Model Management

Vanilla mappings can handle mappings under the restrictions in Section 6.1 and can be used to encode the alternatives described in Section 4.4. For example, given the same relations and conjunctive mappings, Figure 6.11 through Figure 6.13 show how to create a Vanilla mapping that will satisfy three different alternatives: Figure 6.11 shows how to create mappings if MSC 1: Completeness is relaxed (Section 4.4.1.1), Figure 6.12 shows a mapping that satisfies the base semantics (Definition 4.18), and Figure 6.13 shows a Vanilla mapping where MSC 5: minimality (Section 4.4.1.2) is relaxed.

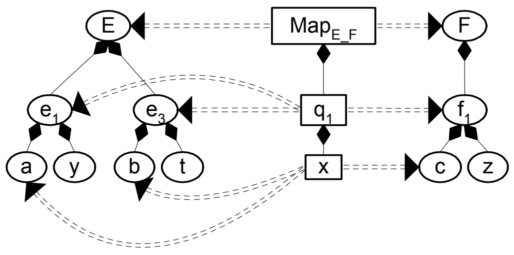


Figure 6.11: A Vanilla mapping in which the mediated schema retains only relations defined in the mapping

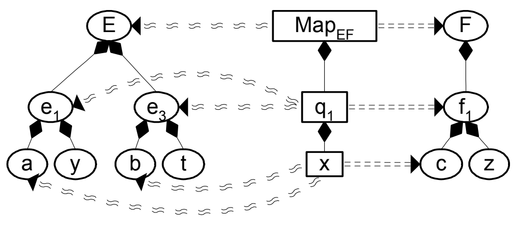


Figure 6.12: A Vanilla mapping for the base semantics: non-mapping-included relationships represented by a relation in the mediated schema

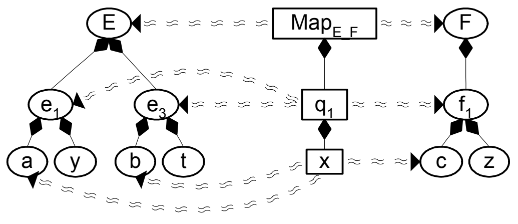


Figure 6.13: A Vanilla mapping where all relations in the input are retained

In addition, Vanilla mappings can express some of the kinds of relationships described in Section 4.5.3 as being inexpressible for conjunctive mappings even restricted to the case of relational schemas.

6.6.2 Creating Mapping Views Difficult Using Model Management

While Model Management and Merge are adept at specifying and creating a mediated schema, they are lacking when it comes to creating the mapping views that allow queries to be translated. In the simplest case, conjunctive mappings using only two mapping statements per IDB, creating the mapping is straightforward. However, as is shown in Section 6.4 even in the case where there are more than two mapping statements per IDB this rapidly becomes very difficult to understand or even impossible to encode in Model Management models, mappings, and operators without resorting to the Expression property of the mapping elements – which is not very generic. It is difficult to envision any generic representation that would be able to handle all such subtleties needed here. This thesis provides the first test of such an end-to-end application; clearly more work is needed at this boundary to see what application support Model Management is capable of providing beyond just creating schemas.

Chapter 7
Conclusions and Future Directions

7.1 Conclusions

As more and more data is stored by more and more people, the number of overlapping sources of data about any given topic is going to increase. In that milieu, it will be increasingly important to have systems that are capable of rapidly querying multiple databases simultaneously. In this thesis we explore how both how to create such systems based on information about how the data sources are related to one another and also how to query those systems efficiently once the system is built. In particular, this thesis makes the following contributions:

- In Chapter 3 we provide the MiniCon algorithm that proves that conjunctive queries can be efficiently answered in data integration systems that use Local-As-View (LAV) mappings. The query rewriting for such systems is done by answering queries using views, where queries are answered using stored queries rather than the underlying relations used to define the views. We show in the first large-scale evaluation of algorithms for answering queries using views that our MiniCon algorithm is faster than previous algorithms for the same problem, sometimes by an order of magnitude. We extend the MiniCon algorithm to include arithmetic comparison predicates, and offer a sound but incomplete solution to this problem. Finally, we show how MiniCon can be extended to rewriting queries in query optimization, which both requires equivalent rewritings and allows access to the relations that define the views.
- In Chapter 4 we leave behind the assumption that the mediated schema is given to us a priori, and discuss how to create the mediated schema given that the source schemas are related to one another using conjunctive mappings – mappings that consist of conjunctive queries. We define a set of mediated schema criteria that describe desirable features in the mediated schema, regardless of how the mediated schema is created, and show a mediated schema and mappings that adhere to these

requirements. We also show by examining the mappings required by the relationships between the local sources, even when the relationships between local sources are very simple, the traditional LAV and GAV mappings from mediated schema to source schemas may not be sufficiently expressive to express the mapping from mediated schema to source schemas. While others have shown that GAV and LAV may not be expressive enough to form the mapping from mediated schema to source schema, Chapter 4 gives an underlying reason for why this occurs.

- In Chapter 5 we extend the problem of mediated schema creation using conjunctive mappings in Chapter 4 to Merge: the problem of merging models for many applications, including data integration, view integration, and ontology merging. Unlike previous algorithms that concentrate on solving meta-model specific problems, we describe how to handle conflicts that occur in merging two models. We provide a generic definition for Merge which includes a first class mapping between input models and describe when we expect that it will be able to be fully automatic and when user intervention is expected. We show how with other Model Management operators Merge in a specific meta-meta-model can subsume previous work from view integration, ontology merging, and programming languages.
- In Chapter 6 we show that the Merge defined in Chapter 5 can be used along with simple import and export operators to mimic the semantic mediated schema creation used in Chapter 4. We show that given an input mapping between the sources (which is outside the scope of this thesis), creating the mediated schema is simple and intuitive in most cases, but that creating the mappings from mediated schema to sources is more difficult and sometimes impossible to be done in a comprehensible fashion using Model Management. This provides not only verification of Merge, but also a first glimpse at how Model Management can be expected to help solve more of the requirements of the applications other than simply creating the mediated schema.

7.2 Future Directions

There are a number of future directions that we are interested in pursuing, all in the realm of extending and Merge other Model Management operators to be more useful and concrete in helping with the semantics of given applications rather than being as divorced from the applications as it is now.

