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

A Schema Transformation Based Approach to Generic Model Management

Andrew Charles Smith

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, May 2009

Abstract

Shared databases made up of numerous heterogeneous components and used by large numbers of people are wide spread in both industry and academia. Writing programs to access and maintain these databases is a time consuming and difficult task that can take up a significant proportion of an enterprise IT manager's resources. The situation has worsened recently as new **Data Definition Languages** (**DDLs**) like XML and RDFS have come to be used. In general, solutions to these problems are specified at the data level and have to be rewritten if the schema is changed, cannot be applied to other application areas and are generally language and implementation specific.

Model Management (**MM**) is an approach that provides a way of overcoming the problems with these data level solutions. The motivation behind MM is to raise the level of abstraction in these application areas from the data level to the schema level. The key idea is to develop a set of *operators* that can be applied to schemas, and the mappings between them, as a whole rather than to individual data elements. The operators should be applicable to a wide range of problems in database management and work on schemas and mappings specified in a wide range of DDLs. Solutions to database management problems can then be specified at a high level of abstraction by combining these operators into a concise and reusable script.

A system that implements the MM operators is called a **Model Management System** (**MMS**). Two key abstractions are required for a such a system: firstly, a common language that can describe the schemas from the different DDLs, called a **Common Data Model** (**CDM**), and second, a way of describing the mappings between those schemas, called a mapping language. The main contribution of this thesis is the implementation of a MMS that uses a CDM with some unique features and adopts a new approach to the design of the mapping language based on **schema transformation**.

The CDM we use is the **Hypergraph Data Model** (**HDM**). The HDM can represent schemas from a wide range of existing DDLs and is differentiated from other CDMs by supporting a generic constraint language. We show in this thesis how this offers significant advantages particularly in the implementation of the ModelGen operator which translates schemas from one DDL to another. One of the contributions of this thesis is the addition of a type system, based on a primitive type hierarchy, which allows us to accurately materialise schemas created by ModelGen.

Our mapping language is **Both As View** (**BAV**). A BAV mapping is made up of a sequence of *bidirectional* primitive transformations that together form a **pathway** and describes precisely how instances of each **schema object** in the source schema are mapped to instances in the target schema and *vice versa*. This is an implementation of the schema transformation approach and has advantages over methods currently used in MMSs because the transformation pathways we create tell us precisely how *individual* schema objects are transformed from the source to the target schema. The work presented in this thesis takes advantage of these features to create novel implementations of the current model management operators.

Acknowledgements

I would like to thank the following people for the help they have given me while writing this thesis:

- My supervisor, Peter McBrien
- My colleagues, Nikos Rizopoulos and Duc Minh Le

Dedication

I would like to dedicate this thesis to my daughter Marie who has helped keep a smile on my face even during the most stressful times.

Contents

List of Tables

List of Figures

Acronyms

Below are a list of the acronyms used in this thesis:

BAV Both As View

CDM Common Data Model

CT Composite Transformation

CTH Common Type Hierarchy

DDL Data Definition Language

GAV Global As View

GLAV Global Local As View

HDM Hypergraph Data Model

IQL Intermediate Query Language

LAV Local As View

MM Model Management

MMS Model Management System

SO Second Order

SO tgd Second Order tuple generating dependency

s-t tgd source to target tuple generating dependency

tgd tuple generating dependency

t-s tgd target to source tuple generating dependency

Chapter 1

Introduction

1.1 Motivation

Writing programs to access and maintain large shared databases is one of the most time-consuming and difficult tasks in database management. Anecdotal evidence suggests that up to 40% of the work done in enterprise IT departments is dedicated to this task [BM07]. Examples of tasks that fall into this category include data integration, schema evolution and change propagation. Difficulties arise because shared databases are often made up of heterogeneous components that may use different **Data Definition Languages** (**DDLs**), use different object names to identify constructs that already exist in the shared database, and so on.

The question of how best to access and maintain this heterogeneous data has been highlighted as an important research area in all the recent database-research self-assessments [BBC+98, BDD+89, AAB+05]. The problem has gained an even greater significance over recent years with the rise of the use of semi-structured DDLs, such as XML, and the need to combine this data with existing relational systems [AAB+05].

Most programs written to solve these problems make use of low-level programming interfaces that provide access to individual schema objects, but provide no way of manipulating the schema as a whole. This means a lot of effort is expended writing programs that navigate around schemas to locate the objects that need to be changed even before a change can be made. This is time consuming, prone to error and has the knock-on effect that any changes to the schema mean a change to the program. A second drawback is that these programs tend to be language and application specific.

For example a program designed for relational databases using SQL to process the schema objects will be of no use if an XML document needs to be processed. Finally, programs written to address one aspect of database management, for example data integration, can seldom be used to do any other database management tasks, such as schema evolution, even though many of the basic operations such as merging schemas and extracting differences between schemas are the same.

As early as 1959 McGee [McG59] realised that identifying generic operations and making them available to programmers would greatly ease the application development task. In the 1970's this idea culminated in the pioneering work by Codd [Cod70] on the relational model and algebra. Instead of navigational access to individual records and data values, Codd suggested a set of algebraic operations on entire relations, such as selection, projection, and join. This approach freed application code from ordering, indexing, and access path dependencies. The relational algebra helped to drastically simplify the programming of data-intensive applications and has been called the single most important development in the database field [Dat95].

Now that many different DDLs are used in addition to the relational model, a further level of abstraction is needed to once again identify those generic operations that can be applied *across* DDLs. These operators need to be able to manipulate the *metadata artifacts* such as schemas and mappings between schemas as a whole, rather than the individual objects within a schema.

1.2 Model Management

A way of raising the level of abstraction in metadata intensive application areas, called **Model Management** (**MM**), was proposed by Bernstein *et al.* in [BHP00]. The key idea behind MM is to develop a set of **generic algorithmic operators** that can be used together to solve a wide range of data management problems in a DDL independent way. Each operator is designed to perform a common data management task that can be applied to a wide range of problems in the field. The initial hope was that MM operators could be applied to a range of 'models' including things like work flows and programming interfaces as well as schemas [BHP00]. More recent work has narrowed the focus somewhat [BM07]. In common with the other current MM prototypes [KQLL07, ACB05, MRB03, MBHR05], we have focused solely on schema based DDLs in this thesis.

Using MM, solutions to data management problems can be specified at a high level

of abstraction by combining the operators into a concise script which can then be applied to any particular instance of the problem. A change in the schema or even the DDL used does not necessitate any change to the script which can simply be run using the new schemas as input.

Instead of individual schema objects the MM operators work on schemas as a whole. This is in contrast to Codd's relational operators that work on individual objects and only within a *relational* schema. The original set of operators proposed by Bernstein [BHP00] has been refined [Mel04, MBHR05, BM07, BH07] and now includes operators that perform schema translation, merging, materialised view selection and view complement as well as operators to manipulate the mappings between schemas.

The challenge in MM is to design a framework in which these operators can be implemented in a DDL independent way such that the output of one task can be used as input to subsequent tasks thereby allowing the operators to be combined into a script.

It is hoped that model management will offer the same large improvement in programmer productivity in metadata intensive applications that Codd's work did in data intensive applications [Mel04]. However, only when a commercial MMS succeeds – possibly based on the work in this thesis – will we be able to determine if this claim is true.

1.3 Example

To help motivate the model management approach, consider the following database management scenario illustrated in Figure 1.1. This example is based on that given in [MBHR05] but is extended to include different DDLs.

The IT department of a large organisation designs a small database to store information about employees. In step 1 on the figure the ER design is translated into a SQL database which is materialised. Data is added to the database and it is used for a while. A request then comes from HR for a list of names and ids of employees who work in the finance department to be sent to them as an XML file. The DBA creates the necessary SQL view which is translated into XML and sent. This is shown in steps 2 and 3.

HR uses this schema but later decides it needs some additional information about employee marital status and date of birth. These attributes are added to the XML

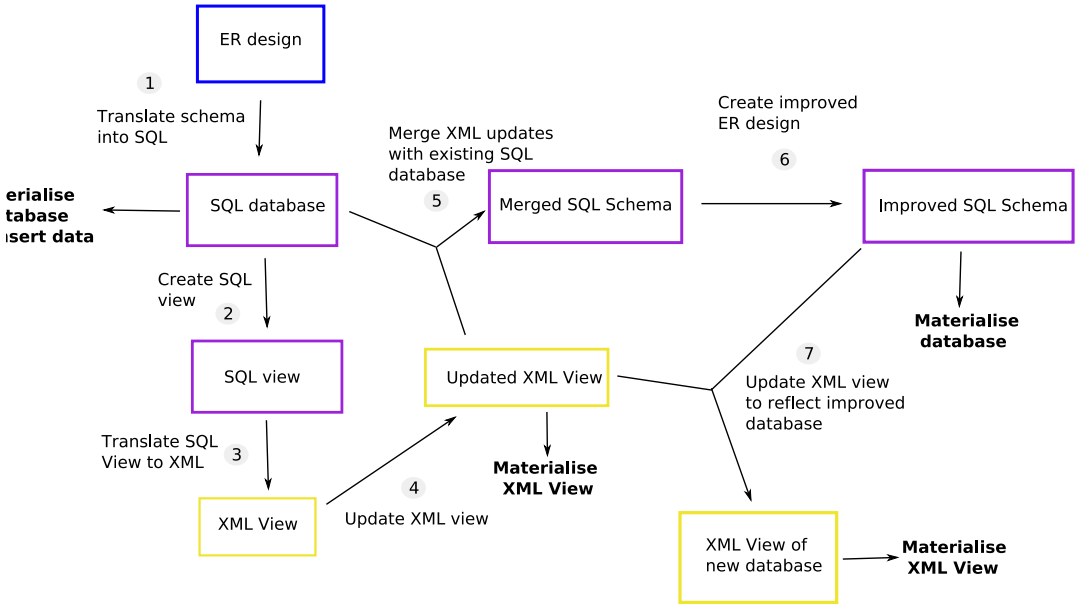


Figure 1.1: An Inter DDL Change Propagation Scenario

schema at step 4. In step 5 of the process the additions are merged with the SQL database to keep it up to date.

Before materialising the newly merged schema the database designer looks at the merged schema and realises some improvements can be made so in step 6, she updates the design and the new improved SQL schema is materialised. Finally the XML schema sent to HR is updated so that it correctly represents a view of the new database. This is shown in step 7.

As we can see there are a number of steps in this process and doing each one manually would be time consuming and prone to error. It also likely that at a later date a similar set of changes may need to be done. A degree of automation would clearly be beneficial.

We will show, as we progress through this thesis, that human involvement is only necessary in this process when design decisions need to be made, *i.e.* steps 2, 4 and 6 in Figure 1.1. The steps that do not involve design, for example the merging and schema translations, can all handled by the MM operators. This represents a significant reduction in effort for the DBA. For example she does not have to manually identify the fields added by HR to the XML view and then add them to the SQL database. This can be handled automatically by the MMS. It also means that if this process needs to be done again the same script can be run and the DBA need again only be involved in 3 steps of the process. This is in stark contrast to *ad hoc* solutions that need to be reengineered at the schema object level for each new

change.

1.4 Existing Approaches

A system that implements the operators and supports the running of scripts is called a **Model Management System** (**MMS**) [BHP00]. Two key abstractions are necessary to create the necessary framework for a MMS. Firstly a **Common Data Model** (**CDM**)[1] capable of expressing schemas from a wide range of DDLs to allow schemas from any DDL to be handled in a uniform way. Secondly we require a flexible and DDL independent **mapping language** able to describe the relationships between the schemas in the system.

The current approaches to the creation of this framework can be divided into two classes. Firstly there are those that support only **intensional** or **structural** mappings. These mappings do not provide access to the instances of the schemas they map between and so cannot be applied to data management problems such as data integration or change propagation where such access is required. Rondo [MRB03], the earliest MMS, and AutoGen [SKZ06] fall into this class.

Moda [MBHR05] falls into the second class of system. It supports **extensional** or **instance-based** semantics, allowing the manipulation of the data held in the schemas. It also implements a wide range of operators but is limited to the relational model. MIDST [ACB05] allows instance based schema translation within a framework that is extensible but so far the other operators have not been implemented. *GeRoMeSUITE* [KQLL07] is another system that provides instance based implementations of a number of operators in a DDL independent framework, but it does not support scripting or the automatic translation of schemas from one DDL to another.

1.5 A MMS based on AutoMed

The approach we propose to the creation of the framework required for a MMS is based on the AutoMed system [BKL+04]. AutoMed is a framework and software

[1] In some MM literature [Mel04] the term Universal Meta Model (UMM) is used instead of CDM. In this thesis we will use the term CDM as this is more commonly adopted in wider database literature [Dat95]

package made up of graphical tools, and a Java API, that has been used successfully to address a wide range of data management problems such as schema based data integration, schema evolution [MP02] and data warehousing [FP04]. We make use of the existing features of AutoMed to produce the first full implementation of a MMS that supports instance based mappings and a wide range of DDLs.

Bernstein and Melnik highlight a number of problem areas [BM07] that affect all the current systems that are addressed by our approach:

1. None of the CDMs currently used provide a generic constraint language which means that during schema translation specific translators must be written for constraints.
2. Compose($\circ$) is the most commonly used operator [MBHR05] but it has been shown that first order mapping languages like relational algebra, are not closed under composition [Kol05, FKPT05]. The mapping language used by a full MMS should therefore be second order.
3. No current system supports DDL independent instance based schema translations, and none return a mapping from the source schema to the target which means the results cannot be used as part of a script execution.

We now briefly introduce the AutoMed system, describing how schemas and mappings are represented in it and the particular advantages it gives us over existing approaches.

The CDM underlying AutoMed is the **Hypergraph Data Model** (**HDM**) [MP98] which provides a way of representing schemas from a wide range of DDLs, including XML, SQL, ER, ORM and UML class diagrams [BM05, MP01] in a simple hypergraph structure. Figure 1.3 shows an HDM schema that is a representation of the SQL schema shown in Figure 1.2. It is made up of three types of construct: **nodes**, represented by circles; **edges**, the lines linking the nodes and **constraints**, the symbols contained in the dashed grey boxes. Each object in the source schema is represented in the system as a combination of instances of these three constructs.

An **instance** of an HDM schema is a structure that includes a function that maps nodes and edges in the schema to values in the domain of discourse of the data source the HDM schema is representing. These values are called the **extent** of the node or edge. Example 1.1 shows an instance of the schema in Figure 1.3[2].

[2]schema objects in BAV operations are surrounded by double chevrons

Emp

eid	name	dept
1	Peter Smith	100
21	Susan Brown	101

Figure 1.2: An instance of the SQL schema S_{eg}

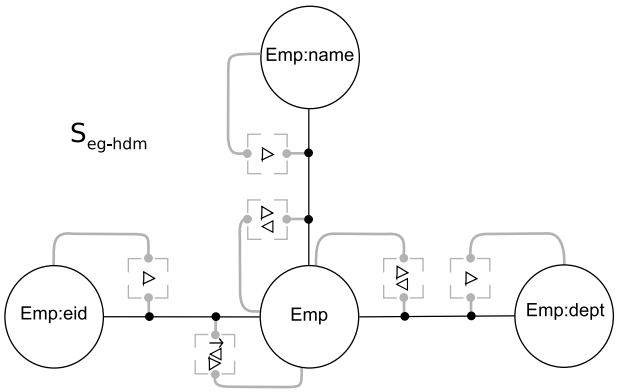


Figure 1.3: HDM representation of a SQL table

Example 1.1 Original Schema Instance

$Ext_{S_{eg-hdm},I_1}(\text{node:}\langle\langle\text{Emp}\rangle\rangle) = \{(1),(21)\}$
$Ext_{S_{eg-hdm},I_1}(\text{node:}\langle\langle\text{Emp:eid}\rangle\rangle) = \{(1),(21)\}$
$Ext_{S_{eg-hdm},I_1}(\text{edge:}\langle\langle\_,\text{Emp},\text{Emp:eid}\rangle\rangle) = \{(1,1),(21,21)\}$
$Ext_{S_{eg-hdm},I_1}(\text{node:}\langle\langle\text{Emp:name}\rangle\rangle) = \{(\text{`Peter Smith'}),(\text{`Susan Brown'})\}$
$Ext_{S_{eg-hdm},I_1}(\text{edge:}\langle\langle\_,\text{Emp},\text{Emp:name}\rangle\rangle) = \{(1,\text{`Peter Smith'}),(21,\text{`Susan Brown'})\}$
$Ext_{S_{eg-hdm},I_1}(\text{node:}\langle\langle\text{Emp:dept}\rangle\rangle) = \{(100),(101)\}$
$Ext_{S_{eg-hdm},I_1}(\text{edge:}\langle\langle\_,\text{Emp},\text{Emp:dept}\rangle\rangle) = \{(1,100),(21,101)\}$

□

The constraint operators, shown in dashed boxes in the figure, make up a language capable of representing the constraints of high level DDLs in a general way, and so address the first open issue raised above. The constraint operators are used to restrict the allowable extents of the nodes and edges they connect. Three are shown in the figure: mandatory ($\triangleleft$), unique ($\triangleright$) and reflexive ($\rightarrow$). The mandatory constraint ensures that any value in the extent of a node is also in the extent of the edge. Unique ensures that a value in the extent of a node appears at most once in the extent of the edge. Reflexive ensures that if a value appears in the extent of the node there must be an identity tuple containing it in the extent of the edge. This constraint language is capable of expressing constraints in a DDL independent way thereby addressing the first drawback of current systems mentioned at the beginning of this section.

In common with other MMSs [MBHR05, KQLL07] we use a high level declarative language to describe our mappings. The AutoMed mapping language is called **Both-As-View** (**BAV**) and is an implementation of the **schema transformation** [MP98] approach that is at the core of our system. Indeed Bernstein suggested in 2007 [BM07] that the techniques described in [MP98] could form the basis for further work in MM.

Schema transformation is the process of transforming a schema by applying a primitive transformation that adds or removes a single schema object thereby creating a new schema. Sequences of these transformations can be used to perform complex transformations. This is a new approach to MM that offers two specific advantages over existing systems. Firstly, the primitive transformations tell us exactly how an object in a target schema was derived from the source schema. This is in contrast to query based approaches where the semantics associated with a particular schema object in a query may not be obvious. The fine grained nature of the transformations allows us to split the processing of a schema into small steps that are easier to process. Secondly, the mappings and resultant schema are created at the same time whereas in existing approaches this requires two separate steps. This is useful when implementing MM operators that require both a mapping and schema as a result.

In AutoMed this sequence of transformations is called a **BAV transformation pathway**. It describes in detail how a source schema, along with its data instances, is transformed into a target schema. At each step, one primitive transformation is applied to the current schema to create a new schema that differs from the old one by one schema object. Each primitive BAV operation is automatically *reversible* allowing us to invert a transformation pathway if required.

Associated with each primitive transformation, is a query expressing the extent of the new or deleted schema object in terms of elements in the current schema. The queries can be posed in any query language, for example XQuery or SQL. These DDL specific languages, however, are not suitable for a MMS. In our implementation the queries are posed in the functional list comprehension language **Intermediate Query Language** (**IQL**) [Pou01, JTMP03], which supports higher order functions and can express queries in a DDL independent manner. The fact that we can use second order queries in our mappings makes them closed for composition [FKPT05] and addresses the second drawback of existing systems mentioned at the beginning of this section.

In this thesis BAV pathways are always made up of a **growth phase**, in which target

schema objects are added to the source schema and a **shrinking phase**, in which the original source schema objects are removed. In the growth phase the queries in the transformations express the extent of the target objects the transformations are creating in terms of source objects and thus provide a **Global As View** (**GAV**)-like interpretation of the target schema. In the shrinking phase the extents of the source objects are expressed in terms of target objects, providing a **Local As View** (**LAV**)-like interpretation. In the implementation presented in this thesis we only support GAV query reformulation.

In summary, the HDM and BAV combine to give us a very accurate and fine-grained description of the schemas and mappings to use in our MMS. The reversible transformation pathways that form the mappings describe not only how the source schema is transformed into a result schema and *vise versa*, but also include the query rewriting rules necessary to populate these schemas. In particular the benefits of the AutoMed approach are:

- The HDM has been shown to be capable of accurately representing many high level DDLs [BM05]. It also includes a constraint language capable of expressing constraints from these high level DDLs in a DDL independent way.
- BAV pathways provide us with a mapping and transformation language that allows us to describe mappings between many DDLs more fully and accurately than previous approaches.
- The IQL allows us to create the second order queries necessary for the implementation of Compose.
- The detailed information contained in each transformation in a BAV pathway helps us to implement the MM operators.
- BAV transformation pathways are bidirectional allowing us to invert them automatically.

NewEmp

eid	name
1	Peter Smith

Figure 1.4: An instance of the SQL schema S'_{eg}

As an example of how we implement MM operators in AutoMed, consider the following BAV pathway fragment that is the growth phase of a pathway that transforms S_{eg} into the schema with instance S'_{eg} shown in Figure 1.4.

① add(table:⟨⟨NewEmp⟩⟩, [{e} | {e} ← table:⟨⟨Emp⟩⟩; e = 1])
② add(column:⟨⟨NewEmp, eid, int, notnull⟩⟩, [{e, e} | {e, e} ← column:⟨⟨Emp, eid⟩⟩; e = 1])
③ add(column:⟨⟨NewEmp, name, varchar, notnull⟩⟩, [{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩; e = 1])

We can use the MM operator Extract to return only the portion of a schema that participates in a mapping, and a mapping from the original schema to the extracted schema. The schema objects from S_{eg} that participate in the mapping are all those that are used to create objects in S'_{eg}. In AutoMed we can work out what these are by analysing the queries in the transformations that add objects to S'_{eg}. For example, Transformation ② uses column:⟨⟨Emp, eid⟩⟩ to define the extent of column:⟨⟨NewEmp, eid⟩⟩ in S'_{eg}, so we know this needs to be in the result schema of Extract. Similarly for table:⟨⟨Emp⟩⟩ and column:⟨⟨Emp, name⟩⟩. In contrast we can see that column:⟨⟨Emp, dept⟩⟩ is not used in any of the queries. It should therefore not be in the result schema. The only instances of the source schema objects that participate in the mapping are those that meet the conditions of the queries in the add transformations. These are the instances we include, giving us the result schema shown in Figure 1.5. The pathway that creates this table is shown below:

④ add(table:⟨⟨extract_Emp⟩⟩, [{e} | {e} ← table:⟨⟨Emp⟩⟩; e = 1])
⑤ add(column:⟨⟨extract_Emp, eid, int, notnull⟩⟩, [{e, e} | {e, e} ← column:⟨⟨Emp, eid⟩⟩; e = 1])
⑥ add(column:⟨⟨extract_Emp, name, varchar, notnull⟩⟩,
[{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩; e = 1])

Emp

eid	name
1	Peter Smith

Figure 1.5: An instance of result schema produced by Extract

We can see how we take advantage of the fine grained nature of BAV transformations to easily see exactly which objects in the source schema participate in the mapping. We also take advantage of the underlying HDM representation to implement the operators in a DDL independent way.

1.6 Contributions

The main contribution of this thesis is the first full implementation of a MMS that supports instance based mappings and implements all the operators proposed by Bernstein [BM07], excluding Match, in a DDL independent way. Match has been excluded because one of our colleagues in the AutoMed group is working independently on this problem [RM05] and we hope to integrate that work with

what we describe in this thesis in the future. The specific contributions made with regards to MMSs are:

- Algorithms to implement instance based versions of the currently defined MM operators [BM07] excluding Match. This addresses the third drawback of existing systems mentioned in Section 1.4.
- A DDL independent algorithm for ModelGen [SM08a, SM08b] that returns a mapping between the source and target schemas, and allows automatic translation between any two DDLs supported by AutoMed. This addresses the final drawback in Section 1.4.
- Implementation of the MM operators, excluding Match, as part of the AutoMed API.
- The addition of a type system to the HDM and a method for translating primitive type information between different DDLs in our MMS, using a novel approach based on a common type hierarchy. This allows us to create more accurate materialised target schemas [SM06].
- New AutoMed wrappers for XML Schema and RDFS.
- A method for translating **Second Order source to target tuple generating dependencies** (**SO s-t tgds**) into executable BAV pathways.

1.7 Outline

The rest of the thesis is laid out as follows. In Chapter 2 we provide some more detailed background on Model Management. We describe the abstractions necessary for implementing a MMS and an example of the widely studied declarative mapping language that we use as a basis for our mapping language, SO s-t tgds. We give the instance based semantics definitions of the current MM operators and we end with brief descriptions of the existing MMS prototypes.

Chapter 3 introduces the existing AutoMed system. We describe the HDM and the BAV mapping and transformation language in detail and give a very brief description of IQL. We go on to show how high level DDLs are represented in the HDM, using SQL, ER, XML Schema and RDFS as examples. The representations of XML Schema and RDFS described in this chapter are new contributions to AutoMed

made by this thesis. We also describe how equivalence preserving composite BAV transformations can be used to restructure HDM schemas. We end the chapter by presenting our method for creating BAV mappings that have the same expressive power as SO s-t tgds.

Chapter 4 presents an addition we have made to AutoMed in the form of a common type hierarchy that allows us to translate primitive data type information between DDLs. We describe how this type hierarchy can be used to spot some data type translation errors and also reduce the need for run time checking of data if a target schema is materialised. The work described in this chapter was published in [SM06].

In Chapter 5 we describe our implementation of the ModelGen operator using a novel approach that can be applied to schemas expressed in a wide range of DDLs. The translation is done via the HDM and makes extensive use of the HDM's unique constraint language. We introduce two algorithms, the first of which matches HDM schema objects with the HDM representation of target DDL constructs and the second transforms any HDM objects that do not match using the composite transformations introduced in Chapter 3, into ones that more closely match the target constructs. Preconditions are placed on the choice of composite transformation to guide the process. The work presented in this chapter was published in [SM08a, SM08b]. We also applied the technique described here to P2P systems in [LSM07, LSM08].

Chapter 6 presents our implementation of the remaining MM operators, excluding Match, within our framework. We make extensive use of the information stored in each of the primitive BAV transformations that make up the input mappings in the implementation of each operator. The underlying HDM representation of all the objects in AutoMed means that our implementations are DDL independent. At the end of the chapter we give a brief description of the MM API we have developed as an extension to the existing AutoMed API, and how to use the to write MM programs. A demonstration of the work in this chapter was published in [SRM08].

In Chapter 7 we describe how MM scripts can be used to solve problems in a number of different domains. Specifically we show in detail how the example introduced in Section 1.3 can be implemented using our system. We illustrate this work with some screen shots from AutoMedas well as the BAV pathways and schemas returned by our operators.

Finally Chapter 8 offers a summary of the work presented in this thesis and some ideas for future work.

Chapter 2

Model Management Systems

In this chapter we discuss the basic requirements for a **Model Management System** (**MMS**). We describe the abstractions necessary to implement a MMS as well as the instance based semantics of the MM operators. We finish the chapter by describing the existing MMSs as described in the literature.

The term Model Management System was coined by Bernstein [BHP00]. It describes a system capable of solving a wide range of data management problems by the application of operators that take schemas and the mappings between them as parameters. Examples of these operators include Merge, an operator that merges two schemas that are linked by a mapping into a single schema, and ModelGen, that translates schemas from one DDL into another DDL. This schema level approach is in contrast to previous solutions where operations are performed on individual schema objects.

In [BHP00] the basic requirements for a MMS are set out as:

- A mechanism for representing schemas from a wide range of **data description languages** (**DDLs**), and storing these representations. This mechanism should be simple, yet expressive enough to fully describe schemas from many different DDLs.
- An appropriate representation for mappings between schemas that is DDL independent. Again this should be expressive enough to describe mappings between many different DDLs.
- Efficient implementations of the MM operators.

- A mechanism for combining these operators into a script or program.

Meeting these requirements allows solutions to metadata management problems to be specified at a high level of abstraction in the form of a script, which can be used to solve the problem regardless of the specific schemas or DDLs involved. These operators can be said to provide a *schema* manipulation language in the same way that SQL and relational algebra provide a *data* manipulation language.

2.1 Schemas

The DDL used for a particular task may depend of a number of factors, such as the data source, schema designer preference, company policy, *etc.* Indeed a number of different DDLs may be used within a single project. The first of the two key abstractions necessary for a MMS is thus a way of representing the **schemas** from all DDLs we wish to support in the system. To overcome the heterogeneity, a **Common Data Model** (**CDM**) [SL90], capable of describing the constructs of all the different DDLs is required.

The design of a CDM falls into two main categories: graphical models and text based models. Graphical solutions are a popular choice as they provide a visual representation of the schema, making them easier to understand. We will discuss each approach along with other issues of importance to CDM design, with the help of a CDM used in a current MMS.

The next decision to make is what type of constructs to include in the CDM. One approach is to create a complex, high-level CDM that includes all the constructs in the DDLs used in the MMS. MIDST is an example of this approach [ACB06]. The alternative approach is to use combinations of a set of simple constructs to create any constructs from the high level DDL. This more flexible approach is the one most commonly adopted, both in MMSs and in systems that map between specific pairs of modelling languages [MZ98, MRB03, ACM02, CR03, BD03, BM05].

Another important feature that the CDM needs is a way of expressing the constraint constructs such as cardinality and integrity constraints from the DDLs in a MMS.

We now define what we mean by a schema and an **instance** of that schema. There are a number of different and seemingly contradictory ways of doing this described in the literature. For example, in some of the literature a schema is defined as a set

of instances [MBHR05, BS81]. However, in this thesis we will use the *unnamed and logic programming* perspective described in [AHV95], where a schema is defined as a non-empty set of relation names. In Definition 2.1 we use the term **schema object** instead of relation in keeping with the way schemas have been described up to this point in AutoMed. We choose the logic programming perspective because much of the work described here has to do with mappings that are frequently described in the literature using this method [BM07, FKP05, MIR94].

Definition 2.1 Schema

A schema is a structure of the form $S = \langle ExtensionalObject, \Sigma \rangle$ where *ExtensionalObject* is a set of extensional **schema objects** and Σ is a set of schema objects that constrain instances of the schema.

Given the following sets of strings: *Labels* for naming objects, *DDLNames* for naming DDLs and *ConstructNames* for naming constructs in a DDL, a schema object, *i.e.* the elements of both *ExtensionalObject* and Σ are defined as follows:

$$Schema{:}\mathsf{DDL}{:}\mathsf{construct}{:}\langle\langle \mathsf{se}_1, \ldots, \mathsf{se}_n \rangle\rangle$$

where

- *Schema* is the schema this schema object is part of
- $\mathtt{DDL} \in DDLNames$ and $\mathsf{construct} \in ConstructNames$ are the names of the DDL and construct of this object
- $\mathsf{se_i} \in SchemeElement$ for $1 \leq i \leq n$, is the *scheme* of the schema object.
- $SchemeElement = ExtensionalObject \cup Labels$. A *SchemeElement* can be an existing member of *ExtensionalObject* or a string from *Labels*. The ordering of the scheme elements is significant.

Differentiating between *Labels* and *ExtensionalObject* allows us to identify atomic schema objects as those which have just elements of *Labels* in the scheme and non-atomic as those which have elements of both *ExtensionalObject* and *Labels*.

The **key scheme** of a schema object includes the names of any extensional objects that it references and a single identifying label. In general we will use only the key scheme of a schema object in mappings and queries.

We use *Schema*, DDL and construct to disambiguate schema objects. If these are not necessary for the disambiguation we will leave them out.

□

Using the notation of Definition 2.1, the SQL schema shown in Figure 2.1 is expressed as: $S_1 = \langle ExtensionalObject, \Sigma \rangle$ where

$ExtensionalObject$ = {table:⟪Emp⟫, column:⟪table:⟪Emp⟫, eid, notnull⟫,
column:⟪table:⟪Emp⟫, name, notnull⟫, column:⟪table:⟪Emp⟫, dept, notnull⟫,
table:⟪Dept⟫, column:⟪table:⟪Dept⟫, did, notnull⟫,
column:⟪table:⟪Dept⟫, dname, notnull⟫,
column:⟪table:⟪Dept⟫, numEmps, null⟫} and
Σ = {primary_key:⟪Emp_pk, table:⟪Emp⟫, eid⟫,
foreign_key:⟪Dept_fk, table:⟪Emp⟫, dept, table:⟪Dept⟫, did⟫,
primary_key:⟪Dept_pk, table:⟪Dept⟫, did⟫}

Consider column:⟪table:⟪Emp⟫, eid, notnull⟫. The scheme of the object is made of the existing schema object table:⟪Emp⟫, the string eid that names the column and the string notnull that specifies that this is a variant of column that should not accept null values. The key scheme of this object is column:⟪table:⟪Emp⟫, eid⟫.

If an existing schema object that is part of a scheme only has a single scheme element, as is the case here for table:⟪Emp⟫, we can simplify the notation and include only the schema object name. We can thus denote the schema object above more simply as column:⟪Emp, eid, notnull⟫. We will apply this simplification in the rest of this thesis.

An **instance** of a schema is defined as follows:

Definition 2.2 Schema Instance

Let S be a schema and $\{\langle\langle \mathsf{so_1} \rangle\rangle, \ldots, \langle\langle \mathsf{so_n} \rangle\rangle\}$ the set of extensional schema objects and Σ be the set of constraint objects in S. An instance, k, of S is a structure for which there exists a function

$$Ext_{S,k}(\langle\langle \mathsf{so_i} \rangle\rangle) \rightarrow \mathcal{P}(Seq(Vals)) \; 1 \leq i \leq n$$

where $\mathcal{P}$ denotes power set and $Seq(Vals)$ is a tuple of values in the Universe of Discourse we wish to describe with the schema S.

Emp

eid	name	dept
1	Peter Smith	100
21	Susan Brown	101

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15

Emp.dept → Dept.did

(a) $\mathsf{Inst}_1(S_{emp})$

Emp

eid	name	dept
5	Joe Brown	100

Dept

did	dname	numEmps?
100	Finance	23

Emp.dept → Dept.did

(b) $\mathsf{Inst}_2(S_{emp})$

Figure 2.1: Two instances of S_{emp}

We now define an instance, $\mathsf{Inst}_k(S)$, of schema, S as:

$$\mathsf{Inst}_k(S) = \{Ext_{S,k}(\langle\langle \mathsf{so_1} \rangle\rangle) \cup \ldots \cup Ext_{S,k}(\langle\langle \mathsf{so_n} \rangle\rangle)\}$$

where $Ext_{S,k}(\langle\langle \mathsf{so_1} \rangle\rangle), \ldots, Ext_{S,k}(\langle\langle \mathsf{so_n} \rangle\rangle)$ satisfy all the constraints in Σ. Each schema object in Σ represents a boolean expression which has as variables the extensional schema objects referenced in its scheme. These expressions must evaluate to true for the constraint to be satisfied.

When writing $\mathsf{Inst}_k(S)$ we will generally include the schema object as a prefix to its extent tuple to allow us to identify the object that the tuple in the extent of the schema is associated with. When writing the extent of a single schema object we do not do this.

We define the set of all possible instances, $\mathsf{AllInst}(S)$, of S as:

$\mathsf{AllInst}(S) = \mathsf{Inst}_1(S) \cup \mathsf{Inst}_2(S) \cup \ldots$

We assume set based semantics for our schemas.

□

When writing an instance of a schema object that references other schema objects in the way that an SQL table references its columns, it is common practice to simply write a predicate whose variables represent the referenced schema objects [AHV95, FKPT05], rather than writing instances of each object in the schema. For example, consider S_{emp} in Figure 2.1, the schema used to store some of the details of the employees in the organisation we introduced in the introduction. We could write a predicate representing the Emp table as $\mathsf{table}{:}\langle\langle \mathsf{Emp} \rangle\rangle(e, n, d)$ where the variables represent the extents of the columns eid, name and dept respectively. A particular

instance of the schema then grounds the variables in these predicates. For example $\mathsf{Inst}_1(S_{emp})$ shown in Figure 2.1 (a) may be written in this way as:

$\mathsf{Inst}_1(S_{emp})$ = {table:⟪Emp⟫(1,'Peter Smith',100),
table:⟪Emp⟫(2,'Susan Brown',101),
table:⟪Dept⟫(100,'Finance',23),table:⟪Dept⟫(101,'HR',15)}

Note that the above representation of the schema instance is not ambiguous as there is a total order over the columns of table:⟪Emp⟫ and table:⟪Dept⟫. We will use this notation for the rest of this chapter.

Any subset of $\mathsf{Inst}_1(S_{emp})$ is also an instance of the schema as long as the values in the subset still satisfy the constraints. So,

$\mathsf{Inst}'_1(S_{emp})$ = {table:⟪Emp⟫(2,'Susan Brown',101), table:⟪Dept⟫(101,'HR',15)}.

is a valid instance of S_{emp} but

{table:⟪Emp⟫(2,'Susan Brown',101),table:⟪Dept⟫(100,'Finance',23)}

is not since it violates the constraint foreign_key:⟪Dept_fk, Emp, dept, Dept, did⟫.

We call a schema along with its current instance a **data source**.

2.2 Mappings

For most high level DDLs, at least one language for describing the relationship between two schemas and their instances has been proposed. In MM these relationships are called **mappings** and the language used to describe them, a **mapping language**. Examples of mapping languages are relational algebra, used with the relational model, and XQuery [BCF+03] used with XML. Using such a DDL specific mapping language in a MMS has obvious limitations. We could not, for example, use relational algebra to define a mapping between a relational schema and an XML schema.

One approach to the design of an inter DDL mapping language often adopted in data integration systems designed to combine data from two specific DDLs is to manipulate specific high level DDLs directly using their native transformation languages. Numerous examples of these systems exist in the literature for the integration of XML and relational schemas [FTS00, SSB+01, BFH+03, CKS+00] where SQL is

used to manipulate the relational schemas and a language like XQuery, the XML schemas. In the more general setting of a MMS this has the obvious disadvantage of not scaling well if more DDLs are added to the MMS.

A more common approach in MM is to use a declarative mapping language that can represent mappings between different DDLs at a high level, abstracting away the implementation details [Mel04, BHP00]. It is the approach adopted by all the current MMSs [KQLL07, MBHR05, MRB03].

A mapping can be viewed as a set of logical dependencies between the schemas it connects [AHV95]. **Extensional** or **instance based** mappings describe the relationship between *instances* of the source schema and *instances* of the target schema. All the mappings we use in this thesis will be extensional.

Definition 2.3 Mapping

We define a **mapping**, map_{S_1,S_2}, to be a tuple of the form:

$$(S_1, S_2, \Sigma_{S_1,S_2})$$

where S_1 is the source schema, S_2 is the target schema, and Σ_{S_1,S_2} is a set of constraints that instances of the mapping must satisfy.

□

Mappings between schemas have been studied in great detail in the context of data integration [Len02] and data exchange [FKMP05]. In both cases it is necessary to create mappings between source and target schemas. The target schema(s) may be virtual as is the case in data integration or materialised as is the case in data exchange.

Two main approaches have been taken to creating mappings between source and target schemas [Len02]. We describe each method briefly below.

Global As View (GAV) The GAV approach is based on the idea that each object in the target (global) schema can be expressed in terms of a view over the source (local) schemas, so when a new object is added to the global schema its extent is defined in terms of objects in the local schemas. Mappings of this type tell us explicitly how to retrieve the data when evaluating a query over an object in the global schema. In most GAV systems query answering is based

on a simple unfolding strategy. This explicit mapping does, however, restrict the design of the global schema.

The GAV approach is most useful when integrating a stable set of sources. Adding a new source or extending one that is already part of the system can mean rewriting existing mappings. Examples of systems adopting the GAV approach are TSIMMIS [GMPQ+97] and Garlic [RAH+96].

Local As View (LAV) In the LAV approach, the content of each local schema is expressed in terms of a view over the global schema. If a new local schema object is added to the existing global schema its extent is expressed in terms of objects in the global schema. This can make it easier to design a good global or target schema because the design can be made independently of the sources. It can, however, make queries over the global schema harder to evaluate [Ull97, Hal01]. LAV works best if there is a stable, well-established global schema. It is easier to add more source schemas to systems based on a LAV approach as the mappings for each component schema are independent of each other. There is no danger that a new mapping assertion will affect the existing system.

Information Manifold [Ull97] is an example of the LAV approach in operation. It expresses the global schema in terms of a Description Logic and uses conjunctive queries to retrieve data from the component schemas. A number of XML-based systems also adopt this approach.

In a MM setting more powerful mapping methods are needed. In general creating a target schema from a given source schema requires the mapping of conjunctive queries on the source schema to conjunctive queries on the target schema, something neither GAV nor LAV can do.

Global local as view (GLAV) [FLM99] uses both GAV and LAV mappings to define the relationship between the global and local schemas. Conjunctions are allowed on both sides of a GLAV rule. The GLAV approach has been widely adopted [BH07, Kol05, KQLJ07] as the mapping language in current work on MM. In particular, s-t tgds [FKMP05] which we describe below, have been used to provide a logical characterisation of the GLAV mappings necessary in a MMS.

To describe how the instances of a source schema are mapped to the target schema we define an *instance* of a mapping as follows:

Definition 2.4 Mapping instance

Given $map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$, an instance of the mapping, $\mathsf{MapInst}_k(map_{S_1,S_2})$, is a set of tuples, $\langle x, y \rangle$, where $x \in \mathsf{AllInst}(S_1)$ and $y \in \mathsf{AllInst}(S_2)$, that satisfy the constraints in Σ_{S_1,S_2}.

We define all the instances of a mapping, $\mathsf{AllMapInst}(map_{S_1,S_2})$, to be *all* sets of tuples, $\langle x, y \rangle$, where $x \in \mathsf{AllInst}(S_1)$ and $y \in \mathsf{AllInst}(S_2)$, that satisfy the constraints in Σ_{S_1,S_2}. □

It is important to note that S_1 and S_2 may have their own sets of constraints, so an instance of the mapping needs to satisfy the constraints on the source and target schemas as well as those in the mapping.

We can make the following statements about map_{S_1,S_2}. If Σ_{S_1,S_2} is empty the mapping is unconstrained. For example, a mapping between two unrelated databases such as a bank customer database and a library catalogue would result in an empty set of constraints. On the other hand if $\mathsf{AllMapInst}(map_{S_1,S_2}) = \emptyset$ the constraint set contains contradictory elements.

In a MMS, the target schema may be generated, by executing the mapping or a MM operator, or it may be an existing schema. We see examples of both of these in the scenario described in the introduction. Examples of a target schema being generated can be found in step 2 where the XML view is created and in step 5, where schemas are merged to create a new schema. An example of a schema that already exists can be found in step 4 where the HR department created a mapping from the existing XML view to the new HR XML Schema. This new XML Schema may have existing constraints that need to be satisfied and may have data in it that is not derived from the source schema.

2.2.1 Second Order s-t tgds - A Mapping Language for MM

Second Order source to target tuple generating dependencies (**SO s-t tgds**) are a declarative way of describing mappings between schemas and have recently emerged as the mapping language of choice in MMSs [BM07, Kol05, BH07] and have been adopted as a way of describing mappings in recent MMS prototypes [KQLJ07, MBHR05]. This declarative approach is particularly useful in a MMS because mappings between a number of different DDLs may be needed and

it allows the implementation details of the individual DDLs to be abstracted away. Before defining SO s-t tgds we review previous approaches.

SO s-t tgds are based on the **tuple generating dependencies** (**tgds**) described in [AHV95]. A tgd is a first-order formula of the form:

$$\forall \vec{x}.\varphi(\vec{x}) \rightarrow \exists \vec{y}.\psi(\vec{x},\vec{y})$$

such that $\varphi(\vec{x})$ is a conjunction of atomic schema objects and constants and $\psi(\vec{x},\vec{y})$ is a conjunction of atomic schema objects and constants with variables in $\vec{x}$ and $\vec{y}$, where $\vec{y}$ may be empty. The instance tuples of the atoms in $\psi(\vec{x},\vec{y})$ are 'generated' by the instance tuples and relations in $\varphi(\vec{x})$, hence the name tuple *generating* dependency.

Fagin et al [FKMP05] extended the use of tgds to describe mappings between two different schemas. Tgds that specifically describe the relationship between a source and target schema are called *source-to-target* tuple-generating dependencies (s-t tgds).

Definition 2.5 s-t tgds and t-s tgds

Let S_1 be the source schema and S_2 be the target schema, such that S_1 and S_2 have no atoms in common. A tgd

$$\forall \vec{x}.\varphi_{S_1}(\vec{x}) \rightarrow \exists \vec{y}.\psi_{S_2}(\vec{x},\vec{y}),$$

is said to be *source-to-target (s-t)* if $\varphi_{S_1}(\vec{x})$ is restricted to use only atoms of S_1 and constants, and $\psi_{S_2}(\vec{x},\vec{y})$ is restricted to use only atoms of S_2 and constants. It is *target to source(t-s)* if S_1 is the target schema and S_2 is the source schema. □

The following are some examples of the different types of simple s-t tgds that we might find in the constraint set of a mapping $map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$, and correspond to Figure 2.2. In both examples schemas d_1 and d_2 consist of a single schema object, $\mathsf{table}{:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d)$. In this and subsequent examples in this section we will not add the construct name to the schema object name as all the constructs we refer to are SQL tables. We also assume here that an SQL table is an atomic object as this is the interpretation used in the literature about embedded dependencies and s-t tgds [AHV95, MIR94, FKMP05]. We show in the next chapter that AutoMed allows us to manipulate a table's columns and constraints separately.

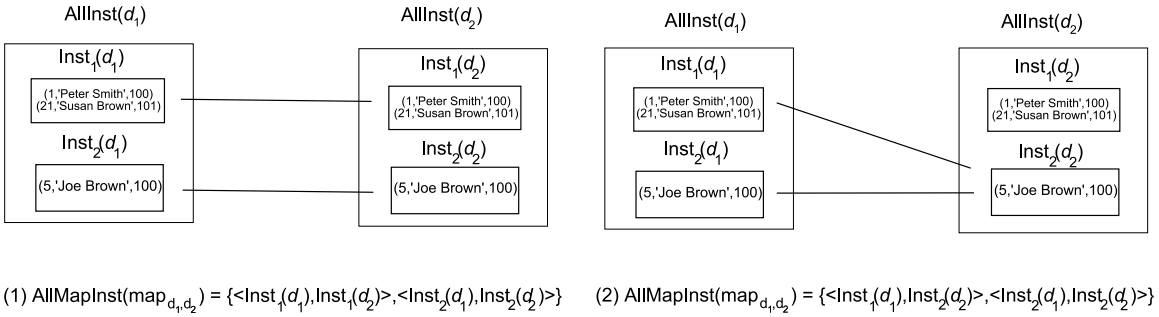


Figure 2.2: Mapping examples

1. $\{\forall e, n, d.(d_1 ::: \langle\langle \mathsf{Emp} \rangle\rangle(e, n, d) \rightarrow d_2 ::: \langle\langle \mathsf{Emp} \rangle\rangle(e, n, d))\}$ - Mapping between identical schemas or d_2 is a superset of d_1.

2. $\{\forall e, n, d.(d_1 ::: \langle\langle \mathsf{Emp} \rangle\rangle(e, n, d) \rightarrow d_2 ::: \langle\langle \mathsf{Emp} \rangle\rangle(\texttt{5,Joe Brown,100}))\}$ - Maps all instances of the source schema to a single instance of the target schema.

Example 2.1 Mapping example

The finance department in our organisation decide they need to have a separate database of all their employees. We will call the schema for this database, S_{finEmp}. Two instances are shown in Figure 2.3. They wish to keep their database synchronised with the main employee database so to allow this they create a bidirectional mapping between S_{emp} in Figure 2.1, and S_{finEmp}.

FinEmp

eid	name
1	Peter Smith

FinDept

did	numEmps
100	23

(a) $\mathsf{Inst}_1(S_{finEmp})$

FinEmp

eid	name
5	Joe Brown

FinDept

did	numEmps
100	23

(b) $\mathsf{Inst}_2(S_{finEmp})$

Figure 2.3: Two instances of S_{finEmp}

The following mapping describes the relationship between instances of S_{emp} and S_{finEmp}. Here, and in all the following equations we will assume the universal quantifiers at the beginning of each s-t tgd are present.

$map_{S_{emp},S_{finEmp}} = (S_{emp}, S_{finEmp}, \Sigma_{S_{emp},S_{finEmp}})$ where $\Sigma_{S_{emp},S_{finEmp}} =$

$$\{S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle(d,\text{'}\mathtt{Finance}\text{'},ne) \rightarrow$$
$$S_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \wedge S_{finEmp}{:::}\langle\langle \mathsf{FinDept}\rangle\rangle(d,ne),$$
$$S_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \wedge S_{finEmp}{:::}\langle\langle \mathsf{FinDept}\rangle\rangle(d,ne) \rightarrow$$
$$S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle(d,\text{'}\mathtt{Finance}\text{'},ne)\}$$

If the instances of S_{emp} are those shown in Figure 2.1 and those of S_{finEmp} are those shown in Figure 2.3 then:

$\mathsf{AllMapInst}(map_{S_{emp},S_{finEmp}}) =$
$\{\langle \mathsf{Inst}_1(S_{emp}), \mathsf{Inst}_1(S_{finEmp})\rangle, \langle \mathsf{Inst}_2(S_{emp}), \mathsf{Inst}_2(S_{finEmp})\rangle\}$ □

Each atom on the RHS of the arrow in the first tgd in $map_{S_{emp},S_{finEmp}}$ can be fully derived from the LHS and no extensional quantifiers are required. This is an example of a *full* tgd and we write it as follows:

$$\forall \vec{\mathrm{x}}.\varphi(\vec{\mathrm{x}}) \rightarrow \psi(\vec{\mathrm{x}}) \tag{2.1}$$

Every full tgd can be rewritten as a finite set of full tgds with a single atom on its right hand side using the Lloyd-Topor transformations [Llo87, FKMP05]. Equation 2.1 can thus be rewritten as shown below:

$$\forall \vec{\mathrm{x}}.\varphi(\vec{\mathrm{x}}) \rightarrow \wedge_{i=1}^{k} R_i(\vec{\mathrm{x}_\mathrm{i}}) \equiv \forall \vec{\mathrm{x}}.\varphi(\vec{\mathrm{x}}) \rightarrow R_i(\vec{\mathrm{x}_\mathrm{i}}) \text{ for } i = 1 \text{ to } k \tag{2.2}$$

This is often a convenient way of writing tgds. Dependencies with a single schema object on the RHS are called *single-head* tgds. Those with more than one schema object on the RHS are called *multi-head*.

It follows from this result that the first tgd, which is an s-t tgd, in Example 2.1 can be rewritten to give us the following set of single head full s-t tgds.

$$\{S_{emp}{:::}\langle\langle \mathsf{Emp}(e,n,d)\rangle\rangle \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}(d,\text{'}\mathtt{Finance}\text{'},ne)\rangle\rangle \rightarrow S_{finEmp}{:::}\langle\langle \mathsf{FinEmp}(e,n)\rangle\rangle,$$
$$S_{emp}{:::}\langle\langle \mathsf{Emp}(e,n,d)\rangle\rangle \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}(d,\text{'}\mathtt{Finance}\text{'},ne)\rangle\rangle \rightarrow S_{finEmp}{:::}\langle\langle \mathsf{FinDept}(d,ne)\rangle\rangle\}$$

The DBA in the finance department has a look at S_{finEmp} and notices that $\mathsf{table}{:}\langle\langle \mathsf{FinDept}\rangle\rangle$ is redundant as all the values are the same and it is not even linked to $\mathsf{table}{:}\langle\langle \mathsf{FinEmp}\rangle\rangle$ by a foreign key. He creates a new schema, S'_{finEmp}, that includes

only table:⟪FinEmp⟫ and updates the mapping accordingly to create $map_{S_{emp},S'_{finEmp}}$

$$(S_{emp}, S'_{finEmp}, \{S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle(d,\text{'Finance'},ne) \rightarrow$$
$$S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n),$$
$$S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow$$
$$\exists d, ne(S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle(d,\text{'Finance'},ne))\})$$

Consider the second tgd in the mapping above which is a t-s tgd. There is an important difference between this tgd and the ones we have been discussing, in that this tgd requires the existential quantifiers $\exists en$ and $\exists d$, on the RHS. This is an example of a tgd that is not full. It is important to note that multi-head tgds that are not full, such as this one, *cannot* always be rewritten as a conjunction of single-head tgds because the existential quantifiers cannot be guaranteed to be consistent across the single-head tgds generated. In our example, if we rewrote the second tgd in the mapping as we have the first we would get

$$\{S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow \exists d(S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d)),$$
$$S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow \exists d, ne(S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle(d,\text{'Finance'},ne))\}$$

If we now added a new member of staff ⟪FinEmp⟫(3,'Paul Jones') we could instantiate these tgds as follows:

$$\{\langle\{S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle\texttt{(3,'Paul Jones')}\},\{S_{emp}{:::}\langle\langle \mathsf{Emp}\rangle\rangle\texttt{(3,'Paul Jones',100)}\}\rangle,$$
$$\langle\{S'_{finEmp}{:::}\langle\langle \mathsf{FinEmp}\rangle\rangle\texttt{(3,'Paul Jones')}\},\{S_{emp}{:::}\langle\langle \mathsf{Dept}\rangle\rangle\texttt{(101,'HR',15)}\}\rangle\}$$

These two tgds do not correctly reflect the fact that, in the original mapping, d is consistent across all the terms of the RHS of the formula.

Creating implications with a single atomic schema object on the RHS is an important step in composing mappings [KQLJ07, FKPT05]. In particular Fagin et al [FKPT05] showed that problems like this make it impossible to correctly compose mappings containing non-full tgds. This is a major problem for a MMS as Compose is one of the most commonly used MM operators [BM07, MBHR05]. Their investigation of the problem led them to introduce SO tgds, a class of existential *second order* formulae. These are tgds extended with existentially quantified function symbols that range over the entire equation. To create an SO tgd we Skolemise any existentially quantified variables on the RHS of a non-full tgd. They are defined as follows:

Definition 2.6 SO s-t tgds and SO t-s tgds

An SO s-t tgd is a formula of the form:

$$\exists\vec{\mathrm{f}}(\forall\vec{\mathrm{x}_1}.\varphi_1 \rightarrow \psi_1 \wedge \cdots \wedge \forall\vec{\mathrm{x}_\mathrm{n}}.\varphi_n \rightarrow \psi_n)$$

where $\vec{\mathrm{f}}$ is a collection of functions symbols. Each $\vec{\mathrm{x}_\mathrm{i}}$ is a vector of variables in the source schema. Each φ_i is a conjunction of schema object predicates and/or equalities over constants and variables defined in the source schema. Each ψ_i is a conjunction of schema object predicates, constants and variables defined in the source schema, and the function symbols in $\vec{\mathrm{f}}$. In a SO t-s tgd the source and target schema are switched.

□

It was shown in [FKPT05] that every tgd is equivalent to an SO tgd where each existentially quantified variable is translated into a Skolem function. We can therefore rewrite the t-s tgd in the example above as follows:

$$\begin{aligned}&\exists dept, numEmp(S'_{finEmp}:::\langle\langle\mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow \\ &\quad S_{emp}:::\langle\langle\mathsf{Emp}\rangle\rangle(e,n,dept(e)) \wedge S_{emp}:::\langle\langle\mathsf{Dept}\rangle\rangle(d,\texttt{'Finance'},numEmp(e)) \wedge d = dept(e))\end{aligned} \quad (2.3)$$

where *dept* is a Skolem function that creates a unique value for the department id column based on their employee id and *numEmp* is a separate Skolem function that creates different unique values for the numEmps column. Note that we can include the Skolem function as part of the definition of the object or as a separate term.

It is also shown in the same paper that every SO tgd is equivalent to an SO tgd in a 'normal form' where the right-hand sides of all the implications, *i.e.* the ψ_i are atomic formulae, as is required by existing composition algorithms [KQLJ07, FKPT05]. We can thus rewrite Equation 2.3 as:

$$\begin{aligned}&\exists dept, numEmp(S'_{finEmp}:::\langle\langle\mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow S_{emp}:::\langle\langle\mathsf{Emp}\rangle\rangle(e,n,dept(e)) \wedge \\ &S'_{finEmp}:::\langle\langle\mathsf{FinEmp}\rangle\rangle(e,n) \rightarrow S_{emp}:::\langle\langle\mathsf{Dept}\rangle\rangle(dept(e),\texttt{'Finance'},numEmp(dept(e))))\end{aligned}$$

As part of this rewriting, any Skolem functions or constants that appear as terms in the original formula are included in the head of the object(s) that use them. In this example $dept(e)$ has been included in the head of $S_{emp}:::\langle\langle\mathsf{Dept}\rangle\rangle$.

Let us again consider the example of mapping $S'_{finEmp}:::\langle\langle\mathsf{FinEmp}\rangle\rangle$(3,'Paul Jones') back into S_{emp}, but this time using the SO t-s tgd we have just created. We use *dept* to create a unique value for the did column based on the employee id, and similarly use *numEmp* to create unique values for the numEmps column. In this case the parameter for the Skolem function is the value created for the department id, as the

number of employees is functionally dependent on the department id. We will call these unique values `uv1` and `uv2`. We therefore have:

$\mathsf{AllMapInst}(map_{S'_{finEmp},S_{emp}}) =$
$\{\langle\{S'_{finEmp}:::\langle\!\langle\mathsf{FinEmp}\rangle\!\rangle\texttt{(3,'Paul Jones')}\},\{S_{emp}:::\langle\!\langle\mathsf{Emp}\rangle\!\rangle\texttt{(3,'Paul Jones',uv1)}\}\rangle,$
$\langle\{S'_{finEmp}:::\langle\!\langle\mathsf{FinEmp}\rangle\!\rangle\texttt{(3,'Paul Jones')}\},\{S_{emp}:::\langle\!\langle\mathsf{Dept}\rangle\!\rangle\texttt{(uv1,'Finance',uv2)}\}\rangle\}$

This allows us to identify `Paul Jones` as a member of `Finance`. This is in contrast to first order non-full tgds where, as we saw above, information is lost if we rewrite them as a conjunction of formulae with one term on the RHS.

2.3 MM Operators

Given a common representation for schemas from the various DDLs in our MMS and a way of describing mapping between them, we are now ready to define operators that work on these two abstractions. In this section we describe the MM operators most commonly cited in the literature [BM07, MBHR05, Mel04, BH07]. These operators cover well known but disparate problems in data management. They have typically been studied in isolation with respect to specific languages. The definitions here distill the essential aspects of these problems into language independent operators that can be used to create scripts and programs.

The operators are shown in Table 2.3 as method calls. The presentation of the operators in this thesis takes a more programmatic approach to the definition of the operators than previous descriptions. We feel this helps to make MM look more like a programming language for metadata rather than a collection of operators.

MM Operator Signature
$\mathsf{Mapping}\, map_{S_1,S_3} = \mathsf{Compose}(\mathsf{Mapping}\, map_{S_1,S_2}, \mathsf{Mapping}\, map_{S_2,S_3})$
$\mathsf{Mapping}\, map_{S_1,S_2} = \mathsf{Confluence}(\mathsf{Mapping}\, map_{S_1,S_2}, \mathsf{Mapping}\, map_{S_1,S_2})$
$\mathsf{Mapping}\, map_{S_1,S_2} = \mathsf{Match}(\mathsf{Schema}\, S_1, \mathsf{Schema}\, S_2)$
$\langle\mathsf{Schema}\, S_m, \mathsf{Mapping}\, map_{S_m,S_1}, \mathsf{Mapping}\, map_{S_m,S_2}\rangle =$
$\quad\mathsf{Merge}(\mathsf{Schema}\, S_1, \mathsf{Schema}\, S_2, \mathsf{Mapping}\, map_{S_1,S_2})$
$\langle\mathsf{Schema}\, S_d, \mathsf{Mapping}\, map_{S_1,S_d}\rangle = \mathsf{Diff}(\mathsf{Schema}\, S_1, \mathsf{Mapping}\, map_{S_1,S_2})$
$\langle\mathsf{Schema}\, S_x, \mathsf{Mapping}\, map_{S_1,S_x}\rangle = \mathsf{Extract}(\mathsf{Schema}\, S_1, \mathsf{Mapping}\, map_{S_1,S_2})$
$\langle\mathsf{Schema}\, S_t, \mathsf{Mapping}\, map_{S_1,S_t}\rangle = \mathsf{ModelGen}(\mathsf{Schema}\, S_1, \mathsf{TargetDDL}\ \ l)$
$\mathsf{ExecutableMapping}\, map_{S_1,S_2} =$
$\quad\mathsf{TransGen}(\mathsf{Mapping}\, map_{S_1,S_2}, \mathsf{ExecutableMappingLanguage}\, l)$

Table 2.1: The MM Operators

Before describing the main model management operators it is necessary to define four auxiliary operators: Invert, Id, Domain, Range.

Definition 2.7 Auxiliary Model Management Operators

Let S_1 and S_2 be schemas and $map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$ be a mapping between them.

- $\mathsf{Domain}(\mathsf{Mapping}\, map_{S_1,S_2}) = \{x \mid \exists y.\langle x, y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2})\}$
- $\mathsf{Range}(\mathsf{Mapping}\, map_{S_1,S_2}) = \{y \mid \exists x.\langle x, y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2})\}$
- $\mathsf{Invert}(\mathsf{Mapping}\, map_{S_1,S_2}) = \{\langle y, x\rangle \mid \langle x, y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2})\}$
- $\mathsf{Id}(\mathsf{Schema}\, S_1) = \{\langle x, x\rangle \mid x \in \mathsf{AllInst}(S_1)\}$

□

Example 2.2 shows an example of each of these auxiliary operators.