7.2.1 Schema Creation for Peer Data Management Systems

Much of the work in dealing with multiple databases recently has moved from data integration to peer data management systems (Aberer et al. 2002; Arenas et al. 2003; Bernstein et al. 2002; Halevy et al. 2003; Ooi et al. 2003). In a peer data management system, there are many different sources that can be added in an ad-hoc fashion. Deciding what schema these sources should be queried in is a notion of much concern. Some work has been done on answering queries given mappings between peers saying how those schemas are related (Tatarinov et al. 2004). This work is a huge leap forward, but can only answer queries where the concepts are common not only to the schema that the user has, but to all the intermediate schemas that give mappings to the destination schemas. The work in Chapter 4 on automatically determining a mediated schema based on mappings between the source schemas could be leveraged to create schemas to use for querying in peer-based data management systems.

7.2.2 Generic Merge for Complex Structures

Within the generic Merge described in Chapter 5, in some of our experiments we encountered a complex structure in one model that expressed a similar concept to a complex structure in another model, but there was no obvious mapping for the individual elements even though the structures as a whole were similar. An open question is how best to express such similarities and exploit them.

7.2.3 EnforceConstraints

We would like to see a model-driven implementation of the EnforceConstraints operator that we proposed in Section 5.3.2. The goal of this operator is to coerce models that are valid in a meta-meta-model (e.g., Vanilla), but not in any particular meta-model (e.g., relational schemas) into being valid in particular meta-models. Preliminary work suggests that some of the work created for the purpose of translating between data models can be leveraged in order to create this operator. In particular some of the work by Atzeni and Torlone in viewing meta-models as consisting of different patterns and changing a schema from one meta-model to another (Atzeni et al. 1996) seems promising, as does some of the similar work in M(DM) (Barsalou et al. 1992).

7.2.4 Semantic Applications for Model Management

Model Management is at a pivotal point: thus far it has focused on schema-level operators without considering the overall context in which they are used. A critical step toward this broader picture is to build a Model Management system that considers an application from end-to-end, *including* the data. One natural target application is three-way merge described in Section 5.7.2. This problem is common not only in core database applications but also in file versioning and versioning support for computer-supported collaborative work. Creating three-way merge requires rigorous definitions and implementations of two Model Management operators: Diff, which takes the difference of two models, and EnforceConstraints, which coerces models to adhere to the constraints of a specific data model. Creating prototypes of these operators and analyzing results at both the schema and data level will show that Model Management can solve real world problems in the context of the entire application and will likely expose new Model Management research problems.

Bibliography

Aberer, K., Cudré-Mauroux, P., and Hauswirth, M. "A Framework for Semantic Gossiping," *SIGMOD Record* (31:4), December 2002, pp 48-53.

Abiteboul, S., and Duschka, O. "Complexity of Answering Queries Using Materialized Views," Symposium on Principles of Database Systems (PODS), 1998, pp. 254-263.

Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1995, p. 685.

Afrati, F., Gergatsoulis, M., and Kavalieros, T. "Answering Queries using Materialized Views with Disjunctions," International Conference on Database Theory (ICDT), 1999, pp. 435-452.

Afrati, F.N., Li, C., and Ullman, J.D. "Generating Efficient Plans for Queries Using Views," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2001, pp. 319 - 330.

Arenas, M., Kantere, V., Kementsietsidis, A., Kiringa, I., Miller, R.J., and Mylopoulos, J. "The Hyperion Project: From Data Integration to Data Coordination," *SIGMOD Record* (32:3), September 2003, pp 53-58.

Atzeni, P., and Torlone, R. "Management of Multiple Models in an Extensible Database Design Tool," International Conference on Extending Database Technology (EDBT), 1996, pp. 79-95.

Avnur, R., and Hellerstein, J.M. "Eddies: Continuously Adaptive Query Processing," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000, pp. 261-272.

Balasubramaniam, S., and Pierce, B.C. "What is a File Synchronizer?," ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), 1998, pp. 98-108.

Barsalou, T., and Gangopadhyay, D. "M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems," International Conference on Extending Database Technology (EDBT), 1992, pp. 218-227.

Batini, C., Lenzerini, M., and Navathe, S.B. "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys* (18:4) 1986, pp 323-364.

Beeri, C., and Milo, T. "Schemas for Integration and Translation of Structured and Semi-Structured Data," ICDT, 1999, pp. 296-313.

Bello, R., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W., Sun, H., Witkowski, A., and Ziauddin, M. "Materialized Views in Oracle," Very Large Data Bases Conference (VLDB), 1998, pp. 659-664.

Bergamaschi, S., Castano, S., and Vincini, M. "Semantic Integration of Semistructured and Structured Data Sources," *SIGMOD Record* (28:1) 1999, pp 54-59.

Berger, M., Schill, A., and Völksen, G. "Coordination Technology for Collaborative Applications: Organizations, Processes, and Agents," Springer, New York, 1998, pp. 177-198.

Berlage, T., and Genau, A. "A Framework of Shared Applications with a Replicated Architecture," ACM Symposium on User Interface Software and Technology, 1993, pp. 249-257.

Bernstein, P.A. "Applying Model Management to Classical Meta Data Problems," Conference on Innovative Data Systems Research (CIDR), 2003, pp. 209-220.

Bernstein, P.A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. "Data Management for Peer-to-Peer Computing: A Vision," International Workshop on the Web and Databases (WebDB), 2002, pp. 89-94.

Bernstein, P.A., Halevy, A.Y., and Pottinger, R.A. "A Vision of Management of Complex Models," *SIGMOD Record* (29:4) 2000, pp 55-63.

Biskup, J., and Convent, B. "A formal view integration method," SIGMOD, 1986, pp. 398-407.

Bracha, G., and Lindstrom, G. "Modularity Meets Inheritance," Computer Languages Conference, 1992, pp. 282-290.

Buneman, P., Davidson, S.B., and Kosky, A. "Theoretical Aspects of Schema Merging," International Conference on Extending Database Technology (EDBT), 1992, pp. 152-167.

Calì, A., Calvanese, D., Giacomo, G.D., and Lenzerini, M. "On the Expressive Power of Data Integration Systems," Conference on Conceptual Modeling (ER), 2002, pp. 338-350.

Calvanese, D., Giacomo, G.D., Lenzerini, M., Nardi, D., and Rosati "Schema and Data Integration Methodology for DWQ," DWQ-UNIROMA-004, DWQ Consortium.