Example 2.2 Auxiliary operators

Assume that we remove table:⟪Dept⟫ from S_{emp} in the previous examples to leave us with S'_{emp} containing table:⟪Emp⟫ and its columns and key. Assume also that the instances of table:⟪Emp⟫ shown in Figure 2.1 remain and we add the following new instance:

$\mathsf{Inst}_3(S'_{emp}) = \{$⟪Emp⟫`(14,'Catherine Thomas',100)`,⟪Emp⟫`(15,'Neal Fischer',100)`$\}$

We define the following mapping

$(S'_{emp}, S'_{finEmp}, \{S'_{emp}{:::}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,100) \rightarrow S'_{finEmp}{:::}\langle\langle\mathsf{FinEmp}\rangle\rangle(e,n)\})$

The following are instances of this mapping:

$\mathsf{MapInst}_1(map_{S'_{emp},S'_{finEmp}}) =$
$\langle\{S'_{emp}{:::}$⟪Emp⟫`(1,'Peter Smith',100)`$\}$,$\{S'_{finEmp}{:::}$⟪FinEmp⟫`(1,'Peter Smith')`$\}\rangle$
$\mathsf{MapInst}_2(map_{S'_{emp},S'_{finEmp}}) =$
$\langle\{S'_{emp}{:::}$⟪Emp⟫`(5,'Joe Brown',100)`$\}$,$\{S'_{finEmp}{:::}$⟪FinEmp⟫`(5,'Joe Brown')`$\}\rangle$
$\mathsf{MapInst}_3(map_{S'_{emp},S'_{finEmp}}) =$
$\langle\{S'_{emp}{:::}$⟪Emp⟫`(14,'Catherine Thomas',100)`$, S'_{emp}{:::}$⟪Emp⟫`(15,'Neal Fischer',100)`$\}$,
$\{S'_{finEmp}{:::}$⟪FinEmp⟫`(14,'Catherine Thomas')`$, S'_{finEmp}{:::}$⟪FinEmp⟫`(15,'Neal Fischer')`$\}\rangle$

The Domain and Range of the mapping are as follows:

$\mathsf{Domain}(map_{S'_{emp},S'_{finEmp}}) = \{\{\langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(1,'Peter Smith',100)}\},\{\langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(5,'Joe Brown',100)}\},$
$\{\langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(14,'Catherine Thomas',100)}, \langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(15,'Neal Fischer',100)}\}\}$

$\mathsf{Range}(map_{S'_{emp},S'_{finEmp}}) = \{\{\langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(1,'Peter Smith')}\},\{\langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(5,'Joe Brown')}\},$
$\{\langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(14,'Catherine Thomas')}, \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(15,'Neal Fischer')}\}\}$

The instances of the mappings returned by the Invert and Id operators are as follows:

$\mathsf{AllMapInst}(\mathsf{Invert}(map_{S'_{emp},S'_{finEmp}})) \supseteq \{$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(1,'Peter Smith')}\},\{S'_{emp} ::: \langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(1,'Peter Smith',100)}\}\rangle,$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(5,'Joe Brown')}\},\{S'_{emp} ::: \langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(5,'Joe Brown',100)}\}\rangle,$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(14,'Catherine Thomas')}, S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(15,'Neal Fischer')}\},$
$\{S'_{emp} ::: \langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(14,'Catherine Thomas',100)}, S'_{emp} ::: \langle\!\langle \mathsf{Emp} \rangle\!\rangle \texttt{(15,'Neal Fischer',100)}\}\rangle\}$

$\mathsf{AllMapInst}(\mathsf{Id}(S'_{finEmp})) \supseteq \{$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(1,'Peter Smith')}\},\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(1,'Peter Smith')}\}\rangle,$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(5,'Joe Brown')}\},\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(5,'Joe Brown')}\}\rangle,$
$\langle\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(14,'Catherine Thomas')}, S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(15,'Neal Fischer')}\},$
$\{S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(14,'Catherine Thomas')}, S'_{finEmp} ::: \langle\!\langle \mathsf{FinEmp} \rangle\!\rangle \texttt{(15,'Neal Fischer')}\}\rangle\}$

□

We will now discuss each of the other operators. It is important to note that the definitions of Merge, Diff and Extract given later in this section capture only the *necessary* conditions on the operators. They do not require the operators to produce unique or minimal solutions. Definitions for minimal solutions for these operators are given in [Mel04]; however, it has been shown [MBHR05, BM07] that these conditions can lead to seemingly correct solutions being rejected and make the materialisation of solutions impossible in some cases even when a useful, non-minimal solution exits. Our emphasis in this thesis is on the efficient implementation of the operators within our framework rather than their theoretical characteristics, so we will not include the minimality conditions in our definitions and will not try to verify minimality when we come to describe our implementations of these operators in Chapter 6.

2.3.1 Compose

It is often necessary to manipulate mappings directly by composing or combining them. The MM operator Compose provides this functionality. It is defined as follows [Mel04]:

Definition 2.8 Compose, $\circ$

Given two mappings map_{S_1,S_2} and map_{S_2,S_3}, map_{S_1,S_3} is a composition if

$$\mathsf{AllMapInst}(map_{S_1,S_3}) = \mathsf{AllMapInst}(map_{S_1,S_2}) \circ \mathsf{AllMapInst}(map_{S_2,S_3})$$

which means that
$\mathsf{AllMapInst}(map_{S_1,S_3}) =$
$\{\langle \mathsf{Inst}_i(S_1), \mathsf{Inst}_k(S_3)\rangle \mid \exists \mathsf{Inst}_j(S_2)$ s. t. $\langle \mathsf{Inst}_i(S_1), \mathsf{Inst}_j(S_2)\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2})$
and $\langle \mathsf{Inst}_j(S_2), \mathsf{Inst}_k(S_3)\rangle \in \mathsf{AllMapInst}(map_{S_2,S_3})\}$

□

Note that this definition of Compose does not provide a semantic mapping. For example assume we have two objects that use different names for the same semantic concept in S_1 and S_3. Unless these two objects are both linked to the same object in S_2 by the mappings map_{S_1,S_2} and map_{S_2,S_3} we assume they are distinct.

Work on composing mappings, which is the most commonly used MM operator [BH07, BM07], led to the introduction of SO s-t tgds. It was shown in [FKPT05] that composition is not closed under first order s-t tdgs. We use Figure 2.4 to describe the results about composition of s-t tgds:

1. If the constraints in both map_{S_1,S_2} and map_{S_2,S_3} are sets of *full* s-t tgds then the constraints in map_{S_1,S_3} are also definable as a set of full s-t tgds. This is shown in Figure 2.4(a) where the composition of $\langle\{\mathsf{Inst}_1(S_1)\},\{\mathsf{Inst}_1(S_2)\}\rangle$ and $\langle\mathsf{Inst}_1(S_2),\mathsf{Inst}_1(S_3)\rangle$ is $\langle\mathsf{Inst}_1(S_1),\mathsf{Inst}_1(S_3)\rangle$ and $\langle\{\mathsf{Inst}_2(S_1)\},\{\mathsf{Inst}_2(S_2)\}\rangle$ and $\langle\mathsf{Inst}_2(S_2),\mathsf{Inst}_2(S_3)\rangle$ is $\langle\mathsf{Inst}_2(S_1),\mathsf{Inst}_2(S_3)\rangle$.

2. If the constraints in map_{S_1,S_2} are full s-t tgds and the constraints in map_{S_2,S_3} are s-t tgds (not necessarily full) then the constraints in map_{S_1,S_3} are definable as a set of s-t tgds that are not necessarily full. This is shown in Figure 2.4(b). $\mathsf{Inst}_2(S_1)$ is mapped to two instances of S_3' each of which have different values for the existentially quantified variable. $\mathsf{AllMapInst}(map_{S_1,S_3'})$ reflects this by mapping $\mathsf{Inst}_2(S_1)$ to both $\mathsf{Inst}_2(S_3')$ and $\mathsf{Inst}_3(S_3')$ using the same existentially quantified variable as was used in $map_{S_2,S_3'}$.

3. Finally, if the constraints in map_{S_1,S_2} and map_{S_2,S_3} are sets of arbitrary s-t tgds (not necessarily full) then the constraints in map_{S_1,S_3} may not be definable in any first order language. This is shown in Figure 2.4 (c) where the tgds that define the instances of $map_{S_1,S_2'}$ include an existentially quantified variable.

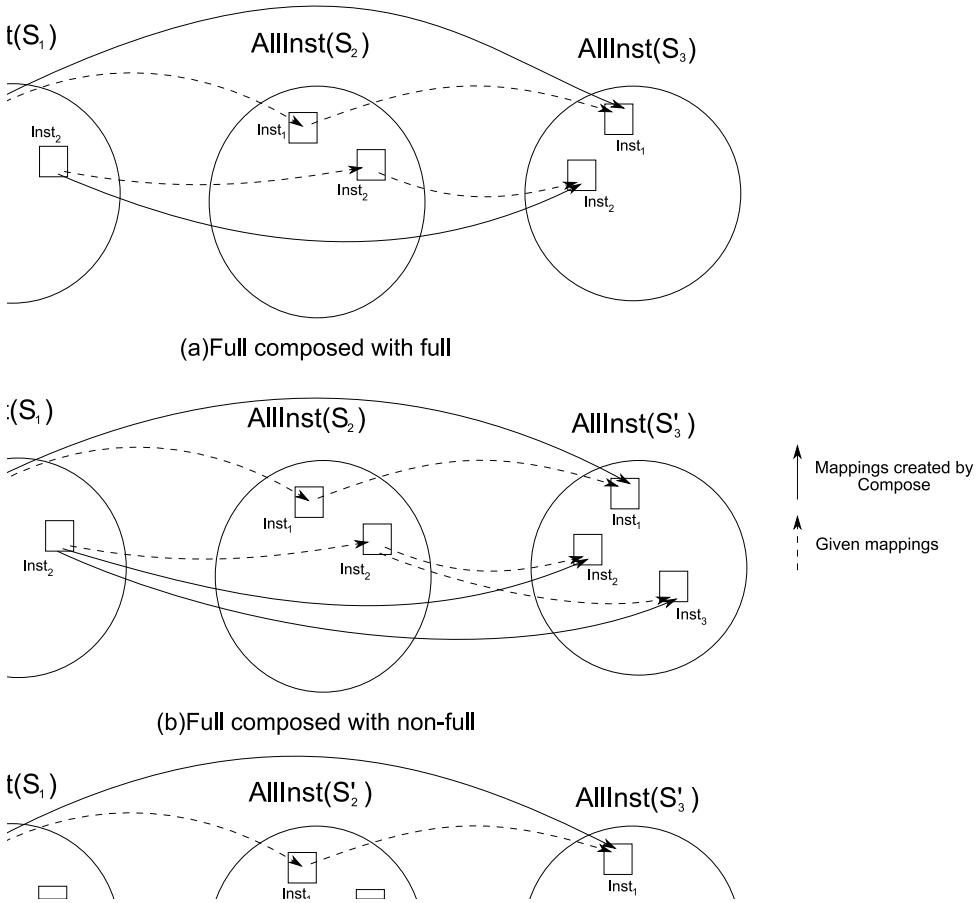


Figure 2.4: Compose

$\mathsf{Inst}_2(S'_2)$ and $\mathsf{Inst}_3(S'_2)$ have different values for this variable. They are then mapped to instances of S'_3. map_{S_1,S'_3} cannot differentiate between the two instances of S'_3 because the existentially quantified variable refers to values in S'_2 and not in S'_3.

We resolve this situation by using the SO tgds introduced in the previous section. They allow us to relate S_1 to S_2 using a Skolem function based on some value in S_1 that is also present in S_3. We will discuss this in more detail when we present our implementation of Compose in Chapter 6.

2.3.2 Confluence

In complex change propagation scenarios it may be the case that changes are made to a schema by a number of different agents, be they users or automatic processes, that need to be propagated. These may result in mappings between two given schemas via different intermediate schemas. Confluence can 'unify' pairs of these mappings

The original definition given in [MBHR05] assumes a first order mapping language with no Skolem functions. As we have described above, our mapping language can make use of Skolem functions. To take this into account we adjust the definition of Confluence to deal with following situation. If one of the input mappings generates a Skolem value for a given target object, and the corresponding transformation in the other mapping maps a data value to the target object, then the domain of the objects in input mappings is the same but the range is different. In one case the Skolem value and in the other the data value. The original definition leads us to reject this mapping instance because the values from the two input mappings conflict. We, however, include this instance using the actual values, as this allows us to create a more complete mapping.

Definition 2.9 Confluence, $\oplus$

Given two mappings map_{S_1,S_2} and map'_{S_1,S_2}, we define confluence as:

$$
\begin{aligned}
map_{S_1,S_2} \oplus map'_{S_1,S_2} \leftarrow\ & (\mathsf{AllMapInst}(map_{S_1,S_2}) \cap \mathsf{AllMapInst}(map'_{S_1,S_2})) \\
& \cup \{\langle x,y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2}) \mid x \notin \mathsf{Domain}(map'_{S_1,S_2}) \wedge y \notin \mathsf{Range}(map'_{S_1,S_2})\} \\
& \cup \{\langle x,y\rangle \in \mathsf{AllMapInst}(map'_{S_1,S_2}) \mid x \notin \mathsf{Domain}(map_{S_1,S_2}) \wedge y \notin \mathsf{Range}(map_{S_1,S_2})\} \\
& \cup \{\langle x,y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2}) \mid x \in \mathsf{Domain}(map'_{S_1,S_2}) \wedge \\
& \qquad \exists z \ such\ that\ \langle x,z\rangle \in \mathsf{AllMapInst}(map'_{S_1,S_2}) \wedge ContainsSkolem(z)\} \\
& \cup \{\langle x,y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2}) \mid y \in \mathsf{Range}(map'_{S_1,S_2}) \wedge \\
& \qquad \exists z \ such\ that\ \langle z,y\rangle \in \mathsf{AllMapInst}(map'_{S_1,S_2}) \wedge ContainsSkolem(z)\} \\
& \cup \{\langle x,y\rangle \in \mathsf{Inst}(map'_{S_1,S_2}) \mid x \in \mathsf{Domain}(map_{S_1,S_2}) \wedge \\
& \qquad \exists z \ such\ that\ \langle x,z\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2}) \wedge ContainsSkolem(z)\} \\
& \cup \{\langle x,y\rangle \in \mathsf{AllMapInst}(map'_{S_1,S_2}) \mid y \in \mathsf{Range}(map_{S_1,S_2}) \wedge \\
& \qquad \exists z \ such\ that\ \langle z,y\rangle \in \mathsf{AllMapInst}(map_{S_1,S_2}) \wedge ContainsSkolem(z)\}
\end{aligned}
$$

where the function $ContainsSkolem(z)$ returns true if z contains any Skolem values.

□

Confluence extracts the submapping on which map_{S_1,S_2} and map'_{S_1,S_2} agree and adds to it the correspondences between all those instances of S_1 and S_2 that participate either only in map_{S_1,S_2} or map'_{S_1,S_2}.

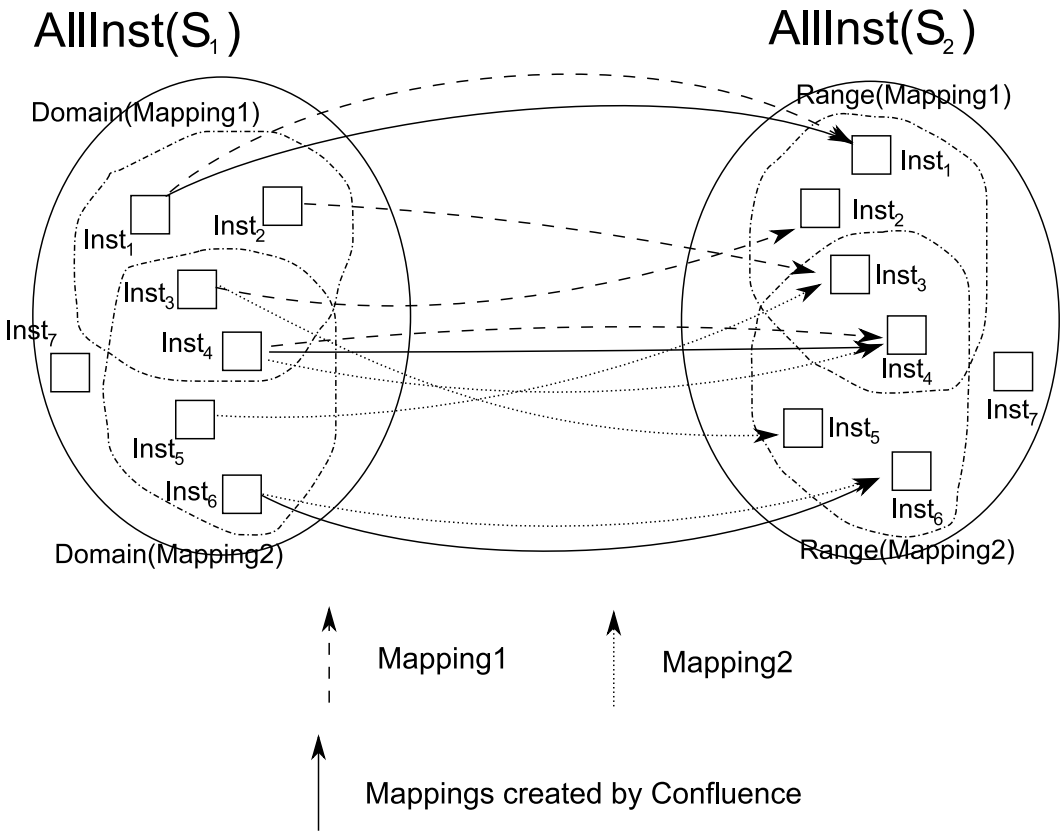


Figure 2.5: Confluence

Figure 2.5 shows two mappings between S_1 and S_2 and the mapping generated by confluence. $\langle\{\mathsf{Inst}_4(S_1)\},\{\mathsf{Inst}_4(S_2)\}\rangle$ is an example of a mapping instance that both mappings agree on. $\langle\{\mathsf{Inst}_1(S_1)\},\{\mathsf{Inst}_1(S_2)\}\rangle$ participates in Mapping1 but not in Mapping2 so is in the confluence mapping. Similarly $\langle\{\mathsf{Inst}_6(S_1)\},\{\mathsf{Inst}_6(S_2)\}\rangle$ participates only in Mapping2. Mapping1 maps $\mathsf{Inst}_2(S_1)$ to $\mathsf{Inst}_3(S_2)$ while Mapping2 maps $\mathsf{Inst}_5(S_1)$ to $\mathsf{Inst}_3(S_2)$, *i.e.* the domains differ but the ranges are the same so we do not include these instances in the confluence mapping. Similarly $\mathsf{Inst}_3(S_1)$ is mapped to two different instances in S_2 so is not included either. Confluence will thus create a mapping with the following instances:

$$\mathsf{AllMapInst}(mapconf_{S_1,S_2}) \supseteq \{\langle\{\mathsf{Inst}_1(S_1)\},\{\mathsf{Inst}_1(S_2)\}\rangle, \langle\{\mathsf{Inst}_6(S_1)\},\{\mathsf{Inst}_6(S_2)\}\rangle, \langle\{\mathsf{Inst}_4(S_1)\},\{\mathsf{Inst}_4(S_2)\}\rangle\ \}$$

If the mappings that are inputs to Confluence are both total and surjective then $map_{S_1,S_2} \oplus map'_{S_1,S_2} = \mathsf{AllMapInst}(map_{S_1,S_2}) \cap \mathsf{AllMapInst}(map'_{S_1,S_2})$.

2.3.3 Match

Applications such as data and schema integration require mappings be created between schemas based on certain criteria. The operator Match takes two schemas as input and returns a set of mappings that holds between the two schemas.

The Match operator is written as follows:

Mapping map_{S_1,S_2} = Match(Schema S_1, Schema S_2) where S_1 and S_2 are the schemas to be matched and map_{S_1,S_2} is the set of mappings between them.

Match differs from the other model management operators in that it has no formal semantics. It returns mappings between schemas in a particular application context. Sometimes these can be discovered semi-automatically [RB01, RM05] but it is always necessary to have some human involvement in the process. It may also be the case that the mappings returned by Match are partial or even inaccurate. Match is not discussed in detail here but work in this area using the methods described in this thesis can be found in [RM05]. We hope to integrate this work with what is described in this thesis in future.

2.3.4 Merge

Merging is a component of two specific problems in data management: view integration and data integration [BLN86]. In view integration the inputs are views of a single schema that is the merged schema. In this case the integration can be driven by mappings between the two views that describes the **overlap** [PB08] between them. In database integration the data sources are independent.

The inputs to the operator defined below addresses the problem of view integration. The first requirement is that all the information from the two input schemas is retained in the merged schema. Secondly, it should be possible to recreate the input schemas from the merged schema. The definition below captures these desiderata:

Definition 2.10 Merge

Let $map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$ be a mapping between S_1 and S_2.

$\langle$Schema S_m, Mapping map_{S_m,S_1}, Mapping $map_{S_m,S_2}\rangle$ =
Merge(Schema S_1, Schema S_2, Mapping map_{S_1,S_2}) holds if and only if:

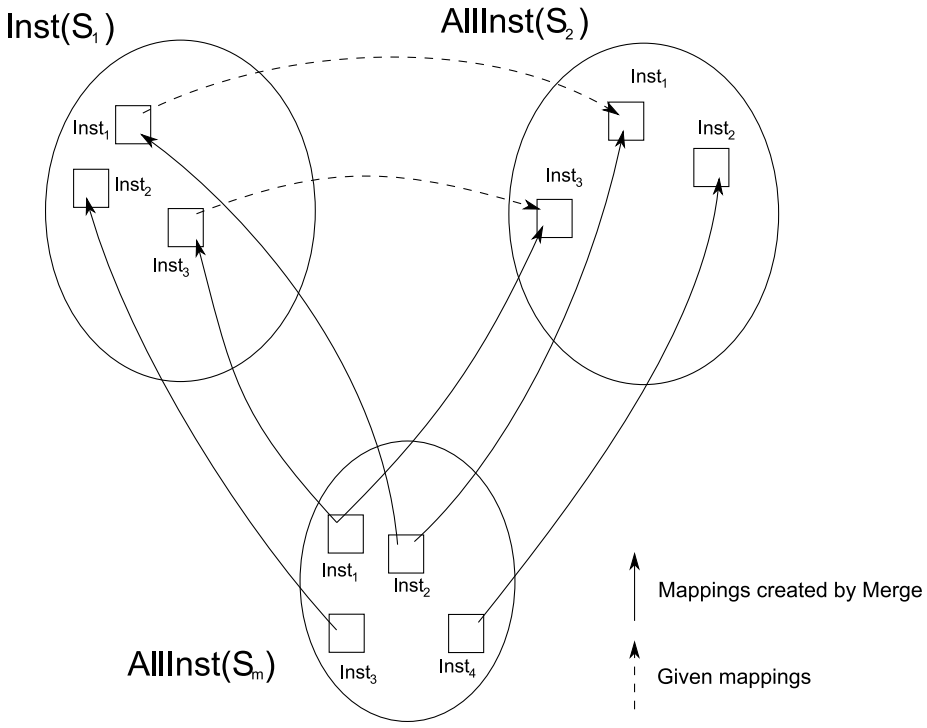


Figure 2.6: Merge

1. map_{S_m,S_1} and map_{S_m,S_2} are (possibly partial) surjective mappings onto S_1 and S_2 respectively, where a surjective mapping is one where every instance of S_1 can be derived from S_m using the mapping map_{S_m,S_1} and similarly for S_2.

2. $map_{S_1,S_2} = \mathsf{Invert}(map_{S_m,S_1}) \circ map_{S_m,S_2}$.

3. $S_m = \mathsf{Domain}(map_{S_m,S_1}) \cup \mathsf{Domain}(map_{S_m,S_2})$.

□

Condition 1 ensures that S_m contains all the information in S_1 and S_2. Condition 2 guarantees that the input mapping can be reconstructed. In other words we can recreate mutually consistent instances of S_1 and S_2. Condition 3 ensures that any instance in the merged schema must be in one of the two mappings we have created.

We see these conditions illustrated in Figure 2.6. In the figure the arrows represent mappings. The start of arrows are the domain of the mapping while the arrow heads are the range. We can see that S_m contains all the instances of S_1 and S_2. Composing the inverse of map_{S_m,S_1} with map_{S_m,S_2} gives us $\{\langle \mathsf{Inst}_1(S_1), \mathsf{Inst}_1(S_2)\rangle, \langle \mathsf{Inst}_3(S_1), \mathsf{Inst}_3(S_2)\rangle\}$ which matches $\mathsf{AllMapInst}(map_{S_1,S_2})$. Finally, each instance of S_m is in the domain of either map_{S_m,S_1} or map_{S_m,S_2}.

The semantics described above are similar to those for view integration, where all the instances in the two input schemas can be related via a mapping, *i.e.* the mapping is complete. These are the semantics we will follow here as they are those given by Melnik and Bernstein [Mel04, MBHR05] in the MM proposal for the Merge operator. Pottinger et al [PB03, PB08] and Gubanov et al [GBM08] provide algorithms for schema merge where the mapping is not complete; this, however, falls outside the scope of what we will discuss here.

2.3.5 Extract

It is often necessary to identify which parts of a source schema have counterparts in the target schema if for example we wish to use part of the source schema in a data warehouse. An operator used to solve this problem should return a subschema of S_1 that participates in map_{S_1,S_2} *i.e.* those parts of S_1 that have a counterpart in S_2.

Finding a subschema of S_1 that can still have map_{S_1,S_2} applied to it is a simplified version of the problem of materialised view selection [ACN00, LBU01, CHS02]. The objective of materialised view selection is to find a set of views that allows us to answer a given query workload. Extract, on the other hand, returns a single view of S_1 that can still be used to answer the specific query map_{S_1,S_2}.

The problem is also related to query rewriting using views [Hal01] where we rewrite the query over S_x, which behaves as the view, rather than S_1, which behaves as the database the view is of.

The DDL independent nature of Extract makes it difficult to create optimal solutions as is the case when SQL or subsets thereof are the mapping language [ACN00]. In [LBU01] it is argued that selecting views that are minimal to a given set of views but not with respect to all views is a good solution, in other words a non-minimal Extract result is still useful.

Definition 2.11 Extract

Let map_{S_1,S_2} be a mapping from S_1 to S_2.

$\langle \mathsf{Schema}\ S_x, \mathsf{Mapping}\ map_{S_1,S_x} \rangle = \mathsf{Extract}(\mathsf{Schema}\ s_1, \mathsf{Mapping}\ map_{S_1,S_2})$ holds if and only if:

1. $map_{S_1,S_x} \circ \mathsf{Invert}(map_{S_1,S_x}) \circ map_{S_1,S_2} = map_{S_1,S_2}$
2. $\mathsf{AllInst}(S_x) = \mathsf{Range}(map_{S_1,S_x})$.

□

Condition 1 ensures that we can get the same result applying map_{S_1,S_x} to S_1 as we do applying map_{S_1,S_2} to S_1 and condition 2 that the only instances in S_x are those that take part in the resultant mapping, map_{S_1,S_x}.

As we said at the beginning of this section, the conditions in this definition provide only the minimal conditions for the operator. Indeed, a solution that will always satisfy the above definition is to make $S_x = S_1$ and the mapping $\mathsf{Id}(S_1)$. It is often possible, however, to come up with a better solution as we will see in Chapter 6.

We see the conditions of Extract illustrated in Figure 2.7. We can see that all instances of S_x are derived from map_{S_1,S_x} as required by condition 2 in the definition. $\mathsf{Inst}_5(S_1)$ is mapped to S_2 by map_{S_1,S_2} so it appears in S_x. In the case of $\mathsf{Inst}_1(S_1)$ not all of the instance is mapped to S_2 *i.e.* only a portion of the instance is part of the domain of the mapping. This will happen when there are select conditions in the constraints in Σ_{S_1,S_2}. In this case only the portion of the instance that is mapped to S_2 is added to the extract schema. In $\mathsf{Inst}_2(S_1)$ and $\mathsf{Inst}_4(S_1)$ it is the key fields of both instances that are mapped to $\mathsf{Inst}_3(S_2)$. The extract mapping maps them both to the same instance in S_x. This will happen when we have a project in our mapping. This leaves us with a schema that contains just those instances that are part of the mapping.

2.3.6 Diff

As well as being able to create a schema that contains only those instances that take part in a mapping, it is also useful in some cases to be able to create a schema that contains only those instances that do *not* take part in a mapping. An example is if a view of a database is used as part of a data warehouse. Updates to this view can cause a heavy load on the database affecting all the relations, including those not taking part in the view. In this scenario we can use the Diff operator to create a view of the database that contains only those relations that are not part of the view which could be materialised and used by those users who do not need the relations in the data warehouse, thus reducing the load on the main database. This is closely related to the concept of the **view complement** [BS81, CP84, LV03].

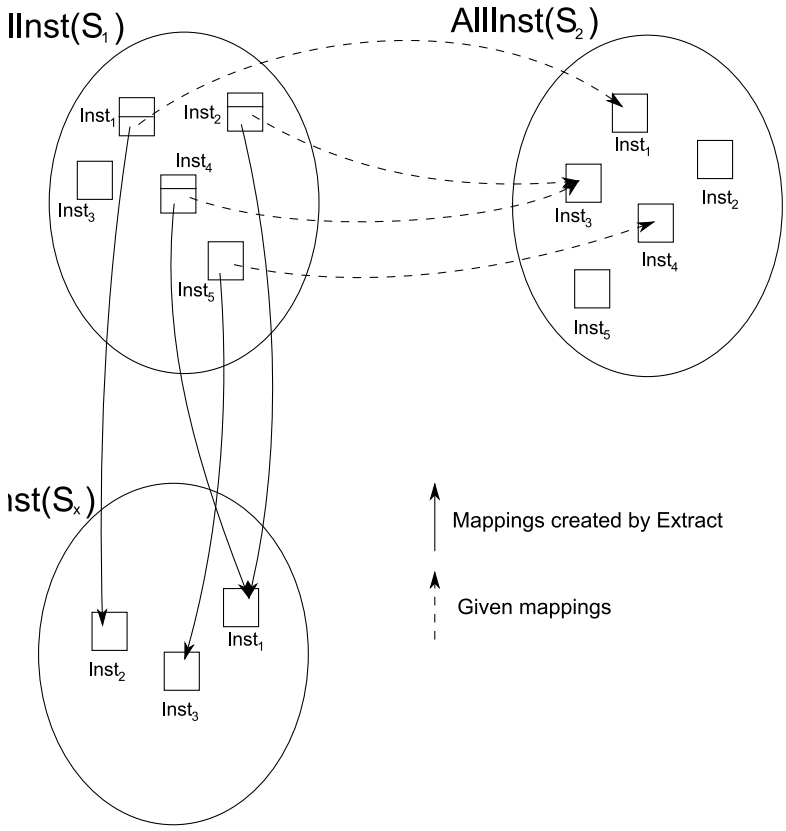


Figure 2.7: Extract

It should be clear from the above that merging the results of Diff with those from Extract will give us back the original schema. Note that the results of Diff and Extract may overlap. The following definition of Diff given in [MBHR05] formalises this notion:

Definition 2.12 Diff

Given S_1, S_2 and map_{S_1,S_2},
$\langle \mathsf{Schema}\ S_d, \mathsf{Mapping}\ map_{S_1,S_d} \rangle = \mathsf{Diff}(\mathsf{Schema}\ S_1, \mathsf{Mapping}\ map_{S_1,S_2})$ holds if and only if, for any S_x and map_{S_1,S_x} satisfying

$$\langle S_x, map_{S_1,S_x} \rangle = \mathsf{Extract}(S_1, map_{S_1,S_2})$$

the following condition holds:

$$\langle S_1, map_{S_1,S_x}, map_{S_1,S_d} \rangle = \mathsf{Merge}(S_x, S_d, \mathsf{Invert}(map_{S_1,S_x}) \circ map_{S_1,S_d})$$

□

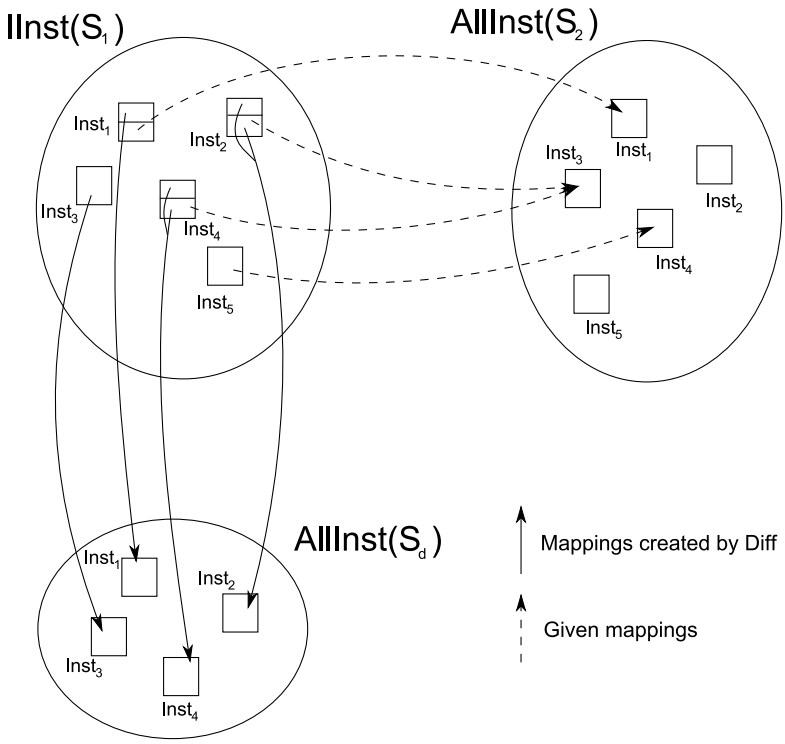


Figure 2.8: Diff

The condition in the definition ensures that Diff returns a schema that, when combined with an extract schema, S_x, allows us to recreate S_1.

As with Extract, the definition above can be trivially satisfied by making $S_d = S_1$ and the mapping $\mathsf{Id}(S_1)$.

The requirement that we should be able to recreate the original schema by merging S_d with the output of Extract means we must be able to uniquely identify the instances of S_d. For example if our DDL were SQL and we were to project only a key column in map_{S_1,S_2} then S_d must include any columns that depend on it, *i.e.* those columns not projected, plus the key. We need to include the key column otherwise we will not know how to relate the data instances together when we merge S_d with an extract schema as required by the definition.

Similarly if we project a non key column with the key column in map_{S_1,S_2} then S_d must contain the columns we have not projected and the key that allows us to uniquely recreate the original instance.

We see examples of Diff illustrated in Figure 2.8. The mappings and schemas in this figure are the same as those shown in Figure 2.7. Note that the whole of $\mathsf{Inst}_2(S_1)$ and $\mathsf{Inst}_4(S_1)$ appear in $\mathsf{AllInst}(S_d)$, *i.e.* the key and non-key values, because we need

to be able to uniquely identify these instances when we merge S_d with S_x to recreate S_1 as required by the definition.

2.3.7 ModelGen

Data management problems often involve data stored in schemas created using more than one DDL. A key feature of a MMS is the ability to process all these schemas in a seamless fashion. ModelGen is the model management [BHP00] operator that provides this functionality. It translates a schema created using one DDL into a corresponding schema expressed in another DDL and also produces a mapping between the schemas. Without this functionality a MMS is restricted to operating on schemas from a single DDL.

There is no formal definition of ModelGen given in the literature. We define it here as follows:

Definition 2.13 ModelGen

Given S_1 and DDL l

$$\langle \mathsf{Schema}\ S_2, \mathsf{Mapping}\ map_{S_1,S_2} \rangle = \mathsf{ModelGen}(\mathsf{Schema}\ S_1, \mathsf{TargetDDL}\ \ l)$$

holds if and only if:

- $\mathsf{AllInst}(S_1) = \mathsf{Domain}(map_{S_1,S_2})$
- $\mathsf{AllInst}(S_2) \supseteq \mathsf{Range}(map_{S_1,S_2})$.

Condition 1 ensures that all the instances of S_1 are translated into the new DDL. Condition 2 ensures that S_2 contains at least the instances that have an equivalent in S_1. □

ModelGen is useful in a number of circumstances. For example, an e-business may wish to move data between its back end SQL database and its XML based web pages without having to re-engineer the mappings every time the database schema or web pages are changed. The advantages become even more apparent if the SQL database was designed using an ER model. The mappings created by ModelGen would also allow the MMS the seamlessly update the ER model based on any changes to the XML view.

2.3.8 TransGen

So far we have seen mappings between existing schemas, and database instances that satisfy those mappings. Ideally, however, our MMS should be able to *create* a target schema and any associated instances, based on a given mapping. For this we need to translate our mappings into *transformations* in an executable mapping language. We will call this executable mapping language the **transformation language**. This translation is the role of the TransGen operator.

Definition 2.14 TransGen

Given S_1 and transformation language l

$$\langle \mathsf{ExecutableMapping}\, map_{S_1,S_2} \rangle = \mathsf{TransGen}(\mathsf{Mapping}\, map_{S_1,S_2} \mathsf{ExecutableMappingLanguage}\, l)$$

if and only if applying $\mathsf{ExecutableMapping}\, map_{S_1,S_2}$ to S_1 creates S_2 along with all its instances. □

TransGen is motivated by the statement: "It's easier to write mappings than write code" [BH07]. It gives us the freedom to describe mappings in a generic, non-implementation specific way. The s-t tgds described above are the most popular formalism but others could be used. TransGen allows us to execute the same mappings in different transformation languages, *i.e.* we can use the same mappings in a system that uses a number of DDL specific transformation languages and a system that uses a single transformation language applied to all schemas in the MMS.

Two approaches have been taken in current MMSs. MIDST [ACB06] uses a single transformation language, Datalog, to directly manipulate the schema descriptions that are stored in an SQL database. Algorithms exist to translate the transformed schemas into their native DDL. *GeRoMeSuite* translates declarative mappings directly into the transformation language of the DDL the schema is expressed in, using a specific algorithm for each DDL [KQLJ07]. Moda uses XSL and XSLT transformations [MBHR05].

SQL DDL is an example of an executable mapping language. The following is the output from $\mathsf{TransGen}(map_{S_{emp},S_{finEmp}}, \mathtt{SQL})$ where $map_{S_{emp},S_{finEmp}}$ is the mapping in Example 2.1.

```
CREATE SCHEMA SFinEmp;

CREATE VIEW SFinEmp.FinEmp(eid,name) AS
SELECT eid,name FROM Emp, Dept
WHERE did = dept AND dname='Finance';

CREATE VIEW SFinEmp.FinDept(did,numEmps) AS
SELECT did,numEmps FROM Emp, Dept
WHERE did = dept AND dname='Finance';
```

It should be noted that the declarative mappings are independent of the transformation language into which they are translated. We can change the value of the l parameter in TransGen to change the transformation to, for example, Datalog [AU79]. The following is the output from TransGen($map_{S_{emp},S_{finEmp}}$,Datalog)

```
finEmp(X,Y) :- emp(X,Y,Z), dept(Z,W,V), W=finance.

finDept(Z,V) :- emp(X,Y,Z), dept(Z,W,V), W=finance.
```

2.4 Model Management Scripts

We saw in the example in the introduction that the maximal benefit of the operators is achieved if a number are executed sequentially in a script. Formally, a model management script is a sequential list of model management operators. In some cases reordering is possible but this is not always the case [MBHR05]. The variables and constants in a script represent schemas and mappings. When a script is executed the variables are replaced with actual schemas and mappings. A script can be said to have completed successfully if the script returns a non-empty result.

Example 2.3 MM script

This example shows how we can use a simple MM script to combine some of the results from examples in the previous sections. Assume we have two databases that store employee data, one contains table:$\langle\langle$Emp$\rangle\rangle$ from S_{emp} in Figure 2.1, while the other contains some additional personal information. The schemas for the two databases are defined as follows:

```
CREATE SCHEMA SEmp;

CREATE TABLE SEmp.Emp(
  eid int not null,
  name varchar not null,
  dept int not null,
  CONSTRAINT Emp_pk PRIMARY KEY (eid)
);

CREATE SCHEMA SPers;

CREATE TABLE SPers.EmpPersonal(
  eid int not null,
  name varchar not null,
  gender varchar not null,
  dob varchar null,
  CONSTRAINT Pers_pk PRIMARY KEY (eid)
);
```

We now wish to create a single database with all the employee information in it. First we need to create a mapping between the two schemas that tells us how the data in one relates to the other. A data expert creates the following mapping:

$map_{S_{emp},S_{pers}} = (S_{emp}, S_{pers}, \Sigma_{S_{emp},S_{pers}})$, where $\Sigma_{S_{emp},S_{pers}} =$
$\{S_{emp}:::\langle\langle \mathsf{Emp} \rangle\rangle(e,n,d) \rightarrow \exists g, dob(S_{pers}:::\langle\langle \mathsf{EmpPersonal} \rangle\rangle(e,n,g(e),dob(e)))\}$

Where g and dob are Skolem functions. As we can see from the mapping there are columns that are common to both schemas. Instead of merging the two schemas in their entirety, we can save ourselves work by adding only those instances from S_{pers} that do not already appear in S_{emp}. We can use the Diff operator to create a schema that only contains schema objects and their instances that do not appear in S_{emp} and then merge it with S_{emp} using the Merge operator. Instead of doing this operation by hand every time a change is made to either schema, we can use our MMS and the following MM script to automatically do the merging for us any time a change is made.

1. $\langle S_d, map_{S_{pers},S_d} \rangle := \mathsf{Diff}(S_{pers}, \mathsf{Invert}(map_{S_{emp},S_{pers}}))$
2. $map_{S_{emp},S_d} := map_{S_{emp},S_{pers}} \circ map_{S_{pers},S_d}$
3. $\langle S_m, map_{S_m,S_{emp}}, map_{S_m,S_d} \rangle := \mathsf{Merge}(S_{emp}, S_d, map_{S_{emp},S_d})$

We can run this script on any pair of schemas that have a mapping between them. Here we just give the results of applying each operator to show how they can be used in subsequent steps. In Chapter 6 we will give more detailed descriptions of how the operators produce their results.

Step 1:

$S_d = \{\text{table:}\langle\langle \text{Pers\_diff} \rangle\rangle, \text{column:}\langle\langle \text{Pers\_diff}, \text{eid} \rangle\rangle, \text{column:}\langle\langle \text{Pers\_diff}, \text{gender} \rangle\rangle, \text{column:}\langle\langle \text{Pers\_diff}, \text{dob} \rangle\rangle,$
$\text{primary\_key:}\langle\langle \text{Pers\_pk}, \text{Pers\_diff}, \text{eid} \rangle\rangle\}$
$map_{S_{pers},S_d} = (S_{pers}, S_d, \{S_{pers}\text{::table:}\langle\langle \text{EmpPersonal}(e,n,g,dob) \rangle\rangle \rightarrow$
$S_d\text{::table:}\langle\langle \text{Pers\_diff}(e,g,dob) \rangle\rangle\})$

Note that we need to retain the primary key column of EmpPersonal in S_d so that we can link the tuples of S_d to those of S_{pers}.

Step 2:

$map_{S_{emp},S_d} = (S_{emp}, S_{pers}, \{S_{emp}\text{::table:}\langle\langle \text{Emp} \rangle\rangle(e, n, d) \rightarrow$
$\exists g, dob(S_{pers}\text{::table:}\langle\langle \text{EmpPersonal} \rangle\rangle(e, n, g(e), dob(e)))\}) \circ$
$(S_{pers}, S_d, \{S_{emp}\text{::table:}\langle\langle \text{EmpPersonal}(e,n,g,dob) \rangle\rangle \rightarrow S_d\text{::table:}\langle\langle \text{EmpDiff}(e,g,dob) \rangle\rangle\})$
$= (S_{emp}, S_d, \{S_{emp}\text{::table:}\langle\langle \text{Emp} \rangle\rangle(e, n, d) \rightarrow$
$\exists g, dob(S_d\text{::table:}\langle\langle \text{EmpDiff}(e,g(e),dob(e)) \rangle\rangle)\})$

Step 3:

$S_m = \{\text{table:}\langle\langle \text{EmpMerge}(e,n,d,g,dob) \rangle\rangle\}$
$map_{S_m,S_{emp}} = (S_m, S_{emp}, \{S_m\text{::table:}\langle\langle \text{EmpMerge}(e,n,d,g,dob) \rangle\rangle \rightarrow S_{emp}\text{::table:}\langle\langle \text{Emp} \rangle\rangle(e, n, d)\})$
$map_{S_m,S_d} = (S_m, S_d, \{S_m\text{::table:}\langle\langle \text{EmpMerge}(e,n,d,g,dob) \rangle\rangle \rightarrow$
$S_d\text{::table:}\langle\langle \text{EmpDiff} \rangle\rangle(e, g, dob)\})$

□

2.5 Other MMSs

We will now summarise the MMSs in the current literature. These systems all take different approaches to the design of both the CDM and the mapping language. Their features are shown in Table 2.2. We will briefly describe each one and show how they represent the SQL database in Figure 2.1 and, where the system supports instance-based mappings, the mapping in Example 2.1. We end each section by assessing how well each of these systems meets the criteria laid out for a MMS in

the first section of this chapter. At the end of this section we discuss how studying these other systems informed decisions we made in the design of our prototype.

MMS Name	Multi DDL support	Automatic schema translation	Executable mappings	Other operators	Supports scripting
Rondo	Y	N	N	Y	Y
MIDST	Y	Y	Y	N	N
Moda	N	N	Y	Y	Y
AutoGen	Y	N	N	Y	N
GeRoMeSuite	Y	N	Y	Y	N

Table 2.2: Current Model Management System Prototypes

2.5.1 Rondo

Rondo [Mel04, MRB03] was the first MMS prototype. It uses a labelled directed graph as the CDM and simple binary tuples called **morphisms** to describe mappings between schemas.

The Rondo CDM provides a common visual representation of schemas for all the DDLs that Rondo supports. Rules exist in the system to translate high level schemas into this generic representation which the MM operators then manipulate. Figure 2.9 shows how the schema in Figure 2.1 is represented in the Rondo CDM. The ovals in the graph denote identifiers, the rectangles denote literals and the links between the various elements are shown using labelled directed edges. This provides an intuitive view of the schema and allows us to easily compare schemas from different DDLs. Rondo can represent SQL and XML schemas in this CDM. Rondo uses an SQL database to store the translated CDM representation.

The morphisms that form the mapping language relate an object in a source schema to one in a target schema. They were chosen as the mapping language for Rondo because their semantics are simple and well understood [AB01, MRB03] but they provide no information about what will happen to the instances of the schema objects they map. It may be the case that all, some or none of the instances of the source schema object should appear in the target schema object. The morphism does not tell us which will be the case. They are useful in tasks such as schema translation where manipulation of the schema instances is not required.

The original version of Rondo [MRB03] implements all the common MM operators but it does not support extensional semantics. This limits its usefulness for a number

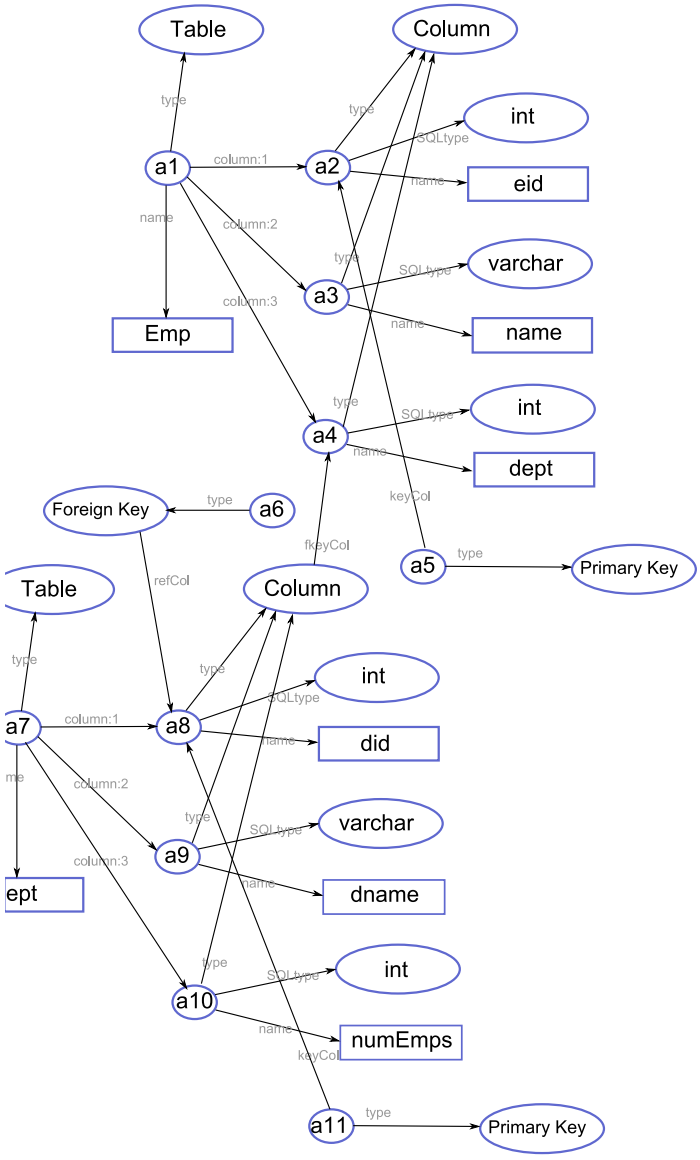


Figure 2.9: Figure 2.1 represented in the Rondo schema language

of common metadata management problems, such as data integration and data exchange.

The following set of morphisms represents an intensional mapping between S_{emp} and S_{finEmp}.

$\langle S_{emp}$:SQL:table:$\langle\langle$Emp$\rangle\rangle, S_{finEmp}$:SQL:table:$\langle\langle$FinEmp$\rangle\rangle\rangle$
$\langle S_{emp}$:SQL:column:$\langle\langle$Emp, eid$\rangle\rangle, S_{finEmp}$:SQL:column:$\langle\langle$FinEmp, eid$\rangle\rangle\rangle$
$\langle S_{emp}$:SQL:column:$\langle\langle$Emp, name$\rangle\rangle, S_{finEmp}$:SQL:column:$\langle\langle$FinEmp, name$\rangle\rangle\rangle$

We can see that these mappings are very simple but they lose the fact that only employees who are in the finance department should be in table:$\langle\langle$FinEmp$\rangle\rangle$. They have been shown to be very efficient to implement [MRB03, Mel04] but their obvious drawback is their lack of extensional semantics.

A limited form of extensional semantics in the form of path morphisms, were added to Rondo [MBHR05], but these are limited in their scope. They support only specific types of relational mapping where the relational schema can be represented as a tree.

Shortcomings:

- Limited extensional semantics

2.5.2 Moda

Moda [MBHR05], a system based on Rondo, supports instance based mappings in the form of logic formulae. Moda was designed to test the viability of using executable mappings to implement the MM operators rather than as a full blown MMS. It only supports relational schemas and does not implement ModelGen.

The implementation of the operators makes use of a library of building blocks that were developed for elementary formula manipulation. The library includes algorithms for unification, resolution, transforming a formula into implicative normal form and others. The tool generates XSL transformations to migrate data between source and target schemas.

Moda only supports relational schemas. It uses first order embedded dependencies to describe its mappings. The Moda representation of the mapping in Example 2.1 is shown below:

SM_Abstracts

OID	sOID	Name
101	1	Emp
102	1	Dept

SM_AttributeOfAbstract

OID	sOID	Name	IsKey	IsNullable	AbsOID	Type
201	1	eid	T	F	101	int
202	1	name	F	F	101	varchar
203	1	dept	F	F	101	int
204	1	dept	T	F	102	int
205	1	dname	F	F	102	varchar
206	1	numEmps	F	T	102	int

Figure 2.10: A MIDST representation of the schema in Figure 2.1(a)

$map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$ where $\Sigma_{S_1,S_2} =$
$\{\forall e, n, d(S_1.\mathsf{Emp}(e, n, d), S_1.\mathsf{Dept}(d, \texttt{'Finance'}, ne) \rightarrow$
$S_2.\mathsf{FinEmp}(e, n), S_2.FinDept(d, ne)),$
$\forall e, n(S_2.\mathsf{FinEmp}(e, n), S_2.FinDept(d, ne) \rightarrow$
$(S_1.\mathsf{Emp}(e, n, d), S_1.\mathsf{Dept}(d, \texttt{'Finance'}, ne)))\}$

We see that the set of embedded dependencies in Σ_{S_1,S_2}, includes a 'reverse' constraint allowing the mapping to be traversed forwards and backwards.

Shortcomings:

- Supports only one DDL.
- Mapping language is first order so mappings are not always composable.

2.5.3 MIDST

A data level implementation of ModelGen based on executable Datalog mappings can be found in MIDST [ACB05]. The common representation of the high level DDL schemas is stored directly in SQL using a relational data dictionary. A representation of the schema in the MIDST CDM is shown in Figure 2.10. This has the advantage of being efficient, but is much more difficult for a user to understand than graphical representations.

MIDST uses a complex, high-level CDM that includes all the constructs of the DDLs used in the MMS. It makes use of a *supermodel* that includes the features of all the constructs of the DDLs known to the system. For example, the `SM_AttributeOfAbstract` construct models attributes from a number of different DDLs and includes features such as IsKey and IsNullable that may be needed by a DDL in the system.

SM_InstOfAbstract

OID	dOID	AbsOID
1001	1	101
1002	1	101
1003	1	102
1004	1	102

SM_InstAttributeOfAbstract

OID	dOID	AttOID	i-AbsOID	Value
2001	1	201	1001	1
2002	1	202	1001	Peter Smith
2003	1	203	1001	100
2004	1	201	1002	21
2005	1	202	1002	Susan Brown
2006	1	203	1002	101
2007	1	204	1003	100
2008	1	205	1003	Finance
2008	1	205	1003	23
2009	1	204	1004	101
2010	1	205	1004	HR

Figure 2.11: A MIDST representation of the SQL database instance in Figure 2.1

Each construct in the supermodel has its own table which stores all the instances of that abstract construct. The data dictionary is visible so new high level constructs can be added to the super model tables as new DDLs are added to the system.

The approach in this thesis, on the other hand, will use a set of simple CDM constructs and use combinations of these to create any complex structures needed. Batini et al. [BLN86], in their survey of data integration methods, suggest that a simpler CDM has advantages over more complex models.

MIDST supports extensional semantics, allowing the manipulation, not only of the schema, but also of the data held in it. Figure 2.11 shows how the database instance in Figure 2.1 can be represented using the MIDST data dictionary.

The translation between DDLs is done as follows: First the source data is copied into the CDM. This representation is then transformed into the target DDL by composing a number of elementary translation steps that are stored in the system. Each step is defined by a specific rule that defines a common restructuring task. The data is then copied from source to CDM to target using instance-level datalog [AU79] rules that mimic the schema transformation rules. Using a generic mapping language like Datalog allows MIDST to support flexible and extensible data translation for a number of DDLs.

MIDST has the disadvantage that data must be copied between the source model and the CDM and again from the CDM to the target system. A further significant disadvantage is that this method does not return a set of mappings between the source and target schemas. This means they cannot be used in a MM script where the mappings between source and target are needed as parameters to other operators.

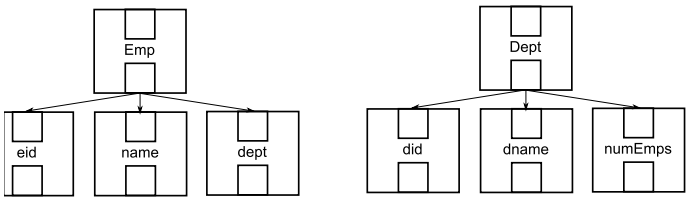


Figure 2.12: The SQL schema from Figure 2.1 represented in RGG

Shortcomings:

- Very complicated supermodel with limited scalability
- Only implements ModelGen at present
- No support for scripting.

2.5.4 AutoGen

AutoGen [SKZ06] takes a different approach to mappings, providing a visual metaphor by using graph transformations as the mapping paradigm. The CDM used by AutoGen is the reserved graph grammar (RGG) [ZZKS05]. Figure 2.12 shows the schema from Figure 2.1 represented in RGG.

As we can see from the figure, schemas are represented as a directed node-edge diagram. A node represents a schema object and an edge denotes a relationship. For example, in the figure, the directed edges represent the table/column relationship between the nodes.

AutoGen provides interfaces for the manipulation of these graphical data models. The approach consists of two levels of graphical operators: low-level customisable operators and high-level generic operators, both of which consist of a set of graph transformation rules. AutoGen automatically produces low-level operators from input schemas, based on the data source the schema came from, and mappings according to a high-level operator.

RGG does not support instance based semantics so the mapping from Example 2.1 cannot be represented. A simple renaming mapping for the Emp table is shown in Figure 2.13.

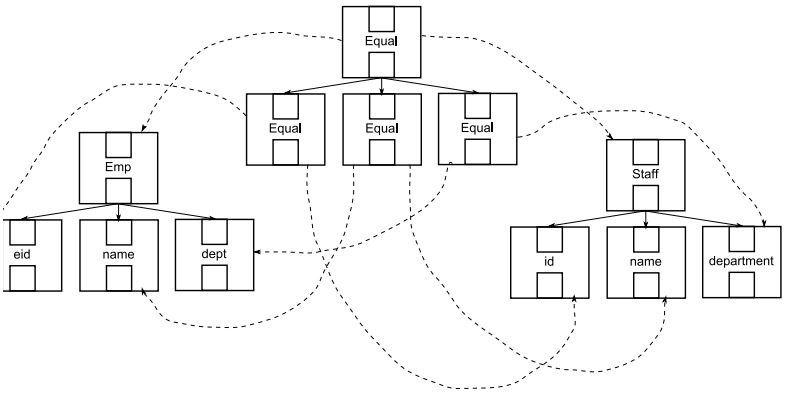


Figure 2.13: A simple mapping represented in AutoGen

The lack of an instance based semantics for RGG and the difficulty of processing graph transformation mappings has prevented any further development of the AutoGen system.

Shortcomings:

- No extensional semantics
- CDM and mappings cannot be implemented efficiently
- No support for scripting

2.5.5 *GeRoMeSuite*

GeRoMeSuite represents the most complete current MMS apart from that described in this thesis. *GeRoMeSuite* [KQLL07] implements a number of the most common operators including Compose, Match and Merge. It additionally provides an environment that simplifies the implementation of other operators. It is based on their generic role based DDL, *GeRoMe* [KQCJ07], in which each schema object has added to it a set of role objects that represent specific properties of the schema object. Roles may be added to or removed from elements at any time to allow for a flexible definition of schemas. Roles can expose different views to different operators on the same schema object. Thus, operators concentrate on features which affect their functionality and can ignore those that do not affect their functionality. The operators, thus, only have to be implemented for the CDM, *GeRoMe*, and not for each specific DDL.

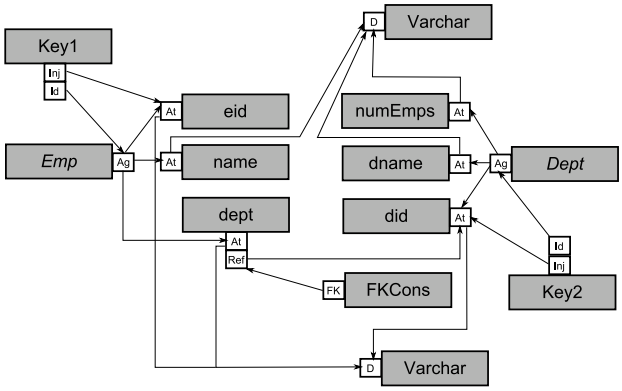


Figure 2.14: The schema from Figure 2.1 represented in *GeRoMe*

Figure 2.14 shows the schema from Figure 2.1 represented in *GeRoMe*. The grey boxes denote schema objects and the white boxes attached to them represent the different roles they play. The two tables Emp and Dept have associated attributes and thus play an *Aggregate* (Ag) role. The attributes themselves play an *Attribute* (At) role. The role also refers to the data type of each attribute. Here the data type domains are represented by the *Domain* (D) role. The key attributes are defined using separate schema objects representing the key constraint. The objects play an *Injective* (Inj) role to indicate that an attribute is unique, and an *Identifier* (Id) role to specify the aggregate for which the attribute is the key. The foreign key constraint in the schema is also represented by a separate schema object which plays a *Foreign Key* (FK) role. This points to a *Reference* (Ref) role which is played by the attribute that references the key of the other table.

The instance of the database in Figure 2.1 is represented in *GeRoMe* as follows:

inst(#0, Emp)
attr(#0, eid, #1), *value*(#1, 1)
attr(#0, eid, #2), *value*(#2, 2)
attr(#0, name, #3), *value*(#3, 'Peter Smith')
attr(#0, name, #4), *value*(#4, 'Susan Brown')
attr(#0, dept, #5), *value*(#5, 100)
attr(#0, dept, #6), *value*(#6, 101)
inst(#7, Dept)
attr(#7, did, #8), *value*(#8, 100)
attr(#7, did, #9), *value*(#9, 101)
attr(#7, dname, #10), *value*(#10, 'Finance')
attr(#7, dname, #11), *value*(#11, 'HR')

attr(#7, numEmps, #12), *value*(#12, 23)
attr(#7, numEmps, #13), *value*(#13, 15)

The *inst* predicate represents an instance of an object that plays an *Aggregate* role, *attr* an attribute and *value* the value of the associated attribute. The abstract identifiers, prefixed with a #, link the various predicates so that a complete instance of the schema can be created.

Mappings in *GeRoMeSuite* are represented using SO s-t tgds. The mapping in Example 2.1 is represented in *GeRoMeSuite* as follows:

$inst(o_0, \texttt{Emp}) \wedge inst(o_1, \texttt{Dept}) \wedge$
$attr(o_0, \texttt{eid}, o_2) \wedge value(o_2, e) \wedge attr(o_0, \texttt{name}, o_3) \wedge value(o_3, n) \wedge attr(o_0, \texttt{dept}, o_4) \wedge value(o_4, d)$
$attr(o_1, \texttt{did}, o_5) \wedge value(o_5, di) \wedge attr(o_1, \texttt{did}, o_6) \wedge value(o_6, 'Finance')$
$\rightarrow inst(o_7, \texttt{FinEmp}) \wedge inst(o_8, \texttt{FinDept}) \wedge$
$attr(o_7, \texttt{eid}, o_9) \wedge value(o_9, e) \wedge attr(o_7, \texttt{name}, o_{10}) \wedge value(o_{10}, n)$
$attr(o_8, \texttt{did}, o_{11}) \wedge value(o_{11}, d) \wedge attr(o_8, \texttt{numEmps}, o_{12}) \wedge value(o_{12}, ne)$

These SO s-t tgds are applicable to *GeRoMe* schema objects which can also be used to represent a wide range of DDLs. However, they do not provide an intuitive description of the mapping. This much is admitted by the authors [KQLJ07]. Indeed they do not expect users of their system to use their formalism to describe mappings, rather these mappings are created using a GUI. They can also be specified as Prolog rules and then imported into the system.

The system provides algorithms to translate these mappings into a suitable transformation in the mapping language for the data source of the original schema. Currently it supports SQL and XQuery.

Shortcomings:

- Queries must be translated into the model specific language of the original schema
- The role based model is complex and difficult for a user to understand.
- There is no current support for creating MM scripts.

2.5.6 ATLAS and Model Driven Engineering

ATLAS [FBJV05] is a **Model-Driven Engineering** (**MDE**) system based on standards published by the **Object Management Group** (**OMG**) [Obj]. It provides a programming environment that treats schemas as first class entities in the same way that MM does. The approach is geared towards MDE rather than databases and the system does not aim to implement the MM operators but rather to provide a programming environment that allows the manipulation of schemas in a more general way. Programs could then potentially be written using this environment to implement the MM operators. This is a different approach to the problems MM aims to address and so a full discussion is beyond the scope of this thesis.

2.5.7 Discussion

Research into the systems mentioned above helped inform the decisions we made when designing our MMS. Early experience with MMSs showed that mappings often needed to be inverted [MBHR05]. With this in mind we followed the approach adopted in Moda of basing our mappings on sets of s-t tgds that include an inverse as part of the mapping and so can be easily inverted. It was also found that Compose was the most commonly used operator and was impossible to implement fully in a first order language [FKPT05]. In common with *GeromeSuite* we use second order mappings which have been shown to be closed under composition [FKPT05].

An attempt to extend the morphisms used in Rondo to executable mappings based on relational algebra was only partially successful because of the DDL specific nature of the executable language they chose. In *GeromeSuite* [KQLL07] mappings are translated into the query language of the source schema rather than being translated into a universal language. This means translators must be written for each DDL in the MMS, an approach that does not scale well. Using a general purpose query language makes these mappings easier to execute. This is the approach taken in MIDST [ACB05], which uses Datalog. We adopt a similar approach using the DDL independent query language IQL [Zam08].

CDMs are central to a DDL independent MMS, but a drawback is that writing mappings using the CDM constructs is often difficult because the relationship between the CDM constructs and the high level constructs they represent is not always obvious. This is particularly apparent in MIDST and *GeromeSuite*. We overcome this by using a unique scheme syntax that allows us to reference the underlying CDM

objects through an intuitive description of the high level schema objects which we describe in detail in the next chapter.

A graphical CDM with a few simple constructs is used to model a wide range of DDLs in Rondo [MRB03], AutoGen [SKZ06] and *GeromeSuite*, and is the approach we have adopted. The choice of a text based model in MIDST which models individual aspects of each DDL in the MMS does not seem to scale well.

As we have seen none of the current systems implement all five of the criteria for a MMS. We will show in this thesis that our system meets all these criteria and so is a more complete MMS than those that exist at the moment. The main limitation of our system is that it does not have an implementation of Match; however, we hope to integrate the ongoing work of Rizopoulos [RM05, MRMM05] in the future to overcome this.

Chapter 3

AutoMed Model Management Abstractions

In this chapter we describe the framework we use for implementing our MMS. We first describe the DDL we use as our CDM in AutoMed and show how schemas from a wide range of structured DDLs can be represented in it. We then go on to describe the schema transformation based mapping and transformation language that we use to represent both inter and intra DDL mappings in AutoMed. Finally we show how we can create transformations in AutoMed that are equivalent to a set of SO s-t tgds.

3.1 The AutoMed CDM

The CDM we use in AutoMed is the **Hypergraph Data Model** (**HDM**) [MP98]. It is a hypergraph based DDL that makes use of a small set of simple constructs to describe other, more complex, DDLs[1] that also includes a DDL independent way of expressing constraints.

The HDM has the following features that make it a good candidate for use as a CDM in a MMS:

- It supports instance-based semantics

[1] In keeping with common terminology, we refer to these more complex DDLs as *higher level* from now on because they have a higher level of complexity

- It has been shown to be able to model a wide range of DDLs including ER, EER, SQL, UML class diagrams, ORM [MP99, BM05] and XML [MP01] documents not constrained by XML Schema.
- It supports simple DDL independent constraints.

In this thesis we extend the definition of the HDM in [MP98, BM05] to include primitive data types, giving in Definition 3.1 a typed HDM.

Definition 3.1 HDM Schema

Given a set of strings called *Labels* that we may use for modelling the real world, and another disjoint set of strings called $TypeNames$ that we may use to name primitive data types, an HDM **schema**, S, is a quadruple $\langle Types, Nodes, Edges, Cons\rangle$ where:

1. $Types \subseteq \{t \mid t \in TypeNames\}$ *i.e.* $Types$ is the set of primitive data types used in this schema. In general we will omit $Types$ from the schema definition unless it is significant to the discussion. Primitive data types in AutoMed are discussed in detail in Chapters 4.
2. $Nodes \subseteq \{\mathsf{hdm{:}node}{:}\langle\langle n_n, t\rangle\rangle \mid n_n \in Names, t \in Types\}$
   i.e. $Nodes$ is the set of nodes in the graph, each denoted by its name enclosed in double chevron marks. t denotes the primitive data type of this node if it has one. If the node is untyped its scheme is simply $\langle\langle n_n\rangle\rangle$.
3. $Edges \subseteq \{\mathsf{hdm{:}edge}{:}\langle\langle n_e, s_1, \ldots, s_n\rangle\rangle \mid n_e \in Names \cup \{\_\} \wedge s_1 \in ExtensionalObject \wedge \ldots \wedge s_n \in ExtensionalObject\}$
   i.e. $Edges$ is the set of edges in the graph where each edge is denoted by its name, together with the list of nodes/edges that it connects, enclosed in double chevron marks. '_' denotes an unnamed edge.
4. $ExtensionalObject = Nodes \cup Edges$
5. $Cons \subseteq \{c(s_1, \ldots, s_n) \mid c \in Funcs \wedge s_1 \in ExtensionalObject \wedge \ldots \wedge s_n \in ExtensionalObject\}$ *i.e.* $Cons$ is a set of boolean-valued functions (*i.e.* constraints) whose variables are members of $ExtensionalObject$ and where the set of functions $Funcs$ forms the HDM constraint language.

   The constraint language used in this thesis is defined in Definition 3.4 but may be extended to handle new DDLs. It includes as many constraints as are

necessary to model the various constraints in the DDLs we wish to process with AutoMed, and to differentiate between the variations of a construct that may exist in a high level DDL.

6. $SchemaObjects = ExtensionalObject \cup Cons$

□

The key scheme of an HDM node does not include the data type while the key scheme of an edge includes all the nodes or edges it links. An example HDM schema, using the key schemes of the objects, is shown in Example 3.1.

Example 3.1 An HDM Schema

Let $S_{eg} = \langle Nodes, Edges, Cons \rangle$ such that

$Nodes = \{\text{node:}\langle\langle \text{Emp} \rangle\rangle, \text{node:}\langle\langle \text{Emp:eid} \rangle\rangle, \text{node:}\langle\langle \text{Emp:name} \rangle\rangle, \text{node:}\langle\langle \text{Emp:dept} \rangle\rangle\}$
$Edges = \{\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle, \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle, \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle\}$
$Cons = \{\}$

□

The HDM supports instance-based semantics. Each node has an extent that is the set of values from the data source object associated with the node. We see later on in this chapter how we create a mapping from a data source to a HDM schema. Each edge also has an extent, where the values the edge extent contains must also appear in the extent of the nodes and edges that the edge connects. The extent of an edge is a collection of tuples with the first element of the tuple coming from the first node or edge in the edge scheme and the second tuple element from the second node or edge. An HDM **instance** is defined as follows:

Definition 3.2 HDM Instance

Let $Vals$ be the set of all values in the domain we wish to model, $Seq(Vals)$ be any sequence of those values, S be an HDM schema, $ExtensionalObject$ the set of extensional objects in S and $Types$ the set of HDM types. An instance k of S is a set

> $Ext_{S,k}(ExtensionalObject) \rightarrow \mathcal{P}(Seq(Vals))$, where $\mathcal{P}$ is the power set, such that the **extent** of a schema object $\langle\langle \text{so} \rangle\rangle \in ExtensionalObject$, written $Ext_{S,k}(\langle\langle \text{so} \rangle\rangle)$, is a set of values containing members of $Seq(Vals)$.

The extent of a node will be a set of values, and the extent of an edge is a set of n-ary tuples where n is the number of nodes/edges the edge connects.

We also add the following restrictions:

1. $Ext_{S,k}(\mathsf{node}{:}\langle\langle n, t\rangle\rangle) \subseteq Ext(t)$ where $t \in Types$
2. $\forall\, 1 \leq i \leq n.(a_1, \ldots, a_n) \in Ext_{S,k}(\mathsf{edge}{:}\langle\langle e, s_1, \cdots s_n\rangle\rangle) \rightarrow a_i \in Ext_{S,k}(s_i)$
3. $\forall c \in Cons$ the expression $c(v_1/Ext_{S,k}(v_1), \cdots, v_n/Ext_{S,k}(v_n))$ evaluates to true, where $v_1, \cdots, v_n$ are the variables of c.

All the instances of S are defined as:

$\mathsf{AllInst}(S) = \mathsf{Inst}_1(S) \cup \mathsf{Inst}_2(S) \cup \ldots$

As in our definition of a general schema instance, cf. Definition 2.2, when writing $\mathsf{Inst}_k(S)$ we will generally include the schema object as a prefix to its extent tuple to allow us to identify the object that the tuple in the extent of the schema is associated with. When writing the extent of a single HDM schema object we do not do this. □

An example of an HDM instance, I_1, of the schema S_{eg} is shown in Example 3.2. Note that when writing the extent of a schema object we will generally only use the key scheme.

Example 3.2 HDM Instance $Ext_{S_{eg},I_1}(\mathsf{node}{:}\langle\langle\mathsf{Emp}\rangle\rangle) = \{(\mathtt{1}), (\mathtt{21})\}$
$Ext_{S_{eg},I_1}(\mathsf{node}{:}\langle\langle\mathsf{Emp{:}eid}\rangle\rangle) = \{(\mathtt{1}), (\mathtt{21})\}$
$Ext_{S_{eg},I_1}(\mathsf{edge}{:}\langle\langle\_, \mathsf{Emp}, \mathsf{Emp{:}eid}\rangle\rangle) = \{(\mathtt{1,1}), (\mathtt{21,21})\}$
$Ext_{S_{eg},I_1}(\mathsf{node}{:}\langle\langle\mathsf{Emp{:}name}\rangle\rangle) = \{(\mathtt{`Peter\ Smith'}), (\mathtt{`Susan\ Brown'})\}$
$Ext_{S_{eg},I_1}(\mathsf{edge}{:}\langle\langle\_, \mathsf{Emp}, \mathsf{Emp{:}name}\rangle\rangle) = \{(\mathtt{1,`Peter\ Smith'}), (\mathtt{21,`Susan\ Brown'})\}$
$Ext_{S_{eg},I_1}(\mathsf{node}{:}\langle\langle\mathsf{Emp{:}dept}\rangle\rangle) = \{(\mathtt{100}), (\mathtt{101})\}$
$Ext_{S_{eg},I_1}(\mathsf{edge}{:}\langle\langle\_, \mathsf{Emp}, \mathsf{Emp{:}dept}\rangle\rangle) = \{(\mathtt{1,100}), (\mathtt{21,101})\}$

□

In the same way that nodes and edges are used to represent the *data* constructs of higher level DDLs, high level *constraint* expressions and any constraints on the data constructs, are represented by six constraint constructs [BM05]. These are summarised in Definition 3.4. When used together, these constructs give a rich

framework in which to express cardinality constraints, keys and other types of constraints found in high level DDLs. In the following, variables that begin with s are assumed to be members of *ExtensionalObject*. For each constraint definition we give both a functional form, *e.g.* **inclusion**(s_1, s_2) that is useful in mapping rules that talk about a constraint in general, and an equivalent infix form, *e.g.* $s_1 \subseteq s_2$, which is used in the diagrams and when we write specific constraint expressions.

In [BM05] a **project** function, that provides a method of producing a view of an HDM edge is defined as shown in Definition 3.3.

Definition 3.3 HDM project

If s is an HDM edge, and t is a tuple such that the elements of t are a subset of the elements in the extent of s, and $\langle s_x, \ldots s_y \rangle$ is a tuple of schemes that appear in s then the HDM **project** function $\pi(\langle s_x, \ldots, s_y \rangle, s, t)$, will return the values of t that correspond to $\langle s_x, \ldots, s_y \rangle$, *i.e.*

$$\pi(\langle s_x, \ldots, s_y \rangle, \langle\langle n_e, s_1, \ldots, s_x, \ldots, s_y, \ldots, s_n \rangle\rangle, \\ (a_1, \ldots, a_x, \ldots, a_y, \ldots, a_n)) = (a_x, \ldots, a_y)$$

Note that for this thesis the project function operates only on single tuples. □

We now review of the definitions of the constraint constructs given in [BM05].

Definition 3.4 HDM Constraint Constructs

The HDM comprises at least the following constraint constructs: $Funcs = \{$**union**$(\cup)$, **inclusion**$(\subseteq)$, **exclusion**$(\not\cap)$, **mandatory**$(\rhd)$, **unique**$(\lhd)$, **reflexive**$(\overset{\mathbf{id}}{\rightarrow})\}$ where the functions are defined as follows:

1. **union**$(s, s_1, \ldots, s_n) \equiv s = s_1 \cup \ldots \cup s_n$: The extent of s equals the unions of the extents of s_1 to s_n *i.e.* $\forall k, Ext_{S,k}(s) = Ext_{S,k}(s_1) \cup \ldots \cup Ext_{S,k}(s_n)$

2. **inclusion**$(s_1, s_2) \equiv s_1 \subseteq s_2$: The extent of s_1 is a subset of s_2, *i.e.* $\forall k, Ext_{S,k}(s_1) - Ext_{S,k}(s_2) = \emptyset$

3. **exclusion**$(s_1, \ldots, s_n) \equiv (s_1 \not\cap \ldots \not\cap s_n)$: The extents of a set of nodes or edges are disjoint. *i.e.* $\forall 1 \leq x < y \leq n$ and $\forall k, Ext_{S,k}(s_x) \cap Ext_{S,k}(s_y) = \emptyset$.

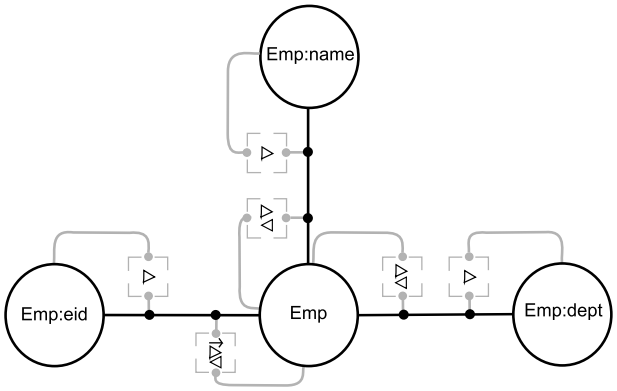


Figure 3.1: S_{eg}, an HDM schema that represents an SQL table with the columns Emp(eid,name,dept)

4. **mandatory**$(\langle s_1, \ldots, s_m \rangle, s) \equiv \langle s_1, \ldots, s_m \rangle \rhd s$: Let $1 \leq i \leq m$, then all nodes and edges s_i are connected by edge s and every combination of values in the extents of $s_1, \ldots, s_m$ must appear at least once in the extent of s. *i.e.* $\forall k$,
$\{(a_1, \ldots, a_m) \mid a_1 \in Ext_{S,k}(s_1) \wedge \ldots \wedge a_m \in Ext_{S,k}(s_m)\} -$
$\{(\pi(s_1, s, t), \ldots, \pi(s_m, s, t)) \mid t \in Ext_{S,k}(s)\} = \emptyset$

5. **unique** $(\langle s_1, \cdots, s_m \rangle, s) \equiv \langle s_1, \cdots, s_m \rangle \lhd s$: Let $1 \leq i \leq m$, then all nodes and edges s_i are connected by edge s, and no combination of values in the extents of $s_1, \ldots, s_m$ may appear more than once in the extent of s, *i.e.* $\forall k$,
$\{t \mid t \in Ext_{S,k}(s) \wedge t' \in Ext_{S,k}(s) \wedge t \neq t' \wedge$
$\pi(s_1, s, t) = \pi(s_1, s, t'), \ldots, \pi(s_m, s, t) = \pi(s_m, s, t')\} = \emptyset$

6. **reflexive**$(s_1, s) \equiv s \xrightarrow{\text{id}} s_1$: If an instance of s_1 appears in edge s, then one of those instances of s must be an identity tuple, *i.e.* $\forall k$,
$\{\pi(s_1, s, t) \mid t \in Ext_{S,k}(s)\} -$
$\{\pi(s_1, s, t) \mid t \in Ext_{S,k}(s) \wedge t = (\pi(s_1, s, t), \pi(s_1, s, t))\} = \emptyset$

□

The constraints defined above are used to determine which possible instances, k, of schema S are valid. For example

node:⟨⟨Emp:dept⟩⟩ ▷ edge:⟨⟨_, Emp, Emp:dept⟩⟩ holds for Example 3.2 since every value that appears in node:⟨⟨Emp:dept⟩⟩ also appears in edge:⟨⟨_, Emp, Emp:dept⟩⟩.

Example 3.3 is an HDM schema with the above constraint added as well as a number of others. Figure 3.1 is a graphical representation of this schema. We can see by inspection that the HDM instance in Example 3.2 satisfies all these constraints.

Example 3.3 An HDM Schema with Constraints

Let $S_{eg} = \langle Nodes, Edges, Cons \rangle$ such that

$Nodes = \{\text{node:}\langle\langle \text{Emp} \rangle\rangle, \text{node:}\langle\langle \text{Emp:eid} \rangle\rangle, \text{node:}\langle\langle \text{Emp:name} \rangle\rangle, \text{node:}\langle\langle \text{Emp:dept} \rangle\rangle\}$

$Edges = \{\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle, \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle, \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle\}$

$Cons = \{\text{node:}\langle\langle \text{Emp:dept} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \lhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp:name} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \lhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp:eid} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \lhd \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle,$

$\text{node:}\langle\langle \text{Emp} \rangle\rangle \xrightarrow{\text{id}} \text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle\}$

□

The constraints described above can be used in combinations to model different types of high level DDL constraint such as keys and cardinality constraints. For example, in S_{eg}, there are mandatory and unique constraints from $\text{node:}\langle\langle \text{Emp} \rangle\rangle$ to $\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle$, $\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:dept} \rangle\rangle$ and $\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:name} \rangle\rangle$. This means that each value of $\text{node:}\langle\langle \text{Emp} \rangle\rangle$ must appear exactly once in each of the edges. When we come to describe how SQL schemas can be translated into HDM we will show that this combination of constraints together with the mandatory constraints from $\text{node:}\langle\langle \text{Emp:eid} \rangle\rangle$, $\text{node:}\langle\langle \text{Emp:dept} \rangle\rangle$ and $\text{node:}\langle\langle \text{Emp:name} \rangle\rangle$ to their respective edges, is equivalent to the SQL constraint on a non nullable column. The extra reflexive constraint to $\text{edge:}\langle\langle \_, \text{Emp}, \text{Emp:eid} \rangle\rangle$ models the fact that the values in $\text{node:}\langle\langle \text{Emp:eid} \rangle\rangle$ match those in $\text{node:}\langle\langle \text{Emp} \rangle\rangle$. This models the concept of a primary key constraint.

3.2 Representing High Level DDLs in AutoMed

To represent a high level DDL in AutoMed we need to create a wrapper [BKL+04] that describes how schemas from the DDL should be represented and processed in our system. These include information about the constructs of the DDL, the

primitive data types and a classification for the DDL which we use when we do inter DDL translation.

In general, the constructs of any DDL can be divided into **extensional** constructs, *i.e.* those that represent sets of data vales from a certain domain, and **constraint** constructs that represent restrictions on the extensional constructs. We divide the extensional constructs of the high level DDL into three classes [MP99]:

- **Nodal**: These may be present independently of any other constructs. An ER entity is an example of a nodal construct as it can be present without requiring the presence of any other particular constructs. The extent of a nodal construct is a simple set of values. The **scheme** of a nodal construct contains the name of an HDM node used to represent it.
- **Link**: These link two other constructs and cannot exist in isolation. An ER relationship is an example of a link construct as it links entities and cannot exist on its own. The extent of a link construct is a set of tuples that form a subset of the cross product of the extents of the constructs it links. The first element in the scheme of a link construct is the name of the HDM edge representing the construct followed by the names of the HDM constructs this edge links and optionally some constraints. These constraints may appear anywhere after the first element of the scheme.
- **Link-Nodal**: These are nodal constructs that can only exist if they are linked to a parent construct. They are represented in the HDM by an edge associating a new node with some existing node or edge. A column in SQL is an example of a link-nodal construct. The extent of a link-nodal construct is a set of binary tuples whose first value comes from the existing node or edge and second from the newly created node. The first element in the scheme of a link nodal construct is the name of the existing HDM node or edge and the second is the name of the high level link nodal object. The name of the new HDM node created is the name of this object prefixed by a colon and the name of the existing HDM node or edge. There may be additional constraints in the scheme.

The production rules defined below describe how each high level construct is translated into an equivalent set of HDM constructs [BM05].

Definition 3.5 HDM Production Rules

HDM production rules take the following form:

$\langle$high level construct$\rangle$ $\langle$high level construct scheme$\rangle(\vec{a}) \rightsquigarrow \langle$HDM scheme$\rangle(a_i)$*
$\langle$condition$\rangle_1 \Rightarrow \langle$HDM constraint$\rangle_1$*$\langle$HDMscheme$\rangle_1$*
$\vdots$
$\langle$condition$\rangle_n \Rightarrow \langle$HDM constraint$\rangle_n$*$\langle$HDMscheme$\rangle_n$*

Where

- $\langle$high level construct$\rangle$ is the name of the construct in the high level DDL.
- $\langle$high level construct scheme$\rangle$ is the scheme of the high level construct.
- $(\vec{a})$ is a vector of variables representing the extent of the construct.
- $\langle$HDMscheme$\rangle(a_i)$* is a list of 0 or more HDM schemes used to represent those aspects of the high level construct that have an extent. Zero such schemes will be denoted by $\perp$.
- $\langle$condition$\rangle$ is a list of 0 or more boolean expressions over elements of $\langle$high level construct scheme$\rangle$. For each $1 \leq i \leq n$, if the $\langle$condition$\rangle_i$ is satisfied then $\langle$HDM constraint$\rangle_i$* and $\langle$HDMscheme$\rangle_i$* are added to the HDM schema.

□

The AutoMed representation of any typed high level DDL will also include a set of primitive data type names that correspond to the data types used in the DDL. The extents of the individuals in this set are allowable values for the primitive data type as defined by the data source. For example, the extent of $\mathsf{short}_{\mathsf{xml}}$ will be that defined by the XML Schema standard in [BM04], *i.e.* the integers from -32768 to 32767.

In the following two sections we review the way SQL and the ER model have been represented in the HDM [BM05]. Then in Sections 3.2.3 and 3.2.4 we present new work that shows how selected constructs in XML Schema and RDFS can be represented in HDM.

Emp

eid	name	dept
1	Peter Smith	100
14	Catherine Thomas	101
21	Susan Brown	101

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15
102	IT	

Emp.dept → Dept.did

Figure 3.2: $\mathsf{Inst}_4(S_{emp})$

3.2.1 SQL

In this section, we review previous work by other members of the AutoMed group that shows how SQL constructs can be defined in terms of HDM constructs [BM05]. We use the database shown in Figure 3.2 to exemplify the discussion.

An SQL **table** is a nodal construct as it can exist independently of any other schema objects. We define the extent of a table to be the extent of its primary key column(s). It is represented as an HDM node. The scheme is simply the name of the table. The production rule is as follows:

$$\textit{nodal}\ :\mathsf{table}{:}\langle\langle T\rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle T\rangle\rangle(x)$$

The extent of the HDM node is the extent of the primary key of the table and the node is untyped. The results of applying the production rule to the tables from Figure 3.2 are as follows:

$$\mathsf{sql{:}table}{:}\langle\langle \mathsf{Emp}\rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle \mathsf{Emp}\rangle\rangle(x)$$

$$\mathsf{sql{:}table}{:}\langle\langle \mathsf{Dept}\rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle \mathsf{Dept}\rangle\rangle(x)$$

An SQL **column** cannot exist independently of its table and so is a link nodal construct represented in the HDM by a node and edge. The edge links the column node to the node representing the table the column is part of. The scheme for a column has four components: first, the name of the table the column is part of, second, a name for the column, third, the data type of the column and finally a constraint specifying whether or not the column accepts null values. The production rule is as follows:

$$\textit{link-nodal}\ \mathsf{sql{:}column}{:}\langle\langle T, C, D, N\rangle\rangle(x, y) \rightsquigarrow$$

$$\begin{array}{ll} & \mathsf{node}{:}\langle\langle T{:}C, typeTrans(D, Types_c)\rangle\rangle(y), \mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle(x, y) \\ \mathsf{true} & \Rightarrow \mathsf{node}{:}\langle\langle T{:}C\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle, \mathsf{node}{:}\langle\langle T\rangle\rangle \lhd \mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle \\ N = \mathsf{notnull} & \Rightarrow \mathsf{node}{:}\langle\langle T\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle \end{array}$$

The $typeTrans(D, Types_c)$ function calculates the HDM equivalent of the SQL data type, D. The method for working this out is described in detail in Chapter 4. The key scheme of this construct is $\mathsf{sql{:}column}{:}\langle\langle T, C\rangle\rangle$. The second and third lines of the production rule add constraints to the HDM graph. The second line adds mandatory and unique constraints that constrain each instance of $\mathsf{node}{:}\langle\langle T{:}C\rangle\rangle$ to be associated with an instance of $\mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle$ and each instance of $\mathsf{node}{:}\langle\langle T\rangle\rangle$ to be associated with at most one instance of $\mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle$. The third line states that if the column is defined as notnull then for each value in $\mathsf{node}{:}\langle\langle T\rangle\rangle$ (*i.e.* for each value of the primary key of the table) there must be an associated value in $\mathsf{edge}{:}\langle\langle \_, T, T{:}C\rangle\rangle$.

When applied to $\mathsf{sql{:}column}{:}\langle\langle \mathsf{Emp}, \mathsf{name}, \mathsf{varchar}, \mathsf{notnull}\rangle\rangle$ in Figure 3.2, the production rule above generates the following expansion:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{sql{:}column}{:}\langle\langle \mathsf{Emp}, \mathsf{name}, \mathsf{varchar}, \mathsf{notnull}\rangle\rangle \rightsquigarrow} \\ & \mathsf{node}{:}\langle\langle \mathsf{Emp{:}name}, \mathit{typeTrans}(\mathsf{varchar}, \mathsf{Types_c})\rangle\rangle, \mathsf{edge}{:}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}name}\rangle\rangle \\ \mathsf{true} & \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp{:}name}\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}eid}\rangle\rangle \\ & \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp}\rangle\rangle \lhd \mathsf{edge}{:}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}eid}\rangle\rangle \\ N = \mathsf{notnull} & \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp}\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}eid}\rangle\rangle \end{array}$$

$typeTrans(\mathsf{varchar}, Types_c)$ returns string so the new node generated by this rule is $\mathsf{node}{:}\langle\langle \mathsf{Emp{:}name}, \mathsf{string}\rangle\rangle$. If we let $I_4 = \mathsf{Inst}_4(S_{emp})$ the extent of $\mathsf{column}{:}\langle\langle \mathsf{Emp}, \mathsf{name}\rangle\rangle$ is {(1,'Peter Smith'),(21,'Susan Brown'),(14,'Catherine Thomas')}

and that of the constructs generated by the rule above are:

$$\begin{array}{l} Ext_{S_{emp-hdm}, I_4}(\mathsf{node}{:}\langle\langle \mathsf{Emp{:}name}\rangle\rangle) = \\ \quad \{(\text{'Peter Smith'}), (\text{'Susan Brown'}), (\text{ 'Catherine Thomas'})\}, \\ Ext_{S_{emp-hdm}, I_4}(\mathsf{edge}{:}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}name}\rangle\rangle) = \\ \quad \{(1, \text{'Peter Smith'}), (21, \text{'Susan Brown'}), (14, \text{'Catherine Thomas'})\} \end{array}$$

If we a applied the rule to $\mathsf{column}{:}\langle\langle \mathsf{Dept}, \mathsf{numEmps}\rangle\rangle$ we would not get a mandatory constraint from $\mathsf{node}{:}\langle\langle \mathsf{Dept}\rangle\rangle$ to $\mathsf{edge}{:}\langle\langle \_, \mathsf{Dept}, \mathsf{Dept{:}numEmps}\rangle\rangle$ because the column is nullable.

The **primary key** construct is a constraint construct and so has no extent. Primary keys may be made up of more than one column. In order to correctly model these

compound keys, it is necessary to use an **edge natural join** [BM05], defined in Definition 3.6.

Definition 3.6 The edge natural join

A view over HDM edges may be formed by joining edges together to form a new *virtual* edge:

$$\langle\langle E, A, B\rangle\rangle(x, y) \bowtie \langle\langle E, A, C\rangle\rangle(x, z) = \langle\langle E, A, B, C\rangle\rangle(x, y, z)$$

□

Using the definition above, the production rule for a primary key is as follows:

$$\begin{array}{l} \textit{constraint}\ \mathsf{sql{:}primary\_key{:}}\langle\langle PK, T, C_1, \cdots, C_n\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \mathsf{node{:}}\langle\langle T\rangle\rangle \xrightarrow{\text{id}} \mathsf{edge{:}}\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \cdots \bowtie \mathsf{edge{:}}\langle\langle \_, T, T{:}C_n\rangle\rangle \end{array}$$

PK is the name of the primary key, T is the table this is the key of and $C_1 \cdots C_n$ are the columns that make up the key. This construct has no extent, so no HDM extensional constructs are produced. The reflexive constraint, when combined with unique and mandatory constraints entails that the extent of the key column(s) gives the extent of the table.

The expansion of the production rule when applied to the primary key on the $\mathsf{sql{:}column{:}}\langle\langle \mathsf{Emp}, \mathsf{eid}\rangle\rangle$ in Figure 3.2 is

$$\begin{array}{l} \mathsf{sql{:}primary\_key{:}}\langle\langle \mathsf{Emp\_pk}, \langle\langle \mathsf{Emp}, \mathsf{eid}\rangle\rangle\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \mathsf{node{:}}\langle\langle \mathsf{Emp}\rangle\rangle \xrightarrow{\text{id}} \mathsf{edge{:}}\langle\langle \_, \mathsf{Emp}, \mathsf{Emp{:}eid}\rangle\rangle \end{array}$$

A **foreign key** is also a constraint construct. The production rule to produce the equivalent HDM constraints is as follows. Note that we use π here in the conventional relational algebra manner:

$$\begin{array}{l} \textit{constraint}\ \mathsf{sql{:}foreign\_key{:}}\langle\langle FK, T, C_1, \ldots, C_n, T_f, C_{f_1}, \ldots, C_{f_n}\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \pi_{\mathsf{node{:}}\langle\langle T:C_1\rangle\rangle,\ldots,\mathsf{node{:}}\langle\langle T:C_n\rangle\rangle}(\mathsf{edge{:}}\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie, \ldots, \bowtie \mathsf{edge{:}}\langle\langle \_, T, T{:}C_n\rangle\rangle) \subseteq \\ \qquad\qquad \pi_{\mathsf{node{:}}\langle\langle T_f:C_{f_1}\rangle\rangle,\ldots,\mathsf{node{:}}\langle\langle T_f:C_{f_n}\rangle\rangle}(\mathsf{edge{:}}\langle\langle \_, T_f, T_f{:}C_{f_1}\rangle\rangle \bowtie, \ldots, \bowtie \mathsf{edge{:}}\langle\langle \_, T_f, T_f{:}C_{f_n}\rangle\rangle) \end{array}$$

FK is the name of the foreign key, T is the table the foreign key is from, $C_1, \ldots, C_n$ are the columns making up the foreign key, while T_f and $C_{f_1}, \ldots, C_{f_n}$ are the table

and columns the foreign key references. In the common case where the foreign keys are not compound keys, *i.e.* $n = 1$, the constraint simplifies to $\mathsf{node}{:}\langle\langle T{:}C_1\rangle\rangle \subseteq \mathsf{node}{:}\langle\langle T_f{:}C_{f_1}\rangle\rangle$. The result of applying the production rule to the foreign key linking $\mathsf{sql{:}column}{:}\langle\langle \mathsf{Emp}, \mathsf{dept}\rangle\rangle$ in to $\mathsf{sql{:}column}{:}\langle\langle \mathsf{Dept}, \mathsf{did}\rangle\rangle$ in Figure 3.2 is shown below:

$$\mathsf{sql{:}foreign\_key}{:}\langle\langle \mathsf{Dept\_fk}, \mathsf{Emp}, \langle\langle \mathsf{Emp}, \mathsf{dept}\rangle\rangle, \mathsf{Dept}, \langle\langle \mathsf{Dept}, \mathsf{did}\rangle\rangle\rangle\rangle \rightsquigarrow \bot$$
$$\mathsf{true} \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp{:}dept}\rangle\rangle \subseteq \mathsf{node}{:}\langle\langle \mathsf{Dept{:}did}\rangle\rangle$$

We can now fully represent Figure 3.2 in terms of the HDM. Figure 3.3 shows the resulting graph.

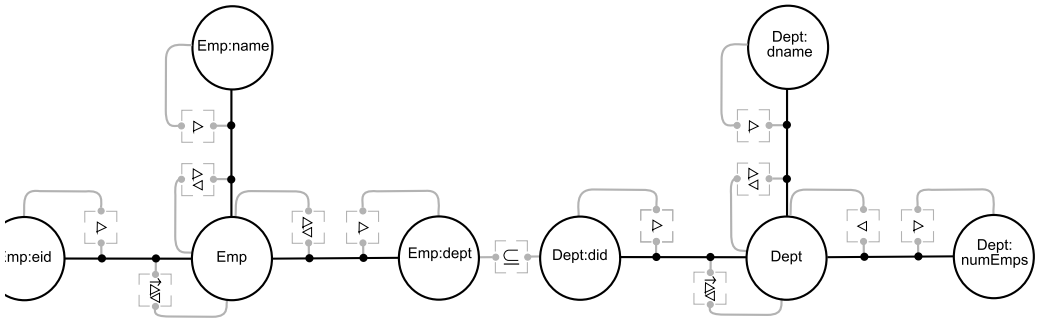


Figure 3.3: The SQL schema from Figure 3.2 as represented in the HDM

3.2.2 An ER Modelling Language

In this section we review how other members of the AutoMed group showed how a typed, extensional ER model can be represented in the HDM [BM05] (see [Pat04, IYEP95] for surveys of variations of ER modelling languages). We will assume that the types are those in the SQL 2003 standard [EMK+04]. We will use Figure 3.4 to help describe this process.

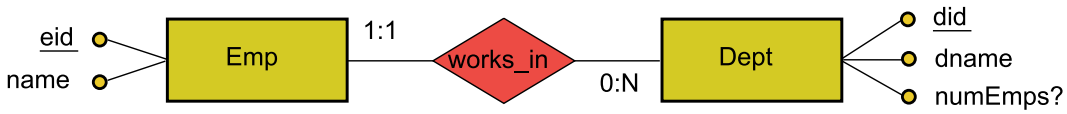


Figure 3.4: An ER schema

An ER **entity** is a nodal construct. Its scheme is simply the name of the entity. It is represented as an HDM node with the same name as the entity. The production rule is as follows:

$$\textit{nodal}\ \mathsf{ER{:}entity}{:}\langle\langle E\rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle E\rangle\rangle(x)$$

As with SQL, the extent of an entity is that of the key attribute of that entity. The entities in Figure 3.4 are transformed into HDM constructs using the production rules shown below:

$$\mathsf{ER{:}entity}{:}\langle\langle \mathsf{Emp} \rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle \mathsf{Emp} \rangle\rangle(x)$$

$$\mathsf{ER{:}entity}{:}\langle\langle \mathsf{Dept} \rangle\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\langle \mathsf{Dept} \rangle\rangle(x)$$

An ER **attribute** is a link-nodal construct. The scheme includes the entity the attribute is associated with, a name for the attribute, and a constraint specifying whether or not the attribute accepts null values or whether it is a key attribute. It is represented in the HDM by a node and an edge. The edge links the attribute node to the node representing the entity the attribute is part of. The production rule is as follows:

$$\begin{array}{l}
\textit{link-nodal}\ \mathsf{ER{:}attribute}{:}\langle\langle E, A, D, C \rangle\rangle(x, y) \rightsquigarrow \\
\qquad \mathsf{node}{:}\langle\langle E{:}A, typeTrans(D, Types_c) \rangle\rangle(y), \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle(x, y) \\
\quad \mathsf{true} \qquad \Rightarrow \mathsf{node}{:}\langle\langle E{:}A \rangle\rangle \rhd \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle, \mathsf{node}{:}\langle\langle E \rangle\rangle \lhd \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle \\
\quad C = \mathsf{notnull} \quad \Rightarrow \mathsf{node}{:}\langle\langle E \rangle\rangle \rhd \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle \\
\quad C = \mathsf{key} \qquad \Rightarrow \mathsf{node}{:}\langle\langle E \rangle\rangle \rhd \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle, \mathsf{node}{:}\langle\langle E \rangle\rangle \xrightarrow{\text{id}} \mathsf{edge}{:}\langle\langle _, E, E{:}A \rangle\rangle
\end{array}$$

This production rule is similar to that for an SQL column, however we include the key constraint as part of the attribute definition. When applied to $\mathsf{ER{:}attribute}{:}\langle\langle \mathsf{Dept}, \mathsf{dname} \rangle\rangle$ in Figure 3.4 the production rule generates:

$$\begin{array}{l}
\mathsf{ER{:}attribute}{:}\langle\langle \mathsf{Dept}, \mathsf{dname}, \mathsf{varchar}, \mathsf{notnull} \rangle\rangle(x, y) \rightsquigarrow \\
\qquad \mathsf{node}{:}\langle\langle \mathsf{Dept{:}dname}, typeTrans(\mathsf{varchar}, \mathsf{Types_c}) \rangle\rangle(y), \mathsf{edge}{:}\langle\langle _, \mathsf{Dept}, \mathsf{Dept{:}dname} \rangle\rangle(x, y) \\
\quad \mathsf{true} \quad \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Dept{:}dname} \rangle\rangle \rhd \mathsf{edge}{:}\langle\langle _, \mathsf{Dept}, \mathsf{Dept{:}dname} \rangle\rangle \\
\quad \mathsf{notnull} \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Dept} \rangle\rangle \lhd \mathsf{edge}{:}\langle\langle _, \mathsf{Dept}, \mathsf{Dept{:}dname} \rangle\rangle \\
\qquad \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Dept} \rangle\rangle \rhd \mathsf{edge}{:}\langle\langle _, \mathsf{Dept}, \mathsf{Dept{:}dname} \rangle\rangle
\end{array}$$

As in the example above, the $typeTrans$ function returns string.

An ER **relationship** construct like works_in,Emp,Dept in Figure 3.4 is a link construct. The scheme comprises a name for the relationship as well as a name and cardinality constraint for each of the entities the relationship associates. The production rule is as follows:

$$\begin{aligned}
&\mathsf{ER{:}relationship}{:}\langle\langle R, E_1, L_1{:}U_1, \ldots, E_n, L_n{:}U_n\rangle\rangle(\vec{x_n}) \rightsquigarrow \mathsf{edge}{:}\langle\langle R, E_1, \cdots, E_n\rangle\rangle(\vec{x_n})\\
&\quad L_1 = 1 \Rightarrow \mathsf{node}{:}\langle\langle E_1\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle R, E_1, \ldots, E_n\rangle\rangle\\
&\quad U_1 = 1 \Rightarrow \mathsf{node}{:}\langle\langle E_1\rangle\rangle \lhd \mathsf{edge}{:}\langle\langle R, E_1, \ldots, E_n\rangle\rangle\\
&\qquad\quad \Rightarrow \vdots\\
&\quad L_n = 1 \Rightarrow \mathsf{node}{:}\langle\langle E_n\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle R, E_1, \ldots, E_n\rangle\rangle\\
&\quad U_n = 1 \Rightarrow \mathsf{node}{:}\langle\langle E_n\rangle\rangle \lhd \mathsf{edge}{:}\langle\langle R, E_1, \ldots, E_n\rangle\rangle
\end{aligned}$$

The HDM schema objects created by applying the production rule to $\mathsf{ER{:}relationship}{:}\langle\langle \mathsf{works\_in}, \mathsf{Emp}, \mathsf{Dept}\rangle\rangle$ are shown below.

$$\begin{aligned}
&\mathsf{ER{:}relationship}{:}\langle\langle \mathsf{works_in}, \mathsf{Emp}, \mathsf{1{:}1}, \mathsf{Dept}, \mathsf{1{:}N}\rangle\rangle(x_1, x_2) \rightsquigarrow \mathsf{edge}{:}\langle\langle \mathsf{works_in}, \mathsf{Emp}, \mathsf{Dept}\rangle\rangle(x_1, x_2)\\
&\quad L_1 = 1 \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp}\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \mathsf{works_in}, \mathsf{Emp}, \mathsf{Dept}\rangle\rangle\\
&\quad L_2 = 1 \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Dept}\rangle\rangle \rhd \mathsf{edge}{:}\langle\langle \mathsf{works_in}, \mathsf{Emp}, \mathsf{Dept}\rangle\rangle\\
&\quad U_1 = 1 \Rightarrow \mathsf{node}{:}\langle\langle \mathsf{Emp}\rangle\rangle \lhd \mathsf{edge}{:}\langle\langle \mathsf{works_in}, \mathsf{Emp}, \mathsf{Dept}\rangle\rangle
\end{aligned}$$

The final ER construct we encounter in Figure 3.4 is the **key** construct. This is a constraint construct and is very similar to a relational primary key as we can see from the production rule, a difference being that the schema for an ER key does not include a name.

$$\begin{aligned}
&\textit{constraint}\ \mathsf{ER{:}key}{:}\langle\langle E, A_1, \cdots, A_n\rangle\rangle \rightsquigarrow \bot\\
&\quad \mathsf{true} \Rightarrow \mathsf{node}{:}\langle\langle E\rangle\rangle \xrightarrow{\text{id}} \mathsf{edge}{:}\langle\langle _, E, E{:}A_1\rangle\rangle \bowtie \ldots \bowtie \mathsf{edge}{:}\langle\langle _, E, E{:}A_n\rangle\rangle
\end{aligned}$$

Using the rules described above we can translate the ER model in Figure 3.4 into the HDM graph shown in Figure 3.5.

A subset constraint states that one entity is a subset of another entity. The production rule is as follows:

$$\begin{aligned}
&\textit{constraint}\ \mathsf{ER{:}subset}{:}\langle\langle E, E_s\rangle\rangle \rightsquigarrow \bot\\
&\quad \mathsf{true} \Rightarrow \mathsf{node}{:}\langle\langle E_s\rangle\rangle \subseteq \mathsf{node}{:}\langle\langle E\rangle\rangle
\end{aligned}$$

A generalisation forms a containment relationship between an entity and one or more other entities. In other words these other entities form disjoint subsets of the first entity. The production rule for a generalisation is as follows:

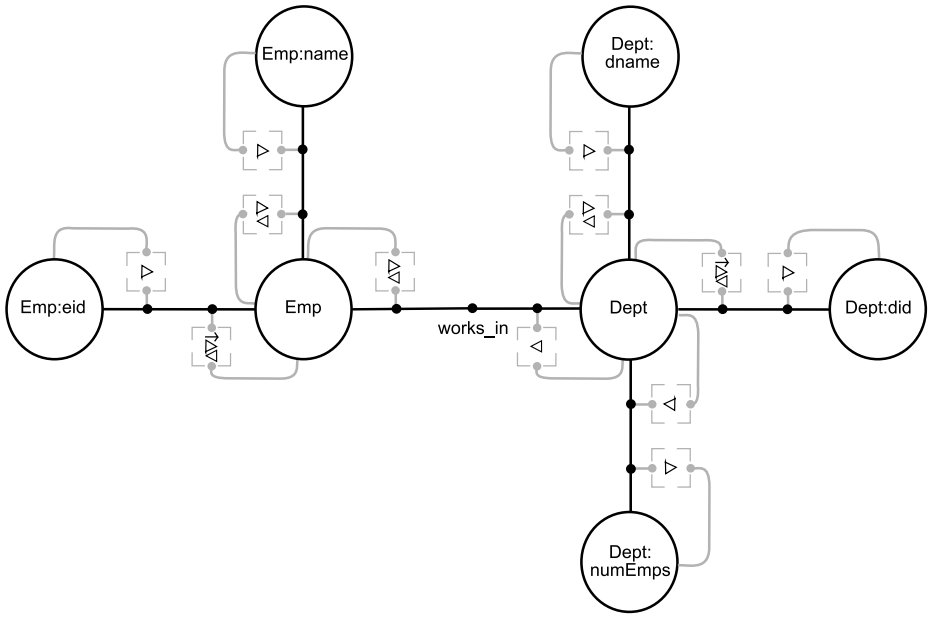


Figure 3.5: The ER schema from Figure 3.4 as represented in the HDM

constraint ER:generalisation:$\langle\langle E, E_1, \cdots E_n \rangle\rangle \rightsquigarrow \bot$

$$\begin{aligned}
\text{true} &\Rightarrow \text{node}{:}\langle\langle E_1 \rangle\rangle \subseteq \text{node}{:}\langle\langle Emp \rangle\rangle \\
\text{true} &\Rightarrow \vdots \\
\text{true} &\Rightarrow \text{node}{:}\langle\langle E_n \rangle\rangle \subseteq \text{node}{:}\langle\langle Emp \rangle\rangle \\
\text{true} &\Rightarrow \text{node}{:}\langle\langle E_1 \rangle\rangle \not\cap \cdots \not\cap \text{node}{:}\langle\langle E_n \rangle\rangle \\
\text{true} &\Rightarrow \text{node}{:}\langle\langle E \rangle\rangle = \text{node}{:}\langle\langle E_1 \rangle\rangle \cup \cdots \cup \text{node}{:}\langle\langle E_n \rangle\rangle
\end{aligned}$$

3.2.3 Selected XML Schema Constructs

In this section we present new work that describes how we use AutoMed to wrap XML documents constrained by a schema expressed in the XML Schema DDL [2] [TBMM01]. In common with other MMSs that can process XML [MRB03, KQLL07], we do not model all the XML Schema constructs. We have chosen constructs that represent the main extensional and constraint constructs of XML Schema. They are `element, attribute, complexType, all, key` and `keyref`.

We will use the XML document and schema shown in Figure 3.6 as our example in this section. The AutoMed representation of the schema is shown in Figure 3.7. The schema S_{xml} allows either `<staff>` or `<dummy>` to be root elements, however, as

[2]Note that when we talk about XML Schema, with a uppercase S, we are referring to the DDL. An XML schema, with a lowercase s, is a specific instance of a schema expressed in the XML Schema DDL.

```
<xsd:complexType name = "emp_type">
 <xsd:all>
  <xsd:element name = "name" type = "xsd:string" />
 </xsd:all>
 <xsd:attribute name = "eid" type = "xsd:int" use = "required"/>
</xsd:complexType>
<xsd:element name = "staff">
 <xsd:complexType>
  <xsd:all>
   <xsd:element name = "dept" maxOccurs = "unbounded">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name = "dname" type = "xsd:string" />
      <xsd:element name = "numEmps" type = "xsd:string" minOccurs = "0" />
      <xsd:element name = "emp" type = "emp_type"
            minOccurs = "0" maxOccurs = "unbounded" />
     </xsd:all>
     <xsd:attribute name = "did" type = "xsd:string"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:all>
 </xsd:complexType>
 <xsd:key name = "empKey">
  <xsd:selector xpath = "staff/dept/emp" />
  <xsd:field xpath = "@eid" />
 </xsd:key>
</xsd:element>
<xsd:element name = "dummy" />

<staff> &0
  <dept did = "100"> &1
    <dname>Finance</dname> &2
    <numEmps>23</numEmps> &3
    <emp eid = "1"> &4
      <name>Peter Smith</name> &5
    </emp>
  </dept>
  <dept did = "101"> &6
    <dname>HR</dname> &7
    <dname>Human Resources</dname> &8
    <numEmps>15</numEmps> &9
    <emp eid = "21"> &10
      <name>Susan Brown</name> &11
    </emp>
    <emp eid = "14"> &12
      <name>Catherine Thomas</name> &13
    </emp>
  </dept>
  <dept did = "102"> &14
    <dname>IT</dname> &15
  </dept>
</staff>
```

Figure 3.6: The schema S_{xml} (top) and an instance $\mathsf{Inst}_1(S_{xml})$ (bottom)

we are wrapping the instance document and it has `<staff>` as its root element, we use this in our schema representation. The empty circles in the figure are attributes, the filled in circles with arrows attached to them are complex elements and the filled in circles attached to lines without arrows are simple elements. The dashed rectangle is a key constraint. We will explain what we mean by complex and simple elements below.

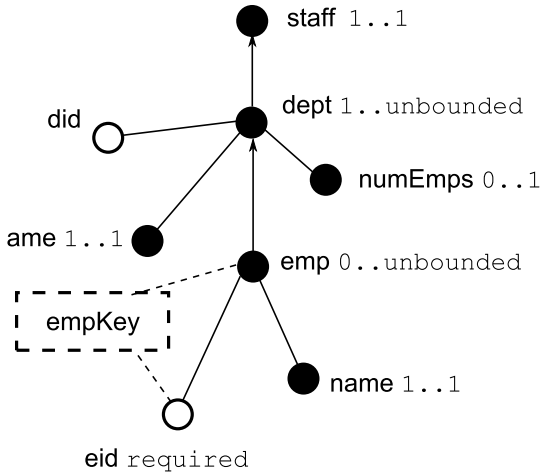


Figure 3.7: The graphical AutoMed representation of S_{xml}

We generate the necessary AutoMed schema objects by executing Algorithm 3.1. It performs a depth first traversal of the XML instance document, starting with the root element, and creates AutoMed objects based on the schema associated with the XML instance document. It takes three parameters, an XML element P which will initially be null, the XML document X we are wrapping, and an AutoMed schema S that will initially be empty.

Lines 11, 18 and 24 all add objects to the AutoMed schema. The precise objects they add are described in the production rule definitions given in the rest of this section (the descriptions being an extension of the definitions given in [MP01]).

Each XML element has a unique object identifier (OID) associated with it, corresponding to its place in its XML instance document. In the bottom part of Figure 3.6, these OIDs are shown next to the elements. The function $getOID(e)$ returns the set of OIDs associated with the element e. For example

$$getOID(\mathsf{complexElement}{:}\langle\langle \mathsf{staff}, \mathsf{dept}\rangle\rangle)$$

will return {`&1,&6,&14`}. Note that we preface each OID with a & to differentiate these values from data values.

Algorithm 3.1: xml_to_automed(Element P, Document$X$, Schema S)

Input: Element P, Document X, Schema S
Output: The AutoMed schema

```
1  if P is null then
2      Let E[1] := X.root;
3      Let n := 1;
4  else
5      Let E[n] be an array of the child elements of P;
6  for i := 1 to n do
7      Let e := E[i];
8      Let L be the value for the minOccurs attribute of e;
9      Let U be the value for the maxOccurs attribute of e;
10     if e is an element of complexType then
11         S := S ∪ complexElement:⟨⟨P, e, L, U⟩⟩;
12         if P is null then
13             xml_to_automed(e, X, S);
14         else
15             xml_to_automed(P/e, X, S);
16     else if e is an element of simpleType then
17         Let D be the data type of e;
18         S := S ∪ simpleElement:⟨⟨P, e, D, L, U⟩⟩;
19 Let A[m] be an array of all the attributes associated with P;
20 for i:=1 to m do
21     Let a := A[i];
22     Let D be the data type of a;
23     Let U be the value for the required attribute of a;
24     S := S ∪ attribute:⟨⟨P, a, D, U⟩⟩;
25 return S;
```

The exact representation of an XML element in the AutoMed will vary according to its type. The first type of element we discuss is the complexElement. We use the term complexElement to describe an element of schema definition is of `complexType`. We do this because we are modelling the *elements* that have been defined to be of `complexType` rather than the type itself. Complex types are used to define the hierarchical structure of an XML document. There are two variations of the construct: named `complexType`s, such as emp_type in Figure 3.6, and unnamed `complexType`s. The `complexType` definition under `dept` is an unnamed `complexType`.

A complexElement is written as complexElement:$\langle\langle P, E, L, U\rangle\rangle$, where P is the XPath expression of the parent schema object or null if E is the root element. In the following production rules we use the function $sc(P)$ to translate any slashes in the XPath expression of the parent schema object, P, into colons. We do this to maintain consistency with the naming conventions of the other DDLs in AutoMed. The production rule for complexElement whose parent element is not null is as follows:

link-nodal xml:complexElement:$\langle\langle P, E, L, U\rangle\rangle(x, y) \wedge P$ is not null $\rightsquigarrow$

$$
\begin{array}{ll}
 & \text{node}{:}\langle\langle sc(P){:}E\rangle\rangle(y), \text{edge}{:}\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle(x, y) \\
\text{true} & \Rightarrow \text{node}{:}\langle\langle sc(P){:}E\rangle\rangle \rhd \text{edge}{:}\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle, \\
 & \Rightarrow \text{node}{:}\langle\langle sc(P){:}E\rangle\rangle \lhd \text{edge}{:}\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle \\
(L \geq 1) & \Rightarrow \text{node}{:}\langle\langle sc(P)\rangle\rangle \rhd \text{edge}{:}\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle \\
(U \leq 1) & \Rightarrow \text{node}{:}\langle\langle sc(P)\rangle\rangle \lhd \text{edge}{:}\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle
\end{array}
$$

This type of complexElement is represented in the HDM by an untyped node and an edge. The remaining lines of the production rule give various cases by which additional structures are added to the HDM schema depending on the exact definition of the complex element. The second line generates mandatory and unique constraints because each instance of node:$\langle\langle P{:}E\rangle\rangle$ must be associated to exactly one instance of node:$\langle\langle P\rangle\rangle$, *i.e.* each instance of a complex element must be associated with exactly one instance of the parent element. The third and fourth lines determine if a mandatory or unique constraint should be added to the HDM depending on the cardinality of the element. These are defined by the values of `minOccurs` and `maxOccurs`.

`<dept>` in Figure 3.6 is a complexElement whose parent is `<staff>`. The result of applying the production rule to it is shown below:

$$
\begin{array}{l}
\text{xml:complexElement}{:}\langle\langle \text{staff}, \text{dept}, 1, \text{unbounded}\rangle\rangle(x, y) \rightsquigarrow \\
\quad \text{node}{:}\langle\langle \text{staff:dept}\rangle\rangle(y), \text{edge}{:}\langle\langle \_, \text{staff}, \text{staff:dept}\rangle\rangle(x, y)
\end{array}
$$

$$\begin{array}{ll} \text{true} & \Rightarrow \text{node:}\langle\langle \text{staff:dept} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{staff}, \text{staff:dept} \rangle\rangle \\ & \Rightarrow \text{node:}\langle\langle \text{staff:dept} \rangle\rangle \lhd \text{edge:}\langle\langle \_, \text{staff}, \text{staff:dept} \rangle\rangle \\ L = 1 & \Rightarrow \text{node:}\langle\langle \text{staff} \rangle\rangle \rhd \text{edge:}\langle\langle \_, \text{staff}, \text{staff:dept} \rangle\rangle \end{array}$$

The extent of complexElement:⟨⟨ce⟩⟩ is some set of made up values that allow us to uniquely identify the position of a given instance of the object. Its exact value depends on whether there is a key associated with the complexElement we are modelling or not. In AutoMed we define this value to be a binary tuple whose first element is the extent of the parent schema object and the second is either given by the function getOID(complexElement:⟨⟨ce⟩⟩) described above, if there is no key, or the value of the key object if there is. For example, the extent of complexElement:⟨⟨staff, dept⟩⟩ which does not have a key, is {(&0,&1),(&0,&6),(&0,&14)} making the extent of node:⟨⟨staff:dept⟩⟩ {(&1),(&6),(&14)} and of edge:⟨⟨_, staff:dept, staff:dept:employee⟩⟩ {(&0,&1),(&0,&6),(&0,&14)}. The extent of complexElement:⟨⟨staff/dept, emp⟩⟩ which does have a key associated with it is {(&1,1),(&6,21),(&6,14)}

If the root element of the XML instance document we are wrapping is a complexElement, as is the case in our example, it is represented by a single untyped HDM node with the same name. The cardinality of the root element is always 1..1. The production rule is as follows:

link-nodal xml:complexElement:⟨⟨*null,E,1,1*⟩⟩$(x, y) \rightsquigarrow$ node:⟨⟨E⟩⟩(y)

The result of applying the production rule above to the root element of $\text{Inst}_1(S_{xml})$ is shown below:

$$\text{xml:complexElement:}\langle\langle \text{null}, \text{staff}, 1, 1 \rangle\rangle \rightsquigarrow \text{node:}\langle\langle \text{staff} \rangle\rangle$$

The extent of the root element will always be the OID &0, so here the extent of node:⟨⟨staff⟩⟩ is &0.

We call an element, that is not of complex type, a simpleElement. Each simpleElement that is not the root element is nested inside a parent complexElement. The production rule is as follows:

link-nodal xml:simpleElement:⟨⟨P, E, D, L, U⟩⟩$(x, y) \wedge P$ is not null $\rightsquigarrow$

$$\begin{array}{ll} & \text{node:}\langle\langle sc(P)\text{:}E, typeTrans(D, Types_c) \rangle\rangle(y), \text{edge:}\langle\langle \_, sc(P), sc(P)\text{:}E \rangle\rangle(x, y) \\ \text{true} & \Rightarrow \text{node:}\langle\langle sc(P)\text{:}E \rangle\rangle \rhd \text{edge:}\langle\langle \_, sc(P), sc(P)\text{:}E \rangle\rangle \\ (L \geq 1) & \Rightarrow \text{node:}\langle\langle sc(P) \rangle\rangle \rhd \text{edge:}\langle\langle \_, sc(P), sc(P)\text{:}E \rangle\rangle \\ (U \leq 1) & \Rightarrow \text{node:}\langle\langle sc(P) \rangle\rangle \lhd \text{edge:}\langle\langle \_, sc(P), sc(P)\text{:}E \rangle\rangle \end{array}$$

D is the data type of the element and L and U are the values for `minOccurs` and `maxOccurs` respectively. The extent of node:$\langle\langle sc(P){:}E\rangle\rangle$ is the set of values that appear in the simpleElement. These values can be repeated so, unlike in the production rule for complexElement, there is no unique constraint between node:$\langle\langle sc(P){:}E\rangle\rangle$ and edge:$\langle\langle \_, sc(P), sc(P){:}E\rangle\rangle$. The result of applying the production rule to simpleElement:$\langle\langle$staff/dept, dname$\rangle\rangle$ in Fig. 3.6 is shown below:

$$
\begin{aligned}
&\text{xml:simpleElement}:\langle\langle \text{staff/dept}, \text{dname}, \text{string}, 1, 1\rangle\rangle(x,y) \rightsquigarrow \\
&\quad \text{node}:\langle\langle sc(\text{staff/dept}){:}\text{dname}, typeTrans(\text{xsd:string}, \text{Types}_\text{c})(y)\rangle\rangle, \\
&\quad \text{edge}:\langle\langle \_, sc(\text{staff/dept}), sc(\text{staff/dept}){:}\text{dname}\rangle\rangle(x,y) \\
&\text{true} \quad \Rightarrow \text{node}:\langle\langle sc(\text{staff/dept}){:}\text{dname}\rangle\rangle \rhd \text{edge}:\langle\langle \_, sc(\text{staff/dept}), sc(\text{staff/dept}){:}\text{dname}\rangle\rangle \\
&U = 1 \Rightarrow \text{node}:\langle\langle sc(\text{staff/dept})\rangle\rangle \lhd \text{edge}:\langle\langle \_, sc(\text{staff/dept})sc(\text{staff/dept}){:}\text{dname}\rangle\rangle
\end{aligned}
$$

The result of $typeTrans$(xsd:string,Types_c) is string. If we let $I_1 = \text{Inst}_1(S_{xml-hdm})$ where $S_{xml-hdm}$ is the schema in Figure 3.8, the extents of the HDM constructs generated by the rule above are:

$$
\begin{aligned}
&Ext_{S_{xml-hdm},I_1}(\text{node}:\langle\langle \text{staff:dept:dname}, \text{string}\rangle\rangle) = \{\texttt{(Finance),(HR)}\} \\
&Ext_{S_{xml-hdm},I_1}(\text{edge}:\langle\langle \_, \text{staff:dept}, \text{staff:dept:dname}\rangle\rangle) = \\
&\quad \{\texttt{(\&1,Finance),(\&6,HR),(\&6,Human Resources)}\}
\end{aligned}
$$

If the root element is a simpleElement we use the following production rule to translate it into the HDM:

$$\textit{link-nodal}\ \text{xml:simpleElement}:\langle\langle null,E,D,L,U\rangle\rangle(x,y) \rightsquigarrow \text{node}:\langle\langle E, typeTrans(D, Types_c)\rangle\rangle(y)$$

If the root element of the XML instance document is a simpleElement it is represented by a single typed HDM node with the same name.