Calvanese, D., Giacomo, G.D., Lenzerini, M., and Vardi, M. "Rewriting of regular expressions and regular path queries," Symposium on Principles of Database Systems (PODS), 1999, pp. 194-204.

Chandra, A.K., and Merlin, P.M. "Optimal Implementation of conjunctive queries in relational databases," ACM Symposium on Theory of Computing (STOC), 1977, pp. 77-90.

Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K. "Optimizing Queries with Materialized Views," International Conference on Data Engineering (ICDE), 1995, pp. 190-200.

Chaudhuri, S., and Vardi, M. "On the Equivalence of Recursive and Nonrecursive Datalog Programs," Symposium on Principles of Database Systems (PODS), 1992, pp. 55-66.

Chaudhuri, S., and Vardi, M. "On the Complexity of Equivalence between Recursive and Nonrecursive Datalog Programs," Symposium on Principles of Database Systems (PODS), 1994, pp. 55-66.

Chirkova, R., Halevy, A.Y., and Suciu, D. "A Formal Perspective on the View Selection Problem," Very Large Data Bases Conference (VLDB), 2001, pp. 59-68.

Cohen, S., Nutt, W., and Serebrenik, A. "Rewriting Aggregate Queries Using Views," Symposium on Principles of Database Systems (PODS), 1999, pp. 155-166.

Dhamankar, R., Lee, Y., Doan, A., Halevy, A.Y., and Domingos, P. "iMAP: Discovering Complex Mappings between Database Schemas," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2004, pp. 383-394.

Duschka, O.M., and Genesereth, M.R. "Answering Recursive Queries Using Views," Symposium on Principles of Database Systems (PODS), 1997a, pp. 109-116.

Duschka, O.M., and Genesereth, M.R. "Query Planning in Infomaster," ACM Symposium on Applied Computing (SAC), 1997b, pp. 109-111.

Duschka, O.M., and Genesereth, M.R. "Query Planning with Disjunctive Sources," AAAI Workshop on AI and Information Integration, AAAI Press, 1998, pp. 23-28.

Duschka, O.M., and Levy, A.Y. "Recursive Plans for Information Gathering," The International Joint Conference on Artificial Intelligence (IJCAI), 1997c, pp. 778-784.

Fagin, R., Kolatis, P.G., Popa, L., and Tan, W.C. "Composing Schema Mappings: Second-Order Dependencies to the Rescue," Symposium on Principles of Database Systems (PODS), 2004, pp. 83-94.

Farquhar, A., Fikes, R., and Rice, J. "The Ontolingua Server: A Tool for Collaborative Ontology Construction," KSL-96-26, Stanford University Knowledge Systems Laboratory.

Fikes, R. "Ontologies: What Are They, and Where's The Research?," Principles of Knowledge Representation and Reasoning (KR), 1996, pp. 652-653.

Florescu, D., Raschid, L., and Valduriez, P. "A Methodology for Query Reformulation in CIS using Semantic Knowledge," *International Journal of Intelligent & Cooperative Information Systems* (5:4) 1996, pp 431-468.

Friedman, M., Levy, A., and Millstein, T. "Navigational Plans for Data Integration," Proceedings of the National Conference on Artificial Intelligence (AAAI), 1999, pp. 67-73.

Friedman, M., and Weld, D. "Efficient Execution of Information Gathering Plans," International Joint Conference on Artificial Intelligence (IJCAI), 1997, pp. 785-791.

Grahne, G., and Mendelzon, A.O. "Tableau Techniques for Querying Information Sources Through Global Schemas," International Conference on Database Theory (ICDT), 1999, pp. 332-347.

Grumbach, S., Rafanelli, M., and Tininini, L. "Querying Aggregate Data," Symposium on Principles of Database Systems (PODS), 1999, pp. 174-184.

Gryz, J. "Query Folding with Inclusion Dependencies," International Conference on Data Engineering (ICDE), 1998, pp. 126-133.

Gupta, A., Harinarayan, V., and Quass, D. "Aggregate-query processing in data warehousing environments," Very Large Data Bases Conference (VLDB), 1995, pp. 358-369.

Halevy, A.Y. "Answering Queries Using Views: A Survey," *VLDB Journal* (10:4), December 2001, pp 270-294.

Halevy, A.Y., Ives, Z.G., Suciu, D., and Tatarinov, I. "Piazza: Data Management Infrastructure for Semantic Web Applications," International Conference on Data Engineering (ICDE), 2003, pp. 505-516.

Harinarayan, V., Rajaraman, A., and Ullman, J.D. "Implementing Data Cubes Efficiently," ACM SIGMOD International Conference on Management of Data (SIGMOD), 1996, pp. 205-216.

Harrison, W., and Ossher, H. "Subject-Oriented Programming (A Critique of Pure Objects)," ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 1993, pp. 411-428.

He, B., and Chang, K.C.-C. "Statistical Schema Matching across Web Query Interfaces," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003, pp. 217-228.

Hull, R. "Relative Information Capacity of Simple Relational Database Schemata," Symposium on Principles of Database Systems (PODS), 1984, pp. 97-109.

Hull, R. "Relative Information Capacity of Simple Relational Database Schemata," *SIAM Journal of Computing* (15:3), August 1986, pp 856-886.

Hull, R., and Yoshikawa, M. "ILOG: Declarative Creation and Manipulation of Object Identifiers," Very Large Data Bases Conference (VLDB), 1990, pp. 455-468.

Ives, Z.G. "Efficient Query Processing for Data Integration," in: *Computer Science and Engineering*, University of Washington, Seattle, WA, USA, 2002, p. 186.

Kalinichenko, L.A. "Methods and Tools for Equivalent Data Model Mapping Construction," International Conference on Extending Database Technology (EDBT), 1990, pp. 92-119.

Kang, J., and Naughton, J.F. "On Schema Matching with Opaque Column Names and Data Values," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003, pp. 205-216.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., and Irwin, J. "Aspect-Oriented Programming," Object-Oriented Programming, 11th European Conference (ECOOP), 1997, pp. 220-242.

Klug, A. "On Conjunctive Queries Containing Inequalities," *Journal of the ACM (JACM)* (35:1) 1988, pp 943-962.

Kolaitis, P., Martin, D., and Thakur, M. "On the Complexity of the Containment Problem for Conjunctive Queries with Built-in Predicates," Symposium on Principles of Database Systems (PODS), 1998, pp. 197-204.