XML attributes are represented in a similar manner to simpleElements. The production rule is as follows:

$$
\begin{aligned}
&\textit{link-nodal}\ \text{xml:attribute}:\langle\langle P, A, D, U\rangle\rangle(x,y) \rightsquigarrow \\
&\qquad \text{node}:\langle\langle sc(P){:}A, typeTrans(D, Types_c)\rangle\rangle(y), \text{edge}:\langle\langle \_, sc(P), sc(P){:}A\rangle\rangle(x,y) \\
&\text{true} \qquad\qquad \Rightarrow \langle\langle sc(P){:}A\rangle\rangle \rhd \langle\langle \_, sc(P), sc(P){:}A\rangle\rangle, \langle\langle sc(P)\rangle\rangle \lhd \langle\langle \_, sc(P), sc(P){:}A\rangle\rangle \\
&(U = \text{required}) \quad \Rightarrow \langle\langle sc(P)\rangle\rangle \rhd \langle\langle \_, sc(P), sc(P){:}A\rangle\rangle
\end{aligned}
$$

A is the name of the attribute and U is the XML Schema use attribute. Note that since each attribute can only have one parent instance, there is always a unique constraint between node:$\langle\langle sc(P)\rangle\rangle$ and edge:$\langle\langle \_, sc(P), sc(P){:}A\rangle\rangle$. The only variation in the mapping depends on the presence of an XML Schema `use` attribute, which if set to `required` would imply the presence of a mandatory constraint between the

HDM node representing the element, and the edge to the node. For example, the result of applying the production rule to the did attribute in Figure 3.6 is shown below

$$\begin{array}{l}\mathsf{xml{:}attribute{:}}\langle\langle \mathsf{staff/dept}, \mathsf{did}, \mathsf{xsd{:}int}, \texttt{required}\rangle\rangle(x,y) \rightsquigarrow \\ \quad \mathsf{node{:}}\langle\langle sc(\mathsf{staff/dept}){:}\mathsf{did}, typeTrans(\mathsf{xsd{:}int}, \mathsf{Types_c})(y)\rangle\rangle, \\ \quad \mathsf{edge{:}}\langle\langle \_, sc(\mathsf{staff/dept}), sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle(x,y) \\ \mathsf{true} \Rightarrow \mathsf{node{:}}\langle\langle sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle \rhd \mathsf{edge{:}}\langle\langle \_, sc(\mathsf{staff/dept}), sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle \\ \quad \Rightarrow \mathsf{node{:}}\langle\langle sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle \lhd \mathsf{edge{:}}\langle\langle \_, sc(\mathsf{staff/dept}), sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle \\ U = required \Rightarrow \mathsf{node{:}}\langle\langle sc(\mathsf{staff/dept})\rangle\rangle \rhd \mathsf{edge{:}}\langle\langle \_, sc(\mathsf{staff/dept}), sc(\mathsf{staff/dept}){:}\mathsf{did}\rangle\rangle\end{array}$$

The result of $typeTrans(\mathsf{xsd{:}int}, \mathsf{Types_c})$ is integer.

The key and keyref constructs in XML Schema are constraint constructs. As we stated above, if there is a key associated with a complexElement then the extent of that complexElement is the extent of the key object. This allows us to create the following production rule for key:

$$\begin{array}{l}constraint\ \mathsf{xml{:}key}{:}\langle\langle K, S, F\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \langle\langle sc(S)\rangle\rangle \xrightarrow{\text{id}} \langle\langle \_, sc(S), sc(S){:}F\rangle\rangle\end{array}$$

Where K is the name of the key, S the selector, which will be a complexElement and F the field of the key definition which will be the key object whose extent the complexElement referenced in S takes. If F is an attribute then we ignore the '@' at the beginning of the attribute name.

The expansion of the production rule for the key construct in Figure 3.6 is shown below

$$\begin{array}{l}\mathsf{xml{:}key}{:}\langle\langle \mathsf{employeeKey}, \mathsf{staff/dept/emp}, \mathsf{eid}\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \mathsf{node{:}}\langle\langle sc(\mathsf{staff/dept/emp})\rangle\rangle \xrightarrow{\text{id}} \mathsf{edge{:}}\langle\langle \_, sc(\mathsf{staff/dept/emp}), sc(\mathsf{staff/dept/emp}){:}\mathsf{eid}\rangle\rangle\end{array}$$

The keyref construct behaves in a similar way to an SQL foreign key and translates to an inclusion constraint in the HDM. We limit ourselves to keyref objects link a single pair of constructs.

$$\begin{array}{l}constraint\ \mathsf{xml{:}keyref}{:}\langle\langle KR, KS, KF, S, F\rangle\rangle \rightsquigarrow \bot \\ \quad \mathsf{true} \Rightarrow \langle\langle sc(S){:}F\rangle\rangle \subseteq \langle\langle sc(KS){:}KF\rangle\rangle\end{array}$$

KR is the name of the keyref, KS is the selector of the key construct this keyref refers to, and KF the field. S and F are the selector and field of the keyref itself.

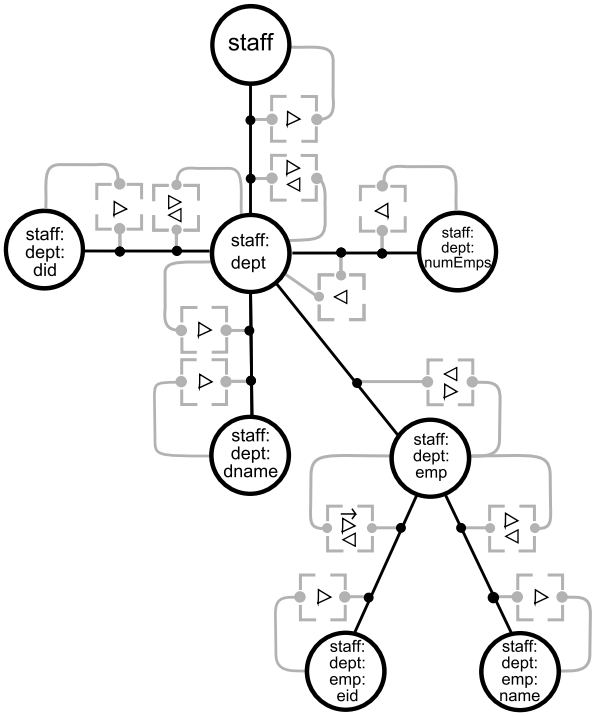


Figure 3.8: $S_{hdm-xml}$

There are no examples of `keyref` constructs in our schema.

Figure 3.8 shows the full HDM representation of the XML Schema in Figure 3.6.

3.2.4 Selected RDFS constructs

In this section we present new work that describes how selected RDFS [BG04] constructs can be modelled in the HDM. RDFS is the schema language for RDF which was created as the foundation language for the Semantic Web [BLHL01]. RDF allows us to make statements about resources on the web. Each statement contains the following three parts:

the subject identifies the resource or literal about which the statement is made

the predicate defines the relationship between the subject and the object of the statement

the object represents a resource or literal which is the 'target' of the predicate

As an example, assume the URI, `http://www.acme.com/staff/ps203.html`, points to Peter Smith's home page and acts as a unique id for him, and that the predi-

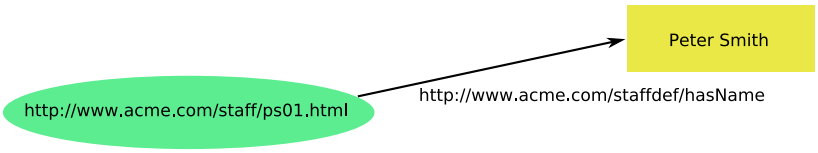


Figure 3.9: An RDF representation of a person's name

```
staffdef:Emp rdf:type rdfs:Class .

staffdef:Dept rdf:type rdfs:Class .

staffdef:hasName rdf:type rdf:Property .
staffdef:hasName rdfs:domain staffdef:Emp .
staffdef:hasName rdfs:range xsd:string .

staffdef:hasId rdf:type rdf:Property .
staffdef:hasId rdfs:domain staffdef:Emp .
staffdef:hasId rdfs:range xsd:int .

staffdef:hasDept rdf:type rdf:Property .
staffdef:hasDept rdfs:domain staffdef:Emp .
staffdef:hasDept rdfs:range staffdef:Dept .

staffdef:hasDid rdf:type rdf:Property .
staffdef:hasDid rdfs:domain staffdef:Dept .
staffdef:hasDid rdfs:range xsd:int .

staffdef:hasNumEmps rdf:type rdf:Property .
staffdef:hasNumEmps rdfs:domain staffdef:Dept .
staffdef:hasNumEmps rdfs:range xsd:int .

staffdef:hasDname rdf:type rdf:Property .
staffdef:hasDname rdfs:domain staffdef:Dept .
staffdef:hasDname rdfs:range xsd:string .
```

Figure 3.10: S_{rdfs}, a representation of the SQL schema from Figure 2.1 in RDFS

cate `http://www.acme.com/staffdef/hasName` is used to indicate the name of an employee. We can make the following statement about Peter Smith using an RDF triple:

Peter's name is Peter Smith :

(`http://www.acme.com/staff/ps01.htm1`,

`http://www.acme.com/staffdef/hasName, Peter Smith`)

RDF can also be shown in a graphical format. Figure 3.9 shows the statement above represented graphically.

RDFS [BG04] is the DDL for RDF. It provides constructs to describe the subjects, objects and predicates in a RDF document as well as relationships between them.

The subject and object components of an RDF triple are declared as `rdfs:class` constructs in RDFS. We model classes as nodes in the HDM, as shown in the fol-

```
staffinst:http://www.acme.com/staff/ps01.htm1 rdf:type staffdef:Emp
staffinst:http://www.acme.com/staff/sb21.htm1 rdf:type staffdef:Emp
staffinst:http://www.acme.com/staff/ct14.htm1 rdf:type staffdef:Emp

staffinst:http://www.acme.com/dept/finance.htm1 rdf:type staffdef:Dept
staffinst:http://www.acme.com/dept/hr.htm1 rdf:type staffdef:Dept

staffinst:http://www.acme.com/staff/ps01.htm1 staffdef:hasName Peter Smith
staffinst:http://www.acme.com/staff/sb21.htm1 staffdef:hasName Susan Brown
staffinst:http://www.acme.com/staff/ct14.htm1 staffdef:hasName Catherine Thomas

staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasDid 100
staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasDid 101

staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasDName Finance
staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasDName HR

staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasNumEmps 23
staffinst:http://www.acme.com/dept/finance.htm1 staffdef:hasNumEmps 15

staffinst:http://www.acme.com/staff/ps01.htm1 staffdef:hasDept
     staffinst:http://www.acme.com/dept/finance.htm1
staffinst:http://www.acme.com/staff/sb21.htm1 staffdef:hasDept
     staffinst:http://www.acme.com/dept/hr.htm1
staffinst:http://www.acme.com/staff/ct14.htm1 staffdef:hasDept
     staffinst:http://www.acme.com/dept/hr.htm1
```

Figure 3.11: $\mathsf{Inst}_1(S_{rdfs})$

lowing production rule:

$$nodal\ \mathsf{rdfs{:}class}{:}\langle\!\langle C\rangle\!\rangle(x) \rightsquigarrow \mathsf{node}{:}\langle\!\langle C\rangle\!\rangle(x)$$

For example $\mathsf{rdfs{:}class}{:}\langle\!\langle \mathsf{staffdef} : \mathsf{Emp}\rangle\!\rangle$ in Figure 3.10 will generate $\mathsf{node}{:}\langle\!\langle \mathsf{staffdef{:}Emp}\rangle\!\rangle$. In the figure we abbreviate the namespace, `http://www.acme.com/staffdef#`, to `staffdef`. The extent of a class is the subject component of any `rdf:type` predicates whose object matched the class name. If we let $I_1 = \mathsf{Inst}_1(S_{rdfs})$ in Figure 3.11 then

$Ext_{S_{rdfs},I_1}(\mathsf{class}{:}\langle\!\langle \mathsf{staffdef} : \mathsf{Emp}\rangle\!\rangle) =$

```
  {staffinst:http://www.acme.com/staff/ps01.htm1,
        staffinst:http://www.acme.com/staff/sb21.htm1,
        staffinst:http://www.acme.com/staff/ct14.htm1}
```

Here the namespace http://www.acme.com/staffinstances# is abbreviated as `staffinst`.

One can define subclass relationships between classes in RDFS. They are represented in the HDM by the inclusion constraint, as shown in the production rule below:

constraint rdfs:subClassOf:$\langle\langle C_{sub}, C\rangle\rangle \rightsquigarrow \bot$
$true \Rightarrow$ node:$\langle\langle$C$_{\text{sub}}\rangle\rangle \subseteq$ node:$\langle\langle$C$\rangle\rangle$

`rdfs:Resource` is a superclass of all RDFS classes. It is represented in the HDM as node:⟨⟨Resource⟩⟩, which is present in all HDM schemas that represent an RDFS schema.

`rdfs:Literal` is the class of all literal values such as integers and strings. Literals in RDFS can be plain or typed and are modelled by a single node called node:⟨⟨rdfs:Literal⟩⟩ in the HDM, which is always present in the same way as node:⟨⟨rdfs:Resource⟩⟩ is. The permissible types in RDFS are the datatypes defined for XML Schema [BM04].

The predicates in an RDF triple are described in RDFS by the **property** construct. It defines a binary predicate, $P(x, y)$. There are two types of properties in RDFS:

object properties which are relations whose range is an object class or is undefined

datatype properties which are relations whose range is a datatype

As both of these depend on the existence of the classes and/or data types they link, we represent them in the HDM as **link** constructs.

An **object property** links two classes. The first argument of the predicate is a member of an object class which may be defined to be the domain class of the property. The second is a member of an object class which may be defined to be the range class. If the domain of a property is not specified, it is assumed to be node:⟨⟨rdfs:Resource⟩⟩. If the range is not specified it is assumed to be node:⟨⟨rdfs:Resource⟩⟩.

A **datatype property**, with range D, states that the value of the property, *i.e.* the range, is a subset of the values allowed by the datatype D. Permissible datatypes include RDF literals and XML Schema datatypes. The individuals in a datatype class are members of the sets defined for the equivalent XML Schema datatype or a set of RDF literals. Datatype properties essentially assign a data type to a class. We represent this in the HDM by assigning a type to the HDM node representing the class. We discuss how we translate high level datatypes such as this into the HDM in Chapter 4.

The production rules for object properties in RDFS are as follows, where C_D is the domain class of the property and C_R the range class:

link rdfs:property:$\langle\langle P, C_D, C_R\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P, C_D, C_R\rangle\rangle(x, y)$

link rdfs:property:$\langle\langle P, null, C_R\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P,$ rdfs:Resource$, C_R\rangle\rangle(x, y)$

link rdfs:property:$\langle\langle P, C_D, null\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P, C_D,$ rdfs:Resource$\rangle\rangle(x, y)$

link rdfs:property:$\langle\langle P, null, null\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P,$ rdfs:Resource, rdfs:Resource$\rangle\rangle(x, y)$

The production rules for a datatype property are:

link rdfs:property:$\langle\langle P, C_D, DataType\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P, C_D,$ rdfs:Literal$\rangle\rangle(x, y)$

link rdfs:property:$\langle\langle P, null, DataType\rangle\rangle(x, y) \rightsquigarrow$ edge:$\langle\langle P,$ rdfs:Resource, rdfs:Literal$\rangle\rangle(x, y)$

Below are the production rules for the staffdef:hasDept and staffdef:hasName properties in Figure 3.10. As in the case of XML Schema, we translate any slashes in the names of the properties or classes into colons.

rdfs:property:$\langle\langle$staffdef:hasDept, staffdef:Emp, staffdef:Dept$\rangle\rangle(x, y) \rightsquigarrow$
edge:$\langle\langle$staffdef:hasDept, staffdef:Emp, staffDef:Dept$\rangle\rangle(x, y)$

rdfs:property:$\langle\langle$staffdef:hasName, staffdef:Emp, xsd:string$\rangle\rangle(x, y) \rightsquigarrow$
edge:$\langle\langle$staffdef:hasName, staffdef:Emp, rdfs:Literal$\rangle\rangle(x, y)$

Given the instance in Figure 3.11, the extent of
property:$\langle\langle$staffdef:hasDept, staffdef:Emp, staffdef:Dept$\rangle\rangle$ is

```
{(staffinst:http://www.acme.com/staff/ps01.htm1,
    staffinst:http://www.acme.com/dept/finance.htm1),
(staffinst:http://www.acme.com/staff/sb21.htm1,
    staffinst:http://www.acme.com/dept/hr.htm1),
```

(staffinst:http://www.acme.com/staff/ct14.htm1,
staffinst:http://www.acme.com/dept/hr.htm1)}

The rdfs:subPropertyOf construct defines a predicate whose tuples are a subset of those of the parent property. It is modelled in the HDM using an inclusion constraint, as shown in the following production rule:

$$\begin{array}{l} \textit{constraint}\ \mathsf{rdfs{:}subPropertyOf{:}}\langle\langle P_{sub}, C_D, C_R\rangle\rangle \rightsquigarrow \bot \\ \quad true \ \ \Rightarrow \mathsf{edge{:}}\langle\langle P_{sub}, C_D, C_R\rangle\rangle \subseteq \mathsf{edge{:}}\langle\langle P, C_D, C_R\rangle\rangle \end{array}$$

We have no instances of rdfs:subProperty constructs in our example.

Figure 3.12 shows the HDM representation of the RDF schema in Figure 3.10.

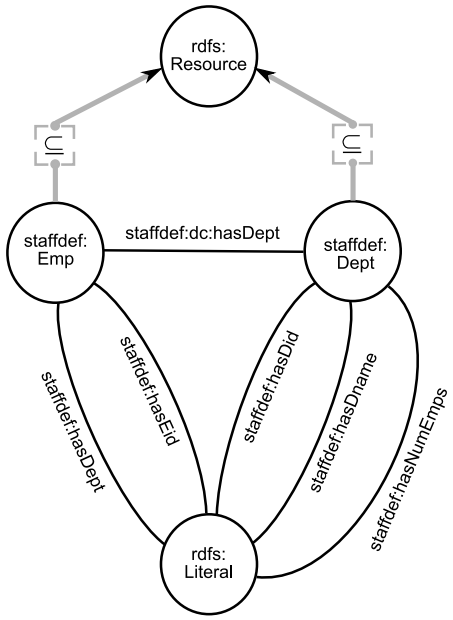


Figure 3.12: An HDM representation of the RDFS schema in Figure 3.10

There are other RDFS constructs that allow users to specify the class that a specific individual is a member of and add labels and comments to a schema that we do not model in the HDM at present. There are also constructs that allow the representation of containers in RDFS. We do not explicitly model these but the functionality they provide is also provided by the IQL query language [JTMP03] that is part of the AutoMed system.

3.3 Mapping and Transformation Language

In the previous section we saw how AutoMed allows us to express schemas from a wide range of high level DDLs. In this section we describe a declarative language that allows us to define mappings and transformations between those schemas and their instances.

We use the schema transformations technique **Both-As-View** (**BAV**) [MP03], as the mapping and transformation language for our MMS framework. BAV uses a DDL independent combination of primitive transformations to create and remove schema objects, and a query language that can be applied to any AutoMed schema object to specify the instances of those objects. This allows us to create target schemas made up of objects expressed in any DDL supported by AutoMed.

A transformation in BAV is made up of a sequence of *bidirectional* primitive transformations that together describe how instances of each schema object in the target schema are derived from instances of schema objects in the source schema and *vice versa*. Each primitive transformation either adds, deletes or renames a single schema object (such as a single SQL column, SQL primary key definition, XML element, HDM node *etc*), thereby incrementally generating a new schema from an old schema. The scheme of each new object must be different from those of the existing objects in the schema so that we can uniquely identify each object. The extent of the schema object being added or deleted is defined as a complete or partial query on the extents of the existing schema objects. The sequence of transformations is called a **pathway**. It is important to note that the schemas created during the execution of a BAV pathway are intensional and are not intended to be materialised, so do not place too heavy a burden on the system. Only the target schema at the end of a pathway is ever materialised.

In this thesis a BAV pathway has two distinct phases. We first execute transformations that add target objects to the source schema. This is called the **growth phase** of the pathway. At the end of the growth phase the schema created will have all the schema objects of the source and target schemas. We then execute transformations to remove the source objects from this schema. This is called the **shrinking phase** of the pathway.

The queries in the growth phase of a pathway describe the target objects in terms of source objects while in the shrinking phase, the queries provide a description of the source schema objects in terms of target objects. These two phases of the pathway

are equivalent to GAV and LAV mappings as described in the previous chapter. At present only GAV query reformulation is implemented in AutoMed [Zam08].

The primitive BAV transformations are defined below. In the definitions, s is an AutoMed schema object and q the query that specifies the extent of s.

1. add($\langle\langle s\rangle\rangle$,$q$): When applied to a schema S this transformation generates a new schema S' that differs from S by having a new object $\langle\langle s\rangle\rangle$. The query q specifies the extent of $\langle\langle s\rangle\rangle$ in terms of the existing object in S. If $\langle\langle s\rangle\rangle$ is a constraint then q is omitted.

2. delete($\langle\langle s\rangle\rangle$,$q$): This transforms S into schema S' that differs from S in that a object $\langle\langle s\rangle\rangle$ is missing. The query q specifies the extent of the object to be removed in terms of the remaining objects in S. This allows us to recover the extent of $\langle\langle s\rangle\rangle$. delete is the inverse of the add transformation. If we start with S and perform an add($\langle\langle s\rangle\rangle$,$q$) and then a delete($\langle\langle s\rangle\rangle$,$q$) we will end up with S again. As above if s is a constraint q is omitted.

3. rename($\langle\langle s\rangle\rangle$,$\langle\langle s'\rangle\rangle$): Given S this transformation creates S' that has a new object with scheme $\langle\langle s'\rangle\rangle$ but is missing the object $\langle\langle s\rangle\rangle$. rename is its own inverse *i.e.* if we start with S a rename($\langle\langle s\rangle\rangle$,$\langle\langle s'\rangle\rangle$) followed by a rename($\langle\langle s'\rangle\rangle$,$\langle\langle s\rangle\rangle$) will get us back to S.

4. extend($\langle\langle s\rangle\rangle$,Range q_l q_u): When applied to schema S this produces a new schema S' that differs from S in that there is a new object $\langle\langle s\rangle\rangle$. The minimum extent of $\langle\langle s\rangle\rangle$ is given by the query q_l and may take the special value Void if, for all I, no values in $Ext_{S',I}(\langle\langle s\rangle\rangle)$ can be derived from S. The maximum extent of $\langle\langle s\rangle\rangle$ is given by q_u which may take the special value Any if, for all I, no limit on $Ext_{S',I}(\langle\langle s\rangle\rangle)$ can be derived from S.

5. contract($\langle\langle s\rangle\rangle$,Range q_l q_u): This is the inverse of extend. Given S it produces S' that differs from S in that a object $\langle\langle s\rangle\rangle$ is missing. The queries q_l and q_u have the same meaning as for extend.

6. ident(*S,S',os*): This transformation creates a pathway between two schemas with identical schema objects and allows us to link two data sources. The *os* parameter allows us to set different operational semantics for the transformation [JTMP04] which determines how the extent of an object in S or S' is calculated if the extent contains values from both data sources. In this thesis we use union semantics which means a value will be returned if it appears in either of the data sources.

Each successful transformation t is reversible by a transformation t^{-1}. For example the transformation $\mathsf{add}(\langle\langle s \rangle\rangle, q)$ can be reversed by $\mathsf{delete}(\langle\langle s \rangle\rangle, q)$.

It is important to note that if there is no delete or contract transformation for an object in the source schema that object will still be present, unchanged, in the target schema.

The add, delete and rename transformations are examples of **exact** transformations [Len02]. If the new schema we create with the transformation is S', the schema object is s and the result of the query in the transformation for instance I is $q(I)$, then the logical semantics of add, delete and rename are as follows:

$$\forall I. Ext_{S',I}(\langle\langle s \rangle\rangle) = q(I)$$

The extend and contract transformation on the other hand change the information capacity of the schema. After an extend the information capacity of S' will be greater than S and after a contract it will be less. If $q_l(I)$ represents a lower bound on the extent of $\langle\langle s \rangle\rangle$ and $q_u(I)$ an upper bound then the semantics of extend and contract are:

$$\forall I. q_l(I) \subseteq Ext_{S',I}(\langle\langle s \rangle\rangle) \subseteq q_u(I)$$

If we are unable to place a lower bound on the extent of $\langle\langle s \rangle\rangle$, $q_l(I)$ is replaced by the keyword Void; similarly if we are unable to place an upper bound on the extent of $\langle\langle s \rangle\rangle$, $q_u(I)$ is replaced by the keyword Any. Table 3.1 shows the information capacity of the primitive transformations when applied to the HDM constructs we described in Section 3.1. Note that we do not apply the contract or extend primitives to constraint constructs as these do not have extents and so there is no need to be able to define upper and lower bound queries. This applies to any constraint construct in AutoMed.

If we have queries defining both the upper and lower bounds of the new schema we have a **complete transformation**. If we only have a lower bound the transformation is **sound** [Len02].

The embedded queries q, q_l and q_u can be expressed in any query language, for instance the DDL specific languages SQL or XPath. In the AutoMed MMS we use the **Intermediate Query Language** (**IQL**) [JTMP03], a typed comprehension-based functional language based on FPL [PS93]. This offers clean semantics and

Primitive Transform $S \rightarrow S'$	Reverse Transformation $S' \rightarrow S$	Conditions on S, S'	Info capacity
add(node:$\langle\langle n \rangle\rangle$,$q$)	delete(node:$\langle\langle n \rangle\rangle$,$q$)	$\langle\langle n \rangle\rangle \notin Nodes, \langle\langle n \rangle\rangle \in Nodes'$	$S \equiv S'$
add(edge:$\langle\langle e \rangle\rangle$,$q$)	delete(edge:$\langle\langle e \rangle\rangle$,$q$)	$\langle\langle e \rangle\rangle \notin Edges, \langle\langle e \rangle\rangle \in Edges'$	$S \equiv S'$
add(cons:$\langle\langle c \rangle\rangle$)	delete(cons:$\langle\langle c \rangle\rangle$)	$\langle\langle c \rangle\rangle \notin Cons, \langle\langle c \rangle\rangle \in Cons'$	$S \equiv S'$
rename(node:$\langle\langle n \rangle\rangle$, node:$\langle\langle n' \rangle\rangle$)	rename(node:$\langle\langle n' \rangle\rangle$, node:$\langle\langle n \rangle\rangle$)	$\langle\langle n \rangle\rangle \in Nodes, \langle\langle n \rangle\rangle \notin Nodes'$ $\langle\langle n' \rangle\rangle \notin Nodes, \langle\langle n' \rangle\rangle \in Nodes'$	$S \equiv S'$
rename(edge:$\langle\langle e \rangle\rangle$, edge:$\langle\langle e' \rangle\rangle$)	rename(edge:$\langle\langle e' \rangle\rangle$, edge:$\langle\langle e \rangle\rangle$)	$\langle\langle e \rangle\rangle \in Edges, \langle\langle e \rangle\rangle \notin Edges'$ $\langle\langle e' \rangle\rangle \notin Edges, \langle\langle e' \rangle\rangle \in Edges'$	$S \equiv S'$
extend(node:$\langle\langle n \rangle\rangle$,$q_l$,$q_u$)	contract(node:$\langle\langle n \rangle\rangle$,$q_l$,$q_u$)	$\langle\langle n \rangle\rangle \notin Nodes, \langle\langle n \rangle\rangle \in Nodes'$	$S \subset S'$
extend(edge:$\langle\langle e \rangle\rangle$,$q_l$,$q_u$)	contract(edge:$\langle\langle e \rangle\rangle$,$q_l$,$q_u$)	$\langle\langle e \rangle\rangle \notin Edges, \langle\langle e \rangle\rangle \in Edges'$	$S \subset S'$

Table 3.1: BAV primitive transformations for HDM schema $S = \langle Nodes, Edges, Cons \rangle$ to generate new schema $S' = \langle Nodes', Edges', Cons' \rangle$.

the ability to represent the type of query one finds in many DDL specific languages [BLS+94]. Using IQL allows us to express inter-schema properties across different DDLs. A full discussion of IQL is unnecessary for this thesis, but we describe the features we need below (detailed information about the language can be found in [Zam08]).

IQL supports list, bag and set semantics, allowing us to represent queries from a wide range of high level query languages. It allows the use of the keyword **distinct** in front of a query to remove duplicates. We assume a set based representation for the instances in our schemas so we use this **distinct** keyword to enforce this when necessary.

IQL queries are comprehensions of the form $[h \mid q_1; \ldots; q_n]$, where h is an expression termed the **head** and $q_1; \ldots; q_n$ where $n > 0$, are **qualifiers**. The result of the query can be found in the head. A qualifier may be either a filter or a generator. Generators are of the form $p \leftarrow e$ and iterate a pattern p over an expression e. Filters are boolean-valued expressions that act as filters on the variable instantiations generated by the generators of the comprehension.

The following is an example of a comprehension that returns the employee id and names of the employees in Figure 3.2, who work in the finance department. In all the mappings, transformations and queries in this section we will only use the key scheme of the schema objects.

$$[\{e, n\} \mid \{e, n\} \leftarrow \text{column:}\langle\langle \text{Emp, name} \rangle\rangle; \{e, d\} \leftarrow \text{column:}\langle\langle \text{Emp, dept} \rangle\rangle;$$
$$\{d, d\} \leftarrow \text{column:}\langle\langle \text{Dept, did} \rangle\rangle; \{d, dn\} \leftarrow \text{column:}\langle\langle \text{Dept, dname} \rangle\rangle; dn = \texttt{'Finance'}]$$

The generators in the query iterate their patterns over the extents of the respective columns. These results are filtered by $dn =$ `'Finance'` to give the final instantiation

of the variables in the head as [{`1,'Peter Smith'`}].

If we have a query that contains a single generator whose pattern exactly matches the head then we abbreviate the query to be just the expression. For example $[\{e, n\} \mid \{e, n\} \leftarrow \mathsf{column{:}\langle\langle Emp, name\rangle\rangle}]$ can be abbreviated to $\mathsf{column{:}\langle\langle Emp, name\rangle\rangle}$.

We now describe an example that shows how a BAV transformation pathway can be used to restructure an XML document.

Assume we start with the following schema (we will only include the key scheme of the schema objects for bevity):

$S_{staff} = \{\mathsf{xs{:}complexElement{:}\langle\langle null, staffByEmp\rangle\rangle, xs{:}complexElement{:}\langle\langle staffByEmp, Emp\rangle\rangle},$
$\mathsf{xs{:}attribute{:}\langle\langle staffByEmp/Emp, eid\rangle\rangle, xs{:}simpleElement{:}\langle\langle staffByEmp/Emp, dept\rangle\rangle}\}$

We now apply the following transformation pathway that includes the function generateGID. generateGID acts like a Skolem function in that it generates unique identifiers which we will call **Skolem values** from now on. It takes four parameters: *schemaName*, *sid*, a list of data values and an arbitrary string value. We set the *schemaName* parameter to be the source schema if we use a generateGID in the growth phase of a pathway, and the name of the target schema if we use a generateGID in the shrinking phase. The value of *sid* is stored in a hashmap and can be recovered by using the generateSID function [KM05]. We do not make use of the value in this thesis so the value of this parameter can be any variable that appears in the pattern of the one of the generators in the query. The third parameter is the list of data values upon which the Skolem values will be based. These will be values that can be used to identify the tuples in the object being created, such as a key value or an OID. The final parameter will generally be the name of the schema object we are adding or removing. The same set of parameters given to generateGID will always create the same set of unique values, a different set of parameters will always create a different set of values.

① add(complexElement:⟨⟨null, staffByDept⟩⟩, complexElement:⟨⟨null, staffByEmp⟩⟩)
② add(complexElement:⟨⟨staffByDept, Dept⟩⟩, [{x, $doid$} | {x} ← complexElement:⟨⟨null, staffByEmp⟩⟩;
{$eoid$, d} ← simpleElement:⟨⟨staffByEmp/Emp, dept⟩⟩;
$doid$ ← generateGID(S, d, [d], 'Dept')])
③ add(attribute:⟨⟨staffByDept/Dept, did⟩⟩,
[{$doid$, d} | {$eoid$, d} ← simpleElement:⟨⟨staffByEmp/Emp, dept⟩⟩;
$doid$ ← generateGID(S, d, [d], 'Dept')])
④ add(complexElement:⟨⟨staffByDept/Dept, Emp⟩⟩,
[{$doid$, $eoid$} | {$eoid$, e} ← complexElement:⟨⟨staffByEmp, Emp⟩⟩;
{$eoid$, d} ← simpleElement:⟨⟨staffByEmp/Emp, dept⟩⟩;
$doid$ ← generateGID(S, d, [d], 'Dept')])
⑤ add(attribute:⟨⟨staffByDept/Dept/Emp, eid⟩⟩, attribute:⟨⟨staffByEmp/Emp, eid⟩⟩)
⑥ delete(xs:simpleElement:⟨⟨staffByEmp/Emp, dept⟩⟩,
[{$eoid$, did} | {$doid$, $eoid$} ← complexElement:⟨⟨staffByDept/Dept, Emp⟩⟩;
{$doid$, did} ← attribute:⟨⟨staffByDept/Dept, did⟩⟩])
⑦ delete(attribute:⟨⟨staffByEmp/Emp, eid⟩⟩, attribute:⟨⟨staffByDept/Dept/Emp, eid⟩⟩)
⑧ delete(xs:complexElement:⟨⟨staffByEmp/Emp⟩⟩,
[{x, $eoid$} | {$eoid$, e} ← attribute:⟨⟨staffByDept/Dept/Emp, eid⟩⟩;
{x} ← xs:complexElement:⟨⟨null, staffByDept⟩⟩])
⑨ delete(xs:complexElement:⟨⟨null, staffByEmp⟩⟩, xs:complexElement:⟨⟨null, staffByDept⟩⟩)

will create this schema:

S'_{staff} = {xs:complexElement:⟨⟨null, staffByDept⟩⟩, xs:complexElement:⟨⟨staffByDept, Dept⟩⟩,
xs:attribute:⟨⟨staffByDept/Dept, did⟩⟩, xs:complexElement:⟨⟨staffByDept/Dept, Emp⟩⟩,
xs:attribute:⟨⟨staffByDept/Dept/Emp, eid⟩⟩}

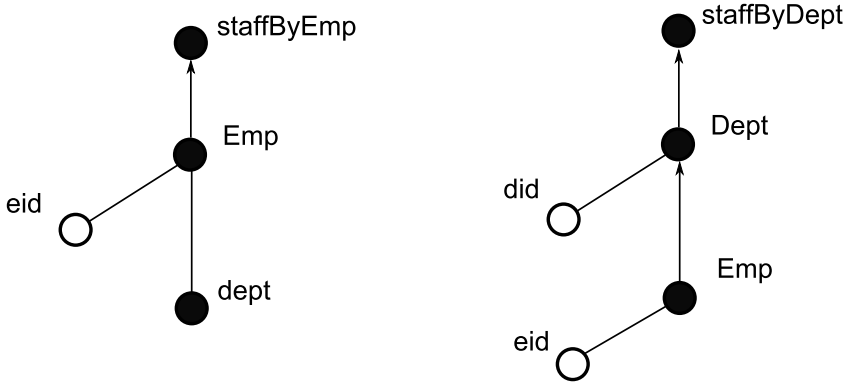


Figure 3.13: XML schemas restructured using a BAV pathway

The two schemas are shown in Figure 3.13. Each transformation creates a new schema. For example, Transformation ① creates a schema that differs from S in that there is a new schema object complexElement:⟨⟨null, staffByDept⟩⟩. The extent of the new schema object is defined *exactly* by complexElement:⟨⟨null, staffByEmp⟩⟩ so we use an add transformation. This query is in terms of existing schema ob-

jects and so does not change the information capacity of the newly created schema. Transformations (2) to (5) similarly create intermediate schemas that have equivalent information capacity. Together transformations (1) to (5) make up the **growth** phase of the pathway. The remaining transformations remove the old schema objects one at a time to leave us with just the target schema. The queries associated with these transformations are used by the query processor to answer queries posed in the target schema.

The queries in Transformations (1) to (5) provide a GAV description of the target schema. A conjunction of the queries in these transformations together describe the instances of the target schema in terms of the source. On the other hand the queries in Transformations (6) to (9) provide a LAV description. The queries in these transformations describe instances of the source schema in terms of the target.

Recall that the extent of a complexElement that does not have a key construct associated with it is not a data value but rather a synthetic value used to uniquely identify the object in XML document hierarchy. In Transformations (2) to (4) we use generateGID to create the required unique identifiers for the new complexElement:$\langle\langle$staffByDept, Dept$\rangle\rangle$ object in the target schema.

To get unique identifiers for instances of complexElement:$\langle\langle$staffByDept/Dept, Emp$\rangle\rangle$ we use the values from complexElement:$\langle\langle$staffByEmp, Emp$\rangle\rangle$ as we know there is a unique value for each instance of the object.

Executing the transformations above in AutoMed for the following XML document:

```
<staffByEmp> &0
  <Emp eid="1"> &1
    <dept>6</dept> &2
  </Emp>
  <Emp eid="2"> &3
    <dept>6</dept> &4
  </Emp>
  <Emp eid="3"> &5
    <dept>10</dept> &6
  </Emp>
  <Emp eid="4"> &7
    <dept>12</dept> &8
  </Emp>
</staffByEmp>
```

gives these extents for the newly created schema objects, where I_1 is the instance of the target schema that corresponds to the source instance above:[3]

$Ext_{S'_{staff},I_1}(\text{complexElement:}\langle\langle\text{staffByDept, Dept}\rangle\rangle) = \{(\&0,\#1001),(\&0,\#1002),(\&0,\#1003)\}$
$Ext_{S'_{staff},I_1}(\text{attribute:}\langle\langle\text{staffByDept/Dept, did}\rangle\rangle) = \{(\#1001,6),(\#1001,6),(\#1002,10),(\#1003,12)\}$
$Ext_{S'_{staff},I_1}(\text{complexElement:}\langle\langle\text{staffByDept/Dept, Emp}\rangle\rangle) =$
$\{(\#1001,\&1),(\#1001,\&3),(\#1002,\&5),(\#1003,\&7)\}$
$Ext_{S'_{staff},I_1}(\text{attribute:}\langle\langle\text{staffByDept/Dept/Emp, eid}\rangle\rangle) = \{(\&1,1),(\&3,2),(\&5,3),(\&7,4)\}$

If the extent of a schema object contains Skolem values as described above, and it exists in the source or target data source, the Skolem values may be post processed by the IQL query processor to unify them with any corresponding data values that may exist.

If S'_{staff} is materialised, as shown below, and becomes a data source in our pathway then any query on complexElement:⟨⟨staffByDept, Dept⟩⟩ that uses that data source will return with the OIDs from the newly materialised XML instance document rather than the Skolem values. Similarly queries on complexElement:⟨⟨staffByDept/Dept, Emp⟩⟩ will return the OIDs from the target schema.

```
<staffByDept> &0
  <Dept did="6"> &1
    <Emp eid="1" /> &2
    <Emp eid="2" /> &3
  </Dept> &4
  <Dept did="10"> &5
    <Emp eid="3" /> &6
  </Dept>
  <Dept did="12"> &7
    <Emp eid="4" /> &8
  </Dept>
</staffByDept>
```

The relationship between the extents of the complexElements generated by the pathway and those in the materialised target schema are shown in Table 3.2. The value generated by the pathway is on the LHS and the value from the materialised target schema is on the RHS of the arrow.

[3]We preface Skolem values with # in the same way as we preface the OIDs in an XML instance with &, to differentiate them from data values

⟨⟨staffByDept/Dept, Emp⟩⟩	⟨⟨staffByDept, Dept⟩⟩
&1 → &2	#1001 → &1
&3 → &3	#1002 → &5
&5 → &6	#1003 → &7
&7 → &7	

Table 3.2: Post processing Skolem values

3.3.1 Translating High Level Schemas into HDM using BAV

In this section we describe the function HLtoHDM(S), implemented in our MMS, that executes the production rules described in Section 3.2, using the BAV transformations described above, to translate a high level schema into the HDM.

Each production rule specifies the HDM constructs that should be created as well as the extent of the objects. We first translate nodal constructs. These production rules are of the form

$$\textit{nodal}\ \mathsf{DDL{:}c{:}}\langle\langle so\rangle\rangle(x) \rightsquigarrow \mathsf{node{:}}\langle\langle so\rangle\rangle(x)$$

Each of these rules generates a single BAV transformation:

$$\mathsf{add}(\mathsf{node{:}}\langle\langle so\rangle\rangle,\mathsf{c{:}}\langle\langle so\rangle\rangle)$$

We next translate link-nodal constructs. These have production rules of the general form:

$$\begin{aligned}&\textit{link-nodal}\ \mathsf{DDL{:}c{:}}\langle\langle so_1, so_2\rangle\rangle(x,y) \rightsquigarrow\\ &\qquad \mathsf{node{:}}\langle\langle so_2{:}so_2\rangle\rangle(y), \mathsf{edge{:}}\langle\langle \_, so_1, so_1{:}so_2\rangle\rangle(x,y)\\ &\quad true \ \Rightarrow \mathsf{node{:}}\langle\langle so_1{:}so_2\rangle\rangle \rhd \mathsf{edge{:}}\langle\langle \_, so_1, so_1{:}so_2\rangle\rangle\end{aligned}$$

As we have seen there may also be other parameters such as a data type and labels that determine whether or not constraints should be added. We include a constraint in this example to illustrate the adding of constraints but there need not always be a constraint added as part of a link-nodal production rule. Each high level link-nodal object is translated into an HDM node and edge and possibly some constraints. The BAV transformations are as follows:

$$\mathsf{add}(\mathsf{node{:}}\langle\langle so_1{:}so_2\rangle\rangle,\mathsf{distinct}\ [\{y\} \mid \{x,y\} \leftarrow \mathsf{c{:}}\langle\langle so_1, so_2\rangle\rangle])$$
$$\mathsf{add}(\mathsf{edge{:}}\langle\langle \_, so_1, so_1{:}so_2\rangle\rangle,\mathsf{distinct}\ \mathsf{c{:}}\langle\langle so_1, so_2\rangle\rangle)$$
$$\mathsf{add}(\mathsf{mandatory{:}}(\mathsf{node{:}}\langle\langle so_1{:}so_2\rangle\rangle, so_1{:}so_2))$$

There may be additional transformations to add more constraints if the production rule calls for it.

Next we translate link structures which have production rules of the general form:

$$link\ \mathsf{DDL{:}c{:}}\langle\langle label, so_1, so_2\rangle\rangle(\vec{x_n}) \rightsquigarrow \mathsf{edge{:}}\langle\langle label, so_1, so_n\rangle\rangle(\vec{x_n})$$

This generates the following transformation:

$$\mathsf{add}(\mathsf{edge{:}}\langle\langle label, so_1, so_2\rangle\rangle, \mathsf{distinct\ c{:}}\langle\langle label, so_1, so_2\rangle\rangle)$$

Again there may be additional parameters and constraints which we leave out here as they are created in the same way as they are for link-nodal constructs.

Finally we translate any constraint structures. Each one specifies exactly the HDM constraints to be added in the same way as the second line of the link-nodal example did above.

At the end of this process $\mathsf{HLtoHDM}(S)$ returns a pathway that describes how S is translated into the HDM.

3.3.2 Composite BAV Transformations

Each BAV transformation step only changes one schema object so a large number of transformations are needed for most operations. To avoid the need to programme each transformation step separately, *information preserving* **composite transformations** (**CT**s) can be defined that are template transformation pathways, describing common patterns of transformation steps. A number of CTs that we use in this thesis are defined in [BM05]. The pathways that create these transformations can be composed allowing us to use a sequence of CTs to perform complex transformations. As with all BAV pathways each CT has an automatically derivable inverse. CTs are most commonly used when translating schemas from one DDL to another, as we will see in Chapter 5.

Two examples CTs from [BM05] are shown in Figure 3.14. inclusion_expand (going from left to right in the figure) and its inverse inclusion_merge are shown in Figure 3.14 (a) and um_redirection, which is its own inverse, in Figure 3.14 (b).

inclusion_merge allows us to merge node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ together, where the extent of node:$\langle\langle A\rangle\rangle$ is a subset of the extent of node:$\langle\langle B\rangle\rangle$ and there is a mandatory

constraint from node:$\langle\langle A\rangle\rangle$ to edge:$\langle\langle \_, A, C\rangle\rangle$. As there may be values in the extent of node:$\langle\langle B\rangle\rangle$ that do not appear in edge:$\langle\langle \_, A, C\rangle\rangle$ we drop the mandatory constraint while performing an inclusion_merge, we also redirect any edges and constraints on node:$\langle\langle A\rangle\rangle$ to node:$\langle\langle B\rangle\rangle$. Conversely, inclusion_expand allows us to create a new node:$\langle\langle A\rangle\rangle$ whose extent contains only those values from node:$\langle\langle B\rangle\rangle$ that are in the extent of edge:$\langle\langle \_, B, C\rangle\rangle$. We create a new edge from node:$\langle\langle C\rangle\rangle$ to node:$\langle\langle A\rangle\rangle$ and add a mandatory constraint from node:$\langle\langle A\rangle\rangle$ to edge:$\langle\langle \_, B, C\rangle\rangle$. We link node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ with an inclusion constraint.

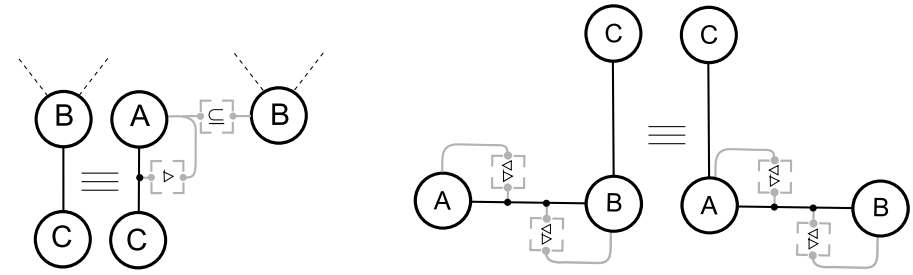
um_redirection allows us to move edge:$\langle\langle \_, A, C\rangle\rangle$ from node:$\langle\langle A\rangle\rangle$ to node:$\langle\langle B\rangle\rangle$ because both node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ have a unique and mandatory constraint on the common edge edge:$\langle\langle \_, A, B\rangle\rangle$. These constraints together are equivalent to stating that there is a one to one correspondence between the values in node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ so whatever is related to a value of node:$\langle\langle A\rangle\rangle$ through edge:$\langle\langle \_, A, C\rangle\rangle$ is equally related to the corresponding value in node:$\langle\langle B\rangle\rangle$.

Algorithm 3.2: expand_multi_value(Schema S, edge:$\langle\langle \_, A, B\rangle\rangle$,String T)

1 $S' := S$.add(node:$\langle\langle T\rangle\rangle$,edge:$\langle\langle \_, A, B\rangle\rangle$)
2 $S' := S'$.add(node:$\langle\langle T{:}A\rangle\rangle$,[{$x$} | {$x, y$} $\leftarrow$ edge:$\langle\langle \_, A, B\rangle\rangle$])
3 $S' := S'$.add(edge:$\langle\langle \_, T, T{:}A\rangle\rangle$,[{{$x, y$}, x} | {x, y} $\leftarrow$ edge:$\langle\langle \_, A, B\rangle\rangle$])
4 $S' := S'$.add(edge:$\langle\langle \_, T, B\rangle\rangle$,[{{$x, y$}, y} | {x, y} $\leftarrow$ edge:$\langle\langle \_, A, B\rangle\rangle$])
5 $S' := S'$.add(subset:(node:$\langle\langle TA\rangle\rangle$, node:$\langle\langle A\rangle\rangle$))
6 $S' := S'$.add(unique:(node:$\langle\langle T\rangle\rangle$, edge:$\langle\langle \_, T, T{:}A\rangle\rangle \bowtie$ edge:$\langle\langle \_, T, T{:}A\rangle\rangle$))
7 $S' := S'$.add(mandatory:(node:$\langle\langle T\rangle\rangle$, edge:$\langle\langle \_, T, B\rangle\rangle \bowtie$ edge:$\langle\langle \_, T, B\rangle\rangle$))
8 $S' := S'$.add(reflexive:(node:$\langle\langle T\rangle\rangle$, edge:$\langle\langle \_, T, T{:}A\rangle\rangle \bowtie$ edge:$\langle\langle \_, T, B\rangle\rangle$))
9 $S' := S'$.add(mandatory:(node:$\langle\langle T{:}A\rangle\rangle$, edge:$\langle\langle \_, T, T{:}A\rangle\rangle$))
10 $S' := S'$.add(mandatory:(node:$\langle\langle B\rangle\rangle$, edge:$\langle\langle \_, T, B\rangle\rangle$))
11 **foreach** $c \in$ *Cons forwhich contains* (edge:$\langle\langle \_, A, B\rangle\rangle, c$) **do**
12 $S' := S'$.delete(c)
13 $S' := S'$.delete(edge:$\langle\langle \_, A, B\rangle\rangle$,node:$\langle\langle T\rangle\rangle$);
14 **return** $p_{S,S'}$;

In Algorithm 3.2 we present a new CT, expand_multi_value which is illustrated in Figure 3.15. The presentation of this CT differs slightly from that in [BM05] because we wish CTs in our MMS to return a BAV pathway. To allow this we keep track of the current schema in the transformation and return a pathway from the input schema to the final schema in the transformation.

The parameters to this CT are an edge, and a string that will be used as the name of a new node created by the CT. It allows us to remove the edge between node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ and replace it with the new nodes node:$\langle\langle T\rangle\rangle$ and node:$\langle\langle T{:}A\rangle\rangle$ and edges linking node:$\langle\langle T\rangle\rangle$ to node:$\langle\langle T{:}A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$. The new edges contain identity tuples and so have the constraints shown in the figure. We also add an



(a) inclusion_expand and inclusion_merge

(b) um_redirection

Figure 3.14: Composite Transformations

inclusion constraint between the newly created node:$\langle\langle T \rangle\rangle$ and node:$\langle\langle A \rangle\rangle$. This CT is useful when translating from a DDL that supports multi-valued attributes (such as XML Schema or some variants of the ER model) into a target DDL that does not (such as SQL). In the figure, the $\bowtie$ symbol represents a join operation between edge:$\langle\langle \_, T, B \rangle\rangle$ and edge:$\langle\langle \_, T, T{:}A \rangle\rangle$. This means that the constraints linked to the join apply to both edges.

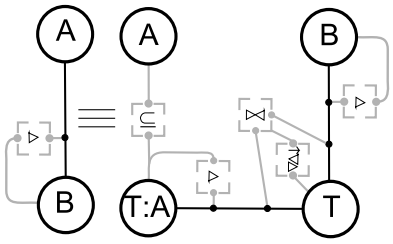


Figure 3.15: expand_multi_value and contract_multi_value

The *contains* predicate used in line 11 holds when its first argument appears as a construct in the formula that is in the second argument. As with any BAV transformation pathway, this CT has an automatically derivable inverse, contract_multi_value.

3.4 Translating SO s-t tgds into BAV pathways

In this section we describe how we create a BAV pathway, equivalent to the set of SO s-t tgds in a mapping, that *transforms* the source schema into the target schema. This is an implementation of TransGen(map_{S_1,S_2}, BAV), where S_1 is the source schema and S_2 the target schema.

Suppose that the finance department of the organisation we discussed in the previous chapter has decided to assign each employee in the department a special finance id

for added security. They decide to keep the association between the employee id and finance id in a separate, secure table. The mapping shown below is between S_{emp} (shown in Figure 2.1) and the new database, S_{finId}.

$map_{S_{emp},S_{finId}} = (S_{emp}, S_{finId}, \Sigma_{S_{emp},S_{finId}})$ where $\Sigma_{S_{emp},S_{finId}} =$

$$\{\exists finId(S_{emp}\text{::table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge S_{emp}\text{::table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{`Finance'}) \rightarrow$$
$$S_{finId}\text{::table:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,finId(e)) \wedge S_{finId}\text{::table:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(finId(e),n)),$$
$$\exists dept(S_{finId}\text{::table:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge S_{finId}\text{::table:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,dept(e)) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(dept(e),ne(dept(e)),\texttt{`Finance'}))\}$$

$finId$,$dept$ and ne are Skolem functions which create different unique values for each unique employee id.

The translation methodology we use to translate a mapping like the one above into BAV transformations is as follows:

1. Translate the SO s-t tgds in the mapping into the 'normal form' described in Section 2.2.1.
2. Translate the LHS of each SO s-t tgd into IQL to create a query q.
3. Create a BAV transformation for each implication in the normalised SO s-t tgds, using q as the query.
4. Simplify the transformation pathway by removing any transformations that simply copies an object in the source schema to the target schema.

Step 1: First we translate any SO s-t tgds in the mapping into the 'normal form' we discussed in Section 2.2.1. There we showed that each SO s-t tgd can be rewritten as a conjunction of implications with a single term on the RHS of each implication.

$$\exists fid(\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow \text{table:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e) \wedge$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow \text{column:}\langle\langle\mathsf{EmpIdFinId},\mathsf{eid}\rangle\rangle(e,e) \wedge$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow \text{column:}\langle\langle\mathsf{EmpIdFinId},\mathsf{fid}\rangle\rangle(e,fid(e)) \wedge$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow \text{table:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(fid(e)) \wedge$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow$$
$$\text{column:}\langle\langle\mathsf{FinEmpName},\mathsf{fid}\rangle\rangle(fid(e),fid(e)) \wedge$$
$$\text{table:}\langle\langle\mathsf{Emp}\rangle\rangle(e,n,d) \wedge \text{table:}\langle\langle\mathsf{Dept}\rangle\rangle(d,ne,\texttt{'Finance'}) \rightarrow$$
$$\text{column:}\langle\langle\mathsf{FinEmpName},\mathsf{name}\rangle\rangle(fid(e),n))$$

and

$$\begin{aligned}
&\exists dept, ne(\mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \mathsf{table}{:}\langle\langle\mathsf{Emp}\rangle\rangle(e) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{eid}\rangle\rangle(e,e) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{name}\rangle\rangle(e,n) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{dept}\rangle\rangle(e,dept(e)) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \mathsf{table}{:}\langle\langle\mathsf{Dept}\rangle\rangle(dept(e)) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \\
&\qquad \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{did}\rangle\rangle(dept(e),dept(e)) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \\
&\qquad \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{numEmps}\rangle\rangle(dept(e),ne(dept(e))) \wedge \\
&\quad \mathsf{table}{:}\langle\langle\mathsf{EmpIdFinId}\rangle\rangle(e,f) \wedge \mathsf{table}{:}\langle\langle\mathsf{FinEmpName}\rangle\rangle(f,n) \rightarrow \\
&\qquad \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{dname}\rangle\rangle(dept(e),\texttt{`Finance'}))
\end{aligned}$$

Step 2 : Create the IQL query. We create our IQL query by translating the LHS of each SO s-t tgd in the mapping into IQL using the following rules:

- Each '$\wedge$' becomes a ';'.
- DDL:construct:$\langle\langle$so$\rangle\rangle(\vec{x})$ becomes a generator of the form $\{\vec{x}\} \leftarrow$ DDL:construct:$\langle\langle$so$\rangle\rangle$.
- Conditions of the form $x\ op\ c$ are translated as they are and become a filter in the IQL query.
- Any constants among $\vec{x}$ are added to the end of the query and become equality filters.

For example the query used to define the extent of table:$\langle\langle$EmpIdFinId$\rangle\rangle$ in the mapping above would be as follows when we have expanded the table object out to specify each AutoMed object:

$$\begin{aligned}
&[\{e\} \mid \{e\} \leftarrow \mathsf{table}{:}\langle\langle\mathsf{Emp}\rangle\rangle; \{e,e\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{eid}\rangle\rangle; \{e,n\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{name}\rangle\rangle; \\
&\quad \{e,d\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Emp},\mathsf{dept}\rangle\rangle; \{d\} \leftarrow \mathsf{table}{:}\langle\langle\mathsf{Dept}\rangle\rangle; \{d,d\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{did}\rangle\rangle; \\
&\quad \{d,ne\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{numEmps}\rangle\rangle; \{d,dn\} \leftarrow \mathsf{column}{:}\langle\langle\mathsf{Dept},\mathsf{dname}\rangle\rangle; dn = \texttt{`Finance'}]
\end{aligned}$$

We can then simplify the query to include only those generators whose pattern variables affect the value of the variables in the head of the query, assuming that we have constraints in the schema that inform us that none of the generators we are leaving out change the result. For example the IQL query above can be simplified

to:

$$[\{e\} \mid \{e,d\} \leftarrow \mathsf{column{:}\langle\langle Emp, dept\rangle\rangle}; \{d,dn\} \leftarrow \mathsf{column{:}\langle\langle Dept, dname\rangle\rangle}; dn = \texttt{'Finance'}]$$

We must retain at least one generator for e and some way of restricting the values of e to those that meet the conditions imposed by the filter. e is the primary key of $\mathsf{table{:}\langle\langle Emp\rangle\rangle}$ and SQL:table and SQL:column, the same value of e appears in $\mathsf{column{:}\langle\langle Emp, eid\rangle\rangle}$,$\mathsf{column{:}\langle\langle Emp, name\rangle\rangle}$ and $\mathsf{column{:}\langle\langle Emp, dept\rangle\rangle}$ so we only need one of these objects. We must retain $\mathsf{column{:}\langle\langle Emp, dept\rangle\rangle}$ because it contains the foreign key linking $\mathsf{table{:}\langle\langle Emp\rangle\rangle}$ to $\mathsf{table{:}\langle\langle Dept\rangle\rangle}$. d is the primary key of $\mathsf{table{:}\langle\langle Dept\rangle\rangle}$ so by the same argument as above we only need one of the columns. We must retain $\mathsf{column{:}\langle\langle Dept, dname\rangle\rangle}$ because it contains the pattern for the filter we need.

The generators for these objects, along with the filter $dn =$ `'Finance'`, are sufficient to generate the values required, while maintaining the constraint that all the employees returned by the query are from the Finance department.

Step 3: Create the BAV transformations. We now use the following rules to create a transformation for each object that appears on the RHS of an implication in the normalised SO s-t tgds. In the following, q is the query we created by translating LHS into IQL as described above in Step 2 of the process.

If there are no Skolem functions or constants on the RHS of the implication we use the following translation:

$$LHS \rightarrow S\text{:DDL:construct:}\langle\langle \mathsf{so}\rangle\rangle(\vec{x}) \Rightarrow op(S\text{:DDL:construct:}\langle\langle \mathsf{so}\rangle\rangle, [\{\vec{x}\} \mid q])$$

where LHS is the left hand side of the implication.

If there are Skolem functions used on the RHS of the implication we use this rule:

$$\exists \vec{f}.LHS \rightarrow S\text{:DDL:construct:}\langle\langle \mathsf{so}\rangle\rangle(\vec{x}, f(x_i) \in \vec{f}) \Rightarrow$$
$$op(S\text{:DDL:construct:}\langle\langle \mathsf{so}\rangle\rangle, \{\vec{x'}\} \mid q; y \leftarrow \mathsf{generateGID}(S, x_i, [x_i], \text{`}f\text{'}))$$

where $x_i \in \vec{x}$ and $\vec{x'}$ contains all the elements of $\vec{x}$ as well as an additional element y which is the pattern variable for the generator containing the generateGID function. y will contain Skolem values created by generateGID.

If there are constants used on the RHS of the implication we use this rule:

$$LHS \rightarrow S{:}\mathsf{DDL{:}construct{:}}\langle\langle \mathsf{so} \rangle\rangle(\vec{x}, c) \Rightarrow op(S{:}\mathsf{DDL{:}construct{:}}\langle\langle \mathsf{so} \rangle\rangle, \{\vec{x'}\} \mid q; y \leftarrow c)$$

where c is a constant. $\vec{x'}$ contains all the elements of $\vec{x}$ as well as an additional element y which is the pattern variable for the generator containing the constant. Its value will be that of the constant.

In all these rules *op* is add for all the tgds that map from the source to the target schema. The values in the target schema are all either directly derived from values in the source or are Skolem values which have been calculated using data values from the source schema. This use of the add primitive in transformation queries involving Skolem values created by generateGID was introduced in [KM05].

For the tgds that map from the target to the source, *op* is delete if q fully defines the extent of $\langle\langle \mathsf{so} \rangle\rangle$ and contract if it does not. The decision whether q does indeed fully define the extent of $\langle\langle \mathsf{so} \rangle\rangle$ is made by the person creating the transformations.

Using these rules we generate the following transformation pathway from the mapping given above:

⑩ add(table:⟨⟨EmpIdFinId⟩⟩, table:⟨⟨Emp⟩⟩)
⑪ add(column:⟨⟨EmpIdFinId, eid⟩⟩, column:⟨⟨Emp, eid⟩⟩)
⑫ add(column:⟨⟨EmpIdFinId, fid⟩⟩, [{e, f} | {e, e} ← column:⟨⟨Emp, eid⟩⟩];
f ← generateGID(S$_{emp}$, e, [e], *'fid'*))
⑬ add(table:⟨⟨FinEmpName⟩⟩, [{f} | {e, e} ← column:⟨⟨Emp, eid⟩⟩;
f ← generateGID(S$_{emp}$, e, [e], *'fid'*)])
⑭ add(column:⟨⟨FinEmpName, fid⟩⟩, [{f, f} | {e, e} ← column:⟨⟨Emp, eid⟩⟩;
f ← generateGID(S$_{emp}$, e, [e], *'fid'*)])
⑮ add(column:⟨⟨FinEmpName, name⟩⟩, [{f, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩;
f ← generateGID(S$_{emp}$, e, [e], *'fid'*)])

Note that tgds cannot be used to specify constraints like primary or foreign keys. We can, however, sometimes infer these constraints from the mapping. For example, from the first tgd in $map_{S_{emp},S_{finId}}$ we can infer a foreign key between column:⟨⟨EmpIdFinId, fid⟩⟩ and column:⟨⟨FinEmpName, fid⟩⟩. We can also infer that column:⟨⟨EmpIdFinId, eid⟩⟩ is a primary key because its extent is derived from a primary key column, column:⟨⟨Emp, eid⟩⟩ in the source schema. In other case constraints on the target schema must be added manually. For example, here we add a primary key to column:⟨⟨FinEmpName, fid⟩⟩. The transformations to perform these actions are as follows:

Emp

eid	name	dept
1	Peter Smith	100
3	Paul Jones	100
5	Joe Brown	100
21	Susan Brown	101

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15
102	IT	

Figure 3.16: $\mathsf{Inst}_5(S_{emp})$

⑯ add(primary_key:⟨⟨EmpIdFinId_key, EmpIdFinId, ⟨⟨EmpIdFinId, eid⟩⟩⟩⟩)
⑰ add(primary_key:⟨⟨FinEmpName_key, FinEmpName, ⟨⟨FinEmpName, fid⟩⟩⟩⟩)
⑱ add(foreign_key:⟨⟨FinEmp_fkey, EmpIdFinId, ⟨⟨EmpIdFinId, fid⟩⟩, FinEmpName, ⟨⟨FinEmpName, fid⟩⟩⟩⟩)

Transformations ⑩ to ⑱ make up the growth phase of the transformation pathway. If we assume we start with the instance of S_{emp} shown in Figure 3.16, then the values created by the pathway are shown in Figure 3.17. We can see that for a given value of the key column column:⟨⟨EmpIdFinId, eid⟩⟩, the same Skolem value is generated for column:⟨⟨EmpIdFinId, fid⟩⟩ and column:⟨⟨FinEmpName, fid⟩⟩. These values ensure that the correct row in table:⟨⟨EmpFinId⟩⟩ is associated with the correct row in table:⟨⟨FinEmpName⟩⟩.

If we now populate column:⟨⟨EmpIdFinId, fid⟩⟩ as shown in Figure 3.18(a), the query processor can unify the Skolem values in column:⟨⟨FinEmpName, fid⟩⟩ with the data values in column:⟨⟨EmpIdFinId, fid⟩⟩, using data recorded by generateGID [KM05], to give us the database instance shown in Figure 3.18(b).

EmpIdFinId

eid	fid
1	#1000
3	#1001
5	#1002

FinEmpName

fid	name
#1000	Peter Smith
#1001	Paul Jones
#1002	Joe Brown

Figure 3.17: Values generated in S_{finId} by the growth phase of the pathway where $\mathsf{Inst}_5(S_{emp})$ is the source schema

EmpIdFinId

eid	fid
1	F244
3	F167
5	F153

FinEmpName

fid	name
#1000	Peter Smith
#1001	Paul Jones
#1002	Joe Brown

(a) Before post processing

EmpIdFinId

eid	fid
1	F244
3	F167
5	F153

FinEmpName

fid	name
F244	Peter Smith
F167	Paul Jones
F153	Joe Brown

(b) After post processing

Figure 3.18: $\mathsf{Inst}_5(S_{finId})$

Because constraints are dealt with as schema objects in their own right, BAV pathways allow us to map constraints from source to target and also create constraints

on the target schema if they are required. We see both cases illustrated in the pathway above. The employee id is used as a key on table:⟨⟨EmpIdFinId⟩⟩ as it was on table:⟨⟨Emp⟩⟩ and a new key on table:⟨⟨FinEmpName⟩⟩ is created along with a foreign key linking the tables.

After these transformations have been executed we have a schema that contains objects from both the source and target schemas. In the shrinking phase of the pathway we remove the source schema objects from this combined schema. The SO t-s tgds in our mapping tell us how the target objects in the combined schema can be mapped to the existing source objects. We use them to create a query in terms of target schema objects that allows us to recover the extent of the source objects we remove in the shrinking phase. The transformations are as follows:

⑲ delete(foreign_key:⟨⟨EmpDept_fkey, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
⑳ delete(primary_key:⟨⟨Emp_key, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
㉑ contract(column:⟨⟨Emp, dept⟩⟩, Range $[\{e, d\} \mid \{e, e\} \leftarrow$ column:⟨⟨EmpIdFinId, eid⟩⟩;
$d \leftarrow$ generateGID($S_{EmpDept}$, e, [e], 'dept′)] Any)
㉒ contract(column:⟨⟨Emp, eid⟩⟩, Range column:⟨⟨EmpIdFinId, eid⟩⟩ Any)
㉓ contract(column:⟨⟨Emp, name⟩⟩, Range column:⟨⟨FinEmpName, name⟩⟩ Any)
㉔ contract(table:⟨⟨Emp⟩⟩, Range column:⟨⟨EmpIdFinId, eid⟩⟩ Any)
㉕ contract(column:⟨⟨Dept, dname⟩⟩, Range $[\{d, dn\} \mid \{e, e\} \leftarrow$ column:⟨⟨EmpIdFinId, eid⟩⟩;
$d \leftarrow$ generateGID($S_{EmpDept}$, e, [e], 'dept′); $dn \leftarrow$ `'Finance'`] Any)
㉖ contract(column:⟨⟨Dept, numEmps⟩⟩, Range $[\{d, ne\} \mid \{e, e\} \leftarrow$ column:⟨⟨EmpIdFinId, eid⟩⟩;
$d \leftarrow$ generateGID($S_{EmpDept}$, e, [e], 'dept′); $ne \leftarrow$ generateGID($S_{EmpDept}$, d, [d], 'numEmp′)] Any)
㉗ contract(column:⟨⟨Dept, did⟩⟩, Range $[\{d, d\} \mid \{e, e\} \leftarrow$ column:⟨⟨EmpIdFinId, eid⟩⟩;
$d \leftarrow$ generateGID($S_{EmpDept}$, e, [e], 'dept′)] Any)
㉘ contract(table:⟨⟨Dept⟩⟩, Range $[\{d\} \mid \{e, e\} \leftarrow$ column:⟨⟨EmpIdFinId, eid⟩⟩;
$d \leftarrow$ generateGID($S_{EmpDept}$, e, [e], 'dept′)] Any)

Note that we use the employee id to generate unique values for department id using generateGID. Ideally we would like to use the department name but this is not a candidate key so we would not be guarenteed unique values.

The person creating the transformations for the shrinking phase needs to make all the transformations for the non-constraint objects contract because the instances of the source schema objects we are removing contain employees who are not in the `Finance` department, but the queries we are using to define their extents contain only employees from the `Finance` department. This means we cannot fully define the extents of the source schema objects with queries containing target schema objects. The queries we use define a lower bound for the extents of the source schema objects but not an upper bound so we add the keyword Any to indicate this fact.

3.5 Chapter Summary

We have shown in this chapter how we can use the HDM as an effective CDM in a MMS. We have also shown that declarative mappings given in the form of the widely used SO s-t tgds can be simply translated into executable BAV transformations using IQL as our query language. This is in contrast to using a language like SQL for which there is no straightforward translation [MBHR05].

There are several advantages to using the BAV approach as a mapping and transformation language in our MMS framework.

- The normalised form of SO tgds translates easily into BAV pathways.
- BAV pathways can be automatically inverted, making it trivial to calculate the inverse of a mapping.
- Constraints are dealt with as objects in their own right that can easily be copied or translated from a source to target schema.
- The target schema and mapping are created as part of the same process. There is no need to create the mapping in a separate step after creating the target schema.
- The method can be applied to a range of DDLs.

Chapter 4

Translating Primitive Data Types in AutoMed

In the previous chapter we showed how we can do inter DDL schema and data translation using AutoMed. We did not, however, describe how the data types of the various objects in the source schema are translated into data types in the target schema. We address this issue in this chapter.

The inter DDL translation of data types has received attention in the field of database programming languages [AB87], but little has been written about how primitive data types, *i.e.* integer, float, string *etc.*, should be translated between DDLs in a multi DDL system such as a MMS. Primitive, atomic or simple data types, referred to from now on as **primitive data types**, are the simplest kind of data type. They are used in DDLs to constrain the values that can appear in an instance of a schema object to which they are assigned. Examples of primitive data types include integer, float, boolean, string, *etc.* They represent a set of literals and the extent of a schema object of a given primitive type must be a subset of that set of literals. We refer to the set of primitive data types supported by a DDL as its **primitive type system**.

In a MMS, where a number of different DDLs are supported, it is necessary to have some way of translating the data types of one DDL to another. Existing MMSs like Rondo and MIDST do simple pair-wise type name translations based on the names of the data types. This is also a common approach in inter DDL data integration [RB01, LVLG03]. This approach does not scale well as the number of DDLs supported by a MMS increases and it can lead to some other more subtle

problems:

- The data type of the source object may allow a greater range of values than that of the target object. This could lead to run-time errors when the materialised target schema is populated with data from the source schema.
- Types representing an identical concept may support a different range of values. For example, a boolean in one DDL may represent 'true' as 1 while another may represent it as T. Again ignoring this when populating the target schema would cause errors.

As a solution to the problem of translating primitive data types between DDLs in a MMS, we present a graph based *logical* type hierarchy that can represent the primitive data types of multiple DDLs based on the range of values they support rather than just their names, and that links similar data types from different DDLs by defining bi-directional mappings from the high level data types to a **common, extensible hierarchy** of data types. We use this hierarchy as a type transfer syntax that helps us choose a primitive type for a target schema object in an inter DDL schema translation. We do not attempt to model the physical representation of different data types in different DDLs. In addition we restrict the type hierarchy to character-based types and those that can logically be represented as characters such as integers, floats, chars *etc.* We do not provide a transfer syntax for things like SQL BLOBs and CLOBs.

In addition to the above, the contributions this chapter makes are as follows:

- A way of translating data between schemas whose data types support different values.
- A means of classifying transformations involving typed constructs into those that are illegal, those that need to be checked at run time for data errors and those for which no run time checks for data errors are required.

4.1 The AutoMed Type System

We saw in Chapter 2 that a data type can be included in the scheme of a schema object and that this constrains the extent of that object to be a subset of the data values associated with the type.

During an inter DDL schema translation it is necessary to change the data type of an object from a source DDL type to a target DDL type. The operation of explicitly changing the data type of an object is called **casting**. Whether or not a type cast will result in a loss of information is related to the question of whether two types are **equivalent** and the notion of a **subtype**. Data types A and B are said to be equivalent if all values in A can legally appear in B and *vice versa*. A subtype can be defined as follows: A is a subtype of B if all values in A can legally appear in B. From the above we can deduce that type A can be *safely* cast to type B, *i.e.* with no loss of information, if A is equivalent to B or A is a subtype of B. For example, casting from a `short` to a `long` in SQL is type safe, as is casting an `integer` to a `string`, but going the other way is not.

Deciding whether types are equivalent or subtypes of each other is one of the tasks in **type checking**. This can be either **static** or **dynamic**. Static type checking can be done on declarative mappings that include type information. If these are then translated into executable mappings, additional static checking can be done before the mappings are executed to make sure no errors have been introduced in the translation process. Dynamic checking is done when the executable mappings are executed and is concerned with the actual data values being added to the target schema. Minimising the amount of dynamic checking done is one of the goals of our type hierarchy.

We define our own primitive type system that can link those of a source and target DDL, as a type hierarchy in the form of a **directed acyclic graph** (**DAG**). We also define a number of operations on the graph. Hierarchies exist as a way of describing data types in some well known type systems, for example, Figure 4.1 is a portion of the XML Schema [BM04] type hierarchy. However the familiar tree hierarchy of Figure 4.1 has several shortcomings because the hierarchy does not fully model the domain of data values in XML. In particular:

- Figure 4.1 does not distinguish between types which are disjoint in their extent, such as positiveInteger (representing all positive integer values) and negativeInteger (representing all negative integer values), and those which overlap, such as int (representing $-2^{31}..2^{31}-1$) and unsignedInt (representing $0..2^{32}-1$).

  A mapping between source and target objects where the types are disjoint should be ruled illegal, unless an explicit conversion has been defined, but a mapping between objects that overlap should be allowed, with run-time range checking.

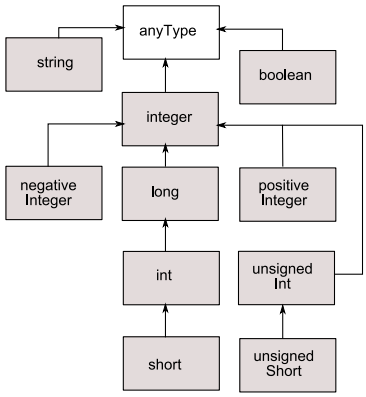


Figure 4.1: A portion of the built-in data type hierarchy as defined for XML Schema in [BM04]

- Figure 4.1, which shows a portion of the type hierarchy defined for XML Schema, fails to identify all isa associations in the hierarchy. For example, all unsignedShort values (which are in the range $0..2^{16}-1$) are a subset of int values. It is also the case that all values in XML can be stored in an object of type string.

  If the source object type is a subtype of the target object type, then the values may be cast without run-time range checking, since the cast will never fail.

Representing the types in Figure 4.1 as a *logical* hierarchy allows us more flexibility in how we define relationships between different types. In the following sections we introduce a definition of the type hierarchy we use in AutoMed that is suited to a MMS. In particular, it builds hierarchies solely on the range of data values a type may take rather than relying on the names of the data types that may be misleading. We then demonstrate how AutoMed translations between schemas in different DDLs may be made type safe.

4.2 The Type Hierarchy

To facilitate the precise representation of types, we introduce in Definition 4.1, a logical type hierarchy to be used to describe the types of a single or collection of DDLs in a manner that allows us to address the problems discussed above. We will show that this type hierarchy can also be used to capture some of the semantics of types that are specific to a particular schema.

Definition 4.1 Type Hierarchy

A type hierarchy TH_x is a tuple $\langle Types_x, Ext_x, \stackrel{t}{=}_x, \prec_x, \not\wedge_x, TypeMapping_x \rangle$ that includes:

- A finite set of type names $Types_x$ that always includes the special value, $anyType$. These will make up the nodes of the DAG.
- An Ext_x function that returns a set of values such that for $t \in Types_x \rightarrow Ext_x(t) \subseteq Ext_x(anyType_x)$

  $Ext_x(anyType)$ represents the set of data values in the domain of discourse of the MMS.
- An **equality** relation, $\stackrel{t}{=}_x$, such that for $t, t' \in Types_x$
  $t \stackrel{t}{=}_x t' \iff Ext_x(t) = Ext_x(t')$
- A **partial ordering** relation $\prec_x$, such that for $t, t' \in Types_x$
  $t \prec_x t' \iff Ext_x(t) \subset Ext_x(t')$

  $\forall t \in Types_x, t \prec_x anyType$

  When all types $t, t' \in Types_x$ that have $t \stackrel{t}{=}_x t'$ are treated as a single node, the $\prec_x$ relation builds the types into a connected directed acyclic graph.

  $\stackrel{t}{=}_x$ and $\prec_x$ together make up the edges of the graph.
- A **disjoint** operator, $\not\wedge_x$, such that for $t, t' \in Types_x$
  $t \not\wedge_x t' \iff Ext_x(t) \cap Ext_x(t') = \phi$. This implies there is no pathway from t to t' in the graph and so we cannot cast between t and t'. If $t \not\wedge_x t'$ then any subtype of t will also be disjoint from t'.
- A set $TypeMapping_x^{a,b}$ of mapping tables, added on a per schema basis such that
  $\langle t_a, t_b, \{\langle s_a^1, s_b^1\rangle, \ldots, \langle s_a^n, s_b^n\rangle\}\rangle$, where $t_a, t_b \in Types_x$, $s_a^1, \ldots, s_a^n$ are subsets of $Ext(t_a)$ and are disjoint. Similarly $s_b^1, \ldots, s_b^n$ are subsets of $Ext(t_b)$ and are disjoint.

  These mapping tables allow us to map specific values, such as those used to denote booleans, from one DDL to another.

  We overload $TypeMapping_x$ to be used as a function with the following definition:

  $$TypeMapping_x(t_a, t_b, v_a) = First\{(s_b^i) \mid \langle t_a, t_b, map\rangle \in TypeMapping_x \wedge \langle s_a^i, s_b^i\rangle \in map \wedge v_a \in s_a^i\}$$

$$TypeMapping_x^{-1}(t_a, t_b, v_a) = First\{(s_b^i) \mid \langle t_b, t_a, map\rangle \in TypeMapping_x \wedge \langle s_b^i, s_a^i\rangle \in map \wedge v_a \in s_a^i\}$$

where i is any value from 1 to n and *First* returns the first element of a set according to a sort order that is fixed for the system. We use $TypeMapping_x(type_a, type_b)$ to denote the specific mapping table that maps $type_a$ to $type_b$.

We will leave the subscripts off the operators when their value is clear from the context.

□

From the definitions above we can make statements about the types in an AutoMed type hierarchy. For example we can say that t is a **subtype** of t' if $t \prec_x t'$.

Examples 4.1 and 4.2 illustrate how we create the type hierarchies for XML and Postgres.

Example 4.1 XML Schema Logical Type Hierarchy

The logical type hierarchy for XML Schema, TH_{xml}, shown in Figure 4.2, can be derived from Figure 4.1 as follows (note that in this and the next example we only include an indicative subset of the types and relationships):

$$\begin{aligned}
Types_{xml} &= \{anyType, string_{xml}, boolean_{xml}, integer_{xml}, positiveInteger_{xml}, \\
&\quad negativeInteger_{xml}, long_{xml}, int_{xml}, short_{xml}, unsignedShort_{xml}\} \\
Ext_{xml} &= \{string_{xml} \rightarrow \text{The set of finite-length sequences of valid XML characters} \\
&\quad boolean_{xml} \rightarrow \{true, false, 1, 0\}, integer_{xml} \rightarrow \{\ldots, \text{-1,-2,0,1,2, } \ldots\}, \\
&\quad positiveInteger_{xml} \rightarrow \{1,2,3, \ldots\}, \\
&\quad negativeInteger_{xml} \rightarrow \{\ldots, \text{-3,-2,-1}\}, \\
&\quad long_{xml} \rightarrow \{-2^{63}, \ldots, 2^{63}-1\}, int_{xml} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, \\
&\quad short_{xml} \rightarrow \{\text{-32768, } \ldots, \text{32767}\}, unsignedShort_{xml} \rightarrow \{0, \ldots, 65535\}\} \\
\stackrel{t}{=}_{xml} &= \{anyType \stackrel{t}{=} string_{xml}\} \\
\prec_{xml} &= \{short_{xml} \prec int_{xml}, unsignedShort_{xml} \prec int_{xml}, \\
&\quad int_{xml} \prec long_{xml}, long_{xml} \prec integer_{xml}, integer_{xml} \prec string_{xml}, \ldots\} \\
\not\wedge_{xml} &= \{negativeInteger_{xml} \not\wedge positiveInteger_{xml}, \\
&\quad negativeInteger_{xml} \not\wedge unsignedShort_{xml}\} \\
TypeMapping_{xml} &= \{\}
\end{aligned}$$

□

In Example 4.2 we create a logical type hierarchy for Postgres based on the built-in casting rules in Postgres that give us isa relationships. The resultant hierarcharchy is shown in Figure 4.3.

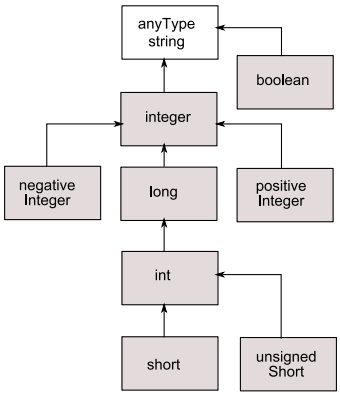


Figure 4.2: TH_{xml} showing the extra isa relationships we have created

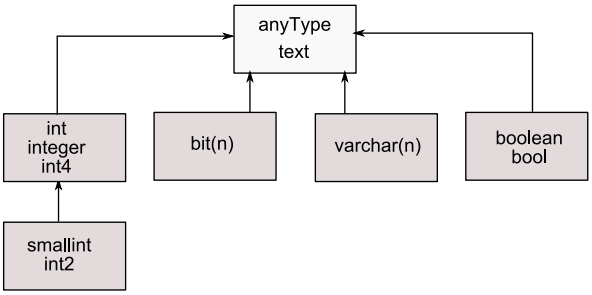


Figure 4.3: A portion of the Postgres type system represented as a logical hierarchy

Note that in this logical hierarchy, integers are a subset of the type text, implying that we can store an integer in a column of type text.

Example 4.2 Postgres Logical Type Hierarchy

$$
\begin{aligned}
Types_{pg} &= \{anyType, text_{pg}, varchar(n)_{pg}, boolean_{pg}, bool_{pg}, integer_{pg}, int_{pg}, int4_{pg}, \\
&\quad smallint_{pg}, int2_{pg}, bit(n)\} \\
Ext_{pg} &= \{text_{pg} \rightarrow \text{The set of all valid Postgres strings} \\
&\quad varchar(n)_{pg} \rightarrow \text{The set of all valid Postgres strings up to length } n \\
&\quad boolean_{pg} \rightarrow \{0,1,y,n,yes,no,t,f,true,false\}, \\
&\quad bool_{pg} \rightarrow \{0,1,y,n,yes,no,t,f,true,false\}, \\
&\quad integer_{pg} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, int_{pg} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, \\
&\quad int4_{pg} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, \\
&\quad smallint_{pg} \rightarrow \{\text{-}32768, \ldots, 32767\}, int2_{pg} \rightarrow \{\text{-}32768, \ldots, 32767\}, \\
&\quad bit(n)_{pg} \rightarrow \text{The set of bit strings up to length } n\} \\
\stackrel{t}{=}_{pg} &= \{text_{pg} \stackrel{t}{=} anyType, boolean_{pg} \stackrel{t}{=} bool_{pg}, int_{pg} \stackrel{t}{=} integer_{pg}, int_{pg} \stackrel{t}{=} int4_{pg}, \\
&\quad smallint_{pg} \stackrel{t}{=} int2_{pg}\} \\
\prec_{pg} &= \{smallint_{pg} \prec integer_{pg}, integer_{pg} \prec text_{pg}, bit(1)_{pg} \prec bit(2)_{pg}, \ldots\} \\
\not\wr_{pg} &= \{\} \\
TypeMapping_{pg} &= \{\}
\end{aligned}
$$

□

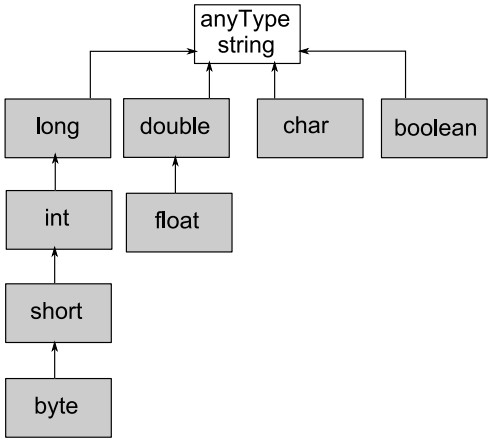


Figure 4.4: The AutoMed common type hierarchy

4.3 Inter DDL Data Type Translation

In this section we describe how we merge the type hierarchies of a source and target schema to create an inter DDL type hierarchy that allows us to cast source data types to target data types.

Our solution is based on a **Common Type Hierarchy** (**CTH**) which acts as an intermediary between the primitive type systems of the source and target DDLs. A user need only create a bidirectional mapping between the types in his DDL and the common type hierarchy rather than having to know how to map to all the other DDLs in the MMS. If the number of DDLs in the MMS is n, the number of bidirectional mappings required using our method is $n-1$ as opposed to the $n(n-1)/2$ that are needed if mappings are created between each pair of DDLs in the MMS. Our approach is similar to a method proposed in [HFG87] for translating between different relational query languages via a common intermediate query language in the PROTEUS system.

4.3.1 The Common Type Hierarchy

If the source and target schemas are defined in different DDLs we need a way of linking the type hierarchy defined for the source model, the **source type hierarchy**, to that defined for the target model, the **target type hierarchy**. We do this using the CTH, the base version of which is shown in Definition 4.2. It is very general but is extended and made more precise as it is used to link other type hierarchies as we will see later. It is shown graphically in Figure 4.4.

Using the CTH as an intermediary means that we only need to create a set of associations between the data types of a DDL in our MMS and those of the CTH. We do not need to define new associations to all existing DDLs each time we add a new DDLs to our MMS. We use the CTH as the primitive type system for the HDM. As AutoMed is implemented in Java, we have chosen the extents of the types in the CTH to be based on the values supported by corresponding Java data types.

Definition 4.2 Common type hierarchy

$$
\begin{aligned}
Types_c &= \{anyType, string_c, long_c, int_c, short_c, byte_c, double_c, float_c, char_c, boolean_c\} \\
Ext_c &= \{string_c \rightarrow \text{A set of all strings up the maximum length allowed by Java}, \\
&\quad long_c \rightarrow \{-2^{63}, \ldots, 2^{63}-1\}, int_c \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, \\
&\quad short_c \rightarrow \{-32768, \ldots, 32767\}, byte_c \rightarrow \{-128, \ldots, 127\}, \\
&\quad double_c \rightarrow \text{The set of 64-bit floating point numbers}, \\
&\quad float_c \rightarrow \text{The set of 32-bit floating point numbers}, \\
&\quad char_c \rightarrow \text{A set of single characters supported by the Java character set}, \\
&\quad boolean_c \rightarrow \{true, false\}\} \\
\stackrel{\text{t}}{=}_c &= \{anyType \stackrel{\text{t}}{=} string_c\} \\
\prec_c &= \{byte_c \prec short_c, short_c \prec integer_c, integer_c \prec long_c, long_c \prec string_c, \\
&\quad float_c \prec double_c, double_c \prec string_c, boolean_c \prec string_c, char_c \prec string_c\} \\
\not\wedge_c &= \{\} \\
TypeMapping_c &= \{\}
\end{aligned}
$$

□

4.3.2 Adding a High Level DDL Type System to AutoMed

Algorithm 4.1 defines the procedure for adding the associations between the high level DDL types and those in the CTH necessary to add the type system of the high level DDL to AutoMed. It is done by hand when adding a new DDL to AutoMed and the resultant definition of the merged type hierarchy becomes part of the wrapper for the new DDL. The parameters of AddTH, are: TH_x, the type hierarchy of the source DDL, TH_c, the current CTH and M_{x-c}, a set of user defined mappings from types in TH_x to those in TH_c.

We first add all the types, extents and type relationships from the new DDL to the CTH (lines 1 to 7). We then try to find any existing data types in the CTH that have an equivalent extent to any of the new data types. If such a data type is found in the CTH we add an equality relationship between it and the new type to the CTH (line 12). We also add any other type relationships that exist between the CTH type and any other data types, to the new type (lines 14 to 18). If no such

Algorithm 4.1: AddTH(TH_x,TH_c,M_{x-c})

```
1  Types_c := Types_x ∪ Types_c;
2  Ext_c := Ext_x ∪ Ext_c;
3  ≐^t_c := ≐^t_x ∪ ≐^t_c;
4  ≺_c := ≺_x ∪ ≺_c;
5  ⊮_c := ⊮_x ∪ ⊮_c;
6  TypeMapping_c := M_{x-c} ∪ TypeMapping_x ∪ TypeMapping_c;
7  ≐^t_c := ≐^t_c ∪ anyType_x ≐^t anyType_c
8  foreach t_x ∈ Types_x do
9      equivalentTypeFound := false;
10     foreach t_c ∈ Types_c do
11         if Ext(t_x) ⊆ Ext(t_c) ∧ Ext(t_c) ⊆ Ext(t_x) then
12             ≐^t_c := ≐^t_c ∪ {t_x ≐^t_c t_c};
13             equivalentTypeFound := true;
14             foreach t'_c ∈ Types_c do
15                 if t_c ⊮_c t'_c then
16                     ⊮_c := ⊮_c ∪ {t_x ⊮ t'_c};
17                 if t_c ≺_c t'_c then
18                     ≺_c:=≺_c ∪{t_x ≺ t'_c};
19     if !equivalentTypeFound then
20         ⟨t_c, t'_c⟩ := placeTypeInHierarchy(t_x,Ext(t_x));
21         ≺_c := ≺_c ∪{t_x ≺ t_c} ;
22         ≺_c := ≺_c ∪{t'_c ≺ t_x} ;
23 return ⟨Types_c, Ext_c, ⊮_c, ≐^t_c, ≺_c, TypeMapping_c⟩
```

equivalent type can be found we need to place the type in the hierarchy by finding the data types in the CTH that most closely bound the extent of the new data type. The function placeTypeInHierarchy performs this task (line 20). At present it must be done by hand as it is not always readily deducible which branch of the CTH a new data type should go into. For example the extent of an XML `unsignedShort` is a subset of both string and int in the CTH. As this process only needs to carried out once, when the wrapper for the DDL is designed, the overhead is very small.

During an inter DDL schema translation the associations created by Algorithm 4.1, between the CTH and the source type hierarchy and between the CTH and the target hierarchy, allow us to form an **inter DDL type hierarchy**. We now describe how we use the type hierarchies defined in Examples 4.1 and 4.2 to create a merged hierarchy that allows us to translate the types of schema objects between the two DDLs.

We add the XML Schema logical hierarchy in Figure 4.2 with the following execution of AddTH.

$$TH_c = \mathsf{AddTH}(TH_{xml}, TH_c, \\ \langle boolean_{xml}, boolean_c, \{\langle\{0,\text{false}\}, \{\text{false}\}\rangle, \langle\{1,\text{true}\}, \{\text{true}\}\rangle\}\rangle)$$

We now add our Postgres type hierarchy from Figure 4.3 to AutoMEDas follows:

$$TH_c = \mathsf{AddTH}(TH_{pg}, TH_c, \\ \langle bool_{pg}, boolean_c, \{\langle\{0,\text{n,no,f,false}\}, \{\text{false}\}\rangle, \langle\{1,\text{y,yes,t,true}\}, \{\text{true}\}\rangle\}\rangle)$$

The two mapping tables we have created, $TypeMapping_c(boolean_{xml}, boolean_c)$ and $TypeMapping_c^{-1}(boolean_{pg}, boolean_c)$ allow us to map from a boolean in XML to a boolean in Postgres.

The CTH generated by the above process, and shown in Figure 4.5, provides us with an inter DDL type hierarchy for the hierarchies in Figures 4.2 and 4.3. Each rectangle represents a set of $\stackrel{t}{=}$ relationships and $TypeMapping$s and the thin lines with arrows on them represent the partial ordering provided by the $\prec$ operator.

4.3.3 Functions Based on the Inter DDL Type Hierarchy

We now define two functions that can be applied to an AutoMed type hierarchy. The first, shown in Algorithm 4.2, translates a data type from one DDL to another based on the CTH. The second, shown in Algorithm 4.3, provides a means of checking

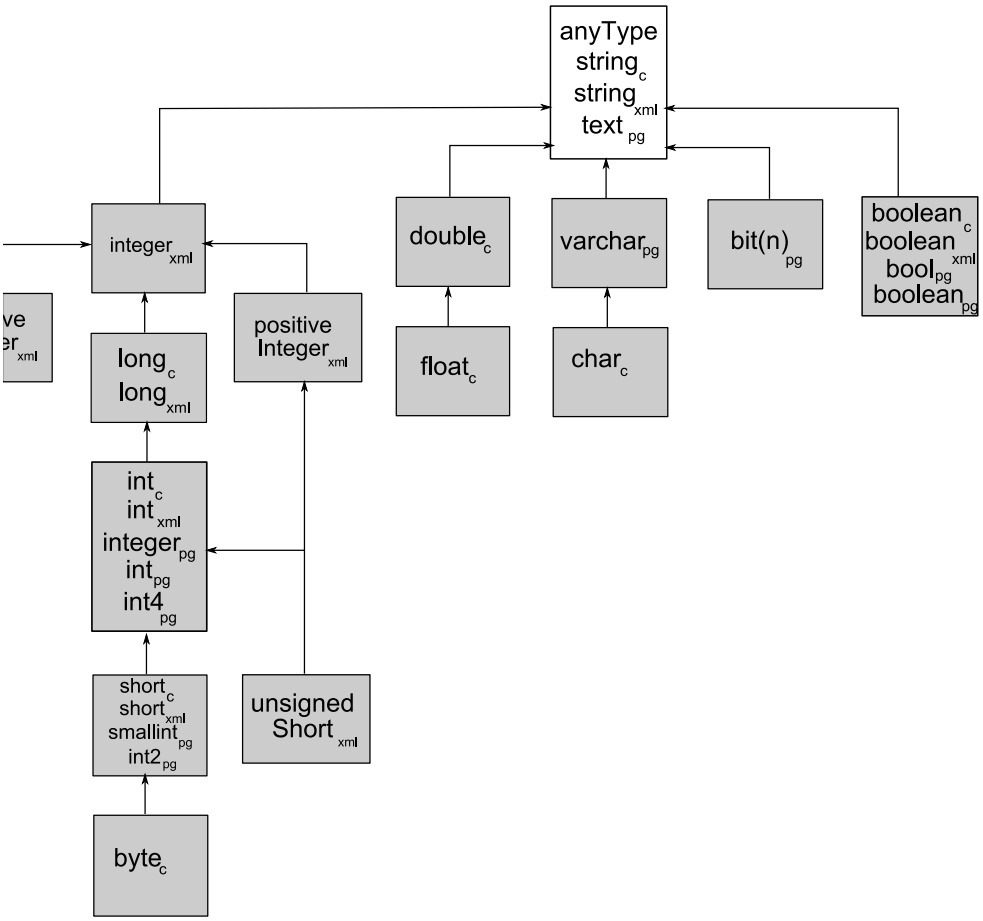


Figure 4.5: The CTH including the types from Figures 4.2 and 4.3

whether a mapping between objects in existing schemas may cause problems because of the data types of the objects being mapped.

As an example, assume our CTH is the one shown in Figure 4.5 and we wish to compute typeTrans$(short_{xml}, Types_{pg})$. The function finds the equality $short_{xml} \overset{t}{=} smallint_{pg}$ to give us the result $smallint_{pg}$ in line 2 of the algorithm. In some cases we may not be able to find a data type from the target DDL in the CTH that matches the source data type. In this case we execute the loop on lines 3 to 6 of the algorithm moving up one level up the type hierarchy at a time, using the partial orderings defined in $\prec_c$, and trying to find a match between types from the source DDL and the target DDL at each level of the hierarchy. We continue doing this until we get to $anyType$. For example typeTrans$(negativeInteger_{xml}, Types_{pg})$ would not find any Postgres types equivalent to $negativeInteger_{xml}$. We use $\prec_c$ to find a Postgres type above $negativeInteger_{xml}$ in the hierarchy. Looking at Figure 4.5 we see there are no Postgres types above $negativeInteger_{xml}$ so we get the result $anyType$.

Algorithm 4.2: typeTrans(t_{sddl},$Types_{tddl}$,$Types_c$)

Input: t_{sddl} : the data type of a schema object in the source schema,
$Types_{tddl}$: the types in the target DDL,
$Types_c$: the common type hierarchy
Output: t_{tddl}: a primitive type $\in Types_{tddl}$

```
1 if ∃t ∈ Types_tddl such that t_sddl =^t_c t then
2     return t;
3 foreach t_c ∈ Types_c such that t_sddl ≺_c t_c do
4     t' := typeTrans(t_c, sddl, tddl);
5     if t' ≠ anyType then
6         return t';
7 return anyType;
```

If the result of typeTrans is *anyType* we have lost the type information in the translation and the target needs to be materialised with some form of generic data type. What precisely this is will depend on the target DDL. For example in XML, the `string` type can represent all values and in Postgres columns can be assigned the special `unknown` type.

Note that typeTrans never lets us move down the CTH as this would mean the target data type is more restrictive than the source and not all the values from the source may be valid, which is a situation we wish to avoid.

Algorithm 4.3: checkType($\langle\langle$so$\rangle\rangle$,$Types_c$,t_{sddl},t_{tddl})

Input: $\langle\langle$so$\rangle\rangle$: the source schema object,
$Types_c$: the common type hierarchy,
t_{sddl}: the type of the source schema object,
t_{tddl}: the type of the target schema object
Output: *cons*: a constraint string

```
1  if t_sddl ↛ t_tddl ∈ ↛_c then
2      throw IllegalCastException;
3  if ∃t_sddl =^t_c t_tddl then
4      return null;
5  foreach t_c ∈ Types_c such that t_sddl ≺_c t_c do
6      if t_c ≠ anyType then
7          cons := checkType(⟨⟨so⟩⟩,t_c,t_sddl);
8          return cons;
9      else
10         cons = Ext(so) ⊆ Ext(t_tddl);
11         return cons;
```

checkType identifies two different types of data type translation problem. Firstly, in lines 1 and 2, it identifies illegal castings. If our pathway from source to target includes data types t and t' such that $t \not\cap t'$ then the cast is illegal from Definition 4.1 and the function throws an exception. For example if the source object was of XML type `positiveInteger` and the target `negativeInteger`, the IllegalCastException would be thrown. Secondly on lines 3 to 11 the algorithm can tell when a translation is possible but with a constraint. We look for the target type higher in this branch of the CTH. If we find it then we return null. If we get to the top of the branch and t_c is *anyType* we know a constraint is necessary. We cannot be certain that the extent of the source object will be within the extent of the target type. The necessary constraint is created on line 10 and must be checked at run-time.

Assume we have an XML source schema that contains
simpleElement:⟨⟨staff/Dept, did, `int`⟩⟩
and that this object is mapped to the Postgres object
column:⟨⟨Dept, did, `smallint`⟩⟩.
We can use the function
checkType (simpleElement:⟨⟨staff/Dept, did, `int`⟩⟩,column:⟨⟨Dept, did, `smallint`⟩⟩)
to see if the mapping is type-safe. We first check to see if we can find the type higher in the hierarchy. In this case we are unable to do this, and to get from int_{xml} to $smallint_{pg}$ we need to go down the hierarchy. Any time it is necessary to move down the CTH to translate from one type to another, as in this case, we may only be able translate a subset of the instances of the object, and so generate a constraint to reflect this. Here we find the XML equivalent of $smallint_{pg}$, *i.e.* $short_{xml}$ and generate the constraint: $Ext(\text{simpleElement:⟨⟨staff/Dept, did, int⟩⟩}) \subseteq Ext(short_{xml})$. If checkType returns null then the mapping is type safe for this object.

4.3.4 Avoiding data errors

One of the aims of the type system in AutoMed is to reduce run time type checking. Transforming the types in a rigorous way allows us to do away with these checks altogether in a number of cases and when some checks are unavoidable we will be able to define what checks need to be done. AutoMed itself does not validate any of the data type constraints but we assume a target system will, and that exceptions will be raised if invalid data is sent to it. This is a situation we are aiming to avoid.

For example $short_{xml}$ and $smallint_{pg}$ are both equivalent to $short_c$. We can say with confidence that any value from a construct of type $short_{xml}$ can be stored in a

Emp

eid(int4)	name(varchar)	pension(bool)	salary(integer)
1	Peter Smith	yes	223400
21	Susan Brown	no	23560

Figure 4.6: Postgres Schema

construct of type $smallint_{pg}$. We thus know that values from objects of these types can be safely interchanged without checking. Similarly if we run checkType on an object whose type is $short_{xml}$ and whose target type is int_{pg} we would get a null result and know that we can safely translate all instances of the source object into the target.

As we saw in the example above, when we needed to move down the CTH to create a data type translation, our system can let us know when run-time checking is needed.

4.4 Type Translation Example

Figure 4.6 shows a Postgres database we wish to translate into XML. We include the data types of the columns in round brackets after the column name.

The initial data type of the AutoMed schema objects match the data type of the corresponding data source objects and are assigned by the wrapper for the relevant high level DDL. If a data source object does not have an associated type, for example a SQL table or an XML complex element, we give it the type *anyType*.

We focus our attention on column:⟨⟨Emp, salary, `integer`⟩⟩ and column:⟨⟨Emp, pension, `bool`⟩⟩. column:⟨⟨Emp, salary, `integer`⟩⟩ can contain any integer up to $2^{31} - 1$. However, the company has introduced a salary cap of 30000 and so has chosen `short` as the data type for the salary element in the XML schema. We thus face the possibility of errors because the largest positive number an XML element with type `short` can hold is 32677. If there is a salary that is greater than 32677 in the database and we created an XML instance document with that value, the XML Schema verification on this document would cause an error. This transformation is not type safe.

Similarly, if we try to put the boolean value `'no'` from the pension column into a `boolean` XML element the operation will fail, since this Postgres boolean value

cannot be stored in a `boolean` XML element (which only supports the values {`0,1,true,false`}). This transformation is also not type safe.

4.4.1 Example BAV Transformations with Data Types

The following example is the growth phase of the transformation pathway that translates the database in Figure 4.6 into XML. The shrinking phase simply removes the SQL schema objects and the data types are not of concern so we do not include the shrinking phase in the example.

Example 4.3 Transformation pathway

In this example the mapping we are given does not include the data types of the target schema objects. We use the typeTrans function and the CTH shown in Figure 4.5 to do the type translation.

① add(complexElement:⟨⟨null, staff, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
② add(complexElement:⟨⟨staff, FinEmp, 0, `unbounded`⟩⟩,
  [{r, e} | {e, e} ← column:⟨⟨Emp, eid⟩⟩; $r \leftarrow$ &0])
③ add(attribute:⟨⟨staff/FinEmp, eid, typeTrans(`int4`, $\text{Types}_{\text{xml}}$), `required`⟩⟩,
  column:⟨⟨Emp, eid⟩⟩)
④ add(simpleElement:⟨⟨staff/FinEmp, name, typeTrans(`varchar`, $\text{Types}_{\text{xml}}$), 1, 1⟩⟩,
  column:⟨⟨Emp, name⟩⟩)
⑤ add(simpleElement:⟨⟨staff/FinEmp, pension, typeTrans(`bool`, $\text{Types}_{\text{xml}}$), 1, 1⟩⟩,
  column:⟨⟨Emp, pension⟩⟩)
⑥ add(simpleElement:⟨⟨staff/FinEmp, salary, typeTrans(`integer`, $\text{Types}_{\text{xml}}$), 1, 1⟩⟩,
  column:⟨⟨Emp, salary⟩⟩)

□

The datatype parameter of the typeTrans function in the transformations above is the datatype of the source schema object whose extent the target object is derived from. For example in Transformation ④ the extent of simpleElement:⟨⟨staff/FinEmp, name⟩⟩ is derived from column:⟨⟨Emp, name⟩⟩ whose data type is `varchar`.

In Transformation ⑤ transType uses $TypeMapping_{TH_c}$ and $TypeMapping^{-1}_{TH_c}$ to explicitly map values of $bool_{xml}$ to $boolean_{pg}$. This overcomes the incompatibility of the XML Schema `boolean` and Postgres `bool` data types discussed earlier. Executing the transformation pathway and materialising the resultant schema will give us the XML Schema shown in Figure 4.7.

```
<xsd:complexType name = "employee_type">
 <xsd:sequence>
  <xsd:element name = "name" type = "xsd:string" />
  <xsd:element name = "pension" type = "xsd:boolean" />
  <xsd:element name = "salary" type = "xsd:int" />
 </xsd:sequence>
 <xsd:attribute name = "eid" type = "xsd:int" use = "required"/>
</xsd:complexType>
<xsd:element name = "staff">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name = "FinEmp" type = "employee_type"
            minOccurs = "0" maxOccurs = "unbounded" />
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Figure 4.7: XML target schema

If we now assume that the XML Schema already exists and that column:⟨⟨Emp, salary, `integer`⟩⟩ is mapped to simpleElement:⟨⟨staff/Emp, salary, `short`⟩⟩, rather than an element of type `int`, we can use checkType to see if this is a type-safe translation. Looking at Figure 4.5 we see that we have to move down the hierarchy to get from $integer_{pg}$ to $short_{xml}$, so we know a constraint must be generated. To work out what the constraint is we need find the equivalent type to $short_{xml}$ in $Types_{pg}$. We can see this is $smallint_{pg}$ or $int2_{pg}$. The function thus generates the constraint:
$Ext(\text{column:}\langle\langle \text{Emp}, \text{salary}, \texttt{integer}\rangle\rangle \subseteq Ext(smallint_{pg}))$
The function also confirms that there are no disjoint pairs of types in the translation. We can check the constraint before we transfer data to the source schema and so be guaranteed that we will not transfer any values that break the type constraints of the XML target schema. This allows us to see the value of 223400 in the salary column of one of the employees was an error, probably caused by incorrect data entry.

Using the type hierarchy has allowed us to translate the primitive data types of our Postgres schema objects into their XML Schema equivalents including translating the values of the Postgres `bool` type into valid values of the XML `boolean` type. In the case of the existing target schema we have also identified a potential type casting problem in one column that will need to be checked during run-time. Without the explicit identification of this problem a MMS that materialised this target schema would either need to check the data values of all the columns or adopt a no-checking policy that could lead to unexpected problems as we saw in the example.

4.5 Related Work

Data types are often discussed in a data integration and exchange context but are seldom dealt with in any detail. TSIMMIS [GMPQ+97] provides for type information to be stored in their Object Exchange Model [PGMW95] but does not use this information to test the safety of transformations. WOL [DK97] is another language for database transformations that stores type information; however, the language is only able to describe transformations in relational and object-relational databases, not generic inter DDL transformations. The Clio system [FKMP05, MHH00] uses s-t tgds without any added type information to specify how and what source data should appear in the target. This data-centric approach does not make use of type information in the source schema to help map to the target schema. Some methods ignore the problem altogether [BFH+03, LVLG03] and make no explicit mention of how primitive data types from one DDL are transformed into the other model. In ignoring type information these systems risk losing expressiveness during the transformations [AB87] and allowing type-incompatible transformations to be written. Rahm and Bernstein [RB01, MBR01] use a special synonym table to match data types between different DDLs in Cupid [MBR01]. This method is effective when mapping between two specific DDLs but does not meet the requirements of a MMS that supports multiple DDLs.

4.6 Chapter Summary

In this chapter we have described how to translate primitive data type information from one DDL to another. We do this using a common type hierarchy that means we do not need to define pair-wise translations for all the DDLs supported by the MMS, but only need to define how the data types of a DDL map to the common type hierarchy. A formal definition of the type system has been provided and it has been shown how this can be included in AutoMed.

Chapter 5

ModelGen in AutoMed

One of the most important features of a MMS is the ability to process schemas from multiple DDLs. The operator ModelGen provides this functionality by translating a schema expressed in one DDL into a corresponding schema in another DDL and also produces a mapping between the schemas. To date, no implementation of ModelGen completely meets these criteria [BM07]. In the example in the introduction we saw how we were able to use ModelGen to translate an ER schema into SQL, and then translate part of that database into XML. We were then able to use the mappings generated as parameters to other MM operators later on in the script. Doing both translations within a single system means we can create a mapping all the way from the ER schema to the XML document.

In this chapter we describe an implementation of ModelGen that creates:

1. data level translations between a source schema and a corresponding schema in a target DDL, by the composition of the CTs described in Section 3.3.2, and
2. a bidirectional mapping between the source and target schemas.

In common with other implementations of ModelGen [ACB05, KQLL07] we do not attempt to show formally that our translations are correct, but rather adopt a heuristic approach that includes rules for detecting when an attempted translation has failed. The majority of this work appeared in [SM08b] and [SM08a].

Figure 5.1 gives an overview of our approach to implementing ModelGen. In Step 1 the source schema S_s is translated into an equivalent HDM schema, S_{hdm-s}, using

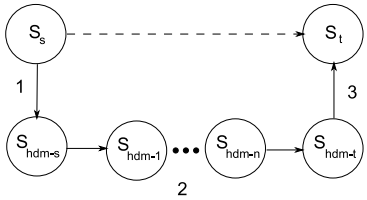


Figure 5.1: Overview of the approach taken

the production rules described in Chapter 3. In Step 2 we apply a series of the composite transformations introduced in Chapter 3 to objects in S_{hdm-s} to transform the schema into S_{hdm-t}, that is an HDM schema equivalent to some schema in the target DDL. The system automatically chooses a suitable CT to apply to any objects that cannot be directly translated into the target DDL. The choice of CTs that can be applied is limited by preconditions relating to the structure of the schema surrounding the object. In Step 3 the constructs in S_{hdm-t} are translated into their equivalent construct in the target DDL, to create S_t.

Note that Steps 1 and 3 are both essentially ModelGen since they translate schemas from the source DDL to the HDM and from the HDM to the target DDL. In the existing literature [ACB05, BM07], however, the translation from source DDL to CDM and from CDM to target DDL has not been called ModelGen, only the overall process. We will maintain that usage here.

We showed in Chapter 3 that we can represent schemas from a wide range of DDLs in AutoMed using the HDM. We have also seen that CTs can be used to transform those HDM schemas into equivalent schemas. Here we present an *automatic* way of transforming the HDM representation of a source schema into an HDM representation that can be translated directly into a target DDL.

We saw in the previous chapter that we can write mappings directly from one high level DDL to another. Adapting this approach to the implementation of ModelGen, however, would mean writing forward and reverse translations between each pair of DDLs in our MMS, resulting in many separate rules and limited scalability. Going via the HDM allows us to take advantage of the common elements between many of the high level DDLs, captured by the HDM thanks to its simple structure and constructs.

The schemas and CTs used in the transformation in Step 2 in Figure 5.1 can be represented as a search graph, as shown in Figure 5.2, whose nodes are the individual HDM schemas and whose edges are the CTs used to transform the schemas. Two algorithms are involved in traversing this graph. The first, described in Section 5.2,

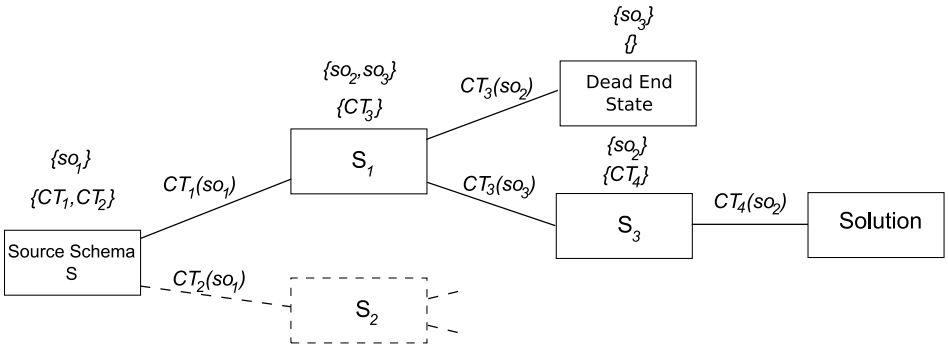


Figure 5.2: The process by which we transform a schema in Step 2 of Figure 5.1

identifies schema objects within the current HDM schema that do not match HDM equivalents of constructs in the target DDL. In the figure these are shown as so_n above the rectangles. The second algorithm, described in Section 5.3, chooses an appropriate CT at each step in the process to transform these unmatched objects into ones that do match HDM equivalents of constructs in the target DDL. The possible CTs are shown as CT_n in the figure. This algorithm performs a depth first search of the possible solution space, starting from the initial state and executing CTs until a solution state or dead end state is reached.

In the figure, the matching algorithm is unable to match one object, so_1, in the source schema to an HDM representation of constructs in the target DDL. Two CTs match the preconditions on so_1 in S. CT_1 is executed first resulting in S_1. Two schema objects are now unmatched. One CT matches the preconditions. It is executed first on so_2. This results in a schema where so_3 is still unmatched but no CTs match the preconditions for so_3 in this schema. We are therefore at a dead end. The transform algorithm now backtracks to S_1 to try CT_3 to so_3. This results in schema S_3 where so_2 is unmatched but CT_4 matches the preconditions. CT_4 is applied to so_2 resulting in a schema where all the schema objects match the HDM representation of constructs in the target DDL and thus a solution has been found.

The process is shown in more detail in Algorithm 5.1.

We first define the two global variables that we use in the Transform algorithm described in Section 5.3. We then apply the HLtoHDM function we defined in Chapter 3 to translate S_s into the HDM. The result of this translation is transformed into an HDM schema that matches the structure of an HDM schema in $TDDL$ using Transform. HDMtoHL, described in Section 5.4, translates the schema that was the result of Transform into $TDDL$. Finally the pathways returned by the three phases of the algorithm are composed to give us a single pathway from S_s to S_t.

Algorithm 5.1: ModelGen(Schema S_s, TargetDDL $TDDL$)

1 TC := The list of constructs in $TDDL$;
2 CT := The list of general purpose CTs;
3 $p_{S_s,S_{hdm-s}}$:= HLtoHDM(S_s);
4 $\langle p_{S_{hdm-s},S_{hdm-t}}, MO, HIC\rangle$:= Transform(S_{hdm-s},new List());
5 p_{S_{hdm-t},S_t} := HDMtoHL($S_{hdm-t}, TDDL$);
6 $p_{S_s,S_t} := p_{S_s,S_{hdm-s}} \circ p_{S_{hdm-s},S_{hdm-t}} \circ p_{S_{hdm-t},S_t}$;
7 **return** $\langle S_t, p_{S_s,S_t}\rangle$;

AutoMed is particularly suited to the task of schema translation for the following reasons:

1. The HDM is a simple and flexible CDM which, as we have shown, can be used to model a wide range of DDLs.

2. The constraint language the HDM uses can be used to differentiate between different variants of a high level construct and, as we will see later in this chapter, provides important clues when it comes to matching HDM schema objects to a target DDL configuration.

3. Our transformation language, BAV, allows us to transform individual schema objects. This means that complex high level constructs are split up into components that can be processed more easily. This allows us to use CTs to break complex restructuring tasks into smaller tasks that can be applied to a number of different translation problems, as well as being composed to perform an entire translation.

A distinguishing feature of our implementation of ModelGen is that the choice of transformations does not rely on knowledge of the source DDL. In addition this work has the advantage that the inverse mapping, *i.e.* from target to source, is directly available because BAV pathways are bidirectional.

So far our experimental work has shown that given the current set of CTs and the preconditions associated with them, our heuristic approach is sufficient to do automatic translation between SQL, XML Schema, RDFS, ER and CSV files, If we wish to add other DDLs to our system, however, it may be necessary to add extra CTs and/or expand the preconditions we use. This is, in fact, likely to be the case if we wish to process languages like OWL [Mik04] that allow the inference of new facts. This is discussed in more detail in the future work section in the conclusion of this thesis.

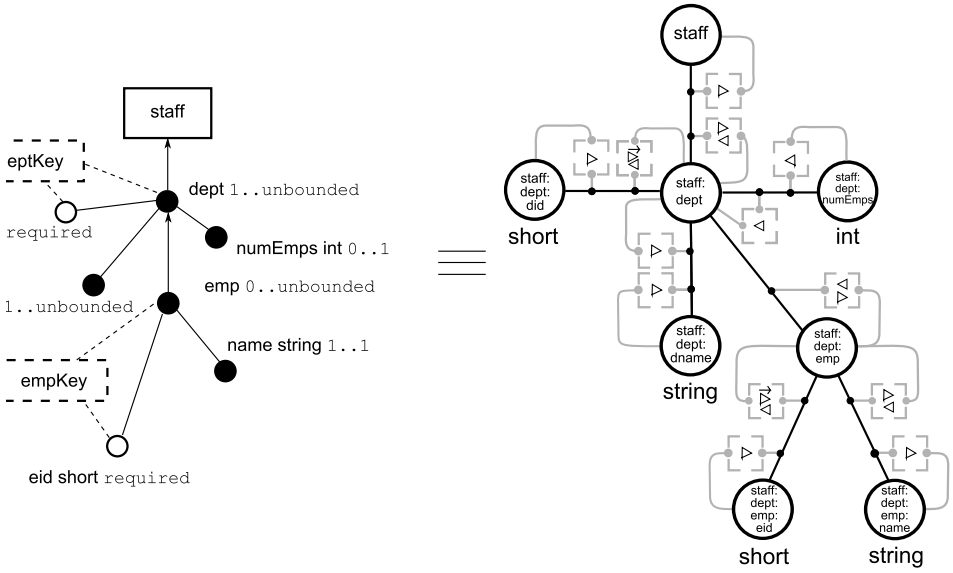


Figure 5.3: An XML schema, S_{xml} and its HDM equivalent, $S_{hdm-xml}$

5.1 Translating from a High Level DDL to the HDM

In this section we introduce the example we use in the rest of this chapter. Figure 5.3 shows an XML schema and HDM schema created by applying the production rules we defined for XML Schema in Section 3.2.3. We annotate the typed HDM nodes with their type name as calculated using the CTH described in the previous chapter. This completes Step 1 in Figure 5.1.

5.2 Match

This section describes how we identify whether the objects in a given HDM schema match those that represent constructs in the target DDL. As we saw in Chapter 3, a given high level construct may have a number of variants. For example an XML attribute may have the `use` attribute set to `required` or not. These variants all generate the same combination of extensional HDM objects but a different set of constraint objects.

For each DDL in AutoMed we store a table similar to that in Table 5.1 that associates a high level construct variant with its scheme and the constraints it generates when translated into the HDM.

Construct	Variant	Construct Scheme	HDM Constraints
table	`table`	table:$\langle\langle T \rangle\rangle$	
column	`column-notnull`	column:$\langle\langle T, C, D, \texttt{notnull} \rangle\rangle$	$\langle\langle T \rangle\rangle \triangleright \langle\langle \_, T, T{:}C \rangle\rangle$, $\langle\langle T \rangle\rangle \triangleleft \langle\langle \_, T, T{:}C \rangle\rangle$, $\langle\langle T{:}C \rangle\rangle \triangleright \langle\langle \_, T, T{:}C \rangle\rangle$
	`column-null`	column:$\langle\langle T, C, D, \texttt{null} \rangle\rangle$	$\langle\langle T \rangle\rangle \triangleright \langle\langle \_, T, T{:}C \rangle\rangle$, $\langle\langle T{:}C \rangle\rangle \triangleleft \langle\langle \_, T, T{:}C \rangle\rangle$
primary_key	`primary_key`	primary_key:$\langle\langle K, T, C \rangle\rangle$	$\langle\langle T \rangle\rangle \stackrel{\text{id}}{\rightarrow} \langle\langle \_, T, T{:}C \rangle\rangle$
foreign_key	`foreign_key`	foreign_key:$\langle\langle K, T, C, T', C' \rangle\rangle$	$\langle\langle T{:}C \rangle\rangle \subseteq \langle\langle T'{:}C' \rangle\rangle$

Table 5.1: SQL constructs, variants and the associated constraints

The Match algorithm, shown Algorithm 5.2, takes two parameters: the HDM schema and a list of the constructs in the target DDL. It uses the constraints associated with the objects in the HDM schema to identify which, if any of a target schema's constructs match the objects. It first attempts to match the *structure* of the graph surrounding the group of HDM schema objects with that of the target construct. For example, an SQL column is represented by an edge attached to a leaf node[1]. So if one of the nodes connected to the edge is not a leaf node the edge cannot represent an SQL column. If the structure matches then the constraints attached to the object are compared to those generated by high level constructs in the target DDL. If we are able to find a match we see if we can infer matches for any of the objects referenced by the current object.

Differences in the semantics of representations in different DDLs mean that it may not be possible to find a target construct that is *equivalent* to the original in a formal way. For example, if we translate a SQL `notnull` column into an ER model that does not support `notnull` attributes. In this case we attempt to match to the target construct that provides the 'tightest fit' to the constraints attached to the schema object. If we have to do one of these 'tightest fit' matches we inform the user that the information capacity of the target schema is greater than that of the source.

Variable	Description
so	An HDM schema object
tc	The construct in the target DDL that *so* has been matched to, `null` if no matching construct has been found
$tcVariant$	The variant of the construct *so* has been matched to
$refObjects$	A hash map of HDM schema objects referenced by *so* and keyed on the construct they will be translated into in the target DDL

Table 5.2: The MatchObject data structure

We now describe the data structures and functions used in the algorithm in detail. Table 5.2 shows the data structure MatchObject that we use in the algorithm. The

[1]Here a leaf node takes its normal meaning as a node attached to only one edge

Algorithm 5.2: Match(Schema S,List TC)