Kossmann, D. "The State of the Art in Distributed Query Processing," *ACM Computing Surveys* (32:4), December 2000, pp 422-469.

Kushmerick, N., Doorenbos, R., and Weld, D. "Wrapper Induction for Information Extraction," International Joint Conference on Artificial Intelligence (IJCAI), 1997, pp. 729-737.

Kwok, C.T., and Weld, D.S. "Planning to gather information," Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI), 1996, pp. 148-155.

Lakshmanan, L.V.S., Sadri, F., and Subramanian, I.N. "SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems," Very Large Data Bases Conference (VLDB), 1996, pp. 239-250.

Lambrecht, E., Kambhampati, S., and Gnanaprakasam, S. "Optimizing Recursive Information Gathering Plans," International Joint Conference on Artificial Intelligence (IJCAI), 1999, pp. 1204-1211.

Larson, J.A., Navathe, S.B., and Elmasri, R. "A Theory of Attribute Equivalence in Databases with Application to Schema Integration," *Transactions on Software Engineering* (15:4), April 1989, pp 449-463.

Lenzerini, M. "Data Integration: A Theoretical Perspective," Symposium on Principles of Database Systems (PODS), 2002, pp. 233-246.

Levy, A.Y., Mendelzon, A.O., Sagiv, Y., and Srivastava, D. "Answering Queries Using Views," Symposium on Principles of Database Systems (PODS), 1995, pp. 95-104.

Levy, A.Y., Rajaraman, A., and Ordille, J.J. "Querying Heterogeneous Information Sources Using Source Descriptions," Very Large Data Bases Conference (VLDB), 1996, pp. 251-262.

Levy, A.Y., and Sagiv, Y. "Queries Independent of Updates," Very Large Data Bases Conference (VLDB), 1993, pp. 171-181.

Madhavan, J., Bernstein, P.A., and Rahm, E. "Generic Schema Matching with Cupid," Very Large Databases Conference (VLDB), 2001, pp. 49-58.

Madhavan, J., and Halevy, A.Y. "Composing Mappings Among Data Sources," Very Large Data Bases Conference (VLDB), 2003, pp. 572-583.

McBrien, P., and Poulovassilis, A. "Data Integration by Bi-Directional Schema Transformation Rules," International Conference on Data Engineering (ICDE), 2003, pp. 227-238.

McGuinness, D.L., Fikes, R., Rice, J., and Wilder, S. "The Chimæra Ontology Environment," National Conference on Artificial Intelligence (AAAI), 2000, pp. 1123-1124.

Melnik, S., Rahm, E., and Bernstein, P.A. "Rondo: A Programming Platform for Generic Model Management," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003, pp. 193-204.

Miller, R.J. "Using Schematically Heterogeneous Structures," ACM SIGMOD International Conference on Management of Data (SIGMOD), 1998, pp. 189-200.

Miller, R.J., Ioannidis, Y.E., and Ramakrishnan, R. "The Use of Information Capacity in Schema Integration and Translation," Very Large Data Bases Conference (VLDB), 1993, pp. 120-133.

Mitra, P. "An Algorithm for Answering Queries Efficiently Using Views," The Australasian Conference on Database Technologies, 1999, pp. 99-106.

Mork, P., Pottinger, R.A., and Bernstein, P.A. "Challenges In Precisely Aligning Models of Human Anatomy," American Medical Informatics Association Annual Symposium, 2004, p. To Appear.

Munson, J.P., and Dewan, P. "A Flexible Object Merging Framework," Conference on Computer Supported Cooperative Work (CSCW), 1994, pp. 231-242.

Noy, N.F., and Musen, M.A. "SMART: Automated Support for Ontology Merging and Alignment," Banff Workshop on Knowledge Acquisition, Modeling, and Management, 1999.

Noy, N.F., and Musen, M.A. "PROMPT: Algorithm and Tool for Ontology Merging and Alignment," Proceedings of the National Conference on Artificial Intelligence (AAAI), 2000, pp. 450-455.

Ooi, B.C., Shu, Y., and Tan, K.-L. "Relational Data Sharing in Peer-Based Data Management Systems," *SIGMOD Record* (23:3), September 2003, pp 59-64.

Ossher, H., and Harrison, W. "Combination of Inheritance Hierarchies," Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1992, pp. 25-40.

Ossher, H., Kaplan, M., Katz, A., Harrison, W., and Kruskal, V. "Specifying Subject-Oriented Composition," *Theory and Practice of Object Systems* (2:3) 1996, pp 179-202.

Papakonstantinou, Y., and Vassalos, V. "Query Rewriting for Semi-Structured Data," ACM SIGMOD International Conference on Management of Data (SIGMOD), 1999, pp. 455-466.

Popa, L., Deutsch, A., Sahuguet, A., and Tannen, V. "A Chase Too Far?," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000, pp. 273-284.

Pottinger, R.A., and Bernstein, P.A. "Merging Models Based on Given Correspondences," Very Large Data Bases Conference (VLDB), 2003, pp. 862-873.

Pottinger, R.A., and Halevy, A.Y. "MiniCon: A scalable algorithm for answering queries using views," *VLDB Journal* (10:2-3) 2001, pp 182-198.

Qian, X. "Query Folding," International Conference on Data Engineering (ICDE), 1996, pp. 48-55.

Rahm, E., and Bernstein, P.A. "A Survey of Approaches to Automatic Schema Matching," *VLDB Journal* (10:4) 2001, pp 334-350.

Rajaraman, A., Sagiv, Y., and Ullman, J.D. "Answering Queries Using Templates with Binding Patterns," Symposium on Principles of Database Systems (PODS), 1995, pp. 105-112.

Rector, A., Gangemi, A., Galeazzi, E., Glowinski, A., and Rossi-Mori, A. "The GALEN CORE Model Schemata for Anatomy: Towards a re-usable application-independent model of medical concepts," The Twelfth International Congress of the European Federation for Medical Informatics, 1994, pp 229-233.

Rosenthal, A., and Reiner, D. "Tools and Transformations - Rigorous and Otherwise - for Practical Database Design," *ACM Transactions on Database Systems* (19:2), June 1994, pp 167-211.

Rosse, C., Shapiro, L.G., and Brinkley, J.F. "The digital anatomist foundational model: principles for defining and structuring its concept domain," AMIA, 1998, pp. 820-824.

Sabetzadeh, M., and Easterbrook, S. "Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach," The IEEE International Conference on Automated Software Engineering (ASE), 2003, pp. 12-21.