```
1  MO := new List();
2  HIC := false;  foreach so in S do
3      mo := new MatchObject(so);
4      MO.add(mo);
5  foreach mo in MO do
6      foreach tc in TC do
7          if matchStructure(mo.so,tc) then
8              cons := getConstraints(S, mo.so);
9              targetConstraints := getTargetConstructConstraints(tc);
10             tcVariant := matchConstraints(cons, targetConstraints);
11             if tcVariant != null then
12                 mo.tc := tc;
13                 mo.tcVariant := tcVariant;
14                 mo.refObjects := labelReferencedSchemaObjects(MO, S, mo);
15                 continue;
16 foreach mo in MO do
17     if mo.tc = null then
18         foreach tc in TC do
19             if matchStructure(so,tc) then
20                 cons := getConstraints(S, so);
21                 targetConstraints := getTargetConstructConstraints(tc);
22                 tcVariant := matchTightestFit(cons, targetConstraints);
23                 if tcVariant != null then
24                     HIC := true;
25                     mo.tc := tc;
26                     mo.tcVariant := tcVariant;
27                     mo.refObjects :=
                       labelReferencedSchemaObjects(MO, S, mo);
28                     continue;
29 return ⟨MO, HIC⟩;
```

constructor for a MatchObject takes a schema object as a parameter and sets tc to null, $tcVariant$ to null and creates a new hash map called $refObjects$.

1. matchStructure returns true if the structure of the extensional objects attached to so match those of tc, false otherwise.

2. getConstraints(S, so) returns the set of constraint schema objects in S that are dependent on so. For example, the constraints dependent on
   edge:⟨⟨_, staff, staff:dept⟩⟩ in Figure 5.3 are
   node:⟨⟨staff⟩⟩ ▷ edge:⟨⟨_, staff, staff:dept⟩⟩,
   node:⟨⟨staff:dept⟩⟩ ▷ edge:⟨⟨_, staff, staff:dept⟩⟩ and
   node:⟨⟨staff:dept⟩⟩ ◁ edge:⟨⟨_, staff, staff:dept⟩⟩.

3. getTargetConstructConstraints(tc) returns a hash map of the set of constraints associated with the high level construct tc keyed on the variants of tc. For example, if tc were a SQL column, the function would return a hash map made up of two sets of constraints, the first with key `column-notnull` would contain the three constraints in the second, third and fourth rows of Table 5.1 and the second with the two constraints in the fifth and sixth rows of the table.

4. matchConstraints($cons, targetConstraints$) returns the variant of the high level construct whose set of constraints match $cons$. It returns `null` if no such match is found. It may be the case that more than one construct in the target DDL shares the same structure and set of constraints. This is the case with the attribute and simpleElement constructs in XML Schema. In cases like this the HDM object will be matched to whichever of the constructs is listed first in TC.

5. matchTightestFit($cons, targetConstraints$). If match is unable to find an exact match we use the matchTightestFit function to see if $cons$ forms a superset of any of the elements in $targetConstraints$. If it does the function returns the construct variant that provides the 'tightest fit' with $cons$. In this case we set the HIC flag to true so we will be able to inform the user that the information capacity of the target schema will be greater than the source.

   Note that we cannot do this 'tightest fit' matching if the structure is different. These objects will need to be transformed before they match the target DDL as discussed in the next section.

6. labelReferencedSchemaObjects(MO, S, mo). Having successfully matched so to tc we may be able to infer additional matches for objects referenced by so

if the scheme of *tc* includes references to other constructs. For example, the scheme of the SQL column construct in Table 5.1 includes a reference to a table construct. By analysing the production rule for *tc* we can work out which of the objects attached to *so* should be matched to the referenced high level construct. In the case of the column construct analysis of its production rules tells us that the node that is the first element of the edge created when a column object is translated into HDM, represents the table construct.

The function performs the checks for these extra matches. It finds the MatchObjects in MO that hold the appropriate HDM schema objects referenced by *mo.so* and sets their *tc* and $tcVariant$ fields to the correct values. It returns an array list containing these referenced HDM schema objects ordered by the position of their corresponding high level construct in the scheme of *mo.tc*. For example, if *mo.tc* is an ER relationship, the HDM node corresponding to the first entity in the scheme will be first in the array list, the second will be second and so on. The array list is stored in the $refObjects$ field of the current match object and is used when we translate the HDM schema into the target DDL.

labelReferencedSchemaObjects also updates the MatchObjects of any HDM objects that are part of *mo.so*. In the example of the SQL column above all the constraint objects on *mo.so* would be matched to *mo.tc* and $mo.tcVariant$ as would the node created when a column object is translated into HDM, that makes up the second element of the edge created.

We now describe a run through of Algorithm 5.2. We start with $S = S_{hdm-xml}$, TL = `SQL` and *so* = edge:⟨⟨_, staff:dept, staff:dept:dname⟩⟩. We will assume *so* is stored in the match object, *mo*. The algorithm identifies the edge as part of a SQL column. Its constraints match those of a `null` column. matchStructure returns `true` since the element in the edge scheme not attached to the unique constraint is a leaf node, as required for a SQL column. *mo.tc* is set to `column-null`. In addition labelReferencedSchemaObjects identifies the fact that the scheme of a column includes a reference to a table construct. It identifies node:⟨⟨staff:dept⟩⟩ as the node that represents this table construct and sets the *tc* field of the MatchObject in MO that holds it to table. $tcVariant$ is also set to `table` and the HDM object is added to the array list to be returned. It also identifies node:⟨⟨staff:dept:dname⟩⟩ as the node in the column definition and sets the *tc* field to column and $tcVariant$ to `column-null` on its match object. It does the same to all the constraints that are part of the `column-null` definition.

5.3 Transform

In most cases the HDM schema created in Step 1 of Figure 5.1 will not translate directly into constructs from the target DDL. In this section we describe an algorithm that transforms an HDM schema that contains objects that were not identified as matching a construct in the target DDL by the Match algorithm, into one where all the objects match a construct in the target DDL.

It is based on a search of the possible schemas that can be created by applying CTs to the unidentified schema objects. The process was shown in Figure 5.2 at the beginning of this chapter. We assume here that applying a CT will return the transformation pathway that executes it.

The CTs we use in this chapter, along with a brief explanation of what they do are shown in Table 5.3. Details of the CTs other than expand_multi_value can be found in [BM05]. New CTs can be created and added to the system. This may be necessary if we add a new DDL to AutoMed that does not fall into any of the classes we have defined.

Transformation	Description
inclusion_merge(node:$\langle\langle B\rangle\rangle$, edge:$\langle\langle E, A, C\rangle\rangle$)	Merges node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ if node:$\langle\langle A\rangle\rangle$ is a subset of node:$\langle\langle B\rangle\rangle$ and there is a mandatory constraint from node:$\langle\langle A\rangle\rangle$ to edge:$\langle\langle E, A, C\rangle\rangle$
inclusion_expand(node:$\langle\langle B\rangle\rangle$, edge:$\langle\langle E, B, C\rangle\rangle$)	Creates a new node node:$\langle\langle A\rangle\rangle$ that is a subset of node:$\langle\langle B\rangle\rangle$, moves edge:$\langle\langle E, B, C\rangle\rangle$ from node:$\langle\langle B\rangle\rangle$ to node:$\langle\langle A\rangle\rangle$ and adds a mandatory constraint from node:$\langle\langle A\rangle\rangle$ to the new edge
id_node_merge(edge:$\langle\langle E, A, B\rangle\rangle$)	Merges node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ if they are identical
id_node_expand(node:$\langle\langle A\rangle\rangle$)	Creates a new node identical to node:$\langle\langle A\rangle\rangle$ and an edge linking it to node:$\langle\langle A\rangle\rangle$
um_redirection(edge:$\langle\langle E_1, A, C\rangle\rangle$,edge:$\langle\langle E_2, A, B\rangle\rangle$)	Moves edge:$\langle\langle E_1, A, C\rangle\rangle$ from node:$\langle\langle A\rangle\rangle$ to node:$\langle\langle B\rangle\rangle$ if node:$\langle\langle A\rangle\rangle$ and node:$\langle\langle B\rangle\rangle$ have unique and mandatory constraints on edge:$\langle\langle E_2, A, B\rangle\rangle$
id_edge_merge(edge:$\langle\langle E_1, A, B_1\rangle\rangle$, edge:$\langle\langle E_2, A, B_2\rangle\rangle$)	Replaces node:$\langle\langle A\rangle\rangle$, edge:$\langle\langle E_1, A, B_1\rangle\rangle$ and edge:$\langle\langle E_2, A, B_2\rangle\rangle$ with edge:$\langle\langle A, B_1, B_2\rangle\rangle$ if for each instance of node:$\langle\langle A\rangle\rangle$ there is one instance of the join of edge:$\langle\langle E_1, A, B_1\rangle\rangle$ and edge:$\langle\langle E_2, A, B_2\rangle\rangle$
expand_multi_value(node:$\langle\langle A\rangle\rangle$, edge:$\langle\langle E, A, B\rangle\rangle$)	Replaces edge:$\langle\langle E, A, B\rangle\rangle$ with a collection of nodes and edges link to node:$\langle\langle A\rangle\rangle$ with an inclusion constraint as long as there is a mandatory constraint from node:$\langle\langle A\rangle\rangle$ to edge:$\langle\langle E, A, B\rangle\rangle$

Table 5.3: The general purpose CTs we use in ModelGen

To limit the number of CTs that need to be applied at each step of the algorithm and to thereby limit the size of the search space a CT must satisfy certain **preconditions** before being applied to an unmatched object. These preconditions rely on the

constraints and structure of the graph surrounding the unmatched object. The preconditions for the CTs in Table 5.3 are shown in Table 5.4. The DNC in the table means we Do Not Care (DNC) whether the precondition is met or not.

Transformation	edge	node	cons	leaf	reflexive	unique
inclusion_merge	N	DNC	DNC	DNC	DNC	DNC
inclusion_expand	Y	N	N	N	N	Y
id_node_merge	Y	N	N	DNC	Y	Y
id_node_expand	N	Y	N	Y	N	DNC
um_redirection	Y	N	N	N	N	DNC
id_edge_merge	Y	N	N	N	Y	DNC
expand_multi_value	Y	N	N	N	N	N

Table 5.4: The CTs and their preconditions

If we assume *so* is the unmatched schema object the preconditions we use are:

edge is *so* an edge construct?
node is *so* a node construct?
cons is *so* a constraint construct?
leaf is *so* a leaf node or connected to a leaf node?
reflexive is there a reflexive constraint attached to *so*?
unique is there a unique constraint attached to *so*?

As an example, consider the expand_multi_value CT that was introduced in Chapter 3. It can only be applied to an edge, the edge must not be attached to a leaf node and there must not be a reflexive or unique constraint on the edge. Some of the preconditions are determined by the parameters that the CT takes. For example, id_node_expand takes a single node as a parameter and so can only be applied to an unmatched node object. Other preconditions are determined by the CT itself, for example id_node_merge can only be applied to an edge that has a reflexive constraint attached to it and id_edge_merge can only be applied to edges that have a join constraint between them.

We have found that these preconditions and CTs are sufficient to translate the schemas we have used in our experiments. However, the list can easily be extended if necessary. Using a greater number of preconditions has an impact on performance as we discuss later.

Transform is shown in Algorithm 5.3. It makes use of two global variables: TC, the list of constructs in the target DDL and CT, the list of general purpose CTs in Table 5.3.

Algorithm 5.3: Transform(Schema S, BAVPathway $pathway$)

```
1  ⟨MO, HIC⟩ := Match(S, TC);
2  foreach mo in MO do
3      if mo.tc = null then
4          num_unmatched++;
5  if num_unmatched = 0 then
6      return ⟨pathway, MO, HIC⟩;
7  else
8      foreach mo in MO do
9          if mo.tc is null then
10             mt := matching_cts(S, mo.so);
11             while mt is not empty do
12                 Let ct := mt.getNextElement();
13                 p_{S,S'} := the result of applying ct to mo.so;
14                 result := Transform(S', pathway + +p_{S,S'});
15                 if result ≠ Fail then
16                     return result;
17     return Fail;
```

The algorithm works as follows: First Match is run to try and match the schema objects in S to constructs in $TDDL$. If Match is able to match all the objects, *num_unmatched* will be zero and the schema has been successfully matched to the target DDL. The algorithm returns a tuple with three elements. Firstly, *pathway*, a pathway from the HDM representation of the source schema to the current schema where all the HDM objects have been matched with the HDM equivalents of constructs in the target DDL. Secondly, MO, a list of successfully matched MatchObjects. Finally, HIC (Higher Information Capacity), a flag that is true if the transformed schema has a higher information capacity than the source, and false if the source and target have the same information capacity.

If there are unmatched objects in MO, the algorithm loops through MO looking for match objects whose target constraint is `null`. When such a match object is found the matching_cts function is called to create a list of CTs whose preconditions match the structure of the subschema made up of *mo.so* and any schema objects it references. For a CT to match the preconditions the subgraph must match each of the 'Y's in the precondition table and must not match the 'N's. The CTs with the fewest DNCs are sorted to the top of the list. For example, if the preconditions of two CTs matched but one had no DNCs and the other had one DNC, the one with

no DNCs would be at the top of the list.

The first CT in the list is applied to *mo.so* to create a pathway between the current schema and S', the final schema in the pathway. The algorithm is then called again with the transformed schema and a new pathway created by concatenating the pathway created on line 14 to the current pathway. If no suitable transformation can be found for *any* of the unmatched schema objects then the translation has failed. This happens if none of the CTs precondition's match the structure of the schema around the unmatched schema objects.

As we discussed in the previous section, the different semantics of DDLs may mean we are unable to create an *equivalent* target schema. We have seen how we handle this situation where the structure of a target construct matches the HDM schema object, *so*, but its constraints only form a subset of those attached to *so*. We may also have a situation where the structure does not match the target DDL and we need to do a transformation.

For example, if we translate an ER `1:1,1:N` relationship into SQL we need to transform the edge that represents this relationship as no SQL construct matches its structure. We will lose some of the constraints because SQL does not support the range of cardinality constraints that the ER model does. These extra constraints sometimes mean that a CT that meets all the preconditions and would perform the required restructuring is prevented from executing. We overcome this by allowing the removal of these constraints before executing the CT. This again creates a target schema with greater information capacity than the source. As part of the execution of the CT in such a case, we inform the user the information capacity has increased as we did in the Match algorithm.

As we have stated previously this is a heuristic process and is not guaranteed to complete successfully. We see the termination conditions for the algorithm illustrated in Figure 5.2. Either all the objects are matched to a target DDL construct in which case a solution has been found or we end up in a state where there are unidentified objects but no CTs meet the preconditions on the schema. In this case the translation has failed. There are two possible reasons for this: firstly it may simply not be possible to translate a given schema expressed in the source DDL into one expressed in the target DDL. So far this has not been the case with the DDLs we have used in our experiments but our method does not guarantee that this will always be the case. The second reason that the translation may fail is that the algorithm has not chosen suitable CTs. If this is the case we need to adjust the

preconditions on the CTs. Indeed, this was necessary throughout our experimental work.

The output of this phase of the translation is a BAV pathway from the HDM representation of the source schema, S_{hdm-s} in Figure 5.1, to an HDM representation of the schema in the target DDL, S_{hdm-t}, as well as a list of match objects that allow us to translate this schema into the target DDL.

5.3.1 Complexity

We analyse the complexity of executing Transform by counting the number of CTs executed. In the worst case, no objects in the source graph are identified as matching a target construct by Match, and we need to iterate num_o times, where num_o is the number of objects in the HDM schema. Let num_{ct} be the number of CTs we have to choose from, and let the search graph we showed in Figure 5.2 have a depth x. In the worst case we will need to execute $(num_o \times num_t)^x$ CTs.

In the worst case x is infinite so it is clearly vital to try to limit it. The more CTs we have the more likely we are to reach a solution quickly because we have more ways of restructuring the HDM schema. There is a trade off here though. Each extra CT will increase the number of choices we have at each stage of the transformation process. To overcome this we use the preconditions on the CTs to limit the number that can be chosen in each recursion of the algorithm. We have used the preconditions to 'direct' the search and to limit the chances of costly backtracking by carefully tuning them based on our experiments.

We cannot place a limit on the depth of the search graph since the number of unidentified objects is not monotonically decreasing. It is possible that applying a CT may increase the number of unidentified objects. For example in Figure 5.2, initially there is only one unidentified object, but after applying $CT_1(so_1)$ there are two unidentified objects. In other words we cannot guarantee that the number of unidentified objects will always reach zero. To prevent the system looping forever in a situation like this, we exit with a failure message if the number of unidentified objects increases for five recursions of Transform in a row. To simplify the presentation of the Transform algorithm, this detail is omitted.

DDL	Class	Class specific CT
XML Schema	Hierarchical	create_root_node
SQL	Natural keys	check_keys
ER	Natural keys	check_keys
RDFS	Semantic Web	create_semweb_nodes

Table 5.5: DDL Classes

5.4 Translating from the HDM to the Target DDL

If Transform completes successfully all the HDM schema objects will be matched to a specific variant of a high level construct in the target DDL. The different semantics and structure of high level DDLs means that we still not be able to create a valid schema in the target DDL. For example, if the target DDL is XML Schema we may need to create a root node. To solve these types of problems we may need to execute specific CTs chosen based on the *class* of the target DDL which forms part of the information contained in a DDL's AutoMed wrapper. The classes for the DDLs used in this thesis are shown in Table 5.5 along with the class specific DDLs we have found necessary.

The check_keys CT, shown in Algorithm 5.4, is run on schemas that use natural keys, such as SQL. It makes sure each node matched to a nodal construct in the target DDL, has an associated object(s) that can be used as a key, *i.e.* is connected to a reflexive constraint. If this is not the case, we add HDM constraints that match a compound key construct to the node and any of its edges that has a mandatory constraint between the edge the node. The extent of the node is updated using the node_reidentify(node:⟨⟨A⟩⟩,*map*) CT presented in [BM05]. This creates a new node:⟨⟨A⟩⟩ with the extent defined by *map*, a set of tuples whose first value comes from node:⟨⟨A⟩⟩, and whose second value is the new value which we wish to assign to node:⟨⟨A⟩⟩.

A hierarchical DDL needs to have a single node that will act as the root. We also need to create an edge from this node to any complex elements that are not children of any other complex elements. If there is more than one such element, *i.e.* the HDM schema is a forest, which could happen if we translated two SQL tables that were not connected by a key, we need to create an edge to all of them. The edge will be matched to a complex element with `minOccurs = 0` and `maxOccurs = unbounded`. We create these objects using the CT create_root_node, shown in Algorithm 5.5, that is only applied to hierarchical DDLs. DDLs in the semantic web class like RDFS and the various flavours of OWL have a similar requirement that extra nodes be

Algorithm 5.4: create_keys(Schema S, ListMO)

```
1  foreach mo in MO do
2      if mo.tc is a nodal construct then
3          node:⟨⟨A⟩⟩ := mo.so;
4          if node:⟨⟨A⟩⟩ is not attached to a reflexive constraint then
5              Let mandatoryEdges := new List();
6              foreach edge:⟨⟨Ei, A, Bi⟩⟩ do
7                  if node:⟨⟨A⟩⟩ ▷ edge:⟨⟨Ei, A, Bi⟩⟩ ∈ Cons then
8                      mandatoryEdges.add(edge:⟨⟨Ei, A, Bi⟩⟩);
9              Let edge:⟨⟨E1, A, B1⟩⟩ ... edge:⟨⟨En, A, Bn⟩⟩ be the edges in mandatoryEdges;
10             S' := S.add(unique:(node:⟨⟨A⟩⟩, edge:⟨⟨E1, A, B1⟩⟩ ⋈ ... ⋈ edge:⟨⟨En, A, Bn⟩⟩));
11             S' :=
               S'.add(mandatory:(node:⟨⟨A⟩⟩, edge:⟨⟨E1, A, B1⟩⟩ ⋈ ... ⋈ edge:⟨⟨En, A, Bn⟩⟩));
12             S := S'.add(reflexive:(node:⟨⟨A⟩⟩, edge:⟨⟨E1, A, B1⟩⟩ ⋈ ... ⋈ edge:⟨⟨En, A, Bn⟩⟩));
13             S := S'.delete(unique:(node:⟨⟨A⟩⟩, edge:⟨⟨Ei, A, Bi⟩⟩));
14             S := S'.delete(mandatory:(node:⟨⟨A⟩⟩, edge:⟨⟨Ei, A, Bi⟩⟩));
15             map = distinct
               [{y, x1, ..., xn} | {y, x1} ← edge:⟨⟨E1, A, B1⟩⟩; ...; {y, xn} ← edge:⟨⟨En, A, Bn⟩⟩];
16             pS',S'' := node_reidentify(node:⟨⟨A⟩⟩,map);
17 return pS,S'';
```

added to the HDM schema before it can be translated. These are node:⟨⟨Literal⟩⟩ and node:⟨⟨Resource⟩⟩ and are created by the create_semweb_nodes CT.

Algorithm 5.5: create_root_node(Schema S, ListMO)