Sagiv, Y. "Optimizing Datalog Programs," in: *Foundations of Deductive Databases and Logic Programming,* J. Minker (ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 659-698.

Sagiv, Y., and Yannakakis, M. "Equivalence Among Relational Expressions With the Union and Difference Operators," *Journal of the ACM* (27:4) 1981, pp 633-655.

Shmueli, O. "Equivalence of Datalog Queries is Undecidable," *Journal of Logic Programming* (15) 1993, pp 231-241.

Shu, N.C., Housel, B.C., and Lum, V.Y. "CONVERT: A High Level Translation Definition Language for Data Conversion," *Communications of the ACM* (18:10) 1975, pp 557-567.

Song, W.W., Johannesson, P., and Bubenko, J.A., Jr. "Semantic Similarity Relations in Schema Integration," *Data Knowledge and Engineering* (19:1) 1996, pp 65-97.

Spaccapietra, S., and Parent, C. "View Integration: A Step Forward in Solving Structural Conflicts," *IEEE Transactions on Data Knowledge and Data Engineering (TKDE)* (6:2), April 1994, pp 258-274.

Srivastava, D., Dar, S., Jagadish, H.V., and Levy, A.Y. "Answering SQL Queries Using Materialized Views," Very Large Data Bases Conference (VLDB), 1996, pp. 318-329.

Steinbrunn, M., Moerkotte, G., and Kemper, A. "Heuristic and Randomized Optimization for the Join," *VLDB Journal* (6:3) 1997, pp 191-208.

Stumme, G., and Maedche, A. "FCA-Merge: Bottom-Up Merging of Ontologies," International Joint Conference on Artificial Intelligence (IJCAI), 2001, pp. 225-234.

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., and Lakhal, L. "Fast Computation of Concept Lattices Using Data Mining Techniques," Knowledge Representation Meets Databases (KRDB), 2000, pp. 129-139.

Tatarinov, I., and Halevy, A.Y. "Efficient Query Reformulation in Peer Data Management Systems," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2004, pp. 539-550.

Theodoratos, D., and Sellis, T. "Data Warehouse Configuration," Very Large Data Bases Conference (VLDB), 1997, pp. 126-135.

Tsatalos, O.G., Solomon, M.H., and Ioannidis, Y.E. "The GMAP: A Versatile Tool for Physical Data Independence," *VLDB Journal* (5:2) 1996, pp 101-118.

Ullman, J.D. *Principles of Database and Knowledge-base Systems, Volumes I, II* Computer Science Press, Rockville MD, 1989.

Ullman, J.D. "Information Integration Using Logical Views," International Conference on Database Theory (ICDT), 1997, pp. 19-40.

Yang, H.Z., and Larson, P.A. "Query Transformation for PSJ-queries," Very Large Data Bases Conference (VLDB), 1987, pp. 245-254.

Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., and Urata, M. "Answering Complex SQL Queries using Automatic Summary Tables," ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000, pp. 105-116.

Zhang, X., and Ozsoyoglu, M.Z. "On Efficient Reasoning with Implication Constraints," Proc. of the Int. Conf. on Deductive and Object-Oriented Databases (DOOD), 1993, pp. 236-252.

Appendix A
Proof of Correctness of the MiniCon Algorithm

A.1 Preliminaries

We consider conjunctive queries and views without built-in predicates. We assume the query has the form $Q(X) \text{ :- } e_1(\overline{X_1}),\ldots,e_n(\overline{X_n})$.

Without loss of generality we assume that no variable appears in more than one view, and the variables used in the views are disjoint from those in the query. Furthermore, we assume that the heads of the views and the query do not contain multiple occurrences of any variable. We apply variable mappings to tuples and to atoms with the obvious meaning, i.e., $Q(\bar{X}) = (\varphi(x_1),\varphi(x_2),\ldots,\varphi(x_n))$ where $\bar{X} = (x_1,\ldots,x_n)$.Recall that a maximally-contained rewriting is, in general, a union of conjunctive rewritings. A conjunctive rewriting has the form

$Q'(\overline{Y}) \text{ :- } V_1(\overline{Y_1}),V_2(\overline{Y_2}),\ldots,V_k(\overline{Y_k})$.

Note that for any $i \neq j$ it is possible that $V_i = V_j$. Given a conjunctive rewriting Q', the *expansion* of Q', denoted by Q'' is the query in which the view atoms are replaced by their definitions (i.e., they are unfolded). Note that when expanding the view definitions we need to create fresh variables for the existential variables in the views. We assume we have a function $f^i(x)$ that returns the i th fresh copy of a variable x. For a given subgoal $g_i \in Q'$, we denote by $exp(i)$ the set of subgoals in Q'' obtained by expanding the definition of V_i. Given two head homomorphisms h_1 and h_2 over the variables of a view V, we say that h_2 is more restrictive than h_1 if whenever $h_1(x)=h_1(y)$, then $h_2(x)=h_2(y)$. Recall that the MiniCon Algorithm produces conjunctive rewritings of the form $Q'(EC(\overline{X})) \text{ :- } V_{C_1}(EC(\Psi_1(\overline{Y_{C_1}}))),\ldots,V_{C_m}(EC(\Psi_m(\overline{Y_{C_m}})))$.

Where for a variable x in Q, $EC(x)$ denotes the representative variable of the set to which x belongs. EC is defined to be the identity on any variable that is not in Q.

Remark 7.1: The following property will be used in the soundness proof. Suppose that a subgoal $g \in Q$ is in G_i, i.e., $\varphi_i(g) \in h_i(V_i)$. The expansion Q'' will contain an atom $\tau(g)$, where, for a variable x:

$\tau(x) = EC(x)$ if $\varphi_i(x)$ is a head variable in $h_i(V_i)$, and

$\tau(x) = f^i(x)$ otherwise □

A.2 Proof of Soundness

We need to show that every conjunctive rewriting Q' that is obtained by the MiniCon Algorithm is contained in Q. To show soundness, we show that there is a containment mapping Y, from Q to Q''. We define an intermediate Y_i for $i = 0, \ldots, k$ by induction as follows. The containment mapping Y will be defined to be Y_k.

U2. For all x where $x \in Vars(Q)$ and $EC(x) \in Vars(Q'')$, $Y_0(x) = EC(x)$.