```
1  S' := S.add(node:⟨⟨R⟩⟩,[&0]);
2  foreach mo in MO do
3      if mo.tc = complexElement then
4          node:⟨⟨A⟩⟩ := mo.so;
5          if node:⟨⟨A⟩⟩ is the root node of a tree in S then
6              S' := S'add(edge:⟨⟨R, A⟩⟩,distinct [{r, x} | {r} ← node:⟨⟨R⟩⟩; {x} ← node:⟨⟨A⟩⟩]);
7              S := S'add(mandatory:(node:⟨⟨R⟩⟩, edge:⟨⟨_, R, A⟩⟩));
8              S := S'add(unique:(node:⟨⟨R⟩⟩, edge:⟨⟨_, R, A⟩⟩))
9  return pS,S';
```

After we have applied any necessary class specific CTs we are ready to use the information stored in the list of MatchObjects, MO, returned by Transform to create the target schema. We loop through all the objects in MO first creating the nodal objects, then the link-nodal constructs, then the link constructs and finally the constraints. For each object in MO we create a BAV transformation that adds the target DDL construct identified in *mo.tc*. The scheme of the newly created object is determined by using the value stored in $mo.tcVariant$ and the table that stores the scheme and HDM constraints of each variant of a high level construct such as that in Table 5.1.

Construct	Variant	Construct Scheme	HDM Constraints
entity	`entity`	entity:$\langle\langle E\rangle\rangle$	
relationship	`rel-0:1-0:N`	relationship:$\langle\langle R, E_1, 0{:}1, E_2, 0{:}N\rangle\rangle$	node:$\langle\langle E_1\rangle\rangle \lhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$
relationship	`rel-1:1-0:N`	relationship:$\langle\langle R, E_1, 1{:}1, E_2, 0{:}N\rangle\rangle$	node:$\langle\langle E_1\rangle\rangle \rhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$ node:$\langle\langle E_1\rangle\rangle \lhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$
relationship	`rel-1:N-1:N`	relationship:$\langle\langle R, E_1, 1{:}N, E_2, 1{:}N\rangle\rangle$	node:$\langle\langle E_1\rangle\rangle \rhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$ node:$\langle\langle E_2\rangle\rangle \rhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$
relationship	`rel-1:1-1:N`	relationship:$\langle\langle R, E_1, 1{:}1, E_2, 1{:}N\rangle\rangle$	node:$\langle\langle E_1\rangle\rangle \rhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$ node:$\langle\langle E_1\rangle\rangle \lhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$ node:$\langle\langle E_2\rangle\rangle \rhd$ edge:$\langle\langle R, E_1, E_2\rangle\rangle$

Table 5.6: Selected variants of the ER relationship construct, their schemes and the associated HDM constraints

For example, assume we have the match objects mo_1, mo_2 and mo_3 shown below and we use Table 5.6 to associate a construct variant to each scheme.

$mo_1.so$ = node:$\langle\langle$R$\rangle\rangle$, $mo_1.tc$ = entity, $mo_1.tcVariant$ = `entity`, $mo_1.refObjects = \{\}$,
$mo_2.so$ = node:$\langle\langle$S$\rangle\rangle$, $mo_2.tc$ = entity, $mo_2.tcVariant$ = `entity`, $mo_2.refObjects = \{\}$,
$mo_3.so$ = edge:$\langle\langle relname, R, S\rangle\rangle$, $mo_3.tc$ = relationship, $mo_3.tcVariant$ = `rel-1:1-1:N`,
$mo_3.refObjects$ = {node:$\langle\langle$R$\rangle\rangle$,node:$\langle\langle$S$\rangle\rangle$}

The target constructs for mo_1 and mo_2 are entities. The production rule for an entity tells us that the extent of the entity is that of the node, and the name of the entity and the node are the same. There are no objects referenced in this scheme so $mo_1.refObjects$ and $mo_2.refObjects$ are empty. The BAV transformations to translate mo_1 and mo_2 into ER are thus:

add(entity:$\langle\langle$R$\rangle\rangle$, node:$\langle\langle$R$\rangle\rangle$)
add(entity:$\langle\langle$S$\rangle\rangle$, node:$\langle\langle$S$\rangle\rangle$)

As part of this process we add the newly created entities to a map, $addedObjects$, that maps HDM objects to ER entities.

$addedObjects$:= $addedObjects \cup$ {$\langle$node:$\langle\langle$R$\rangle\rangle$,entity:$\langle\langle$R$\rangle\rangle\rangle$, $\langle$node:$\langle\langle$S$\rangle\rangle$,entity:$\langle\langle$S$\rangle\rangle\rangle$}

The target construct for mo_3 is relationship. The production rule for a relationship tells us the extent of the relationship object is that of the edge and the name of the relationship and the edge are the same. The variant is `rel-1:1-1:N`. From Table 5.6 the scheme for the object we create is relationship:$\langle\langle R, E_1, 1{:}1, E_2, 1{:}N\rangle\rangle$. The referenced entities that form the second and fourth elements of the scheme are found by looking at $mo_3.refObjects$ and then finding the equivalent high level object in $addedObjects$. The E_1 in the scheme is thus replaced with entity:$\langle\langle$R$\rangle\rangle$ and E_2 with entity:$\langle\langle$S$\rangle\rangle$. The resulting transformation is:

add(relationship:$\langle\langle$relname, R, 1:1, S, 1:N$\rangle\rangle$, edge:$\langle\langle$rel, R, S$\rangle\rangle$)

The output of this phase is the BAV pathway that maps all the HDM objects to their high level DDL equivalents.

5.5 Adding a New DDL

If a new DDL that includes constructs that are not found in the existing DDLs in our system, is added to the AutoMed MMS, it may be necessary to add new CTs and/or preconditions to translate to and from this new DDL. The amount of work required depends on how closely related the new DDL is to an existing DDL in the system.

The set of CTs which we used initially were those described in [BM05]. These were sufficient to translate between the first two DDLs we added to our prototype MMS, namely SQL and the ER model described in Section 3.2.2. Adding XML Schema to the system required us to write a new HDM to high level CT, create_root_node. We also needed to add the expand_multi_value CT described in Algorithm 3.15 to allow us translate multi-valued XML Schema elements into constructs supported by SQL and our ER model. Finally, we needed to add the unique precondition to make sure we did not apply the expand_multi_value CT to an element that was not multi-valued.

On the other hand, subsequently adding a variant of the ER model that supports multi-valued attributes, did not require any changes to be made to our MMS because both the extensional and constraint constructs were sufficiently similar to those found in the existing ER model and XML Schema (which supports multi-valued elements). It also falls into the same DDL class as the existing ER model, *i.e.* Natural Keys, so no new HDM to high level CTs were required.

5.6 Correctness

AutoMed ModelGen returns a single BAV transformation pathway. It was shown in [MP02] that, when considering a pathway, we need only consider one transformation at a time since each pathway T can be written as $t; T'$ where t is the primitive transformation that transforms T' to T. We apply this argument to the three phases of the translation described in this chapter. We can show the translation as a whole

is correct by showing each of the three phases is correct because each phase is simply a part of the pathway.

Firstly we assume that the production rules for translating high level DDL constructs in the HDM are correct and we can therefore assume the translations based on them are correct.

In step two of the translation, as shown in Figure 5.1, the HDM schema is transformed by means of a number of CTs that are composed together. We will take as axiomatically true that all the general purpose CTs we use produce equivalent schemas. By the argument above, applying them in a sequence must also result in an equivalent schema. If we need to apply a class specific CT we may *increase* the information capacity of the target schema as we do when we apply create_root_node but never decrease it.

The translation from the transformed HDM schema to a schema in the target DDL is based on the inverse of the production rules. If the Match algorithm has completed successfully each HDM object is matched to some construct in the target DDL. This matching is based on the production rules for the target DDL. As in the case of the high level to HDM step, we assume these production rules produce a high level schema with equivalent information capacity to the HDM schema it is based on.

These three steps are simply parts of a transformation pathway and so, as each one produces a schema that has at worst higher information capacity than the source schema we know that the target schema must also have an information capacity that is equal to or higher than that of the source schema.

5.7 Example translation from XML to SQL

In this section, we describe how we use AutoMed ModelGen to translate S_{xml} in Figure 5.3, into an SQL schema. We have already described, in Section 5.1, how S_{xml} is translated into $S_{hdm-xml}$. If we look at this schema we see that it cannot be translated directly into SQL. Firstly, it is hierarchical and secondly a department can have more than one name and so node:⟨⟨staff:dept:dname⟩⟩ cannot be translated directly into a SQL column. Some transformations are required.

In the first iteration of the Transform algorithm, Match returns
edge:⟨⟨_, staff:dept, staff:dept:emp⟩⟩ and edge:⟨⟨_, staff:dept, staff:dept:dname⟩⟩ with the

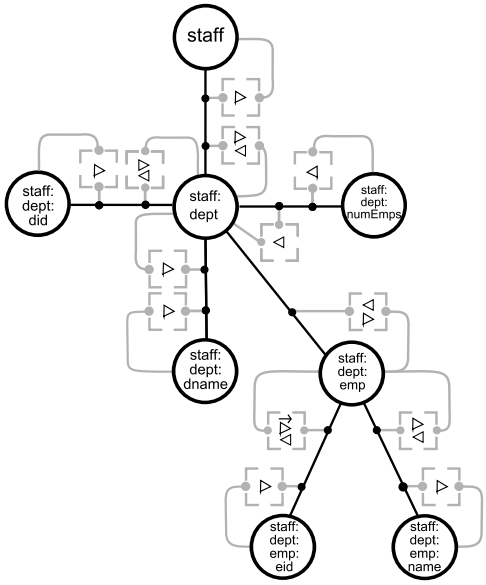


Figure 5.4: $S_{hdm-xml}$

matched label set to false. If we consider edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle$ first, and compare the structure of the surrounding schema with the preconditions in Table 5.4, we see that two CTs match. inclusion_expand matches with no DNCs, whereas um_redirection has one DNC, so inclusion_expand is executed.

This allows us to use the CT below to create a node whose extent is those departments that do have employees and then split the rest of the schema off.

inclusion_expand($S_{hdm-xml}$,node:$\langle\langle$staff:dept:emp$\rangle\rangle$,edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle$)

We show the BAV transformation pathway for this CT as an example:

① add(node:$\langle\langle$staff:dept:emp:dept$\rangle\rangle$,
  $[\{d\} \mid \{d, e\} \leftarrow$ edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle]$)
② add(edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept:emp:dept}\rangle\rangle, [\{e, d\} \mid$
  $\{d, e\} \leftarrow$ edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle]$)
③ add(node:$\langle\langle$staff:dept:emp:dept$\rangle\rangle \subseteq$ node:$\langle\langle$staff:dept$\rangle\rangle$)
④ add(node:$\langle\langle$staff:dept:emp:dept$\rangle\rangle \rhd$ edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept:emp:dept}\rangle\rangle$)
⑤ add(node:$\langle\langle$staff:dept:emp$\rangle\rangle \rhd$ edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept:emp:dept}\rangle\rangle$)
⑥ add(node:$\langle\langle$staff:dept:emp$\rangle\rangle \lhd$ edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept:emp:dept}\rangle\rangle$)
⑦ delete(node:$\langle\langle$staff:dept:emp$\rangle\rangle \rhd$ edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept}\rangle\rangle$)
⑧ delete(node:$\langle\langle$staff:dept:emp$\rangle\rangle \lhd$ edge:$\langle\langle \_, \text{staff:dept:emp}, \text{staff:dept}\rangle\rangle$)
⑨ delete(edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle, [\{d, e\} \mid$
  $\{e, d\} \leftarrow$ edge:$\langle\langle \_, \text{staff:dept}, \text{staff:dept:emp}\rangle\rangle]$)

This shows one of the strengths of the BAV approach. We can evolve the schema by only changing some of the objects rather than having to create a completely

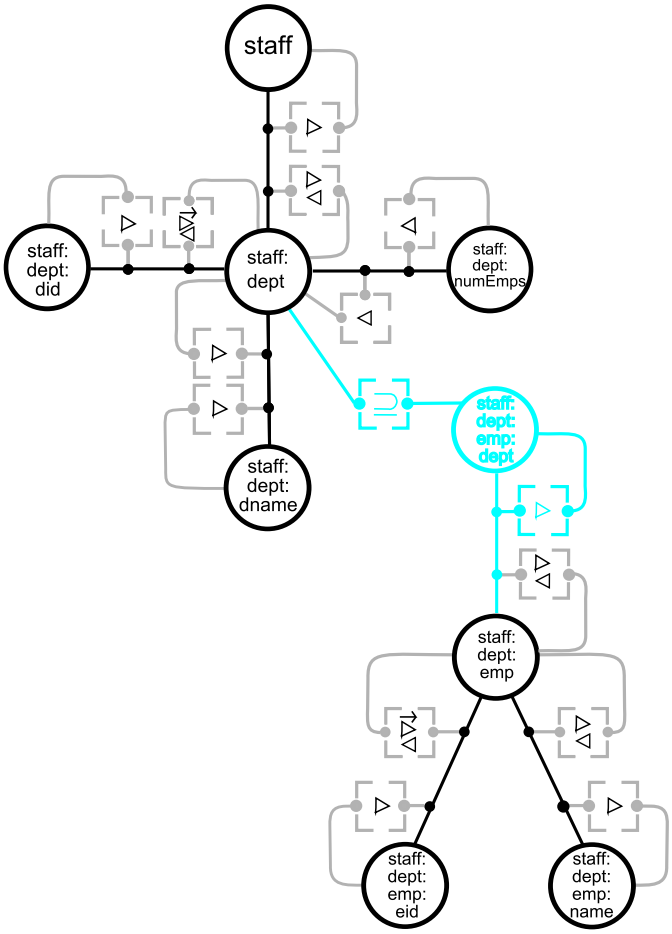


Figure 5.5: After applying inclusion_expand

new schema each time we execute a CT. The result of applying this CT is shown in Figure 5.5. The grey objects are those added or changed by the execution of the CT.

In the second iteration just edge:⟨⟨_, staff:dept, staff:dept:dname⟩⟩ will be returned by Match with its matched label set to false. The only CT whose preconditions are met by this edge is expand_multi_value. We thus execute the following CT:

expand_multi_value($S'_{hdm-xml}$, edge:⟨⟨_, staff:dept, staff:dept:dname⟩⟩,dname)

This creates the schema shown in Figure 5.6. The match algorithm has identified three nodes which can be translated into tables, node:⟨⟨staff:dept:emp⟩⟩, node:⟨⟨staff:dept⟩⟩ and node:⟨⟨dname⟩⟩. The nodes and edges linked to them have been identified as columns. There are also two inclusion constraints that have been identified as foreign keys.

In Figure 5.6 the inclusion constraints are attached to the node that has been iden-

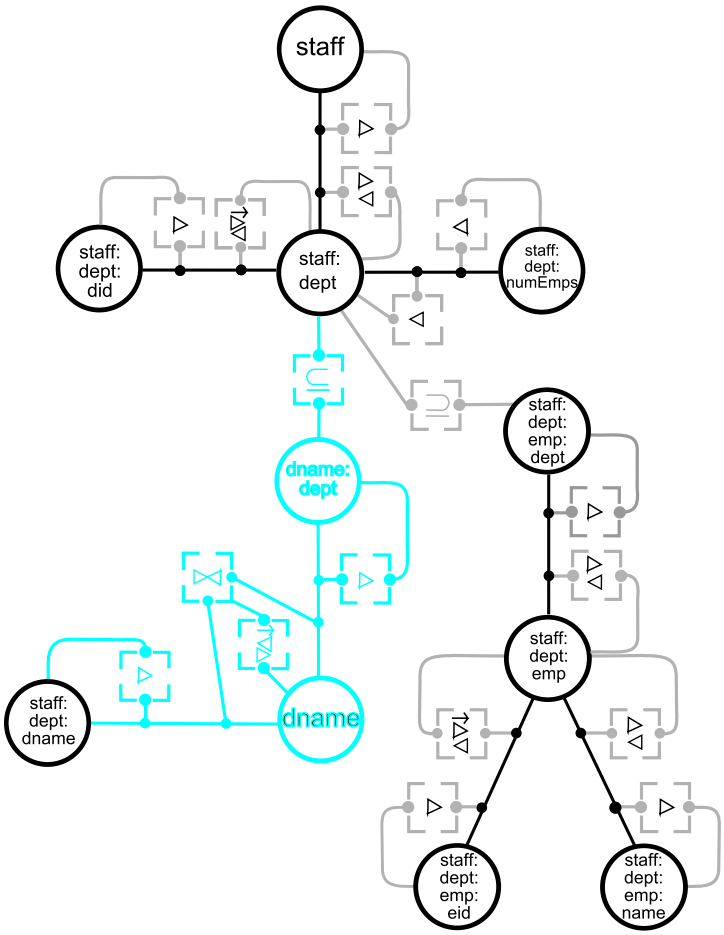


Figure 5.6: After applying expand_multi_value

tified as a table. As part of the translation to SQL these inclusion constraints are moved to the nodes that will be translated into the primary key columns as identified by the **mandatory**, **unique** and **reflexive** constraints from the table node to the edge linking it to the primary key node.

The final SQL schema is shown below. We again use the CTH to calculate the correct data types for the columns. node:⟨⟨staff⟩⟩ is untyped so we leave the type for the SQL column we create to represent it as unknown.

S_{sql} = {table:⟨⟨staff:dept⟩⟩, column:⟨⟨staff:dept, staff, unknown, notnull⟩⟩,
column:⟨⟨staff:dept, staff:dept:did, smallint, notnull⟩⟩,
column:⟨⟨staff:dept, staff:dept:numEmps, integer, null⟩⟩,
primary_key:⟨⟨staff:dept_key,table:⟨⟨staff:dept⟩⟩, column:⟨⟨staff:dept, staff:dept:did⟩⟩⟩⟩,
table:⟨⟨dname⟩⟩, column:⟨⟨dname, dname:dept, short, notnull⟩⟩,
column:⟨⟨dname, staff:dept:dname, string, notnull⟩⟩,
primary_key:⟨⟨dname_key,table:⟨⟨dname⟩⟩, column:⟨⟨dname, dname:dept⟩⟩,
column:⟨⟨dname, staff:dept:dname⟩⟩⟩⟩,
table:⟨⟨staff:dept:emp⟩⟩,column:⟨⟨staff:dept:emp, staff:dept:emp:dept, smallint, notnull⟩⟩,
column:⟨⟨staff:dept:emp, staff:dept:emp:eid, smallint, notnull⟩⟩,
column:⟨⟨staff:dept:emp, staff:dept:emp:name, varchar, notnull⟩⟩,
primary_key:⟨⟨staff:dept:emp_key,table:⟨⟨staff:dept:emp⟩⟩,
column:⟨⟨staff:dept:emp, staff:dept:emp:eid⟩⟩⟩⟩
foreign_key:⟨⟨staff:dept:emp_fk,table:⟨⟨staff:dept:emp⟩⟩,
column:⟨⟨staff:dept:emp, staff:dept:emp:dept⟩⟩,
table:⟨⟨staff:dept⟩⟩, column:⟨⟨staff:dept, staff:dept:did⟩⟩⟩⟩
foreign_key:⟨⟨dname_fk,table:⟨⟨dname⟩⟩,
column:⟨⟨dname, dname:dept⟩⟩,
table:⟨⟨staff:dept⟩⟩, column:⟨⟨staff:dept, staff:dept:did⟩⟩⟩⟩}

The names of the tables and columns could be improved with some post processing. It would also be possible for a human expert to remove
column:⟨⟨staff:dept, staff, unknown, notnull⟩⟩
since it does not add any useful data to the schema.

5.8 Experimental Results

Figure 5.7 shows the number of match operations versus the number of schema objects required to translate various subsets of an XML Schema representation of DBLP into SQL. The gradient is steepest when schema objects from the source DDL that have no direct equivalent in the target DDL are added to the source schema, in this case nested XML Schema complex types. Where the graph is flatter constructs

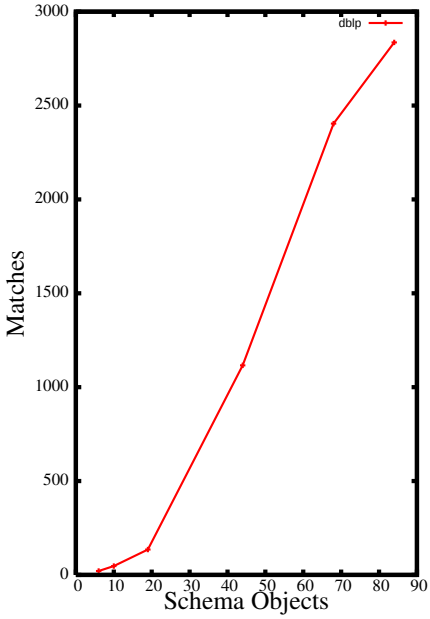


Figure 5.7: DBLP XML Schema to SQL

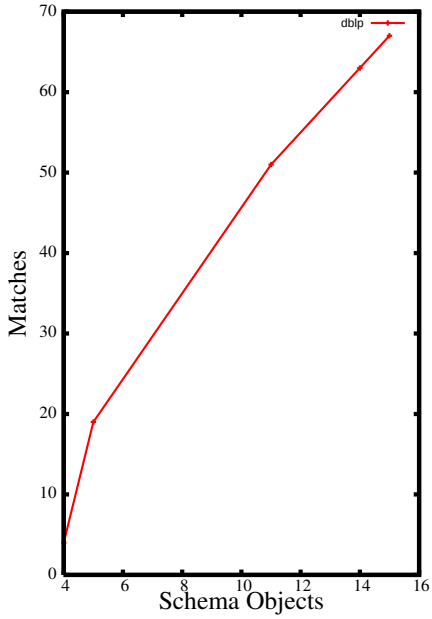


Figure 5.8: SQL database to ER

that could be matched directly with the target model, like XML Schema attributes, were added. Figure 5.8 shows matches vs schema objects for the translation of a SQL database to ER. Again the graph is steeper when tables with foreign keys are added.

5.9 Related Work

The implementation of ModelGen in Rondo uses a specific converter for each pair of DDLs the systems supports. In the most recent papers about Rondo [MRB03, MBHR05] this was limited to just SQL and XML. In addition these converters focus solely on the structural semantics of the input and output schema [MRB03, Mel04] and do not translate the data instances. Another schema only implementation of ModelGen is AutoGen [SKZ06].

The work most closely related to ours is that done by Atzeni et al in their MIDST system [ACB05, ACB06]. They also generate data-level translations by composition of elementary transformations to translate schemas and data between a number of different DDLs using a three stage approach. Their method differs from ours, however, in that the data is copied from source to CDM to target. This is inefficient if the data sets are large. Given the source and target DDLs, *predefined* algorithms choose the most appropriate transformation at each step in the process. In the final

step, the transformed data is copied into the target system. This differs from our approach in that our composite transformations are chosen at run-time.

A significant disadvantage of the MIDST implementation of ModelGen is that it does not return a set of mappings between the source and target schemas [BM07]. This means it cannot be used in a MM script where the mappings between source and target are needed as parameters to other operators.

Kensche *et al* use *GeRoMe* [KQLJ07], in their MMS *GeRoMeSuite*, to describe mappings directly from the source to the target schema using SO s-t tgds. There is a specific materialisation algorithm for each target DDL. The mappings are not generated automatically but are specified by an operator, using a GUI, and translated into the *GeRoMe* CDM by the system.

Numerous examples of systems for translating between specific models exist in the literature: XML and relational schemas [SSB+01, SSK+01] as well as ER and relational [PB94] and ER and XML schemas [SMD03]. More recent work on object relational to SQL translation has been done by Mork and Bernstein [MBM07].

Our method has the following advantageous characteristics:

1. We can do the translation between the DDLs currently supported by our prototype automatically.
2. Our simple CDM means high level structures can be dealt with in a uniform manner.
3. We use a common query language for all our data translation rather than DDL specific query languages.
4. We create a *bidirectional* mapping between the source and target schemas as part of our target schema creation process. This mapping allows us to translate the instances of the source schema into the target schema.

The fact that we do our translation via a CDM means that we need to do a greater number of transformations when compared to a system that translates the source schema directly into the target DDL. To quantify this difference let us assume that there are n schema objects in the source schema and m in the target. A direct bidirectional translation from source to target will require approximately $n + m$ transformations, one to create each of the target schema objects and one to create each of the source schema objects for the reverse translation.

Now assume that in our system, the source schema is represented by o HDM schema objects. Note that $o \geq n$ because the fine grained nature of the HDM means that each high level object is represented by at least one HDM object. We require $n + o$ transformations to translate the source schema into the HDM, one to create each HDM schema object and one to create each source schema object in the reverse translation. We now require p transformations to restructure the HDM schema. If we assume the target schema is represented by q HDM schema objects we require at least $m + q$ transformations to translate from the HDM to the target DDL. As above $q \geq m$. The total number of transformations required by our system is thus at least $2(m + n)$. There will therefore be at least twice as many transformations required using our system when compared to one that translates the source schema directly into the target DDL.

This disadvantage is mitigated by the fact that our transformations are simple and can be automatically generated. As far as we know there are no systems that can automatically generate transformations from a source schema directly to a different target DDL, for anything other than a specific pair of DDLs [BM07, STZ+99].

Another disadvantage is that our translations are heuristic in nature, and the process is not always guaranteed to succeed even if a suitable translation is possible in theory. Once again, this is in common with other MMSs [ACB05, KQLL07].

5.10 Chapter Summary

This chapter has presented a data level implementation of the MM operator ModelGen that returns the translated schema along with its data instances as well as a bidirectional mapping between the source and target schemas. We have shown how a schema and its associated data instances can be translated from one DDL to another by first translating it into the HDM and then applying composite transformations to restructure the HDM schema. This restructured schema is then translated into the target DDL. We have described an algorithm for choosing the most suitable CT at each stage of the transformation process in the HDM and a mechanism for determining when a given schema matches the constructs of the target DDL. Finally, we presented a detailed example and some experimental results.

Chapter 6

MM Operator Implementation in AutoMed

In this chapter we present our implementations of the MM operators, Compose, Confluence, Merge, Extract and Diff, proposed by Bernstein et al [Ber03, BM07]. We take advantage of both the schema transformation technique of our mapping and transformation language, BAV, and the fine grained nature of the HDM, to help us create implementations for each operator. We do not present an implementation of Match in this thesis but work on Match and Merge within the AutoMed framework can be found in [RM05, MRMM05] and is the subject of ongoing work.

The implementation of each MM operator is a difficult task even in isolation. The requirement, in a MMS, that the implementations should all exist within a common framework makes it even harder. The problems the operators aim to solve, *i.e.* schema translation - ModelGen, mapping composition - Compose, schema merging - Merge, view materialisation - Extract and view complement - Diff have been studied in great detail and many different approaches to their implementation have been proposed. They all, however, look at the problems in isolation and are generally only applicable to a single DDL. Our aim in this chapter is to show how we use our MM framework to implement *all* these operators in a DDL independent manner. We do not claim that any of the implementations are necessarily the best way to solve the individual problems associated with each operator, but rather that our technique is flexible enough to cover the wide range of problems necessary to implement a generic MMS. As far as we are aware this has not been done before in a DDL independent way in a system supporting instance based semantics.

The schema transformation approach we use in our mapping and transformation language, BAV, and our CDM provide particular advantages when it comes to implementing these operators:

1. The underlying representation of schema objects in the HDM means that we can process schema objects in all high level DDLs in a uniform way.
2. The primitive transformations that make up a BAV pathway provide specific information about the semantics of each schema object. This allows us to process objects one at a time thereby breaking the problems down into simpler steps.
3. We create the mapping and result schema at the same time. This is particularly helpful in the implementation of Merge, Extract and Diff. Other techniques in the literature require a step to create the result schema and then another to create the mapping.
4. Our framework can support both LAV and GAV query processing but LAV has not been implemented at present.

It is not always necessary for the implementations of the operators to produce results that are minimal in the information theoretic sense. Indeed, it may be the case that non-minimal solutions are of more practical benefit that a strictly minimal one, some minimal solutions do not always justify the added complexity of creating them [LBU01, LV03]. With this in mind, the definitions of Diff, Extract and Merge do not require minimal solutions [MBHR05, MRB03].

Our system also takes this approach. The implementation of the Merge operator we present in Section 6.4 will sometimes produce a result schema that contains repeated data and the implementation of Diff also does not always produce a minimal result. We discuss this in more detail in the sections on Merge and Diff respectively.

6.1 Auxiliary Operators

The implementation of Invert is straightforward in AutoMed as all the pathways are bidirectional. To create the inverse of a pathway we simply make each add transformation a delete and *vice versa* and each contract an extend and *vice versa* and then execute the pathway in reverse.

Id(S) is a pathway that first adds and then removes all the objects in S. It can also be interpreted as a pathway of length zero.

We now show how we implement Domain and Range as defined in Definition 2.7 in Chapter 2. To recap, the Domain of mapping map_{S_1,S_2} is the set of values that make up the first element in the tuples of AllMapInst(map_{S_1,S_2}) and the Range the set of values that make up the second element.

Recall that the growth phase of a BAV pathway defines the extents of objects in the target schema in terms of source objects. If p_{S_1,S_2} is the BAV pathway equivalent to map_{S_1,S_2} then the result of executing Range is the union of the extents of the S_2 objects added in the growth phase of p_{S_1,S_2}. The extents of objects in the source schema that are unchanged are also in the range of the mapping since all their instances are mapped, unchanged, to S_2. Calculating Range is equivalent to GAV query processing as S_2 behaves like the global schema. This is directly supported by our system at present.

For example, consider the following mapping

$$\{\langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge c = \texttt{'Finance'} \rightarrow \langle\langle \mathsf{V}\rangle\rangle(a,c), \langle\langle \mathsf{T}\rangle\rangle(d,e) \rightarrow \langle\langle \mathsf{W}\rangle\rangle(d,e),$$
$$\langle\langle \mathsf{V}\rangle\rangle(a,c) \rightarrow \exists b(\langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge c = \texttt{'Finance'}), \langle\langle \mathsf{W}\rangle\rangle(d,e) \rightarrow \langle\langle \mathsf{T}\rangle\rangle(d,e)\}$$

between S_1 and S_2 which are defined as follows:

S_1={table:⟨⟨R⟩⟩,column:⟨⟨R, A⟩⟩, column:⟨⟨R, B⟩⟩, column:⟨⟨R, C⟩⟩, primary_key:⟨⟨R_pk, R, A⟩⟩,
table:⟨⟨T⟩⟩, column:⟨⟨T, D⟩⟩, column:⟨⟨T, E⟩⟩,primary_key:⟨⟨T_pk, T, D⟩⟩}

S_2={table:⟨⟨V⟩⟩,column:⟨⟨V, A⟩⟩, column:⟨⟨V, C⟩⟩, primary_key:⟨⟨V_pk, V, A⟩⟩,
table:⟨⟨W⟩⟩, column:⟨⟨W, D⟩⟩, column:⟨⟨W, E⟩⟩,primary_key:⟨⟨W_pk, W, D⟩⟩}

The BAV pathway equivalent to this mapping is

① add(table:⟨⟨V⟩⟩, [{a} | {a, c} ← column:⟨⟨R, C⟩⟩; c = 'Finance'])
② add(column:⟨⟨V, A⟩⟩, [{a, a} | {a, c} ← column:⟨⟨R, C⟩⟩; c = 'Finance'])
③ add(column:⟨⟨V, C⟩⟩, [{a, c} | {a, c} ← column:⟨⟨R, C⟩⟩; c = 'Finance'])
④ add(table:⟨⟨W⟩⟩, [{d} | {d, e} ← column:⟨⟨T, E⟩⟩])
⑤ add(column:⟨⟨W, D⟩⟩, [{d, d} | {d, e} ← column:⟨⟨T, E⟩⟩])
⑥ add(column:⟨⟨W, E⟩⟩, [{d, e} | {d, e} ← column:⟨⟨T, E⟩⟩])
⑦ add(primary_key:⟨⟨V_pk, V, A⟩⟩)
⑧ add(primary_key:⟨⟨W_pk, W, D⟩⟩)
⑨ delete(primary_key:⟨⟨R_pk, R, A⟩⟩)
⑩ delete(primary_key:⟨⟨T_pk, T, D⟩⟩)
⑪ contract(column:⟨⟨T, E⟩⟩, Range [{d, e} | {d, e} ← column:⟨⟨W, E⟩⟩] Any)
⑫ contract(column:⟨⟨T, D⟩⟩, Range [{d, d} | {d, e} ← column:⟨⟨W, E⟩⟩] Any)
⑬ contract(table:⟨⟨T⟩⟩, Range [{d} | {d, e} ← column:⟨⟨W, E⟩⟩] Any)
⑭ contract(column:⟨⟨R, C⟩⟩, Range [{a, c} | {a, c} ← column:⟨⟨V, C⟩⟩; c = 'Finance'] Any)
⑮ contract(column:⟨⟨R, B⟩⟩, Range [{a, b} | {a, c} ← column:⟨⟨V, C⟩⟩; c = 'Finance';
b ← generateGID(S_1, a, [a], 'B')] Any)
⑯ contract(column:⟨⟨R, A⟩⟩, Range [{a, a} | {a, c} ← column:⟨⟨V, C⟩⟩; c = 'Finance'] Any)
⑰ contract(table:⟨⟨R⟩⟩, [{a} | {a, c} ← column:⟨⟨V, D⟩⟩; c = 'Finance'])

The range is the union of the extents of the queries in Transformations ① to ⑥. If

AllInst(S_1)⊇{⟨⟨R⟩⟩(1,'Susan','HR'), ⟨⟨R⟩⟩(10,'John','Finance'), ⟨⟨R⟩⟩(20,'Anne','HR'),
⟨⟨T⟩⟩(100,'Finance'),⟨⟨T⟩⟩(101,'HR')}

then

Range(map_{S_1,S_2})⊇{⟨⟨V⟩⟩(10,'Finance')} ∪ {⟨⟨W⟩⟩(100,'Finance'),⟨⟨W⟩⟩(101,'HR')}

The domain is the union of the extents of all the S_1 objects that take part in the mapping. If we have a materialised target schema we could use the queries in the shrinking phase of the pathway to calculate the domain, since these queries return the extent of the S_1 objects that take part in the mapping in terms of S_2 objects.

If, however, the target schema has not been materialised we cannot use the extents of the S_2 objects to work out the domain. Instead, we create a query using S_1 objects. We start with the query from the shrinking phase of the pathway used to remove the S_1 object, and replace the S_2 objects in its body with the queries used in the growth phase to add those S_2 objects. These growth phase queries are made up of S_1 objects. Any S_1 objects that have no shrinking phase transformation must have been mapped unchanged to the target schema and so appear in their entirety in S_2. In this case all their instances appear in the domain.

To calculate the values of column:⟨⟨T, D⟩⟩ that appear in Domain(p_{S_1,S_2}) we create a query where column:⟨⟨W, E⟩⟩ used in the query in Transformation ⑫ that re-

moves column:⟨⟨T, D⟩⟩, is replaced with the query in Transformation ⑥ which adds column:⟨⟨W, E⟩⟩.

$[\{d, d\} \mid \{d, e\} \leftarrow [\{d, e\} \leftarrow \text{column:}\langle\langle \mathsf{T}, \mathsf{E}\rangle\rangle]]$

this simplifies to

$[\{d, d\} \mid \{d, e\} \leftarrow \text{column:}\langle\langle \mathsf{T}, \mathsf{E}\rangle\rangle]$

Similarly to calculate the values of column:⟨⟨R, C⟩⟩ that appear in the domain we use the following query:

$[\{a, a\} \mid \{a, c\} \leftarrow [\{a, c\} \leftarrow \text{column:}\langle\langle \mathsf{R}, \mathsf{C}\rangle\rangle; c = \texttt{`Finance'}]; c = \texttt{`Finance'}]$

which simplifies to

$[\{a, a\} \mid \{a, c\} \leftarrow \text{column:}\langle\langle \mathsf{R}, \mathsf{C}\rangle\rangle; c = \texttt{`Finance'}]$

As we can see we are left with a query which only contains S_1 objects. To calculate the values of column:⟨⟨R, B⟩⟩ that appear in the domain we use the following query. Note that we ignore any generateGID functions in the query to remove column:⟨⟨R, B⟩⟩ as these do not represent data values:

$[\{a, b\} \mid \{a, c\} \leftarrow [\{a, c\} \leftarrow \text{column:}\langle\langle \mathsf{R}, \mathsf{C}\rangle\rangle; c = \texttt{`Finance'}]; c = \texttt{`Finance'}]$

In the query above the b in the head of the query cannot be bound to anything in the body so the result of the query is empty. This means there are no values of column:⟨⟨R, B⟩⟩ in the domain.

We use similar queries to calculate the instances of the other S_1 objects that appear in the domain.

$\text{Domain}(p_{S_1,S_2}) \supseteq \{\text{table:}\langle\langle \mathsf{R}\rangle\rangle(10)\} \cup \{\text{column:}\langle\langle \mathsf{R}, \mathsf{A}\rangle\rangle(10{,}10)\} \cup \{\text{column:}\langle\langle \mathsf{R}, \mathsf{C}\rangle\rangle(10, \texttt{`Finance'})\} \cup \{\text{table:}\langle\langle \mathsf{T}\rangle\rangle(100), \text{table:}\langle\langle \mathsf{T}\rangle\rangle(101)\} \cup \{\text{column:}\langle\langle \mathsf{T}, \mathsf{D}\rangle\rangle(100{,}100), \text{column:}\langle\langle \mathsf{T}, \mathsf{D}\rangle\rangle(101{,}101)\} \cup \{\text{column:}\langle\langle \mathsf{T}, \mathsf{E}\rangle\rangle(100, \texttt{`Finance'}), \text{column:}\langle\langle \mathsf{T}, \mathsf{E}\rangle\rangle(101, \texttt{`HR'})\}$

6.2 Compose

We show in this section how we implement Compose as defined in Definition 2.8 in Chapter 2. Our implementation follows a procedure similar to that outlined in [BGMN06], in which a new mapping whose set of constraints is equivalent to $\Sigma_{S_1,S_2} \cup \Sigma_{S_2,S_3}$ but which contains no objects from S_2, is created.

Equivalence between sets of constraints is defined in [BGMN06] as follows: if S and S' are two schemas such that $\mathsf{AllInst}(S') \subseteq \mathsf{AllInst}(S)$ then the sets of constraints, Σ over S and Σ' over S' are equivalent, denoted $\Sigma \equiv \Sigma$', if:

Soundness Every $\mathsf{Inst}_i(S)$ satisfying Σ, when restricted to the objects in S', yields $\mathsf{Inst}_j(S')$ that satisfies Σ'.

Completeness Every $\mathsf{Inst}_j(S')$ that satisfies Σ' can be extended to $\mathsf{Inst}_i(S)$ satisfying Σ by adding new objects in $S - S'$.

The goal of an implementation of Compose can now be restated as follows: Given a set of constraints Σ_{S_1,S_2} over $S_1 \cup S_2$ and a set of constraints Σ_{S_2,S_3} over $S_2 \cup S_3$, return map_{S_1,S_3} that includes a set of constraints Σ_{S_1,S_3} over $S_1 \cup S_3$ such that $\Sigma_{S_1,S_3} \equiv \Sigma_{S_1,S_2} \cup \Sigma_{S_2,S_3}$ [BGMN06].

As we have seen, in AutoMed we execute the constraints in a mapping as transformation pathways where the constraints on the individual schema objects are defined as IQL queries. We thus define the concept of equivalence for pathways. Given a pathway p that adds and removes objects in S and a pathway p' that adds and removes objects in S' and $S' \subseteq S$, we say $p \equiv p'$ if:

Soundness Every $\mathsf{Inst}_i(S)$ created by p can also be created by p' which only adds or removes objects in S' and whose primitive transformation queries only use objects in S'.

Completeness Every $\mathsf{Inst}_j(S')$ created by p' can be extended to $\mathsf{Inst}_i(S)$ created by p by adding new transformations that add or remove objects in $S - S'$, to p', and whose transformation queries use objects in $S - S'$.

In AutoMed p_{S_2,S_3} can be appended to p_{S_1,S_2} to give us p_{S_1,S_3} [MP02]. We can thus state the goal of Compose in AutoMed as: Given a transformation pathway p_{S_1,S_3} that adds and removes objects in $S_1 \cup S_2 \cup S_3$, create a new pathway p'_{S_1,S_3} that adds and removes objects in $S_1 \cup S_3$ such that $p'_{S_1,S_3} \equiv p_{S_1,S_3}$. Our new pathway p'_{S_1,S_3} is thus p_{S_1,S_3} without the transformations that add and remove objects in S_2 and with any queries that involve S_2 objects rewritten over objects in S_1 or S_3. The rewriting uses a process based on view unfolding [Sto75].

We first split p_{S_1,S_3} into the two input pathways for the Compose operator, *i.e.* p_{S_1,S_2} and p_{S_2,S_3}. For each object, $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$, in S_2 there is either a transformation of the form

$\mathsf{add/extend}(\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle, E_w^1)$

in p_{S_1,S_2}, or the object in S_2 is the same as that in S_1. E_w^1 is an IQL expression made up of S_1 objects and/or constants and Skolem functions, We can be sure that we have such an expression because by the rules of a BAV pathway each object in S_2 must have been added to S_1 by a single transformation in p_{S_1,S_2} or be unchanged between S_1 and S_2.

S_2 objects are used in the growth phase transformation queries of p_{S_2,S_3} to define the extents of the S_3 objects we wish to create in p'_{S_1,S_3}. The transformations are of the form:

$\mathsf{add/extend}(\langle\langle \mathsf{so}_{\mathsf{j}}^3 \rangle\rangle, E_x^2)$

We now replace every occurrence of $\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle$ in E_x^2 with E_w^1, thereby eliminating $\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle$ and giving us transformations that add S_3 objects using transformation queries containing only objects from S_1. If there is no transformation that adds $\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle$ then $\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle$ is an S_1 object. It can thus remain unchanged in E_x^2.

If there is no such transformation for a given S_3 object, this means the S_3 object is the same as some object in S_2. We can therefore use the transformation that added the object in p_{S_1,S_2} as the transformation in p'_{S_1,S_3}. If there is no transformation that adds the S_3 object in p_{S_1,S_2} either then it is unchanged from S_1 to S_3 and so no transformation is necessary.

In the shrinking phase of our new pathway we need to remove the S_1 objects left in our new schema, using queries containing only S_3 objects. In p_{S_1,S_2} there are transformations of the form

$\mathsf{delete/contract}(\langle\langle \mathsf{so}_{\mathsf{k}}^1 \rangle\rangle, E_y^2)$

which remove S_1 objects using queries containing S_2 objects. p_{S_2,S_3} contains transformations of the form

$\mathsf{delete/contract}(\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle, E_z^3)$

By the same argument as we made above we create the transformation in p'_{S_1,S_3} to remove $\langle\langle \mathsf{so}_{\mathsf{k}}^1 \rangle\rangle$ by replacing any $\langle\langle \mathsf{so}_{\mathsf{i}}^2 \rangle\rangle$ in E_y^2 with E_z^3.

If p_{S_1,S_2} or p_{S_2,S_3} are not needed in other steps of the MM script this Compose operation is part of, they can be removed.

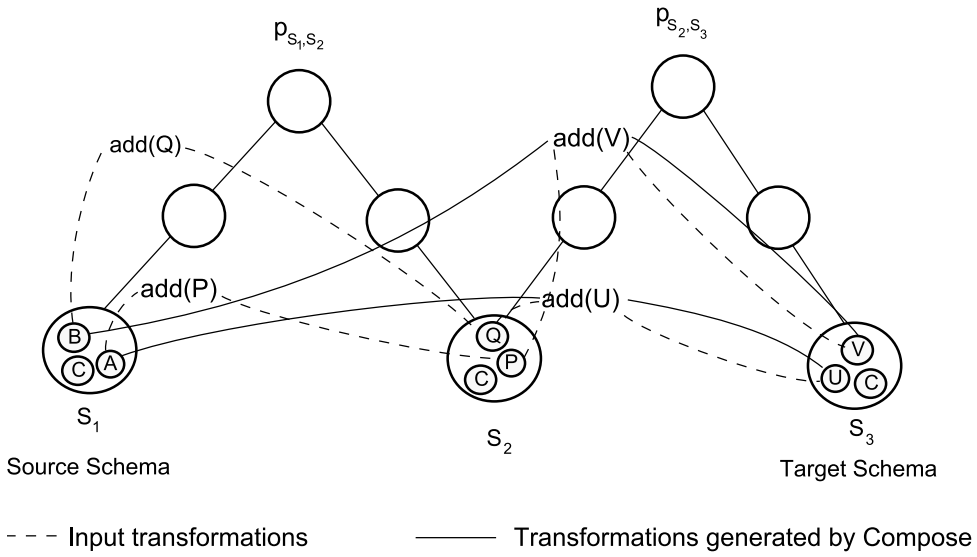


Figure 6.1: Composing two transformation pathways

We see this process illustrated graphically in Figure 6.1. Objects $\langle\langle \mathsf{A} \rangle\rangle$ and $\langle\langle \mathsf{B} \rangle\rangle$ are used in p_{S_1,S_2} to define the extents of the S_2 objects $\langle\langle \mathsf{P} \rangle\rangle$ and $\langle\langle \mathsf{Q} \rangle\rangle$ respectively. $\langle\langle \mathsf{P} \rangle\rangle$ and $\langle\langle \mathsf{Q} \rangle\rangle$ are used in turn to define the extents of the S_3 objects $\langle\langle \mathsf{U} \rangle\rangle$ and $\langle\langle \mathsf{V} \rangle\rangle$ in p_{S_2,S_3}. We use the add transformations in p_{S_2,S_3} to create the S_3 objects in the compose pathway but rewrite the queries over $\langle\langle \mathsf{A} \rangle\rangle$ and $\langle\langle \mathsf{B} \rangle\rangle$ as shown. $\langle\langle \mathsf{C} \rangle\rangle$ is unchanged from S_1 to S_3 so no transformation needs to be added for it to p'_{S_1,S_3}.

These new transformations give us a complete pathway because for each transformation that added an object to S_3 in p_{S_2,S_3} we have one in p'_{S_1,S_3} that adds the same object but does not use objects from S_2 in its transformation query. To show the new pathway is sound we need to make sure that the transformations to add and remove $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ can be recreated. We know that $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ has an extent of E_j^1 and we know we have the objects in E_j^1 in our new pathway because we used them to create the S_3 objects. We can thus use E_j^1 as the transformation query to recreate $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ as required. Similarly for the transformation to remove $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$.

The schema transformation technique helps us implement Compose because the primitive transformations tell us exactly how each S_2 object was added, giving us all the information we need to do the rewriting. The algorithm in [BGMN06] sometimes needs to manipulate the mappings first to get the S_2 object on its own on the left or right hand side of the mapping expression. This is a common feature of other recent algorithms to compose mappings [FKPT05, BGMN08, KQLJ07] which all have a step to 'pull apart' the mappings before further processing is possible. Our second order query language also means that Compose is closed in our system [FKPT05].

6.2.1 Example of Compose

Assume we have following SQL schemas:

$S_1 = \{\text{table:}\langle\langle\text{Takes}\rangle\rangle, \text{column:}\langle\langle\text{Takes, name}\rangle\rangle, \text{column:}\langle\langle\text{Takes, course}\rangle\rangle\}$

$S_2 = \{\text{table:}\langle\langle\text{Takes1}\rangle\rangle, \text{column:}\langle\langle\text{Takes1, name}\rangle\rangle, \text{column:}\langle\langle\text{Takes1, course}\rangle\rangle,$
$\text{table:}\langle\langle\text{Student}\rangle\rangle, \text{column:}\langle\langle\text{Student, name}\rangle\rangle, \text{column:}\langle\langle\text{Student, studentId}\rangle\rangle\}$

$S_3 = \{\text{table:}\langle\langle\text{Enrollment}\rangle\rangle, \text{column:}\langle\langle\text{Enrollment, studentId}\rangle\rangle, \text{column:}\langle\langle\text{Enrollment, course}\rangle\rangle\}$

We define the following mappings: $map_{S_1,S_2} = (S_1, S_2, \Sigma_{S_1,S_2})$ where

$$\Sigma_{S_1,S_2} = \{\langle\langle\text{Takes}\rangle\rangle(n,c) \rightarrow \langle\langle\text{Takes1}\rangle\rangle(n,c), \exists f \langle\langle\text{Takes}\rangle\rangle(n,c) \rightarrow \langle\langle\text{Student}\rangle\rangle(n,f(n))\}$$

and $map_{S_2,S_3} = (S_2, S_3, \Sigma_{S_2,S_3})$ where

$$\Sigma_{S_2,S_3} = \{\langle\langle\text{Student}\rangle\rangle(n,s) \wedge \langle\langle\text{Takes1}\rangle\rangle(n,c) \rightarrow \langle\langle\text{Enrollment}\rangle\rangle(s,c)\}$$

which become the following BAV pathways (We will use the first letter of the attribute names for brevity):

⑱ add(table:⟨⟨Takes1⟩⟩, table:⟨⟨Takes⟩⟩)
⑲ add(column:⟨⟨Takes1, N⟩⟩, column:⟨⟨Takes, N⟩⟩)
⑳ add(column:⟨⟨Takes1, C⟩⟩, column:⟨⟨Takes, C⟩⟩)
㉑ add(table:⟨⟨Student⟩⟩, distinct[$\{n,s\} \mid \{n,c\} \leftarrow$ table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID(S_1, n, $[n]$, 'S')])
㉒ add(column:⟨⟨Student, N⟩⟩, distinct[$\{\{n,s\},n\} \mid \{n,c\} \leftarrow$ table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID(S_1, n, $[n]$, 'S')])
㉓ add(column:⟨⟨Student, S⟩⟩, distinct[$\{\{n,s\},s\} \mid \{n,c\} \leftarrow$ table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID(S_1, n, $[n]$, 'S')])
㉔ delete(column:⟨⟨Takes, C⟩⟩, column:⟨⟨Takes1, C⟩⟩)
㉕ delete(column:⟨⟨Takes, N⟩⟩, column:⟨⟨Takes1, N⟩⟩)
㉖ delete(table:⟨⟨Takes⟩⟩, table:⟨⟨Takes1⟩⟩)

and

㉗ add(table:⟨⟨Enrollment⟩⟩, distinct[$\{s,c\} \mid \{n,s\} \leftarrow$ table:⟨⟨Student⟩⟩; $\{n,c\} \leftarrow$ table:⟨⟨Takes1⟩⟩])
㉘ add(column:⟨⟨Enrollment, S⟩⟩, distinct[$\{\{s,c\},s\} \mid \{n,s\} \leftarrow$ table:⟨⟨Student⟩⟩;
$\{n,c\} \leftarrow$ table:⟨⟨Takes1⟩⟩])
㉙ add(column:⟨⟨Enrollment, C⟩⟩, distinct[$\{\{s,c\},c\} \mid \{n,s\} \leftarrow$ table:⟨⟨Student⟩⟩;
$\{n,c\} \leftarrow$ table:⟨⟨Takes1⟩⟩])
㉚ delete(column:⟨⟨Student, S⟩⟩, distinct[$\{\{n,s\},s\} \mid \{s,c\} \leftarrow$ table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID(S_3, s, $[s]$, 'N')])
㉛ delete(column:⟨⟨Student, N⟩⟩, distinct[$\{\{n,s\},n\} \mid \{s,c\} \leftarrow$ table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID(S_3, s, $[s]$, 'N')])

㉜ delete(table:⟨⟨Student⟩⟩, distinct[$\{n, s\}$ | $\{s, c\}$ ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])
㉝ delete(column:⟨⟨Takes1, C⟩⟩, distinct[$\{\{n, c\}, s\}$ | $\{s, c\}$ ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])
㉞ delete(column:⟨⟨Takes1, N⟩⟩, distinct[$\{\{n, c\}, n\}$ | $\{s, c\}$ ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])
㉟ delete(table:⟨⟨Takes1⟩⟩, distinct[$\{n, c\}$ | $\{s, c\}$ ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])

Assume we have the following schema instances:

$\mathsf{Inst}_1(S_1)$ = {⟨⟨Takes⟩⟩('Alice','Math'),⟨⟨Takes⟩⟩('Alice','Art')}
$\mathsf{Inst}_1(S_2)$ = {⟨⟨Takes1⟩⟩('Alice','Math'),⟨⟨Takes1⟩⟩('Alice','Art'),
⟨⟨Student⟩⟩('Alice',1234)}
$\mathsf{Inst}_1(S_3)$ = {⟨⟨Enrollment⟩⟩(1234,'Math'),⟨⟨Enrollment⟩⟩(1234,'Art')}

This gives us the following mapping instances:

$\mathsf{AllMapInst}(map_{S_1,S_2}) \supseteq \langle \mathsf{Inst}_1(S_1), \mathsf{Inst}_1(S_2) \rangle$
$\mathsf{AllMapInst}(map_{S_2,S_3}) \supseteq \langle \mathsf{Inst}_1(S_2), \mathsf{Inst}_1(S_3) \rangle$

Using the definition above we can see that $\langle \mathsf{Inst}_1(S_1), \mathsf{Inst}_1(S_3) \rangle$ is a valid instance of the composition of $\mathsf{AllMapInst}(map_{S_1,S_2}) \supseteq \langle \mathsf{Inst}_1(S_1), \mathsf{Inst}_1(S_2) \rangle$ and $\mathsf{AllMapInst}(map_{S_2,S_3}) \supseteq \langle \mathsf{Inst}_1(S_2), \mathsf{Inst}_1(S_3) \rangle$.

We wish to create a new pathway that does not have the transformations ⑱ to ㉓ and ㉚ to ㉟ in it. Removing these also means we need to replace these objects in the other transformations.

The generators used in Transformation ㉗ are $\{n, s\} \leftarrow$ table:⟨⟨Student⟩⟩ and $\{n, c\} \leftarrow$ table:⟨⟨Takes1⟩⟩. These objects are added to S_2 using Transformations ⑱ and ㉑ respectively. All these transformations are add transformations so we use an add in the transformation we generate. The resultant transformations for transformations ㉗ to ㉙ are shown below:

㊱ add(table:⟨⟨Enrollment⟩⟩, distinct[$\{s, c\}$ | $\{n, c\}$ ← table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID($S_1, n, [n]$, 'S')])
㊲ add(column:⟨⟨Enrollment, S⟩⟩, distinct[$\{\{s, c\}, s\}$ | $\{n, c\}$ ← table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID($S_1, n, [n]$, 'S')])
㊳ add(column:⟨⟨Enrollment, C⟩⟩, distinct[$\{\{s, c\}, c\}$ | $\{n, c\}$ ← table:⟨⟨Takes⟩⟩;
$s \leftarrow$ generateGID($S_1, n, [n]$, 'S')])

This pathway creates S_3 as required with a student number created by a Skolem function based on the student name.

The qualifiers we use for the shrinking phase are those from the shrinking phase of p_{S_2,S_3}. The transformations we base the shrinking phase of p'_{S_1,S_3} on are those that remove objects in p_{S_1,S_2}, *i.e.* Transformations (24) to (26). In the same way as we did above, we rewrite the queries using Transformations (33) to (35) to give us:

(39) delete(column:⟨⟨Takes, C⟩⟩, distinct[{{n, c}, c} | {s, c} ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])
(40) delete(column:⟨⟨Takes, N⟩⟩, distinct[{{n, c}, n} | {s, c} ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])
(41) delete(table:⟨⟨Takes⟩⟩, distinct[{n, c} | {s, c} ← table:⟨⟨Enrollment⟩⟩;
$n \leftarrow$ generateGID($S_3, s, [s]$, 'N')])

6.3 Confluence

We show in this section how we implement Confluence, as defined in Definition 2.9 in Chapter 2. The only instance based implementation for Confluence that we are aware of is for Moda [MBHR05], which only works for the relational model. The implementation described here is thus the first DDL independent solution.

The definition of Confluence calls for us to return a mapping that combines the instances of the two input mappings, excluding any instances that conflict. In AutoMed this means we need to return a pathway which adds and removes the same objects as the input pathways and whose queries combine their instances according to the rules in Definition 2.9.

Assume that our input pathways are p_{S_1,S_2} and p'_{S_1,S_2}. For each transformation

$$t = \mathsf{op}(\mathsf{c}{:}\langle\langle \mathsf{so} \rangle\rangle, q)$$

in p_{S_1,S_2} there is a transformation

$$t' = \mathsf{op}'(\mathsf{c}{:}\langle\langle \mathsf{so} \rangle\rangle, q')$$

in p'_{S_1,S_2}. We know this must be the case because the source and target schemas of both p_{S_1,S_2} and p'_{S_1,S_2} are the same. The combination of transformation primitives used in the *growth* phase of the two pathways to add $\mathsf{c}{:}\langle\langle \mathsf{so} \rangle\rangle$ to S_2 and the transformation primitive, $\mathsf{op_c}$, we will use in the confluence pathway are as follows:

	1	2	3	4
op	add	add	extend	extend
op'	add	extend	add	extend
op_c	add	add	add	extend

There is an equivalent table for the shrinking phase where add is replaced by delete and extend by contract.

The combination of transformation primitives determines how we compute the query, q_c, that we use for the confluence transformation for c:⟨⟨so⟩⟩. As we explain below, it is not always possible at present for our system to compute the correct query. This is in common with the implementation of Confluence for Moda described in [MBHR05].

First we determine if one or both of q and q' contains the function generateGID. If so, we aim to replace the function with appropriate generators from the other query, if they are available. This allows us to map data values in our confluence mapping rather that Skolem values. We are unable to work out whether there are appropriate generators in the other query in general but can in some simple cases as we show in the second example below. Once any appropriate substitutions have been made we work out what q_c should be as follows:

- In case 1 in the table we know that q and q' completely define the extent of c:⟨⟨so⟩⟩ because both transformations use add primitives. To compute q_c we first need to determine whether $q \equiv q'$. If it is we can pick either q or q' to be q_c. If not then c:⟨⟨so⟩⟩ does not appear in the confluence mapping because the instances of c:⟨⟨so⟩⟩ conflict. At present there is no general method for determining IQL query equivalence. However, some cases of equivalence are easy to identify. For example if q can be made syntactically equal to q' by variable renaming and the reordering of the generators and filters we know q and q' are equivalent.

- Cases 2 and 3 are equivalent. In case 2 we know that q completely defines the extent of c:⟨⟨so⟩⟩ while q' only defines it partially. If $q \supseteq q'$ then we set q_c equal to q, otherwise c:⟨⟨so⟩⟩ does not appear in the confluence mapping. There is no method for determining query subsumption in IQL in general, but we can deal with the following special case: If q and q' can be made syntactically equal, as described above, apart from extra filters in q', we know that $q \supseteq q'$.

- In case 4, we know that neither q nor q' completely defines the extent of c:⟨⟨so⟩⟩ because both transformation primitives are extend. If there are no conflicts

in the results generated by q and q' then $q_c = q$ union q'. At the moment there is no general method for determining when two IQL queries produce conflicting results. As above, however, there are special cases that we can deal with. For example, consider a source schema that contains an object that has a key constraint associated with it. If q and q' contain filters that use different values of the key object to generate different results then we know that there is no conflict. Conversely, if the queries generate different results for the same key then we know there is a conflict.

6.3.1 Examples of Confluence

Consider the following schemas

S={table:⟨⟨P⟩⟩, column:⟨⟨P, A⟩⟩, column:⟨⟨P, B⟩⟩, primary_key:⟨⟨P_pk, P, A⟩⟩}
S'={table:⟨⟨Q⟩⟩, column:⟨⟨Q, A⟩⟩, column:⟨⟨Q, B⟩⟩, column:⟨⟨Q, C⟩⟩, primary_key:⟨⟨Q_pk, Q, A⟩⟩}

AllInst(S)⊇{{⟨⟨P⟩⟩(10,'John'),⟨⟨P⟩⟩(20,'Anne')}, {⟨⟨P⟩⟩(5,'Peter')}, {⟨⟨P⟩⟩(15,'Andrew')}}

and the mapping from S to S', defined by this set of tgds:

$$\{\langle\langle \mathsf{P}\rangle\rangle(a,b) \wedge a = 10 \rightarrow \langle\langle \mathsf{Q}\rangle\rangle(a,b,c) \wedge c = \text{`temp'}, \langle\langle \mathsf{Q}\rangle\rangle(a,b,c) \rightarrow \langle\langle \mathsf{P}\rangle\rangle(a,b)\}$$

The equivalent BAV pathway is shown below:

(42) extend(table:⟨⟨Q⟩⟩, Range [{a} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(43) extend(column:⟨⟨Q, A⟩⟩, Range [{a, a} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(44) extend(column:⟨⟨Q, B⟩⟩, Range [{a, b} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(45) extend(column:⟨⟨Q, C⟩⟩, Range [{a, c} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10; c ← 'temp'] Any)
(46) contract(column:⟨⟨P, B⟩⟩, Range column:⟨⟨Q, B⟩⟩ Any)
(47) contract(column:⟨⟨P, A⟩⟩, Range column:⟨⟨Q, A⟩⟩ Any)
(48) contract(table:⟨⟨P⟩⟩, Range table:⟨⟨Q⟩⟩ Any)

This gives us the following mapping instances:

AllMapInst($map_{S,S'}$)⊇ {⟨{⟨⟨P⟩⟩(10,'John')},{⟨⟨Q⟩⟩(10,'John', 'temp')}⟩}

A second mapping between S and S' is defined by this set of tgds:

$$\{\langle\langle \mathsf{P}\rangle\rangle(a,b) \wedge a = 15 \rightarrow \langle\langle \mathsf{Q}\rangle\rangle(a,b,c) \wedge c = \text{`fulltime'}, \langle\langle \mathsf{Q}\rangle\rangle(a,b,c) \rightarrow \langle\langle \mathsf{P}\rangle\rangle(a,b)\}$$

The equivalent BAV pathway, $p'_{S,S'}$, is shown below:

(49) extend(table:⟪Q⟫, Range [{a} | {a, b} ← column:⟪P, B⟫; a = 15] Any)
(50) extend(column:⟪Q, A⟫, Range [{a, a} | {a, b} ← column:⟪P, B⟫; a = 15] Any)
(51) extend(column:⟪Q, B⟫, Range [{a, b} | {a, b} ← column:⟪P, B⟫; a = 15] Any)
(52) extend(column:⟪Q, C⟫, Range [{a, c} | {a, b} ← column:⟪P, B⟫; a = 15;
c ← 'fulltime'] Any)
(53) contract(column:⟪P, B⟫, Range column:⟪Q, B⟫ Any)
(54) contract(column:⟪P, A⟫, Range column:⟪Q, A⟫ Any)
(55) contract(table:⟪P⟫, Range table:⟪Q⟫ Any)

which gives us the following mapping instances:

AllMapInst($map'_{S,S'}$) ⊇{⟨{⟪P⟫(15,'Andrew')}, {⟪Q⟫(15,'Andrew','fulltime')}⟩}

As we can see from the queries in the growth phase of the two pathways use different filters. The constant values that appear in the heads of the queries in Transformations (45) and (52) are different so we know we do not have any conflicts. The queries for growth phase in the confluence pathway are thus a union of the queries from the two pathways.

(56) extend(table:⟪Q⟫, Range [{a} | {a, b} ← column:⟪P, B⟫; a = 15] union
[{a} | {a, b} ← column:⟪P, B⟫; a = 10] Any)
(57) extend(column:⟪Q, A⟫, Range [{a, a} | {a, b} ← column:⟪P, B⟫; a = 15] union
[{a, a} | {a, b} ← column:⟪P, B⟫; a = 10] Any)
(58) extend(column:⟪Q, B⟫, Range [{a, b} | {a, b} ← column:⟪P, B⟫; a = 15] union
[{a, b} | {a, b} ← column:⟪P, B⟫; a = 10] Any)
(59) extend(column:⟪Q, C⟫, Range [{a, c} | {a, b} ← column:⟪P, B⟫; a = 15; c ← 'fulltime'] union
[{a, c} | {a, b} ← column:⟪P, B⟫; a = 10; c ← 'temp'] Any)

In the shrinking phase the queries from $p_{S,S'}$ and $p'_{S,S'}$ defining the extents of the objects in S share the same generators and there are no filters so we simply use the transformations from $p_{S,S'}$. The final confluence pathway is:

(60) contract(column:⟪P, B⟫, Range column:⟪Q, B⟫ Any)
(61) contract(column:⟪P, A⟫, Range column:⟪Q, A⟫ Any)
(62) contract(table:⟪P⟫, Range table:⟪Q⟫ Any)

The instances of the confluence mapping are:

AllMapInst($mapconf1_{S,S'}$) ⊇{⟨{⟪P⟫(10,'John')}, {⟪Q⟫(10,'John','temp')}⟩,
{⟨{⟪P⟫(15,'Andrew')}, {⟪Q⟫(15,'Andrew','fulltime')}⟩}

The following is an example where one input mapping contains a generateGID function and the other does not. We use the same schemas and instances as above but change $map'_{S,S'}$ to be defined by this set of tgds:

$$\{⟪P⟫(a,b) \wedge a = 10 \rightarrow ⟪Q⟫(a,b,Sk1(a)), ⟪Q⟫(a,b,c) \rightarrow ⟪P⟫(a,b)\}$$

The equivalent BAV pathway, $p'_{S,S'}$, is shown below:

(63) extend(table:⟨⟨Q⟩⟩, Range [{a} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(64) extend(column:⟨⟨Q, A⟩⟩, Range [{a, a} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(65) extend(column:⟨⟨Q, B⟩⟩, Range [{a, b} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10;] Any)
(66) extend(column:⟨⟨Q, C⟩⟩, Range [{a, c} | {a, a} ← column:⟨⟨P, B⟩⟩; a = 10;
c ← generateGID(S, a, [a], *'Sk1'*)] Any)
(67) contract(column:⟨⟨P, B⟩⟩, Range column:⟨⟨Q, B⟩⟩ Any)
(68) contract(column:⟨⟨P, A⟩⟩, Range column:⟨⟨Q, A⟩⟩ Any)
(69) contract(table:⟨⟨P⟩⟩, Range table:⟨⟨Q⟩⟩ Any)

which gives us the following mapping instances:

AllMapInst($map'_{S,S'}$) ⊇{⟨{⟨⟨P⟩⟩(10,'John')}, {⟨⟨Q⟩⟩(10,'John',#1000)}⟩}

where #1000 is the Skolem value generated by generateGID in Transformation (66).

As we can see, $map'_{S,S'}$ now generates a Skolem value for column:⟨⟨Q, C⟩⟩. However, Transformation (45) that adds column:⟨⟨Q, C⟩⟩ in $map_{S,S'}$ does not use any Skolem values so we use Transformation (45) in our confluence pathway. The full pathway is as follows:

(70) extend(table:⟨⟨Q⟩⟩, Range [{a} | {a, a} ← column:⟨⟨P, A⟩⟩; a = 10] Any)
(71) extend(column:⟨⟨Q, A⟩⟩, Range [{a, a} | {a, a} ← column:⟨⟨P, A⟩⟩; a = 10] Any)
(72) extend(column:⟨⟨Q, B⟩⟩, Range [{a, b} | {a, b} ← column:⟨⟨P, B⟩⟩; a = 10] Any)
(73) extend(column:⟨⟨Q, C⟩⟩, Range [{a, c} | {a, a} ← column:⟨⟨P, A⟩⟩; a = 10; c ← 'temp'] Any)
(74) contract(column:⟨⟨P, B⟩⟩, Range column:⟨⟨Q, B⟩⟩ Any)
(75) contract(column:⟨⟨P, A⟩⟩, Range column:⟨⟨Q, A⟩⟩ Any)
(76) contract(table:⟨⟨P⟩⟩, Range table:⟨⟨Q⟩⟩ Any)

The instances of the confluence mapping are:

AllMapInst($mapconf2_{S,S'}$) ⊇{⟨{⟨⟨P⟩⟩(10,'John')}, {⟨⟨Q⟩⟩(10,'John','temp')}⟩}

6.4 Merge

We show in this section how we implement Merge as defined in Definition 2.10 in Chapter 2. Our aim is to create a schema that contains all the information in the input schemas while aiming to reduce the amount of redundant information in it.

AutoMed provides us with two separate implementations of Merge whose usage depends on the type of input available to the operator. If the input is a set of

correspondences describing things like subset and disjoint relationships the implementation described in [MRMM05] can be used. If, on the other hand the input is a BAV pathway, then the implementation we describe here can be used.

A feature of both implementations that is not a general feature of merge algorithms is that a mapping between the merged schema and the input schemas is provided as part of the output. As far as we are aware only the algorithms described in [PB08, MBHR05] provide this functionality. In addition our schema transformation technique makes our implementations unique in that we create the mappings between the input schemas and the merged schema as we go rather than having to create the mapping after the merged schema has been created as needs to be done in [PB08, MRB03, MBHR05].

Our implementation takes advantage of the way BAV pathways are created in our MM system. The schema created at the end of the growth phase of the input pathway contains all the objects of S_1 and S_2. We call this schema $S_{combined}$. The pathways also provide mappings between the two input schemas and $S_{combined}$. This schema thus satisfies the conditions for Merge given in Chapter 2 in that it contains all the objects from S_1 and S_2 and we have mappings from this schema back to S_1 and S_2. However, $S_{combined}$ will contain redundant information if $\mathsf{AllInst}(S_1) \cap \mathsf{AllInst}(S_2) \neq \emptyset$. At present we are unable to calculate $\mathsf{AllInst}(S_1) \cap \mathsf{AllInst}(S_2)$ in general, and so cannot know when we have a minimal solution. A particular problem is if there are objects in S_1 and/or S_2 that do not take part in the input mapping. For example S_1 and S_2 could both contain objects that hold the same information, but if these objects do not take part in the mapping we have no way of knowing they are related and so we cannot remove either of them from our result schema. Our implementation can, however, improve on $S_{combined}$ in some cases.

Transformations in the input pathway that are of the form

$$\mathsf{add}(\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle, q) \tag{6.1}$$

where the only generators in q are schema objects in S_1, imply the extent of $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ is some subset of $\mathsf{AllInst}(S_1)$. If no objects in $S_{combined}$, including constraint objects, reference $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ it does not need to be in the merge schema. We can get the schema that does not contain $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ by rearranging the transformations in p_{S_1,S_2} so that $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ is added last. Our merge schema is then the last schema in the pathway before $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$ is added. An example is shown in Figure 6.2. The instances of $\langle\langle \mathsf{P} \rangle\rangle$ form a subset of those of $\langle\langle \mathsf{A} \rangle\rangle$ and $\langle\langle \mathsf{P} \rangle\rangle$ is not referenced by any other objects.

We rearrange our pathway so that $\langle\langle \mathsf{P} \rangle\rangle$ is added last and our merge schema, S_m, becomes the schema in the pathway just before $\langle\langle \mathsf{P} \rangle\rangle$ is added.

We can also create a more minimal solution if we have transformations of the form

$$\mathsf{extend}(\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle, \mathsf{Range}\ \langle\langle \mathsf{so}_\mathsf{j}^1 \rangle\rangle\ \mathsf{Any}) \tag{6.2}$$

in our input pathway. These imply that the extent of $\langle\langle \mathsf{so}_\mathsf{j}^1 \rangle\rangle$ forms the lower bound of the extent of $\langle\langle \mathsf{so}_\mathsf{i}^2 \rangle\rangle$. $\langle\langle \mathsf{so}_\mathsf{j}^1 \rangle\rangle$ can thus also be left out of the merge schema in the same way as we described above.

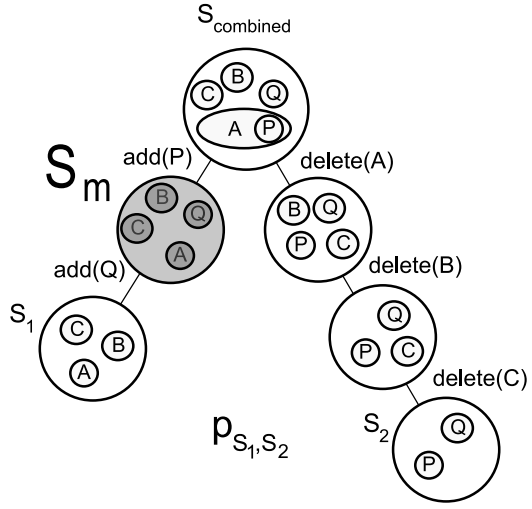


Figure 6.2: Finding a smaller merge schema

S_m, p_{S_m,S_1} and p_{S_m,S_2} meet the conditions of Merge as follows: p_{S_m,S_1} is surjective onto S_1 since executing the pathway leaves us with exactly the S_1 objects as can be seen by the following the pathway in the figure from S_m down to S_1. Similarly the pathway fragment from S_m to S_2 that makes up the rest of the pathway is surjective onto S_2. This meets condition 1 of the definition.

We have a pathway from S_1 to S_2 via S_m as required by condition 2 in the definition. Recall that we showed in Section 6.2 that if we have a pathway p_{S_1,S_3} via S_2 this is equivalent to $p_{S_1,S_2} \circ p_{S_2,S_3}$.

The third condition in the definition requires that any instance in the merged schema must be in one of the two mappings we have created, *i.e.* there can be no instances of S_m not added by one of the pathways. This follows from the way we create S_m. There are no S_1 instances of S_m that are not added by p_{S_1,S_m} and there are no S_2 objects that are not added by p_{S_2,S_m}.

The complexity of the algorithm we have described here is linear in the number of schema object in S_1 plus the number of schema objects in S_2. As we have seen though, it cannot always create a minimal result schema. As we discussed at the beginning of this chapter, this is a tradeoff that is made in other MMS prototypes [MBHR05].

6.4.1 Example of Merge

Consider the schemas S and S' defined below.

S={table:⟨⟨R⟩⟩,column:⟨⟨R, A⟩⟩, column:⟨⟨R, B⟩⟩, column:⟨⟨R, C⟩⟩, primary_key:⟨⟨R_pk, R, A⟩⟩,
table:⟨⟨T⟩⟩, column:⟨⟨T, C⟩⟩},
S'={table:⟨⟨U⟩⟩, column:⟨⟨U, A⟩⟩, column:⟨⟨U, B⟩⟩, column:⟨⟨U, C⟩⟩, primary_key:⟨⟨U_pk, U, A⟩⟩}

AllInst(S)⊇{{⟨⟨R⟩⟩(10,'John',100),⟨⟨R⟩⟩(20,'Anne',101), ⟨⟨T⟩⟩(100)},
{⟨⟨R⟩⟩(20,'Peter',101),⟨⟨T⟩⟩(101)}, {⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100)}}

AllInst(S')⊇{{⟨⟨U⟩⟩(10,'John','temp')}, {⟨⟨U⟩⟩(15,'Andrew','temp')}}

and the mapping defined by this set of tgds:

$$\{\langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge \langle\langle \mathsf{T}\rangle\rangle(c) \wedge c = 100 \rightarrow \langle\langle \mathsf{U}\rangle\rangle(a,b,d) \wedge d = \text{`temp'},$$
$$\langle\langle \mathsf{U}\rangle\rangle(a,b,d) \rightarrow \langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge \langle\langle \mathsf{T}\rangle\rangle(c) \wedge c = 100$$

The equivalent BAV pathway is shown below:

(77) add(table:⟨⟨U⟩⟩, [{a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩; c = 100])
(78) add(column:⟨⟨U, A⟩⟩, [{a, a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩; c = 100])
(79) add(column:⟨⟨U, B⟩⟩, [{a, b} | {a, b} ← column:⟨⟨R, B⟩⟩; {a, c} ← column:⟨⟨R, C⟩⟩;
{c, c} ← column:⟨⟨T, C⟩⟩; c = 100])
(80) add(column:⟨⟨U, C⟩⟩, [{a, d} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩;
c = 100; d ← 'temp'])
(81) add(primary_key:⟨⟨U_pk, U, A⟩⟩)
(82) delete(primary_key:⟨⟨R_pk, R, A⟩⟩)
(83) contract(column:⟨⟨T, C⟩⟩, Range [{c, c} | c ← 100] Any)
(84) contract(table:⟨⟨T⟩⟩, Range [{c} | c ← 100] Any)
(85) contract(column:⟨⟨R, C⟩⟩, Range [{a, c} | {a, a} ← column:⟨⟨U, A⟩⟩; c ← 100] Any)
(86) contract(column:⟨⟨R, B⟩⟩, Range column:⟨⟨U, B⟩⟩ Any)
(87) contract(column:⟨⟨R, A⟩⟩, Range column:⟨⟨U, A⟩⟩ Any)
(88) contract(table:⟨⟨R⟩⟩, Range table:⟨⟨U⟩⟩ Any)

Transformations (77) to (79) are of the form shown in Equation (6.1) so we may be able to leave one of these objects out of our result schema. The transformations imply that the data in table:⟨⟨U⟩⟩, column:⟨⟨U, A⟩⟩ and column:⟨⟨U, B⟩⟩ is a subset of that in

AllInst(S) so we could potentially leave them out of our merge schema. table:⟨⟨U⟩⟩ and column:⟨⟨U, A⟩⟩ are referenced by primary_key:⟨⟨U_pk, U, A⟩⟩ though, so we need to leave them in. We can, however, leave column:⟨⟨U, B⟩⟩ out of the merge schema. We rearrange the transformations so that Transformation (79) is the last transformation in the growth phase.

The merged schema is thus:

S_m = {table:⟨⟨R⟩⟩,column:⟨⟨R, A⟩⟩,column:⟨⟨R, B⟩⟩,column:⟨⟨R, C⟩⟩, primary_key:⟨⟨R_pk, R, A⟩⟩,
table:⟨⟨T⟩⟩,column:⟨⟨T, A⟩⟩,table:⟨⟨U⟩⟩,column:⟨⟨U, A⟩⟩, column:⟨⟨U, C⟩⟩, primary_key:⟨⟨U_pk, U, A⟩⟩}.

The instances are:

AllInst(S_m)⊇{{⟨⟨R⟩⟩(10,'John',100),⟨⟨R⟩⟩(20,'Anne',101), ⟨⟨T⟩⟩(100),⟨⟨U⟩⟩(10,'temp')},
{⟨⟨R⟩⟩(20,'Peter',101),⟨⟨T⟩⟩(101)}, {⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100),⟨⟨U⟩⟩(15,'temp')}}

To work out p_{S_m,S_1} we invert the other transformations in the growth phase to give us:

(89) delete(primary_key:⟨⟨U_pk, U, A⟩⟩)
(90) delete(column:⟨⟨U, C⟩⟩, distinct[{a, d} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩; $c = 100$;
$d \leftarrow$ 'temp'])
(91) delete(column:⟨⟨U, A⟩⟩, distinct[{a, a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩; $c = 100$])
(92) delete(table:⟨⟨U⟩⟩, distinct[{a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, c} ← column:⟨⟨T, C⟩⟩; $c = 100$])

$p_{S_m,S}$ is surjective onto S *i.e.* all the RHS elements in the mapping tuples make up AllInst(S) we can see this by inverting the pathway we have just created which removes exactly the ⟨⟨U⟩⟩ objects we added. Its instances are:

AllMapInst($map_{S_m,S}$) ⊇
{⟨{⟨⟨R⟩⟩(10,'John',100),⟨⟨R⟩⟩(20,'Anne',101),⟨⟨T⟩⟩(100), ⟨⟨U⟩⟩(10,'temp')},
{⟨⟨R⟩⟩(10,'John',100),⟨⟨R⟩⟩(20,'Anne',101),⟨⟨T⟩⟩(100)}⟩,
⟨{⟨⟨R⟩⟩(20,'Peter',101),⟨⟨T⟩⟩(101)}, {⟨⟨R⟩⟩(20,'Peter',101),⟨⟨T⟩⟩(101)}⟩
⟨{⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100), ⟨⟨U⟩⟩(10,'temp')}, {⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100)}⟩}

p_{S_m,S_2} is made up of the remaining transformations in p_{S_1,S_2} as follows:

(93) add(column:⟨⟨U, B⟩⟩, [{a, b} | {a, b} ← column:⟨⟨R, B⟩⟩; {a, c} ← column:⟨⟨R, C⟩⟩;
{c, c} ← column:⟨⟨T, C⟩⟩; c = 100])
(94) delete(primary_key:⟨⟨R_pk, R, A⟩⟩)
(95) contract(column:⟨⟨T, C⟩⟩, Range [{c, c} | $c \leftarrow$ 100] Any)
(96) contract(table:⟨⟨T⟩⟩, Range [{c} | $c \leftarrow$ 100] Any)
(97) contract(column:⟨⟨R, C⟩⟩, Range [{a, c} | {a, a} ← column:⟨⟨U, A⟩⟩; $c \leftarrow$ 100] Any)

(98) contract(column:⟪R, B⟫, Range column:⟪U, B⟫ Any)
(99) contract(column:⟪R, A⟫, Range column:⟪U, A⟫ Any)
(100) contract(table:⟪R⟫, Range table:⟪U⟫ Any)

The instances of this pathway are:

AllMapInst($map_{S_m,S'}$) $\supseteq$
{⟨{{⟪R⟫(10,'John',100),⟪R⟫(20,'Anne',101), ⟪T⟫(100), ⟪U⟫(10,'temp')},
{⟪U⟫(10,'John','temp')}⟩,
⟨{⟪R⟫(20,'Peter',101),⟪T⟫(101)}, ∅⟩
⟨{⟪R⟫(15,'Andrew',100),⟪T⟫(100), ⟪U⟫(10,'temp')}, {⟪U⟫(15,'Andrew','temp')}⟩}

We can see again that this is surjective onto S_2 and since our pathway gets us back to S_2 condition 2 is satisfied too. The union of the domains of both mappings is AllInst(S_m) so condition 3 is met.

6.5 Extract

We show in this section how we implement Extract as defined in Definition 2.11 in Chapter 2. Our approach is similar to those described in [MBHR05] and [MRB03]. We create add transformations for each object in S_1 that occurs in any add queries in p_{S_1,S_2}, as well as any objects transitively referenced by the objects we have added to our extract schema, S_x. In other words we create a schema containing all those objects whose extent is used to define the extent of a target schema object.

To determine which objects should be in S_x we look at the transformation queries in p_{S_1,S_2} used to add S_2 objects. Any S_1 object that is used to define the instances of an S_2 object should appear in S_x. These transformations will be of the form:

add/extend(⟪so_i^2⟫,E_i^1)

E_i^1 is an expression which contains generators for the S_1 objects ⟪so_1^1⟫,...,⟪so_n^1⟫. All these objects must be added to S_x. We also need to add any objects transitively referenced by any of the ⟪so_i^1⟫. Finally we any S_1 objects that are unchanged by p_{S_1,S_2}. This are those objects that are mapped in their entirety from S_1 to S_2 and so must also appear unchanged in S_x. The extent of each object we add to the extract schema is its domain in p_{S_1,S_2}. We calculate this as shown in Section 6.1.

To create the shrinking phase of the extract pathway, in which we remove all the original S_1 objects, we use the S_x objects we create in the growth phase to define

the extents of the queries. To define the extents of the objects in S_1 that do not have counterparts in S_x, we use the generateGID function to create Skolem values based on the values that do exist in S_x. These Skolem values can be unified with the actual values that appear in S_1 by the query processor.

The complexity of this algorithm is linear in the size of S_1. Our algorithm maps only those instances of S_1 that appear in the domain of p_{S_1,S_2} to S_x. These are precisely the instances that p_{S_1,S_2} maps to S_2 so condition 1 of the definition holds. The only instances of S_x are those added by the transformations in p_{S_1,S_x}. These are the instances added in the growth phase and precisely define the Range of the pathway as discussed in Section 6.1, which satisfies condition 2.

The way our mappings are defined in the AutoMed MMS allows us to create minimal results for Extract. Each object in S_1 either has a transformation in the shrinking phase of the pathway containing a query that defines the lower bound of its extent or appears unchanged in the target schema. In the latter case the whole object takes part in the mapping and so needs to appear in S_x. In the former case the query gives us a minimal set of values used from that source object. If this query can be rewritten to a non-empty query over S_1, then the rewritten query is used in a new transformations defining the extent of the object in S_x.

6.5.1 Example of Extract

Consider the following mapping between S and S''

S={table:⟨⟨R⟩⟩, column:⟨⟨R, A⟩⟩, column:⟨⟨R, B⟩⟩, column:⟨⟨R, C⟩⟩, primary_key:⟨⟨R_pk, R, A⟩⟩,
table:⟨⟨T⟩⟩, column:⟨⟨T, C⟩⟩,column:⟨⟨T, D⟩⟩,primary_key:⟨⟨T_pk, T, C⟩⟩}
S''={table:⟨⟨V⟩⟩, column:⟨⟨V, A⟩⟩, column:⟨⟨V, B⟩⟩, primary_key:⟨⟨V_pk, V, A⟩⟩}

AllInst(S)⊇{{⟨⟨R⟩⟩(10,'John',100),⟨⟨R⟩⟩(20,'Anne',101), ⟨⟨T⟩⟩(100,'Finance')},
{⟨⟨R⟩⟩(5,'Peter',101),⟨⟨T⟩⟩(101,'HR')}, {⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100,'Finance')}}

and the mapping between them defined by this set of tgds:

$$\{\langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge \langle\langle \mathsf{T}\rangle\rangle(c,d) \wedge d = \texttt{'Finance'} \rightarrow \langle\langle \mathsf{V}\rangle\rangle(a,d),$$
$$\langle\langle \mathsf{V}\rangle\rangle(a,d) \rightarrow \exists Sk1, Sk2.\langle\langle \mathsf{R}\rangle\rangle(a,b,c) \wedge \langle\langle \mathsf{T}\rangle\rangle(c,d) \wedge b = Sk1(a) \wedge c = Sk2(a)$$

The equivalent BAV pathway is shown below:

(101) add(table:⟨⟨V⟩⟩, [{a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩; d = 'Finance'])
(102) add(column:⟨⟨V, A⟩⟩, [{a, a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(103) add(column:⟨⟨V, B⟩⟩, [{a, d} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(104) add(primary_key:⟨⟨V_pk, V, A⟩⟩)
(105) delete(primary_key:⟨⟨R_pk, R, A⟩⟩)
(106) contract(column:⟨⟨T, D⟩⟩, Range [{c, d} | {a, d} ← column:⟨⟨V, A⟩⟩;
c ← generateID(S', a, [a], *'Sk2'*)] Any)
(107) contract(column:⟨⟨T, C⟩⟩, Range [{c, c} | {a, d} ← column:⟨⟨V, A⟩⟩;
c ← generateID(S', a, [a], *'Sk2'*)] Any)
(108) contract(table:⟨⟨T⟩⟩, Range [{c} | {a, d} ← column:⟨⟨V, A⟩⟩;
c ← generateID(S', a, [a], *'Sk2'*)] Any)
(109) contract(column:⟨⟨R, C⟩⟩, Range [{a, c} | {a, d} ← column:⟨⟨V, A⟩⟩;
c ← generateID(S', a, [a], *'Sk2'*)] Any)
(110) contract(column:⟨⟨R, B⟩⟩, Range [{a, b} | {a, d} ← column:⟨⟨V, A⟩⟩;
b ← generateID(S', a, [a], *'Sk1'*)] Any)
(111) contract(column:⟨⟨R, A⟩⟩, Range [{a, a} | {a, d} ← column:⟨⟨V, A⟩⟩] Any)
(112) contract(table:⟨⟨R⟩⟩, Range table:⟨⟨V⟩⟩ Any)

This pathway generates the following instances:

AllInst(S'')⊇{{⟨⟨V⟩⟩(10,'Finance')}, {⟨⟨V⟩⟩(15,'Finance')}}

If we look at the pathway above we see that column:⟨⟨R, C⟩⟩ and column:⟨⟨T, D⟩⟩ are used in the queries of Transformations (101) to (103). table:⟨⟨R⟩⟩ and table:⟨⟨T⟩⟩ are referenced by column:⟨⟨R, C⟩⟩ and column:⟨⟨T, D⟩⟩. column:⟨⟨R, A⟩⟩ is transitively referenced by column:⟨⟨R, C⟩⟩ via table:⟨⟨R⟩⟩ and primary_key:⟨⟨R_pk, R, A⟩⟩, and column:⟨⟨T, C⟩⟩ is transitively referenced by column:⟨⟨T, D⟩⟩ via table:⟨⟨T⟩⟩ and primary_key:⟨⟨T_pk, T, C⟩⟩ so we need to add all these objects to the extract schema. column:⟨⟨R, B⟩⟩ is not used in any of the queries or transitively referenced by any objects that need to be the extract schema so we can leave it out.

The query for all the add transformations is the same and each of the generators in the query affects all the objects. To calculate the extent of the objects missing from the query we add a generator for them. In each case, however, we can simplify the query (as we showed in Section 6.1) because the values generated by the generator for the new object are already available. We can get the extent of table:⟨⟨R⟩⟩ and column:⟨⟨R, A⟩⟩ from generator for column:⟨⟨R, C⟩⟩ and similarly we get the value of table:⟨⟨T⟩⟩ from the generator for column:⟨⟨T, D⟩⟩. We are thus able to use the same query as we used for the add transformations in p_{S_1,S_2} for each of the add transformations in p_{S_1,S_x}.

The queries for the transformations in the shrinking phase are all based in the corresponding objects in the extract schema except for column:⟨⟨R, B⟩⟩ which does not have a corresponding object in the extract schema. Its extent is the Skolem function used to remove it in p_{S_1,S_2}. The pathway is thus:

(113) add(table:⟨⟨extract_R⟩⟩, [{a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(114) add(column:⟨⟨extract_R, A⟩⟩, [{a, a} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(115) add(column:⟨⟨extract_R, C⟩⟩, [{a, c} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(116) add(primary_key:⟨⟨extract_R_pk, extract_R, A⟩⟩)
(117) add(table:⟨⟨extract_T⟩⟩, [{c} | {c} ← {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(118) add(column:⟨⟨extract_T, C⟩⟩, [{c, c} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(119) add(column:⟨⟨extract_T, D⟩⟩, [{c, d} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
d = 'Finance'])
(120) add(primary_key:⟨⟨extract_T_pk, T, C⟩⟩)
(121) contract(primary_key:⟨⟨T_pk, T, C⟩⟩)
(122) contract(column:⟨⟨T, D⟩⟩, Range column:⟨⟨extract_T, D⟩⟩ Any)
(123) contract(column:⟨⟨T, C⟩⟩, Range column:⟨⟨extract_T, C⟩⟩ Any)
(124) contract(table:⟨⟨T⟩⟩, Range table:⟨⟨extract_T⟩⟩ Any)
(125) contract(primary_key:⟨⟨R_pk, R, A⟩⟩)
(126) contract(column:⟨⟨R, C⟩⟩, Range column:⟨⟨extract_R, C⟩⟩ Any)
(127) contract(column:⟨⟨R, B⟩⟩, Range [{a, b} | {a, a} ← column:⟨⟨extract_R, A⟩⟩;
b ← generateGID(S", a, [a], *'Sk1'*)] Any)
(128) contract(column:⟨⟨R, A⟩⟩, Range column:⟨⟨extract_R, A⟩⟩ Any)
(129) contract(table:⟨⟨R⟩⟩, Range table:⟨⟨extract_R⟩⟩ Any)
(130) rename(table:⟨⟨extract_R⟩⟩, table:⟨⟨R⟩⟩)
(131) rename(table:⟨⟨extract_T⟩⟩, table:⟨⟨T⟩⟩)

The instances of the extract schema created by this pathway are as follows:

AllInst(S_x) = {{⟨⟨R⟩⟩(10,100),⟨⟨T⟩⟩(100,'Finance')}, {⟨⟨R⟩⟩(15,100),⟨⟨T⟩⟩(100,'Finance')}}

p_{S,S_x} thus generates the following mapping instances:

AllMapInst(map_{S,S_x})=
{⟨{⟨⟨R⟩⟩(10,'John',100), ⟨⟨T⟩⟩(100,'Finance')},
{⟨⟨R⟩⟩(10,100), ⟨⟨T⟩⟩(100,'Finance')}⟩,
⟨{⟨⟨R⟩⟩(15,'Andrew',100),⟨⟨T⟩⟩(100,'Finance')}, {⟨⟨R⟩⟩(15,100),⟨⟨T⟩⟩(100,'Finance')}⟩}

Invert(p_{S,S_x}) which we get by inverting the operators and executing the pathway above in reverse order, generates the following mapping instances:

AllMapInst($map_{S_x,S}$)=
{⟨{⟪R⟫(10,100), ⟪T⟫(100,'Finance')},
{⟪R⟫(10,Sk1(10)='John',100), ⟪T⟫(100,'Finance')}⟩,
⟨{⟪R⟫(15,100),⟪T⟫(100,'Finance')}, {⟪R⟫(15,Sk1(15)='Andrew',100),⟪T⟫(100,'Finance')}⟩}

We can see by examining the mapping instances above that composing these two mappings and deskolemising $Sk1$ using the values in the source schema, gives us Id(Domain(map_{S_1,S_2})).

This is equivalent to a pathway of length 0 so Id(S) $\circ$ p_{S_1,S_2} = p_{S_1,S_2} as required by the definition.

Condition 2:

Range(p_{S,S_x}) = {{⟪R⟫(10,100),⟪T⟫(100,'Finance')}, {⟪R⟫(15,100),⟪T⟫(100,'Finance')}}
= AllInst(S_x)

6.6 Diff

We show in this section how we implement Diff as defined in Definition 2.12 in Chapter 2. The definition requires us to be able to recreate S_1 by merging the result of Diff with the result of an Extract with the same input parameters. This means we need to include any objects in S_1 that are used to uniquely identify instances in S_1 in our diff schema, S_d,. We can thus restate the requirements of Diff as follows: to return a schema containing all instances of S_1 that do not participate in the mapping plus any objects and their instances, necessary to uniquely identify instances of S_1. As with Extract, S_1 will meet these requirements.

The implementation we present here follows the same basic principles as we applied in the implementation of Extract but here we select schema objects that are *not* used in the queries of any growth phase transformations in p_{S_1,S_2} as well as those needed to uniquely identify instances of S_1, for example a primary key column.

To determine which objects should be in S_d we look at the queries used to add S_2 objects. Any S_1 object that is not used in any of the queries that define the instances of S_2 must appear in S_d. In addition, any object whose instances are filtered must appear in S_d. This is to allow us to recreate the input schema. As in the case of Extract we need to add any objects transitively referenced by these objects.

If the instances of an object we add are filtered we may only need to add the instances that make up the complement of the query to S_d. For example, if an add query in p_{S_1,S_2} contained a filter, $a = 10$, the query in the transformation to create the S_d object would be $a \neq 10$. The exception is if the object is transitively referenced by an object whose instances all need to appear in S_d, since we need to be able to uniquely identify each instance in S_d to allow us to merge it with the output of Extract as required by the definition.

We create the shrinking phase by using the instances of the objects we have added to S_d to define the extents of the transformation queries used to remove their corresponding S_1 objects. If there is no S_d object corresponding to an object we need to remove from S_1, we use generateGID to create a Skolem function to define the extent of the object. As with the algorithm for Extract, the complexity of this algorithm is linear in the size of S_1.

As we said previously we cannot always guarantee a minimal result schema in an information theoretic sense. In particular we do not analyse the consequences of the input schema having a finite domain. Consider the mapping

$$\langle\langle \mathsf{R} \rangle\rangle(a, b) \rightarrow \langle\langle \mathsf{T} \rangle\rangle(a)$$

between $S = \{\mathsf{table}{:}\langle\langle \mathsf{R} \rangle\rangle, \mathsf{column}{:}\langle\langle \mathsf{R}, \mathsf{A} \rangle\rangle, \mathsf{column}{:}\langle\langle \mathsf{R}, \mathsf{B} \rangle\rangle, \mathsf{primary\_key}{:}\langle\langle \mathsf{R\_pk}, \mathsf{R}, \mathsf{A} \rangle\rangle\}$ and $S' = \{\mathsf{table}{:}\langle\langle \mathsf{T} \rangle\rangle, \mathsf{column}{:}\langle\langle \mathsf{T}, \mathsf{A} \rangle\rangle, \mathsf{primary\_key}{:}\langle\langle \mathsf{T\_pk}, \mathsf{T}, \mathsf{A} \rangle\rangle\}$ and suppose that $\mathsf{column}{:}\langle\langle \mathsf{R}, \mathsf{A} \rangle\rangle$ has a finite domain $\mathcal{I}$. Our algorithm for Diff would return $\mathsf{table}{:}\langle\langle \mathsf{diff\_R} \rangle\rangle{=}\mathsf{table}{:}\langle\langle \mathsf{R} \rangle\rangle$ as the result schema, with the mapping

$$\langle\langle \mathsf{R} \rangle\rangle(a, b) \rightarrow \langle\langle \mathsf{diff\_R} \rangle\rangle(a, b)$$

In our implementation of Diff we include the values from $\mathsf{column}{:}\langle\langle \mathsf{R}, \mathsf{B} \rangle\rangle$ but we also include all the values from $\mathsf{column}{:}\langle\langle \mathsf{R}, \mathsf{A} \rangle\rangle$ in the result schema because it is the key of $\mathsf{table}{:}\langle\langle \mathsf{R} \rangle\rangle$ and its values are used to uniquely identify the rows in the table. We need to do this so we can uniquely recreate the original schema by merging this diff schema with one generated by extract, as required by the definition. However, the minimal result is a set of objects of the form $\mathsf{table}{:}\langle\langle \mathsf{S_i} \rangle\rangle(\mathsf{B})$ where $i \in \mathcal{I}$. These relations do not contain the data values from $\mathsf{column}{:}\langle\langle \mathsf{A} \rangle\rangle$ and so are minimal in an information theoretic sense. The mapping from the source to the result schema is

$$\langle\langle \mathsf{R} \rangle\rangle(a, b) \rightarrow \langle\langle \mathsf{S}_a \rangle\rangle(b)$$

This schema, however, is not always of practical use. If $\mathcal{I}$ is the domain of values that can be stored in a normal signed 32-bit integer then the result schema will contain 2^{31} table definitions.

6.6.1 Example of Diff

We use the same schema and pathway as in the extract section. In $p_{S,S''}$ table:⟨⟨R⟩⟩, column:⟨⟨R, A⟩⟩,column:⟨⟨R, B⟩⟩,table:⟨⟨T⟩⟩ and column:⟨⟨T, C⟩⟩ are not used in any of the queries so must appear in the diff schema, S_d. The instances of column:⟨⟨R, C⟩⟩ and column:⟨⟨T, D⟩⟩ are filtered by the filter d='Finance' and so also need to appear in S_d. We thus need to add all of S_1's objects to S_d. We need to add all the instances of column:⟨⟨R, B⟩⟩ so therefore we need to add all the instances of the objects transitively referenced by column:⟨⟨R, B⟩⟩, *i.e.* table:⟨⟨R⟩⟩, column:⟨⟨R, A⟩⟩,column:⟨⟨R, C⟩⟩. The instances of table:⟨⟨T⟩⟩, column:⟨⟨T, C⟩⟩ and column:⟨⟨T, D⟩⟩ are those generated by inverting the filter d='Finance' so that it becomes $d \neq$'Finance'. p_{S_1,S_d} is thus:

(132) add(table:⟨⟨diff_R⟩⟩, table:⟨⟨R⟩⟩)
(133) add(column:⟨⟨diff_R, A⟩⟩, column:⟨⟨R, A⟩⟩)
(134) add(column:⟨⟨diff_R, B⟩⟩, column:⟨⟨R, B⟩⟩)
(135) add(column:⟨⟨diff_R, C⟩⟩, column:⟨⟨R, C⟩⟩)
(136) add(table:⟨⟨diff_T⟩⟩, [{c} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
$d \neq$ 'Finance'])
(137) add(column:⟨⟨diff_T, C⟩⟩, [{c, c} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
$d \neq$ 'Finance'])
(138) add(column:⟨⟨diff_T, D⟩⟩, [{c, d} | {a, c} ← column:⟨⟨R, C⟩⟩; {c, d} ← column:⟨⟨T, D⟩⟩;
$d \neq$ 'Finance'])
(139) contract(column:⟨⟨T, D⟩⟩, Range column:⟨⟨diff_T, D⟩⟩ Any)
(140) contract(column:⟨⟨T, C⟩⟩, Range column:⟨⟨diff_T, C⟩⟩ Any)
(141) contract(table:⟨⟨T⟩⟩, Range table:⟨⟨diff_T⟩⟩ Any)
(142) contract(column:⟨⟨R, C⟩⟩, Range column:⟨⟨diff_R, C⟩⟩ Any)
(143) contract(column:⟨⟨R, B⟩⟩, Range column:⟨⟨diff_R, B⟩⟩ Any)
(144) contract(column:⟨⟨R, A⟩⟩, Range column:⟨⟨diff_R, A⟩⟩ Any)
(145) contract(table:⟨⟨R⟩⟩, Range table:⟨⟨diff_R⟩⟩ Any)
(146) rename(table:⟨⟨diff_R⟩⟩, table:⟨⟨R⟩⟩)
(147) rename(table:⟨⟨diff_T⟩⟩, table:⟨⟨T⟩⟩)

which generates the following schema:

S_d = {table:⟨⟨R⟩⟩, column:⟨⟨R, A⟩⟩, column:⟨⟨R, B⟩⟩, column:⟨⟨R, C⟩⟩, primary_key:⟨⟨R_pk, R, A⟩⟩,
table:⟨⟨T⟩⟩, column:⟨⟨T, C⟩⟩,column:⟨⟨T, D⟩⟩,primary_key:⟨⟨T_pk, T, C⟩⟩}

The instances of S_d and S_x that we created above are:

AllInst(S_d)⊇{{⟪R⟫(10,'John',100), ⟪R⟫(20,'Anne',101)},
{⟪R⟫(5,'Peter',101),⟪T⟫(101,'HR')}, {⟪R⟫(15,'Andrew',100)}}
AllInst(S_x) ⊇ {{⟪R⟫(10,100),⟪T⟫(100,'Finance')}, {⟪R⟫(15,100),⟪T⟫(100,'Finance')}}

We can see by inspection that merging these two schemas will give us back S as required by the definition.

AllInst(S)⊇{{⟪R⟫(10,'John',100),⟪R⟫(20,'Anne',101), ⟪T⟫(100,'Finance')},
{⟪R⟫(5,'Peter',101),⟪T⟫(101,'HR')}, {⟪R⟫(15,'Andrew',100),⟪T⟫(100,'Finance')}}

6.7 TransGen

The AutoMed implementation of TransGen takes a string representation of the SO s-t tgds in the input mapping and translates them into a BAV pathway. The string representation of the mapping used in the previous two sections is given below:

```
Sd:sql:table:<<R>>(a) and S:sql:column:<<R,A>>(a,a) and
S:sql:column:<<R,B>>(a,b) and S:sql:column:<<R,C>>(a,c) and
S:sql:column:<<T,C>>(c,c) and
S:sql:column:<<T,D>>(c,d) and d='Finance' ->
Sd:sql:table:<<V>>(a) and Sd:sql:column<<V,A>>(a,a) and Sd:sql:column<<V,B>>(a,d)

Sd:sql:table:<<V>>(a) and Sd:sql:column<<V,A>>(a,a) and Sd:sql:column<<V,B>>(a,d)->
S:sql:table:<<R>>(a) and S:sql:column:<<R,A>>(a,a) and
S:sql:column:<<R,B>>(a,b) and S:sql:column:<<R,C>>(a,c) and
S:sql:table:<<T>>(c) and S:sql:column:<<T,C>>(c,c) and
S:sql:column:<<T,D>>(c,d) and b = Sk1(a) and c = Sk2(a)
```

This is translated into the BAV pathway shown in Section 6.5.1 by a simple text parser using the process described in Section 3.4.

6.8 Model Management Scripts

The operators we have described here have been implemented as an addition to the AutoMed Java API and can be used to create programs that directly execute MM scripts. The AutoMed API contains a number of classes that provide the underlying framework of schemas and mappings (in the form of BAV pathways) [BKL+04]

that we need for our MMS. The three main classes that we use in the implementation of our MM API are `Schema`, `Pathway` and `Transformation`. These provide methods to manipulate the schemas and mappings in our MMS. AutoMed also provides a GUI that allows us to visualise the results produced by the operators and manipulate them directly.

The MM API is made up of one class for each operator. Each of these classes extends the basic `Operator` class which contains a number of methods that allow us to manipulate the schemas and transformation pathways that make up the input parameters of the operators. Each operator class contains a method that executes the operator. The signatures of the methods are shown in Table 6.1.

MM Methods
`Pathway` p = transGen(`String` map)
`Pathway` $pcomp$ = `compose`(`Pathway` $p1$, `Pathway` $p2$)
`Pathway` $pcon$ = `confluence`(`Pathway` $p1$, `Pathway` $p2$)
`Pathway` $p[\,]$ = `merge`(`Schema` $S1$, `Schema` $S2$, `Pathway` p)
`Pathway` pd = `diff`(`Schema` S, `Pathway` p)
`Pathway` px = `extract`(`Schema` S, `Pathway` p)
`Pathway` pmg = `modelGen`(`Schema` S, `String` l)

Table 6.1: The MM Method Calls

The inputs for all the MM operators other than `transGen` are `Schema` and `Pathway` objects. The `Schema` objects we use are generally created by wrapping an external data source using the `AutoMedWrapper` class that exists as part of the AutoMed API or by extracting the final schema the pathway returned by the MM operators. In addition, AutoMed provides mechanisms for creating schemas that are not based on a data source either by reading in a textual description of a schema from a file or by directly creating the schema objects using the GUI.

The `Pathway` objects are made up of a sequence of transformations and the schemas created by those transformations. The Transformation class has methods that give us access to the transformation type, the schema object and the transformation query. The user-generated pathways in a MM script are created by using `transGen` which translates the `String` objects representing the mappings into BAV pathways. The other pathways will be the output of other MM operators.

As we can see from the table all the MM operator methods return a `Pathway` object except for `merge` which returns an array of two `Pathway` objects, one for each mapping required by the definition. In the case of the Compose and Confluence operators these are the only outputs we need. However, for Diff, Extract, Merge

and ModelGen we also need a schema as part of the output. In the case of `diff`, `extract` and `modelGen` this schema is the last one in the returned `Pathway` object. We access it by using the `getLastSchema(`p`)` method that returns the final schema in a `Pathway` object. We can obtain the result schema of `merge` by executing the `getFirstSchema(`p`)` method, which returns the first schema in a `Pathway` object, on either of the pathway objects in the array that the method returns, as the pathways are from the merged schema to the input schemas.

We can use the operators directly through the AutoMed GUI or as part of a program. The fragment below shows how we can combine these new classes with the existing framework provided by AutoMed to execute a MM script. This executes the script from Example 2.3 in Section 2.4.