U3. Y_i is an extension of Y_{i-1}, defined as follows: for all x in the $Domain(\varphi_i)$, if $x \notin Domain(Y_{i-1})$ then $Y\_i(x) = f^i(EC(\varphi_i(x)))$.

Now we show that Y is a containment mapping.

- Mapping of the head: we need to show that $Y(\overline{X}) = EC(\overline{X})$. Because of U1, it suffices to show that for every variable in $x \in \overline{X}$, $EC(x)$ appears in Q''. By Property 3.1, clause C1, we know whenever x is in the domain of φ and is a head variable in Q, φ maps x to a head variable in $h(V)$. By Property 3.2, clause D1, we know that given a MCD set, all the head variables in Q are in the domain of *some* MCD in the set. From the definition of Ψ_i, we know that $\overline{X}$ is a subset of the union of the ranges of the Ψ_i's, and hence, $EC(x)$ is in Q'' for every $x \in \overline{X}$.
- Mapping of a subgoal g. We need to show that Q'' includes $Y(g)$. By Remark 7.1 we know that Q'' includes $\tau(g)$. It suffices to show that $Y(g) = \tau(g)$, which follows immediately from the definition of Y.

A.3 Completeness

Let P be a maximally-contained rewriting of Q using $\mathcal{V}$, and let R be the rewriting produced by the MiniCon Algorithm. The MiniCon Algorithm is complete if $R \sqsupseteq P$. Since both R and P are unions of conjunctive queries, it suffices to show that if p' is a conjunctive rewriting in P, then there exists a conjunctive rewriting r' in R, such that $r' \sqsupseteq p'$ (Sagiv et al. 1981). Since p' is part of a maximally-contained rewriting of Q, there exists a containment mapping θ from Q to the expansion p'' of p' (Chandra et al. 1977). We will use θ to show that there exists a set of MCDs that are created by the MiniCon Algorithm such that when the MCDs are combined, we obtain a conjunctive rewriting r' that contains p'.

We proceed as follows:

- $\forall$ subgoals $g_i \in p'$, we define G_i to be the set of subgoals $g \in Q$, such that $\theta(g) \in exp(i)$ (i.e., G_i includes the set of subgoals in Q that are mapped to the expansion of g_i in p''). Note that for $i \neq j$, the sets G_i and G_j are disjoint.
- We denote by θ_i the restriction of the containment mapping θ to the variables appearing in G_i.
- The mapping θ_i is a mapping from $Vars(G_i)$ to $Vars(exp(g_i))$. However, it can be written as a composition of two mappings, one from $Vars(G_i)$ to $h_i(Vars(V_i))$ (where h_i is a head homomorphism on V_i), and another from $h_i(Vars(V_i))$ to $Vars(exp(g_i))$. Formally, there exists a mapping τ_i: $Vars(G_i) \rightarrow h_i(Vars(V_i))$ and a renaming α of the variables in $h_i(Vars(V_i))$, such that $\theta\_i(x) = \alpha(\tau_i(h_i(x)))$ for every variable $x \in G_i$.

  We choose h_i to be the least restrictive head homomorphism on $Vars(V_i)$ for which τ_i and α exist. Note that since we chose h_i to be the least restrictive head homomorphism, then *any* MCD created by the MiniCon Algorithm for V_i would at least as restrictive as τ_i (hence, τ_i depends only on Q and the view V_i, and not on how V_i is used in the rewriting p').
- We show that we now have all the components of a MCD, which we will denote by C_i:

  h_i is a head homomorphism on $Vars(V_i)$,

  $h_i(V_i(\overline{A}))$ is the result of applying h_i to the head variables $\overline{A}$ of V_i.

  τ_i is a partial mapping from $Vars(Q)$ to $h_i(Vars(V_i))$, and

  G_i is a set of subgoals in Q that are covered by τ_i

  Furthermore, the MCD C_i satisfies the conditions of Property 3.1 which are enforced by the MiniCon Algorithm:

C1. For any head variable x of Q, $\tau_i(x)$ is a head variable of $h_i(V_i)$, because $\theta_i(x)$ is a head variable of p''.

C2. It follows from the fact that θ_i is a restriction of a containment mapping from Q to p'', that if $\tau_i(x)$ is an existential variable in $h_i(V_i)$, then for every subgoal $g_1 \in Q$ that includes x (1) all the variables in g_1 are in the domain of τ_i, and (2) $\tau_i(g_1) \in h_i(V_i)$.

In addition, note that $C_1,\ldots,C_k$ satisfy Property 3.2, which is the condition that the MiniCon Algorithm checks before it combines a set of MCDs:

D1. $G_1 \cup \ldots \cup G_k= Subgoals(Q)$ because θ is a containment mapping from Q to p'', and

D2. for every $i \neq j$, $G_i \cap G_j = \varnothing$ because of the way we constructed the G_i's.

- The only difference between the MCD C_i and a MCD created by the MiniCon Algorithm is that τ_i may not be the *minimal* mapping necessary to satisfy Property 3.1. However, this is easy to fix by simply decomposing the MCD C_i into a set of MCDs that satisfy Property 3.1 exactly and contain only minimal mappings for τ_i and minimal sets of subgoals in their fourth component. Note that even after decomposing the MCDs, the G_i's are still disjoint subsets of subgoals in Q, and hence Property 3.2 is still satisfied.
- At this point we have shown that we have a set of MCDs $C_1, \ldots, C_l$, that satisfy Property 3.1 and Property 3.2. Furthermore, each of the mappings τ_i in the MCDs less restrictive than θ in the following sense: for any variables x, y, if $\tau_i(x)=\tau_i(y)$ then $\theta(x)=\theta(y)$

As a result, when procedure **combineMCDs** creates the function EC, it will have the property that $EC(x)=EC(y)$ only if $\theta(x)=\theta(y)$. Consequently, the conjunctive rewriting r' that is produced when $C_1, \ldots, C_l$ are combined will have the same property: whenever the same variable appears in two argument positions in r', those two argument positions will have the same variable in p'. Hence, there is a containment mapping from r' to p', and therefore $p' \sqsubseteq r'$.

Appendix B
Examples of Relational Mathematical Mapping Requirements

- Functionality: It is important to realize functionality is talking about the entire state at once, rather than just about the state of one relation. For example, if

  I(m1) =

  {*m11(<1>)*, *m12(<3>)*}

  {*m11(<1>)*, *m12(<2>)*}

  m1_m2({*m11(<1>)*, *m12(<3>)*} $\rightarrow$ *m21(<3>)*, {*m11(<1>)*, *m12(<2>)*} $\rightarrow$ *m22(<4>)*) is functional despite mapping *m11(<1>)* to two different values in *m2* because the mapping takes the state as a whole (including the state of *m12*) rather than just the state of *m11*. Similar arguments apply to injectivity.
- Totality: For example, *m1_m2(e1(<1, 2>))* $\rightarrow$ *f1(<1>, <2>)* would be a total function if *I(m1)* = σ_{m1} = {*e1(<1, 2>*}.
- Surjectivity: For example, *m1_m2*(*e1(<1, 2>)*) $\rightarrow$ *f1(<1>, <2>)* would be a surjective function if *I(m2)* = σ_{m2} = {*f1(<1>, <2>)*}.

Appendix C
Our Modifications to the BDK Algorithm

BDK combine the representation of our Has-a and Type-of relationships into one relationship. They represent the fact that an element r Has-a element x of type y by an arrow from r to y with the label x. The different representations are shown in Figure C.1 (a single Has-a relationship) and in Figure C.2 (a violation of the one-type restriction). Although they represent both the Vanilla Has-a and Type-of relationships with a single relationship, the BDK algorithm only involves duplicate element types, not duplicate containers, hence our modification involves only transforming the Vanilla Type-of relationships rather than the associated Has-a and Contains relationships.

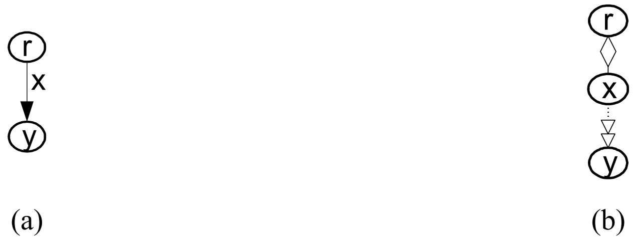


Figure C.1: Modeling Has-a and Type-of relationships. (a) In the BDK meta-meta-model (b) In Vanilla

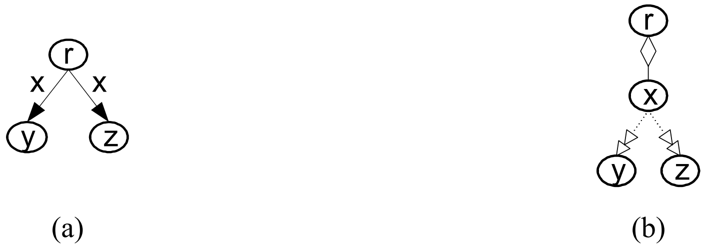


Figure C.2: A violation of the one-type restriction. (a) In the BDK meta-meta-model (b) In Vanilla

Applying the BDK algorithm to a model M expressed in Vanilla requires the following conceptual steps:

1. Apply all implied relationships (listed in Section 5.4.1) that can lead to the inclusion of additional Type-of or Is-a relationships.

2. Apply a transformation T1(M)→N to transform M into a model N in BDK's meta-meta-model. This transformation operates in the following fashion:
    a. Each element m of M becomes a node n in N, where the label of n is the value of the ID property of m.
    b. Each Vanilla Is-a relationship between two elements in M becomes a BDK Is-a relationship between the corresponding nodes in N.
    c. For each Vanilla Type-of relationship in M, a BDK Has-a relationship with label "t" is created between corresponding nodes in N. Note that the label is unimportant except that it must be the same for the BDK algorithm to function correctly.
3. Run the BDK algorithm to change weak schemas (those that do not obey the one-type restriction) into strong schemas (those that obey the one-type restriction).
4. Remove all Type-of and Is-a relationships from M (i.e., all relationships imported into N are removed)
5. Apply a transformation T2(N)→M to add into M:
    a. All nodes from N that do not correspond to elements in M (i.e., nodes that were created to help resolve one-type conflicts).
    b. All relationships in N (i.e., all relationships that were originally imported into N plus all of the changes and additions to resolve one-type conflicts). All Is-a relationships are added directly between corresponding elements since the Is-a relationships are the same in BDK and in Vanilla. All BDK Has-a relationships are transformed into Vanilla Type-of relationships, and their labels are discarded (since we specified that these would all be equal to the arbitrary choice "t", no information is lost in this transformation)
6. All implied relationships are removed.

This algorithm guarantees:

1. All relationships originally in M are retained at least implicitly. All relationships that are not Is-a or Type-of relationships are retained. All Is-a and Type-of relationships are imported from N. Since the BDK transformation only changes the relationships based on the implication rule "If T(q, r) and I(r, s) then T(q, s)," which exists in Vanilla and makes the same guarantee of retaining all relationships at least implicitly, this is retained.
2. No element in M has more than one type. The BDK algorithm ensures that this is true for Has-a relationships in N. No Type-of relationships are included in M other than those in N, and T2's transformation between Has-a and Type-of relationships cannot violate this. The final step of removing implied relationships cannot violate this since no new relationships are added.
3. The associativity and commutativity properties of BDK are retained. Both T1 and T2 are entirely order independent. Note that in order for the conflict resolution to be associative and commutative, the algorithm must be run only *once* at the end of a series of merges (see (Buneman et al. 1992) for an explanation as to why).

Appendix D
Compose

The Compose operator takes a mapping, Map_{A_B}, between models A and B and a mapping, Map_{B_C}, between models B and C and returns, Map_{A_C}, the composed mapping between A and C. We define Compose based on the definition of right composition (i.e., composition driven by the right hand mapping) in (Bernstein 2003).

Let m be a mapping element in mapping Map_{A_B} between models A and B. Define domain(m) (respectively, range(m)) to be all elements e such that $e \in A$ (respectively $e \in B$) and Me(m, e). For each element e in the domain of each mapping element m in Map_{B_C}, Compose must identify the mapping elements in Map_{A_B} that provide input to e. We compose each element $m_{B\_C} \in Map_{B\_C}$ with the union of all elements $m_{A\_B} = m_{AB1}, \ldots, m_{ABn} \in Map_{A\_B}$ where range(m_{A_B}) $\cap$ domain(m_{B_C}) $\neq \varnothing$.

Given this decision, we define the composition Map_{A_C} of Map_{A_B} and Map_{B_C} constructively as follows:

1. (Copy) Create a copy Map_{A_C} of Map_{B_C}. Note that Map_{A_C} and Map_{B_C} have corresponding mapping relationships to B and C and, therefore, the same domains and ranges.
2. (Pre-compute Input) $\forall$ objects m_{A_C} in Map_{A_C}, let Input(m_{A_C}) be the set of all elements m_{A_B} in Map_{A_B} such that range(m_{A_B}) $\cap$ domain(m_{B_C}) $\neq \varnothing$.
3. (Define domains) $\forall\, m_{A\_C} \in Map_{A\_C}$,
   a. If $\bigcup_{m_{ABi} \in Input(m_{AC})} range(m_{ABi}) \supseteq domain(m_{AC})$, then set domain($m_{A\_C}$) = $\bigcup_{m_{ABi} \in Input(m_{AC})} domain(m_{ABi})$.

b. Else if m_{A_C} is not needed as a support element[25] (because none of its descendants satisfies (3a)), then delete it, else set domain(m_{A_C}) = range(m_{A_C}) = $\varnothing$.

Step 3 defines the domain of each object m_{A_C} in Map_{A_C}. Input(m_{A_C}) is the set of all objects in Map_{A_B} whose range intersects the domain of m_{A_C}. If the union of the ranges of Input(m_{A_C}) contains the domain of m_{A_C}, then the union of the domains of Input(m_{A_C}) becomes the domain of m_{A_C}. Otherwise, m_{A_C} is not in the composition, so it is either deleted (if it is not a support object, required to maintain the well-formed-ness of Map_{A_C}), or its domain and range are cleared (since it does not compose with objects in Map_{A_B}).

[25] A support element is an element needed to support the structural integrity of the model (i.e., an element needed to ensure that the result is a model)

Appendix E
Three-Way Merge Algorithm

1. Map_{O_A} = Match(O, A) (can be automatic from History properties)
2. Map_{O_B} = Match(O, B) (can be automatic from History properties)
3. $Map_{O\_A'}$ = Apply(Map_{O_A}) such that if $e \in Map_{O\_A}$ if domain(e) is identical to range(e), then delete e (we are capturing the things changed in A)
4. $Map_{O\_B'}$ = Apply(Map_{O_B}) such that if $e \in Map_{O\_B}$ if domain(e) is identical to range(e), then delete e (we are capturing the things changed in B)
5. $Changed_A$ = range($Map_{O\_A'}$) (the things changed in A)
6. $Changed_B$ = range($Map_{O\_B'}$) (the things changed in B)
7. Map_{ChA_ChB} = Match($Changed_A$, $Changed_B$)
8. Map_{ChB_ChA} = Match($Changed_B$, $Changed_A$)
9. A′ = Diff($Changed_A$, $Changed_B$, Map_{ChA_ChB}) (A′ represents the things changed in A that were not changed in B, and mutatis mutandis for B′ below)
10. B′ = Diff($Changed_B$, $Changed_A$, Map_{ChB_ChA})
11. Map_{A_B} = Match(A,B) (according to OIDs)
12. G = Merge(A, Map_{A_B}, B)
13. $Map_{G\_A'}$ =Match(G,A′)
14. GA = Merge(G, $Map_{G\_A'}$, A′) with preference for A′
15. $Map_{GA\_'B'}$ =Match(GA′,B′)
16. GAB = Merge(GA′, $Map_{GA'\_B'}$, B′) with preference for B′ (GAB represents the full merge with a preference for those things that have changed in either A or B but not both)

17. $Deleted_A$ = Diff(O,A,Map_{O_A})
18. $Deleted_B$ = Diff(O, B, Map_{O_B})
19. $Map_{DeletedA\_ChangedB}$ = Match($Deleted_A$, $Changed_B$)
20. $Map_{DeletedB\_ChangedA}$ = Match($Deleted_B$, $Changed_A$)
21. $ShouldDelete_A$ = Diff($Deleted_A$, $Changed_B$, $Map_{DeletedA\_ChangedB}$)
22. $ShouldDelete_B$ = Diff($Deleted_B$, $Changed_A$, $Map_{DeletedB\_ChangedA}$)
23. Map_{GAB_SDA} = Match(GAB, $ShouldDelete_A$)
24. GABSDA = Diff(GAB, $ShouldDelete_A$, Map_{GAB_SDA})
25. Map_{GABSDA_SDB} = Match(GABSDA, $ShouldDelete_B$)
26. Final result = Diff(GABSDA, $ShouldDelete_B$, Map_{GABSDA_SDB})

Appendix F
The PROMPT Algorithm

1. The user performs setup by loading the models, A and B, and specifying some options. If the operation is merge, the result model C is initialized to be a new model with a new root and A and B as that root's children.

2. PROMPT generates an initial list of suggestions, based largely on content or syntactic information. It examines the objects, but not the structural information (i.e., the position of the objects or their participation in specific relationships as represented by relationships between the objects).

   If the operation is merge:

   a. $\forall$ pairs of objects $a \in A$ and $a \in B$ with identical names PROMPT either merges the a and b in C or removes either a or b from C.

   b. $\forall$ pairs of objects $a \in A$ and $b \in B$ with linguistically similar names a link is created between them in C (with a lower degree of confidence than if the names were identical). This means that both a and b are still in C, but PROMPT suggests that they may need to be merged by adding them to the ToDo list.

   If the operation is match and the user has tagged one model (say, A) as more general during setup, then PROMPT assumes that the objects in the less general model (say, B) should be linked in as sub-objects of the objects in A. If there is a top-level object, t, in B, with the same name as an object in A, then the two objects are merged in C. Otherwise finding a parent object for t is added to the ToDo list.

3. The user selects and performs an operation such as merging an object or resolving an item on the ToDo or Conflict lists.

4. PROMPT performs any automatic updates that it can and creates new suggestions. It has the ability to:

   a. Execute any changes automatically determined as necessary by PROMPT.

   b. Add any conflicts caused by the user's actions in step 3 to the Conflicts list
   c. Add to the ToDo list any other suggested operations or make new suggestions based on linguistic similarity or structural stability.

5. Steps 3 and 4 are repeated until the ontologies are completely merged or matched.

Vita

Rachel Pottinger finished her PhD in Computer Science and Engineering at the University of Washington in 2004. She earned her MS in Computer Science and Engineering from the University of Washington in 1999. She received her BS in Computer Science from Duke University in 1997.Her main research interests are meta-data management and data integration systems. She is a recipient of the Microsoft Research Fellowship and a National Science Foundation Fellowship.