```
AutoMedWrapper sw=AutoMedWrapper.selectNewAutoMedWrapper(
      username,password,null,driver,urlbase+dbName,"SEmp",wf);
Schema SEmp=sw.getSchema();
  TransGen transGenObject = new TransGen();
Pathway pSEmp_SPers = transGenObject.transGen(mapSEmp_SPers);
Schema SPers = pSEmp_SPers.getLastSchema();

Schema[] pathway = SPers.findShortestPath(SEmp);
Pathway mapSPers_SEmp = Pathway.createPathway(pathway);

Diff diffObject = new Diff();
Pathway pSPers_Sd = diffObject.diff(SPers, pSPers_SEmp);
Schema Sd = pSPers_Sd.getLastSchema();
Sd.setName("Sd");

Compose composeObject = new Compose();
Pathway mapSEmp_Sd = composeObject.compose(mapSEmp_SPers, mapSPers_Sd);

Merge mergeObject = new Merge();
Pathway res[] = mergeObject.merge(SEmp, Sd, mapSEmp_Sd);
Schema Sm = res[0].getFirstSchema();
Sm.setName("Sm");
```

We first wrap the `SEmp` schema from the example using `selectNewAutoMedWrapper` in the `AutoMedWrapper` class and access the AutoMed schema created by the wrapping process using the `getSchema()` method on the next line. We then use `transGen` to create the BAV pathway that executes the mapping from `SEmp` to `SPers`.

We create the inverse of this pathway by simply reversing the order of the schemas used to create the pathway. We use the existing AutoMed methods `findShortestPath` and `createPathway` to work out which schemas should be in the pathway and then link them in the correct order.

We next create an instance of the new `Diff()` class and execute its `diff` method to create a BAV pathway from `SPers` to a schema that is the Diff of `SPers` and `SEmp`. This schema is the last in the pathway and is accessed using the existing `getLastSchema` method which is part of the `Pathway` class. We use the `setName` method in `Schema` to set the name of this schema to `Sd`.

We next create an instance of the new `Compose` object to allow us to compose the pathway we have just created with the original pathway we created to give us a pathway directly from `SEmp` to `Sd`.

We then merge `SEmp` and `Sd`. The pathway we just created describes the relationship between their objects. The result of `merge` is an array of two pathways. The merged schema is the first schema in both the pathways. We extract the merged schema from the first pathway in the array using the `getFirstSchema` method we described earlier.

Adding the MM API to the existing AutoMed system provides us with a powerful tool for data management.

6.9 Related Work

To our knowledge there is no MM framework that offers DDL-independent implementations of the instance-based semantics of all the MM operators presented in this chapter. Implementations exist for Moda but these are only for the relational model [MBHR05], and for Rondo but only for structural semantics [MRB03]. *GeromeSuite* implements Match, Merge and Compose for multiple DDLs but not Diff, Extract or ModelGen. At present *GeromeSuite* also does not provide an API that allows the operators to be used together as we do here [KQLL07].

6.10 Chapter Summary

We have shown in this chapter how our framework offers a flexible environment in which to implement MM operators. The operators discussed in this chapter as well as ModelGen have all been implemented in Java and added to AutoMed. They can be combined into programs to solve complex data management problems as we will see in the next chapter.

Chapter 7

Case Studies

So far in this thesis the examples we have presented have all been schema based. This is because, in common with all existing MMS prototypes, the AutoMed MMS can only process schema based DDLs. In this chapter we present some examples of scripts that can be used to solve problems in other application domains, before describing in detail how we use the AutoMed MMS to execute a script to do schema-based change propagation.

The high level of abstraction in MM allows the techniques to be applied to a wide range of domains [BHP00]. We first discuss how these techniques can be applied to a problem associated with middleware messaging. A common task in heterogeneous messaging systems is that of merging messages that conform to different messaging standards. The MM script below could be used to perform this task. Message m_{1-ML1} is expressed in a message standard `ML1` while m_{2-ML2} is expressed using a standard `ML2`.

1. $\langle m_{1-ML2}, map_{m_{1-ML1},m_{1-ML2}} \rangle = \mathsf{ModelGen}(m_{1-ML1}, \texttt{ML2})$
2. $map_{m_{1-ML2},m_{2-ML2}} = \mathsf{Match}(m_{1-ML2}, m_{2-ML2})$
3. $\langle m_{3-ML2}, map_{m_{3-ML2},1-ML2}, map_{m_{3-ML2},2-ML2} =$
   $\mathsf{Merge}(m_{1-ML2}, m_{2-ML2}, map_{m_{1-ML2},m_{2-ML2}})$

The merged message, m_{3-ML2}, is expressed using the `ML2` standard.

Programming interfaces are another domain that we can apply MM techniques to. We could, for example, create a script that extracts the common elements from two programming interfaces, as well as those that only occur in one or other of the

interfaces. The script below takes two interfaces, p_1 and p_2 as input and returns three new interfaces. The interface containing the common elements is p_e, that containing only those elements that only occur in p_1 is p_{d1} and that containing only those that only occur in p_2 is p_{d2}.

1. $\langle map_{p_1,p_2} \rangle = \mathsf{Match}(p_1, p_2)$
2. $\langle p_e, map_{p_1,p_e} \rangle = \mathsf{Extract}(p_1, map_{p_1,p_2})$
3. $\langle p_{d1}, map_{p_1,d1} \rangle = \mathsf{Diff}(p_1, map_{p_1,p_2})$
4. $\langle p_{d2}, map_{p_,,d2} \rangle = \mathsf{Diff}(p_2, \mathsf{Invert}(map_{p_1,p_2}))$

Another type of model we may wish to add to a MMS is an ontology modelling language. OWL-DL [Mik04] is a semantic web language used to define ontologies in which rules can be written that allow a reasoner to infer new facts from those given. If we assume the mapping between an ontology o_1 and its inferred closure, o_{1-INF}, is $map_{o_1,o_{1-INF}}$, we could use the following MM script to work out what the inferred facts in an ontology are:

1. $\langle o_{inf}, map_{o_1,o_{1-INF}} = \mathsf{Diff}(o_1, map_{o_1,o_{1-INF}})$

We have gained an idea of the problems that we would face in adding OWL-DL to our particular MMS prototype in recent work we have done in translating OWL-DL ontologies into SQL [SRM09] where we used specially designed SQL triggers to simulate OWL-DL's inference ability.

Creating *executable* mappings between the different types of models described above has proved to be extremely challenging [BM07, JJNQ09]. We can find no references in any of the MM literature to the *implementation* of a MMS that processes anything other than schema-based DDLs. Since the initial paper [BHP00], a MM 2.0 has been proposed that focuses on creating extensional mappings between schema-based DDLs [ACB06, BM07, JJNQ09, BH07] rather than how MM techniques can be applied to other application domains. All of the existing MMS prototypes limit the DDLs they support to those that are schema-based.

At the moment we also do not have a way of characterising non schema-oriented models in the AutoMed MMS. We also have no way of dealing with things like the timing of messages or events in the messaging domain. Therefore, our implementations of the operators are currently not able to process anything other that schema-oriented models. We believe, however, that the fine-grained transformation based approach to representing schemas and mappings and implementing the

operators described in this thesis, provides the best way of extending MM implementations to process other types of models, when compared with the approaches adopted in the other existing MMS prototypes. The ability to 'divide and conquer' the problems by transforming a single, simple schema object at a time reduces the complexity of the individual steps. Extending our MMS to do this forms part of the future work we discuss in Chapter 8.

7.1 Schema-Based Change Propagation Example

We now present a MM script to solve the schema-based change propagation problem introduced in Section 1.3. Aspects of this script perform the common data management problems view integration, schema translation and data exchange. This example is one we have created ourselves, based on the scenario presented in [MBHR05], because as far as we are aware there are no existing benchmarks for MMSs [BM07].

Recall that the example concerned the maintenance of a company database linked to an XML schema containing just the finance employees used by the HR department. A figure illustrating the full example, annotated with the steps we will execute, is in shown in Figure 7.1. The process can be split into four stages. The first stage translates the ER design into SQL. The output of this stage is a mapping between the ER and SQL schemas shown as the darker arrow on Step 1 in the figure. At the end of each of the next three stages we have a mapping between the current SQL database and the current, consistent XML schema used by HR. The darker arrows on 6, 12 and 16 show these output mappings. S_{emp} and S_{finEmp} are the schemas we have been using throughout the thesis. The ER and XML schemas based on them have an $-er$ or $-xml$ suffix in their names. The full script is shown in Figure 7.2.

The four stages of the process are:

A The first stage consists of Step 1 of the script and is an example of **schema translation**. ModelGen is executed to translate the ER design into SQL.

B In this stage the XML schema required by HR is created and updated. A mapping between the database and the new schema is also produced at the end of the stage. Steps 2 to 5 create the mappings between S_{emp} and the updated XML schema created by HR, $S'_{finEmp-xml}$. The XML schema is linked directly to S_{emp} in Step 6 by composing the mappings created in the previous steps. This is an example of **data exchange**.

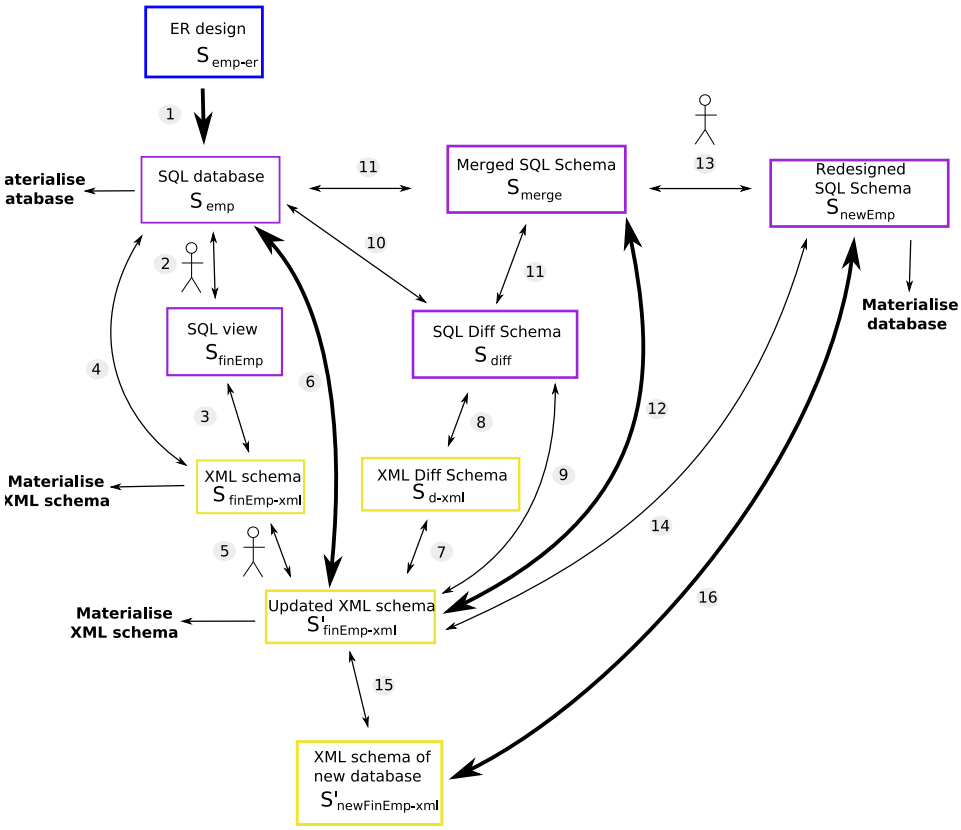


Figure 7.1: Detailed diagram of the Example

1. $\langle S_{emp}, map_{S_{emp-er},S_{emp}} \rangle = \mathsf{ModelGen}(S_{emp-er}, \mathtt{SQL})$
2. $map_{S_{emp},S_{finEmp}}$ = User defined view creation
3. $\langle S_{finEmp-xml}, S_{finEmp}\text{-}S_{finEmp-xml} \rangle = \mathsf{ModelGen}(S_{finEmp}, XML)$
4. $map_{S_{emp},S_{finEmp-xml}} = map_{S_{emp},S_{finEmp}} \circ map_{S_{finEmp},S_{finEmp-xml}}$
5. $map_{S_{finEmp-xml},S'_{finEmp-xml}}$ = User defined additions to XML Schema
6. $map_{S_{emp},S'_{finEmp-xml}} = map_{S_{emp},S_{finEmp-xml}} \circ map_{S_{finEmp-xml},S'_{finEmp-xml}}$
7. $\langle S_{d-xml}, map_{S'_{finEmp-xml},S_{d-xml}} \rangle = \mathsf{Diff}(S'_{finEmp-xml}, \mathsf{Invert}(map_{S_{emp},S'_{finEmp-xml}}))$
8. $\langle S_{diff}, map_{S_{d-xml},S_{diff}} \rangle = \mathsf{ModelGen}(S_{d-xml}, \mathtt{SQL})$
9. $map_{S'_{finEmp-xml},S_{diff}} = map_{S'_{finEmp-xml},S_{d-xml}} \circ map_{S_{d-xml},S_{diff}}$
10. $map_{S_{emp},S_{diff}} = map_{S_{emp},S'_{finEmp-xml}} \circ map_{S'_{finEmp-xml},S_{diff}}$
11. $\langle S_m, map_{S_{merge},S_{emp}}, map_{S_{merge},S_{diff}} \rangle = \mathsf{Merge}(S_{emp}, S_{diff}, map_{S_{emp},S_{diff}})$
12. $map_{S_{merge},S'_{finEmp}} = (map_{S_{merge},S_{emp}} \circ map_{S_{emp},S'_{finEmp-xml}}) \oplus (map_{S_{merge},S_{diff}} \circ \mathsf{Invert}(map_{S'_{finEmp-xml},S_{diff}}))$
13. $map_{S_{merge},S_{newEmp}}$ = User defined improved SQL design
14. $map_{S'_{finEmp-xml},S_{newEmp}} = \mathsf{Invert}(map_{S_{merge},S'_{finEmp-xml}}) \circ map_{S_{merge},S_{newEmp}}$
15. $\langle S'_{newFinEmp-xml}, map_{S'_{finEmp-xml},S'_{newFinEmp-xml}} \rangle = \mathsf{Extract}(S'_{finEmp-xml}, map_{S'_{finEmp-xml},S_{newEmp}})$
16. $map_{S_{newEmp},S'_{newFinEmp-xml}} = \mathsf{Invert}(map_{S'_{finEmp-xml},S_{newEmp}}) \circ map_{S'_{finEmp-xml},S'_{newFinEmp-xml}}$

Figure 7.2: MM Script

C During this stage the database is updated with the new objects added by HR, to their schema. Again a mapping is created between this new database and the existing HR schema is created as the final step in this stage. In Steps 7 to 11 the differences between $S_{finEmp-xml}$ are identified and merged into S_{emp} to give us S_{merge}. This is an example of view integration. This leaves us with two different pathways between $S'_{finEmp-xml}$ and S_{merge}, one via S_{emp} and the other via S_{diff}. Step 12 combines these pathways to link $S'_{finEmp-xml}$ directly to the new SQL database.

D In the final stage a new database is created, the HR schema is updated to reflect these changes and a mapping between the new database and the updated HR schema is created. This is another example of **data exchange**.

7.2 Script Execution in AutoMed

In this section we use the implementations of the operators described in the previous two chapters to execute the script. We also provide some screen shots from the system itself and for completeness we include all the transformations and schemas.

7.2.1 Stage A

The first step in the script

$\langle S_{emp}, map_{S_{emp-er},S_{emp}} \rangle = \mathsf{ModelGen}(S_{emp-er}, \mathtt{SQL})$

translates the ER design into an SQL schema that can be materialised. Only one CT is required to transform the HDM representation of S_{emp-er}, into a schema that can be translated in SQL:

$\mathsf{inclusion\_expand}(S_{emp-er}, \mathsf{node}{:}\langle\langle \mathsf{Dept} \rangle\rangle, \mathsf{edge}{:}\langle\langle \mathsf{works\_in}, \mathsf{Emp}, \mathsf{Dept} \rangle\rangle)$

This splits the schema into two parts linked by an inclusion constraint($\subseteq$). A screenshot of the initial ER schema and its translation into SQL via the HDM is shown in Figure 7.3. The SQL schema is S_{emp}, which we used throughout Chapter 2. We repeat $\mathsf{Inst}_5(S_{emp})$ in Figure 7.4 here for convenience.

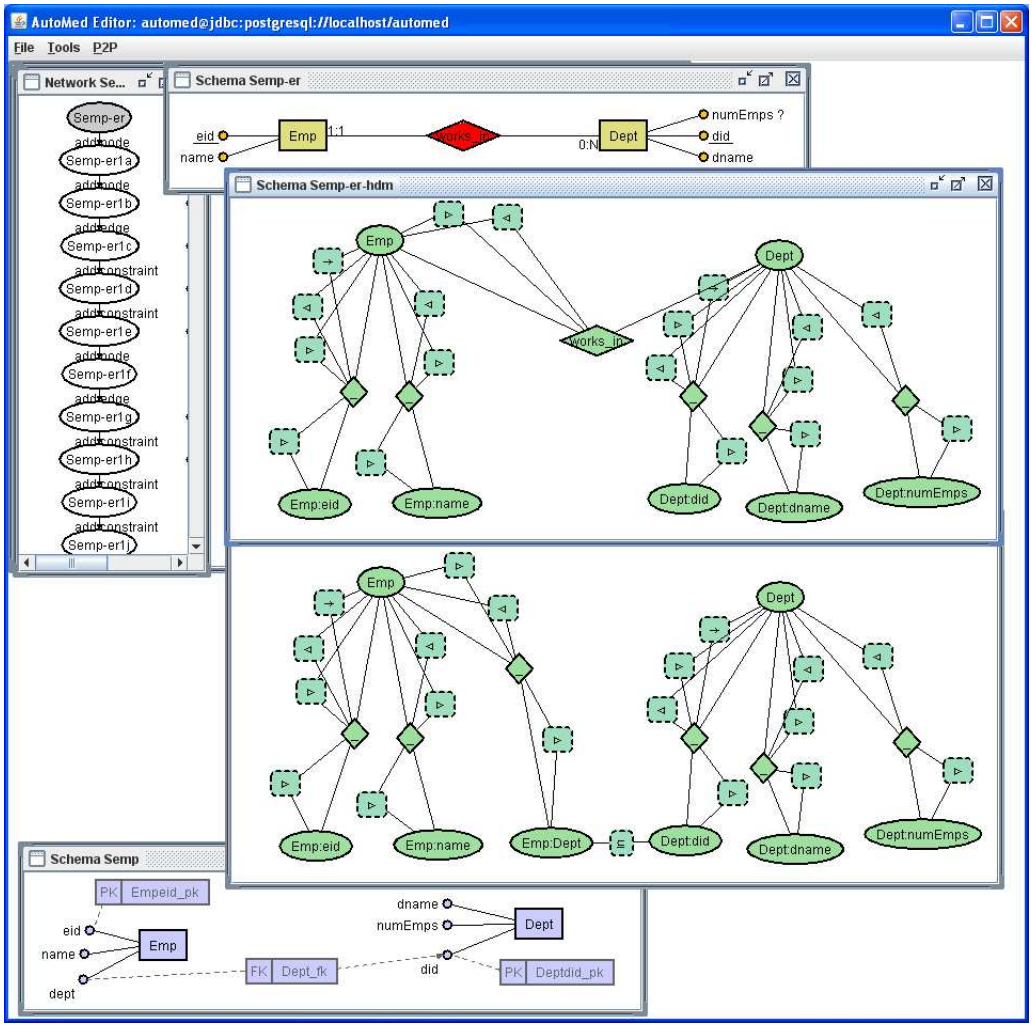


Figure 7.3: A screenshot from AutoMed of the translations of the ER schema, S_{emp-er} into SQL as well as part of the pathway (on the left hand side)

7.2.2 Stage B

In Step 2 of the script the DBA creates the view requested by the HR department using the following mapping:
$map_{S_{emp},S_{finEmp}} = (S_{emp}, S_{finEmp}, \Sigma_{S_{emp},S_{finEmp}})$ where $\Sigma_{S_{emp},S_{finEmp}} =$

$$\{\langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge \langle\langle \mathsf{Dept}\rangle\rangle(d,dn,ne) \wedge dn = \texttt{`Finance'} \rightarrow \langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \wedge \langle\langle \mathsf{FinDept}\rangle\rangle(d,dn),$$
$$\exists dname.\langle\langle \mathsf{FinEmp}\rangle\rangle(e,n) \wedge \langle\langle \mathsf{FinDept}\rangle\rangle(d,ne) \rightarrow \langle\langle \mathsf{Emp}\rangle\rangle(e,n,d) \wedge \langle\langle \mathsf{Dept}\rangle\rangle(d,dname(d),ne)\}$$

We read this into AutoMed as a string and our implementation of the TransGen operator then translates it into a BAV pathway as described in Section 6.7. The pathway is shown below and the instance of the resulting view derived from $\mathsf{Inst}_5(S_{emp})$ is shown in Figure 7.5.

Emp

eid	name	dept
1	Peter Smith	100
3	Paul Jones	100
5	Joe Brown	100
21	Susan Brown	101

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15
102	IT	

Emp.dept → Dept.did

Figure 7.4: $\mathsf{Inst}_5(S_{emp})$

The growth phase of the pathway is:

① add(table:⟨⟨FinEmp⟩⟩, [{e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
② add(column:⟨⟨FinEmp, eid, int, notnull⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
③ add(column:⟨⟨FinEmp, name, varchar, notnull⟩⟩,
[{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩; {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
④ add(table:⟨⟨FinDept⟩⟩, [{d} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
⑤ add(column:⟨⟨FinDept, did, int, notnull⟩⟩, [{d, d} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
⑥ add(column:⟨⟨FinDept, numEmps, int, null⟩⟩, [{d, ne} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, ne} ← column:⟨⟨Dept, numEmps⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩;
dn ← 'Finance'])
⑦ add(primary_key:⟨⟨FinEmpkey, ⟨⟨FinEmp⟩⟩, ⟨⟨FinEmp, eid⟩⟩⟩⟩)
⑧ add(primary_key:⟨⟨FinDeptkey, ⟨⟨FinDept⟩⟩, ⟨⟨FinDept, did⟩⟩⟩⟩)

The shrinking phase of this pathway is shown below. The primitive transformations are all contract because we only have the employees from the finance department in S_{finEmp} so we cannot fully recreate S_{emp} with these transformations.

⑨ delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
⑩ delete(primary_key:⟨⟨Deptdid_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
⑪ delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
⑫ contract(column:⟨⟨Dept, numEmps⟩⟩, Range column:⟨⟨FinDept, numEmps⟩⟩ Any)
⑬ contract(column:⟨⟨Dept, dname⟩⟩, Range [{d, dn} | {d, d} ← column:⟨⟨FinDept, did⟩⟩;
dn ← generateGID(S_{finEmp}, d, [d], 'dname')] Any)
⑭ contract(column:⟨⟨Dept, did⟩⟩, Range column:⟨⟨FinDept, did⟩⟩ Any)
⑮ contract(table:⟨⟨Dept⟩⟩, Range table:⟨⟨FinDept⟩⟩ Any)
⑯ contract(column:⟨⟨Emp, dept⟩⟩,
Range [{e, d} | {e, e} ← column:⟨⟨FinEmp, eid⟩⟩; {d, d} ← column:⟨⟨FinDept, did⟩⟩] Any)
⑰ contract(column:⟨⟨Emp, name⟩⟩, Range column:⟨⟨FinEmp, name⟩⟩ Any)
⑱ contract(column:⟨⟨Emp, eid⟩⟩, Range column:⟨⟨FinEmp, eid⟩⟩ Any)
⑲ contract(table:⟨⟨Emp⟩⟩, Range table:⟨⟨FinEmp⟩⟩ Any)

FinEmp

eid	name
1	Peter Smith
3	Paul Jones
5	Joe Brown

FinDept

did	numEmps?
100	23

Figure 7.5: $\mathsf{Inst}_5(S_{finEmp})$

The generateGID function in Transformation (13) creates Skolem values based on the department id to create unique values for the department name.

In Step 3

$$\langle S_{finEmp-xml}, map_{S_{finEmp},S_{finEmp-xml}} \rangle = \mathsf{ModelGen}(S_{finEmp}, XML)$$

we translate S_{finEmp} into XML, the DDL of choice for the HR department. The HDM schema objects created by translating S_{emp} into the HDM all match the HDM representation of XML schema constructs so all we need to do is execute the create_root_node CT to create a root node for the XML schema. A screenshot of the XML and SQL schemas as well as their HDM equivalents is shown in Figure 7.6.

The growth phase of resultant BAV pathway is shown below:

(20) add(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
(21) add(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, e} ← column:⟨⟨FinEmp, eid⟩⟩; $r \leftarrow$ &0])
(22) add(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, column:⟨⟨FinEmp, eid⟩⟩)
(23) add(simpleElement:⟨⟨root/FinEmp, name, 1, 1, int⟩⟩, column:⟨⟨FinEmp, name⟩⟩)
(24) add(complexElement:⟨⟨root, FinDept, 0, unbounded⟩⟩, [{r, d} | {d, d} ← column:⟨⟨FinDept, did⟩⟩; $r \leftarrow$ &0])
(25) add(simpleElement:⟨⟨root/FinDept, did, 1, 1, int⟩⟩, column:⟨⟨FinDept, did⟩⟩)
(26) add(simpleElement:⟨⟨root/FinDept, numEmps, 0, 1, int⟩⟩, column:⟨⟨Dept, numEmps⟩⟩)
(27) add(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)
(28) add(key:⟨⟨FinDeptkey, ⟨⟨root, FinDept⟩⟩, ⟨⟨root/FinDept, did⟩⟩⟩⟩)

complexElement:⟨⟨root, FinEmp⟩⟩ and complexElement:⟨⟨FinDept⟩⟩ both have key constructs associated with them so we do not need to create unique OIDs: we use the values from the keys as the unique identifying value in their extents. The shrinking phase of this pathway is:

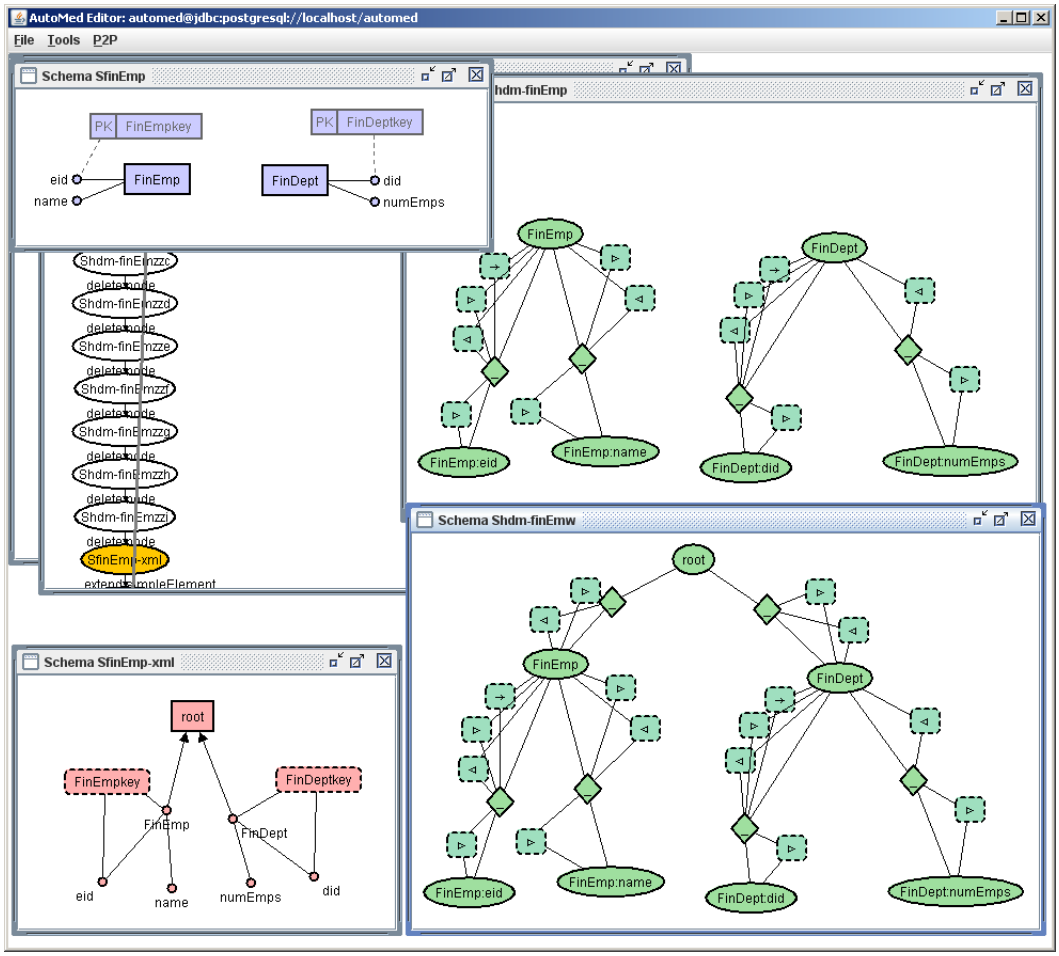


Figure 7.6: A screenshot of the translation from S_{finEmp} to $S_{finEmp-xml}$

(29) delete(primary_key:⟨⟨FinEmpkey, Emp, ⟨⟨FinEmp, eid⟩⟩⟩⟩)
(30) delete(primary_key:⟨⟨FinDeptkey, Dept, ⟨⟨FinDept, did⟩⟩⟩⟩)
(31) delete(column:⟨⟨FinDept, numEmps⟩⟩, simpleElement:⟨⟨root/FinDept, numEmps⟩⟩)
(32) delete(column:⟨⟨FinDept, did⟩⟩, simpleElement:⟨⟨root/FinDept, did⟩⟩)
(33) delete(table:⟨⟨FinDept⟩⟩,
$[\{d\} \mid \{d, d\} \leftarrow$ simpleElement:⟨⟨root/FinDept, did⟩⟩$]$)
(34) delete(column:⟨⟨FinEmp, name⟩⟩, simpleElement:⟨⟨root/FinEmp, name⟩⟩)
(35) delete(column:⟨⟨FinEmp, eid⟩⟩, simpleElement:⟨⟨root/FinEmp, eid⟩⟩)
(36) delete(table:⟨⟨FinEmp⟩⟩, $[\{e\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩$]$)

Transformations (1) to (36) give us a pathway from S_{emp} to $S_{finEmp-xml}$.

In Step 4

$$map_{S_{emp},S_{finEmp-xml}} = map_{S_{emp},S_{finEmp}} \circ map_{S_{finEmp},S_{finEmp-xml}}$$

we compose the mappings created in Steps 2 and 3 above to create a pathway from S_{emp} directly to $S_{finEmp-xml}$ without any S_{finEmp} objects in it. We use the

transformations from the growth phase of $p_{S_{finEmp},S_{finEmp-xml}}$ rewritten over objects in S_{emp} to get this result. For example Transformation (23) creates simpleElement:⟨⟨root/FinEmp, name⟩⟩ with an extent of column:⟨⟨FinEmp, name⟩⟩. In $p_{S_{emp},S_{finEmp}}$, column:⟨⟨FinEmp, name⟩⟩ is created by Transformation (3).
In $p_{S_{emp},S_{finEmp-xml}}$, created in this step, we thus add simpleElement:⟨⟨root/FinEmp, name⟩⟩ with the extent query from Transformation (3) to give us Transformation (40) below. The other objects are created in the same way.

(37) add(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
(38) add(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'; $r \leftarrow$ &0])
(39) add(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(40) add(simpleElement:⟨⟨root/FinEmp, name, 1, 1, string⟩⟩,
[{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩; {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(41) add(complexElement:⟨⟨root, FinDept, 0, unbounded⟩⟩, [{r, d} | {d, dn} ← column:⟨⟨Dept, dn⟩⟩;
$dn \leftarrow$ 'Finance'; $r \leftarrow$ &0])
(42) add(simpleElement:⟨⟨root/FinDept, did, 1, 1, int⟩⟩, [{d, d} | {d, dn} ← column:⟨⟨Dept, dn⟩⟩;
$dn \leftarrow$ 'Finance'])
(43) add(simpleElement:⟨⟨root/FinDept, numEmps, 1, 1, int⟩⟩, [{d, ne} | {d, dn} ← column:⟨⟨Dept, dn⟩⟩;
{d, ne} ← column:⟨⟨Dept, numEmps⟩⟩; $dn \leftarrow$ 'Finance'])
(44) add(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)
(45) add(key:⟨⟨FinDeptkey, ⟨⟨root, FinDept⟩⟩, ⟨⟨root/FinDept, did⟩⟩⟩⟩)

We create the transformations in the shrinking phase in a similar way to that described above for the growth phase. We use the queries in the shrinking phase of $p_{S_{emp},S_{finEmp}}$, *i.e.* Transformations (9) to (19) but with the queries rewritten over objects in $S_{finEmp-xml}$ to produce:

(46) delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
(47) delete(primary_key:⟨⟨Deptdid_pk, Emp, ⟨⟨Dept, did⟩⟩⟩⟩)
(48) delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(49) contract(column:⟨⟨Dept, numEmps⟩⟩, Range simpleElement:⟨⟨root/FinDept, numEmps⟩⟩ Any)
(50) contract(column:⟨⟨Dept, dname⟩⟩, Range [{d, dn} | {d, d} ← simpleElement:⟨⟨root/FinDept, did⟩⟩;
$dn \leftarrow$ generateGID(S_{finEmp}, d, [d], 'dname')] Any)
(51) contract(column:⟨⟨Dept, did⟩⟩, Range simpleElement:⟨⟨root/FinDept, did⟩⟩ Any)
(52) contract(table:⟨⟨Dept⟩⟩, Range complexElement:⟨⟨root, FinDept⟩⟩ Any)
(53) contract(column:⟨⟨Emp, dept⟩⟩,
Range [{e, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
{d, d} ← simpleElement:⟨⟨root/FinDept, did⟩⟩] Any)
(54) contract(column:⟨⟨Emp, name⟩⟩, Range simpleElement:⟨⟨root/FinEmp, name⟩⟩ Any)
(55) contract(column:⟨⟨Emp, eid⟩⟩, Range simpleElement:⟨⟨root/FinEmp, eid⟩⟩ Any)
(56) contract(table:⟨⟨Emp⟩⟩, Range complexElement:⟨⟨root, FinEmp⟩⟩ Any)

$p_{S_{emp},S_{finEmp}}$ and $p_{S_{finEmp},S_{finEmp-xml}}$ are no longer used in the script and can thus be removed.

In Step 5 the HR DBA realises that the FinDept branch of $S_{finEmp-xml}$ is redundant and so removes it from the schema. He also adds extra elements for date of birth and marital status to the `emp_type` complex type. The SO s-t tgds that describe this mapping are shown below and the resultant XML Schema is shown in Figure 7.7.

$$\begin{aligned}\{&\exists dob, isMarried.\langle\langle \text{root/FinEmp}\rangle\rangle(e,n) \wedge \langle\langle \text{root/FinDept}\rangle\rangle(d,ne) \rightarrow\\ &\quad \langle\langle \text{root/FinEmp}\rangle\rangle(e,n,dob(e),isMarried(e)),\\ &\exists did, dname.\langle\langle \text{root/FinEmp}\rangle\rangle(e,n,dob,im) \rightarrow\\ &\quad \langle\langle \text{root/FinEmp}\rangle\rangle(e,n) \wedge \langle\langle \text{root/FinDept}\rangle\rangle(did(e),dname(did(e)))\}\end{aligned}$$

The transformations in the growth phase are shown below:

(57) add(simpleElement:⟨⟨root/FinEmp, dob, 0, 1, string⟩⟩,
[{e, dob} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
dob ← generateGID($S_{finEmp-xml}$, e, [e], 'dob')])
(58) add(simpleElement:⟨⟨root/FinEmp, isMarried, 0, 1, boolean⟩⟩,
[{e, im} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
im ← generateGID($S_{finEmp-xml}$, e, [e], 'isMarried')])

The extents of simpleElement:⟨⟨root/FinEmp, dob⟩⟩ and simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩ are generated by the function calls generateGID($S_{finEmp-xml}$, e, $[e]$, 'dob') and generateGID($S_{finEmp-xml}$, e, $[e]$, 'isMarried') which create a unique identifying value based on the employee id for each date of birth and marital status. These can be replaced with the actual data values when the schema is materialised.

The shrinking phase is:

(59) delete(key:⟨⟨FinDeptkey, ⟨⟨root, FinDept⟩⟩, ⟨⟨root/FinDept, did⟩⟩⟩⟩)
(60) delete(simpleElement:⟨⟨root/FinDept, did⟩⟩,
[{d, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')])
(61) delete(simpleElement:⟨⟨root/FinDept, numEmps⟩⟩,
[{d, ne} | {e, d} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did');
ne ← generateGID($S'_{finEmp-xml}$, d, [d], 'numEmps')])
(62) delete(complexElement:⟨⟨root, FinDept⟩⟩,
[{r, d} | {e, d} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did'); r ← &0])

In Step 6

$$map_{S_{emp},S'_{finEmp-xml}} = map_{S_{emp},S_{finEmp-xml}} \circ map_{S_{finEmp-xml},S'_{finEmp-xml}}$$

```
<xsd:complexType name = "emp_type">
 <xsd:all>
  <xsd:element name = "eid" type = "xsd:int"/>
  <xsd:element name = "name" type = "xsd:string" />
  <xsd:element name = "dob" type = "xsd:string" minOccurs = "0"/>
  <xsd:element name = "isMarried" type = "xsd:boolean" minOccurs = "0"/>
 </xsd:all>
</xsd:complexType>
<xsd:element name = "root">
 <xsd:complexType>
  <xsd:all>
    <xsd:element name = "FinEmp" type = "emp_type"
            minOccurs = "0" maxOccurs = "unbounded" />
  </xsd:all>
 </xsd:complexType>
 <xsd:key name = "FinEmpkey">
  <xsd:selector xpath =  "root/FinEmp" />
  <xsd:field xpath =  "eid" />
 </xsd:key>
</xsd:element>
```

Figure 7.7: $S'_{finEmp-xml}$

```
<root>
 <FinEmp eid = "1">
  <name>Peter Smith</name>
  <dob>20/05/1980</dob>
  <isMarried>true</isMarried>
 </FinEmp>
 <FinEmp eid = "3">
  <name>Paul Jones</name>
  <dob>12/02/1972</dob>
  <isMarried>true</isMarried>
 </FinEmp>
 <FinEmp eid = "5">
  <name>Joe Brown</name>
  <dob>03/06/1975</dob>
  <isMarried>false</isMarried>
 </FinEmp>
</root>
```

Figure 7.8: $\mathsf{Inst}_5(S'_{finEmp-xml})$

we create a pathway directly from the original database to the updated HR schema by composing the pathways created in Steps 4 and 5. We create the transformations in the same way as we did in Step 4 by rewriting the queries in the growth phase transformations in $p_{S_{finEmp-xml},S'_{finEmp-xml}}$ over objects in S_{emp}. For those objects in $S'_{finEmp-xml}$ that do not have a transformation in $p_{S_{finEmp-xml},S'_{finEmp-xml}}$ we use the transformation that added them in $p_{S_{emp},S_{finEmp-xml}}$. Consider simpleElement:⟨⟨root/FinEmp, dob⟩⟩ added by Transformation (57) in $p_{S_{finEmp-xml},S'_{finEmp-xml}}$. The $S_{finEmp-xml}$ object in the transformation query, simpleElement:⟨⟨root/FinEmp, eid⟩⟩, is added to $S_{finEmp-xml}$ by Transformation (39) whose query is:

$$[\{e,e\} \mid \{e,d\} \leftarrow \text{column:}\langle\langle \text{Emp, dept}\rangle\rangle; \{d,dn\} \leftarrow \text{column:}\langle\langle \text{Dept, dname}\rangle\rangle; dn \leftarrow \texttt{'Finance'}]$$

We replace simpleElement:⟨⟨root/FinEmp, eid⟩⟩ in Transformation (57) with this query to give us Transformation (67) below. We create a new Skolem function for the values of the object based on the rewritten query.

(63) add(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
(64) add(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'; $r \leftarrow$ &0])
(65) add(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(66) add(simpleElement:⟨⟨root/FinEmp, name, 1, 1, string⟩⟩, [{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩;
{e, d} ← column:⟨⟨Emp, dept⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(67) add(simpleElement:⟨⟨root/FinEmp, dob, 0, 1, string⟩⟩, [{e, dob} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance';
$dob \leftarrow$ generateGID($S_{emp}, e, [e]$, 'dob')])
(68) add(simpleElement:⟨⟨root/FinEmp, isMarried, 0, 1, boolean⟩⟩,
[{e, im} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance';
$im \leftarrow$ generateGID(S_{emp}, e, [e], 'isMarried')])
(69) add(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)

The shrinking phase is as follows:

(70) delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
(71) delete(primary_key:⟨⟨Deptdid_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(72) delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(73) contract(column:⟨⟨Dept, numEmps⟩⟩,
Range [{d, ne} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did');
ne ← generateGID($S'_{finEmp-xml}$, d, [d], 'numEmps')] Any)
(74) contract(column:⟨⟨Dept, dname⟩⟩,
Range [{d, dn} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did');
dn ← generateGID($S'_{finEmp-xml}$, d, [d], 'dname')] Any)
(75) contract(column:⟨⟨Dept, did⟩⟩,
Range [{d, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(76) contract(table:⟨⟨Dept⟩⟩,
Range [{d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(77) contract(column:⟨⟨Emp, dept⟩⟩,
Range [{e, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(78) contract(column:⟨⟨Emp, name⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, name⟩⟩ Any)
(79) contract(column:⟨⟨Emp, eid⟩⟩, Range simpleElement:⟨⟨root/FinEmp, eid⟩⟩ Any)
(80) contract(table:⟨⟨Emp⟩⟩, Range [{e} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩] Any)

Transformations (63) to (80) constitute the output pathway of Stage B. Transformations (37) to (62) can now be removed.

7.2.3 Stage C

We now merge the additional objects added to the original HR schema, $S_{finEmp-xml}$, by the HR department into the database, S_{emp}.

In Step 7

$$\langle S_{d-xml}, map_{S'_{finEmp-xml},S_{d-xml}} \rangle = \mathsf{Diff}(S'_{finEmp-xml}, \mathsf{Invert}(map_{S_{emp},S'_{finEmp-xml}}))$$

we use our implementation of the Diff operator to get the new information added to $S'_{finEmp-xml}$, *i.e.* the portion of $S'_{finEmp-xml}$ that is not fully derived from S_{emp}. We use the inverse of the pathway created above as the input pathway.

To decide which objects should appear in S_{d-xml} we look at the add phase of the

input pathway, *i.e.* Transformations (70) to (80). We see that the $S'_{finEmp-xml}$ elements not used in the queries in these transformations are
simpleElement:⟨⟨root/FinEmp, dob⟩⟩, simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩ and
complexElement:⟨⟨root, FinEmp⟩⟩ so we need to add these to S_{d-xml}.
simpleElement:⟨⟨FinEmp, eid⟩⟩ is transitively referenced by
simpleElement:⟨⟨root/FinEmp, dob⟩⟩ via complexElement:⟨⟨root, FinEmp⟩⟩ and
key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩ so we need to add it too. We also see that there are no filters or joins on any of the queries involving objects we need to add so all the instances of objects will be in S_{d-xml}. Here we can take advantage of the schema transformation technique that means we do not need a transformation for the objects in our target schema that are not changed. We therefore only need a single transformation in $map_{S'_{finEmp-xml}, S_{d-xml}}$ that removes simpleElement:⟨⟨root/FinEmp, name⟩⟩.

(81) delete(simpleElement:⟨⟨root/FinEmp, name⟩⟩,
[{e, n} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
n ← generateGID(S_{d-xml}, e, [e], 'name')])

In Step 8

$$\langle S_{diff}, map_{S_{d-xml}, S_{diff}} \rangle = \mathsf{ModelGen}(S_{d-xml}, \mathtt{SQL})$$

we translate S_{d-xml} into SQL using ModelGen, to create S_{diff}. We need to use a single CT, inclusion_expand to split the root node from the complex element. A screenshot of the translation is shown Figure 7.9 and the resultant schema is shown in Figure 7.10.

The pathway is as follows:

(82) add(table:⟨⟨root_FinEmp⟩⟩, [{e} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩])
(83) add(column:⟨⟨root_FinEmp, eid, int, notnull⟩⟩, simpleElement:⟨⟨root/FinEmp, eid⟩⟩)
(84) add(column:⟨⟨root_FinEmp, dob, varchar, null⟩⟩, simpleElement:⟨⟨root/FinEmp, dob⟩⟩)
(85) add(column:⟨⟨root_FinEmp, isMarried, bool, null⟩⟩, simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩)
(86) add(column:⟨⟨root_FinEmp, root⟩⟩, [{e, r} | {r, e} ← complexElement:⟨⟨root, FinEmp⟩⟩])
(87) add(primary_key:⟨⟨root_FinEmpeid_pk, root_FinEmp, ⟨⟨root_FinEmp, eid⟩⟩⟩⟩)
(88) add(table:⟨⟨root⟩⟩, complexElement:⟨⟨null, root, 1, 1⟩⟩)
(89) add(column:⟨⟨root, root, varchar, notnull⟩⟩, [{r, r} | {r} ← complexElement:⟨⟨null, root, 1, 1⟩⟩])
(90) add(primary_key:⟨⟨rootroot_pk, root, ⟨⟨root, root⟩⟩⟩⟩)
(91) add(foreign_key:⟨⟨root_fk, root_FinEmp, ⟨⟨root_FinEmp, root⟩⟩, root, ⟨⟨root, root⟩⟩⟩⟩)

The shrinking phase is:

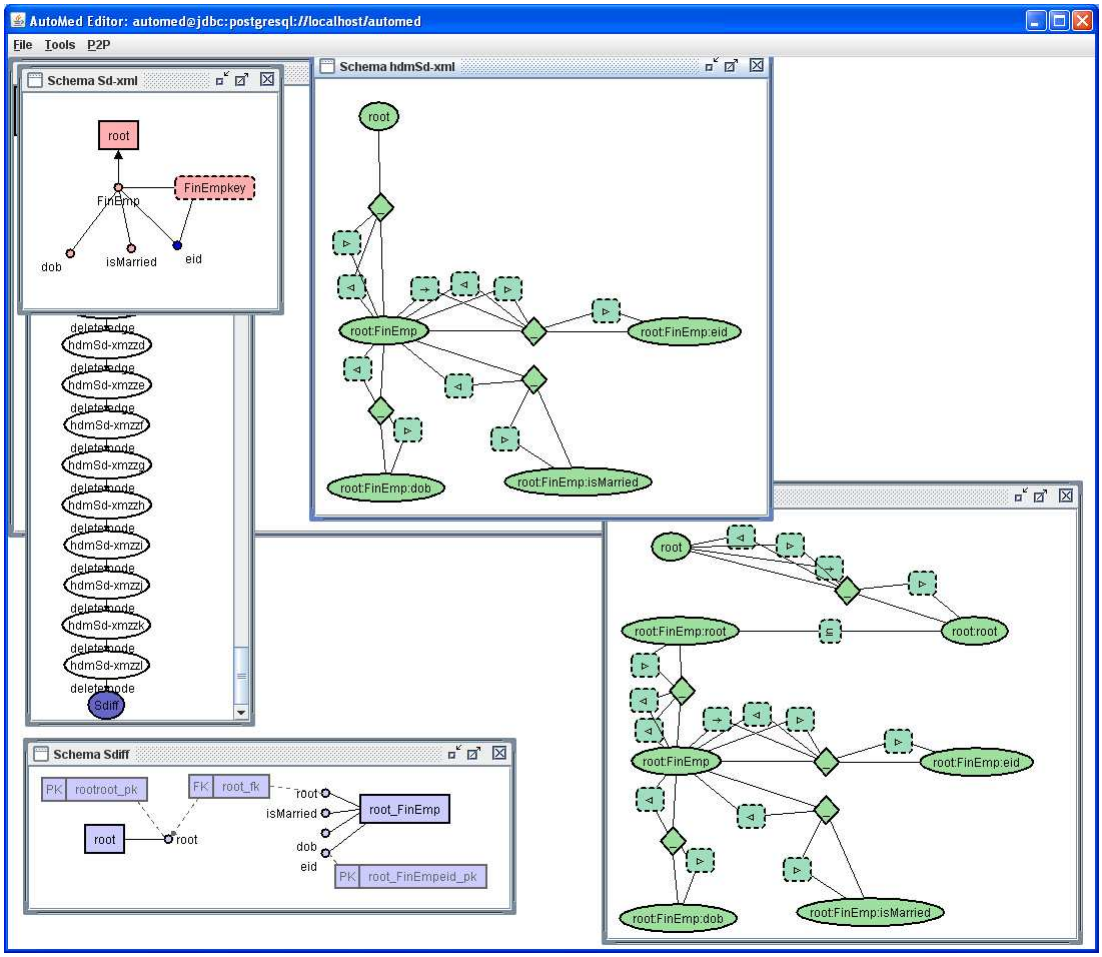


Figure 7.9: A screenshot of the translation from S_{d-xml} to S_{diff}

(92) delete(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)
(93) delete(simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩, column:⟨⟨root_FinEmp, isMarried⟩⟩)
(94) delete(simpleElement:⟨⟨root/FinEmp, dob⟩⟩, column:⟨⟨root_FinEmp, dob⟩⟩)
(95) delete(simpleElement:⟨⟨root/FinEmp, eid⟩⟩, column:⟨⟨root_Emp, eid⟩⟩)
(96) delete(complexElement:⟨⟨root, FinEmp⟩⟩, [{r, e} | {e, r} ← column:⟨⟨root_Emp, root⟩⟩])
(97) add(complexElement:⟨⟨null, root, 1, 1⟩⟩, table:⟨⟨root⟩⟩)

In Step 9

$$map_{S'_{finEmp-xml},S_{diff}} = map_{S'_{finEmp-xml},S_{d-xml}} \circ map_{S_{d-xml},S_{diff}}$$

we compose the pathways created in Steps 7 and 8 to create a pathway directly from $S'_{finEmp-xml}$ to S_{diff}. In the same way as we have done before we rewrite the queries from the growth phase of $p_{S_{d-xml},S_{diff}}$ over objects in $S'_{finEmp-xml}$ to give us:

root_FinEmp

eid	dob?	isMarried?	root
1	20/05/1980	true	&0
3	12/02/1972	true	&0
5	03/06/1975	false	&0

root_FinEmp.root → root.root

root

root
&0

Figure 7.10: $\mathsf{Inst}_5(S_{diff})$

(98) add(table:⟨⟨root_FinEmp⟩⟩, [{e} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩])
(99) add(column:⟨⟨root_FinEmp, eid⟩⟩,
simpleElement:⟨⟨root/FinEmp, eid⟩⟩)
(100) add(column:⟨⟨root_FinEmp, dob⟩⟩,
simpleElement:⟨⟨root/FinEmp, dob⟩⟩)
(101) add(column:⟨⟨root_FinEmp, isMarried⟩⟩,
simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩)
(102) add(column:⟨⟨root_FinEmp, root⟩⟩, [{e, r} | {r, e} ← complexElement:⟨⟨root, FinEmp⟩⟩])
(103) add(primary_key:⟨⟨root_FinEmpeid_pk, root_FinEmp, ⟨⟨root_FinEmp, eid⟩⟩⟩⟩)
(104) add(table:⟨⟨root⟩⟩, [{r} | {r} ← complexElement:⟨⟨null, root, 1, 1⟩⟩])
(105) add(column:⟨⟨root, root⟩⟩, [{r, r} | {r} ← complexElement:⟨⟨null, root, 1, 1⟩⟩])
(106) add(primary_key:⟨⟨rootroot_pk, root, ⟨⟨root, root⟩⟩⟩⟩)

The shrinking phase is:

(107) delete(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)
(108) delete(simpleElement:⟨⟨root/FinEmp, isMarried, 0, 1, boolean⟩⟩,
column:⟨⟨root_FinEmp, isMarried⟩⟩)
(109) delete(simpleElement:⟨⟨root/FinEmp, dob, 0, 1, string⟩⟩,
column:⟨⟨root_FinEmp, dob⟩⟩)
(110) delete(simpleElement:⟨⟨root/FinEmp, name, 1, 1, string⟩⟩, [{e, n} | {e, e} ← column:⟨⟨root_Emp, eid⟩⟩;
n ← generateGID(S_{diff}, e, [e], 'name')])
(111) delete(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, column:⟨⟨root_Emp, eid⟩⟩)
(112) delete(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, e} ← column:⟨⟨root_Emp, eid⟩⟩;
r ← &0])
(113) delete(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | r ← &0])

$p_{S'_{finEmp-xml},S_{d-xml}}$ and $p_{S_{d-xml},S_{diff}}$ are not used again in the script and so can be removed.

In Step 10

$$map_{S_{emp},S_{diff}} = map_{S_{emp},S'_{finEmp-xml}} \circ map_{S'_{finEmp-xml},S_{diff}}$$

we compose the results of Steps 6 and 9. The resultant pathway includes the transformations in the growth phase of $map_{S'_{finEmp-xml},S_{diff}}$ with the queries rewritten over S_{emp} objects to give us the following transformations:

(114) add(table:⟨⟨root_FinEmp⟩⟩, [{e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(115) add(column:⟨⟨root_FinEmp, eid, int, notnull⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(116) add(column:⟨⟨root_FinEmp, dob, varchar, null⟩⟩, [{e, dob} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance';
dob ← generateGID(S_{emp}, e, [e], 'dob')])
(117) add(column:⟨⟨root_FinEmp, isMarried, bool, null⟩⟩, [{e, im} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance';
im ← generateGID(S_{emp}, e, [e], 'isMarried')])
(118) add(column:⟨⟨root_FinEmp, root, varchar, notnull⟩⟩, [{e, r} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'; r ← &0])
(119) add(primary_key:⟨⟨root_FinEmpeid_pk, root_FinEmp, ⟨⟨root_FinEmp, eid⟩⟩⟩⟩)
(120) add(table:⟨⟨root⟩⟩, distinct[{r} | r ← &0])
(121) add(column:⟨⟨root, root, varchar, notnull⟩⟩, distinct[{r, r} | r ← &0])
(122) add(primary_key:⟨⟨rootroot_pk, root, ⟨⟨root, root⟩⟩⟩⟩)
(123) add(foreign_key:⟨⟨root_fk, root_FinEmp, ⟨⟨root_FinEmp, root⟩⟩, root, ⟨⟨root, root⟩⟩⟩⟩)

The transformations in the shrinking phase are those used to remove the S_{emp} rewritten over S_{diff} objects.

The transformations for non-constraint objects in the shrinking phase are all contracts because the transformations in $p_{S_{emp},S'_{finEmp-xml}}$ are all contracts.

(124) delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
(125) delete(primary_key:⟨⟨Deptdid_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(126) delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(127) contract(column:⟨⟨Dept, numEmps⟩⟩,
Range [{d, ne} | {e, e} ← column:⟨⟨root_FinEmp, eid⟩⟩;
d ← generateGID(S_{diff}, e, [e], 'did'); ne ← generateGID(S_{diff}, d, [d], 'numEmps')] Any)
(128) contract(column:⟨⟨Dept, dname⟩⟩,
Range [{d, dn} | {e, e} ← column:⟨⟨root_FinEmp, eid⟩⟩;
d ← generateGID(S_{diff}, e, [e], 'did'); dn ← generateGID(S_{diff}, d, [d], 'dname')] Any)
(129) contract(column:⟨⟨Dept, did⟩⟩,
Range [{d, d} | {e, e} ← column:⟨⟨root_FinEmp, eid⟩⟩;
d ← generateGID(S_{diff}, e, [e], 'did')] Any)
(130) contract(table:⟨⟨Dept⟩⟩,
Range [{d} | {e, e} ← column:⟨⟨root_FinEmp, eid⟩⟩;
d ← generateGID(S_{diff}, e, [e], 'did')] Any)
(131) contract(column:⟨⟨Emp, dept⟩⟩,
Range [{e, d} | {e, e} ← column:⟨⟨root_FinEmp, eid⟩⟩;
d ← generateGID(S_{diff}, e, [e], 'did')] Any)
(132) contract(column:⟨⟨Emp, name⟩⟩, Range column:⟨⟨root_FinEmp, name⟩⟩ Any)
(133) contract(column:⟨⟨Emp, eid⟩⟩, Range column:⟨⟨root_FinEmp, eid⟩⟩ Any)
(134) contract(table:⟨⟨Emp⟩⟩, Range table:⟨⟨root_FinEmp⟩⟩ Any)

Emp

eid	name	dept
1	Peter Smith	100
3	Paul Jones	100
5	Joe Brown	100
21	Susan Brown	101

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15
102	IT	

root_FinEmp

eid	dob?	isMarried?	root
1	20/05/1980	true	&0
3	12/02/1972	true	&0
5	03/06/1975	false	&0

root

root
&0

Emp.dept → Dept.did root_FinEmp.root → root.root

Figure 7.11: $\mathsf{Inst}_5(S_{merge})$

We cannot remove either of the input pathways after this step because they are both used in Step 12 later on in the script.

In Step 11

$$\langle S_m, map_{S_{merge},S_{emp}}, map_{S_{merge},S_{diff}} \rangle = \mathsf{Merge}(S_{emp}, S_{diff}, map_{S_{emp},S_{diff}})$$

we merge S_{emp} with S_{diff} to create a schema that includes all the objects from S_{emp} and those added by the HR department in Step 5.

We are unable to optimise the schema obtained at the top of the above pathway because there are no objects in S_{diff} whose instances are subsets of something in S_{emp} and that are not referenced by some other object in S_{diff}. Our merged schema is thus the one shown in Figure 7.11 and includes all the objects and instances from both S_{emp} and S_{diff}. As we can see, the only repeated values are the employee ids which are referenced by the key constraint on table:⟨⟨Emp⟩⟩ and table:⟨⟨FinEmp⟩⟩ and so cannot be removed.

We do not create any new transformations in this step. $p_{S_{merge},S_{diff}}$ is made up of Transformations (124) to (134) and $p_{S_{merge},S_{emp}}$ is the inverse of Transformations (114) to (123).

Step 12

$$map_{S_{merge},S'_{finEmp}} = (map_{S_{merge},S_{emp}} \circ map_{S_{emp},S'_{finEmp-xml}}) \oplus (map_{S_{merge},S_{diff}} \circ \mathsf{Invert}(map_{S'_{finEmp-xml},S_{diff}}))$$

uses the Confluence operator to combine the two pathways that exist between S_{merge} and $S'_{finEmp-xml}$, one via the result of Steps 11 and 6 and the other via the results

of Steps 11 and 9.

We need to perform two Compose operations before we can do the Confluence. In $map_{S_{merge},S_{emp}} \circ map_{S_{emp},S'_{finEmp-xml}}$ there is no growth phase in $p_{S_{merge},S_{emp}}$ so the resultant transformations are simply copies of those used to add $S'_{finEmp-xml}$ objects in $p_{S_{emp},S'_{finEmp-xml}}$, *i.e.* Transformations (63) to (69). The shrinking phase is made up of the shrinking phase transformations of $p_{S_{merge},S_{emp}}$ rewritten over objects in $S'_{finEmp-xml}$.

The resultant pathway uses a generateGID function in the transformations to add simpleElement:⟨⟨root/FinEmp, dob⟩⟩ and simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩.

In $map_{S_{merge},S_{diff}} \circ map_{S_{diff},S'_{finEmp-xml}}$, again there is no growth phase in $p_{S_{merge},S_{diff}}$. We thus take the transformations for the composition from the growth phase of $p_{S_{diff},S'_{finEmp-xml}}$.

In this case the transformation queries for the transformations that add simpleElement:⟨⟨root/FinEmp, dob⟩⟩ and simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩ do not have generateGID functions but there is one in the query for the transformation that adds simpleElement:⟨⟨root/FinEmp, name⟩⟩.

When we execute Confluence we replace the generateGIDs with the generators of the equivalent transformation in the other pathway. This gives a pathway between S_{merge} to $S'_{finEmp-xml}$ where none of the generators are generateGID functions, *i.e.* the whole of $S'_{finEmp-xml}$ can be derived from S_{merge} using $map_{S_{merge},S'_{finEmp-xml}}$

Due to its length the result pathway has been put into the appendix in Section A.1. This forms the output pathway for Stage C. We can now remove the pathways created in Steps 6,9 and 11 as these are no longer required.

7.2.4 Stage D

To comply with a new company policy of not storing the dates of birth of employees, a new version of the schema, S_{newEmp}, with the date of birth removed is created. The DBA takes this opportunity to add the isMarried column to the Emp table and remove the redundant table:⟨⟨root⟩⟩. The redesign is Step 13 in the script. The SO s-t tgds used in the mapping are shown below and the new schema is shown in

Emp

eid	name	dept	isMarried?
1	Peter Smith	100	true
3	Paul Jones	100	true
5	Joe Brown	100	false
21	Susan Brown	101	

Dept

did	dname	numEmps?
100	Finance	23
101	HR	15
102	IT	

Emp.dept → Dept.did

Figure 7.12: $\mathsf{Inst}_5(S_{newEmp})$

Figure 7.12.

$$\langle\langle \mathsf{Emp} \rangle\rangle(e,n,d) \wedge \langle\langle \mathsf{Dept} \rangle\rangle(d,dn,ne) \wedge \langle\langle \mathsf{root\_FinEmp} \rangle\rangle(e,dob,im,r) \wedge \langle\langle \mathsf{root} \rangle\rangle(r) \rightarrow$$
$$\langle\langle \mathsf{Emp} \rangle\rangle(e,n,d,im) \wedge \langle\langle \mathsf{Dept} \rangle\rangle(d,dn,ne),$$
$$\exists dob.\langle\langle \mathsf{Emp} \rangle\rangle(e,n,d,im) \wedge \langle\langle \mathsf{Dept} \rangle\rangle(d,dn,ne) \rightarrow$$
$$\langle\langle \mathsf{Emp} \rangle\rangle(e,n,d) \wedge \langle\langle \mathsf{Dept} \rangle\rangle(d,dn,ne) \wedge \langle\langle \mathsf{root\_FinEmp} \rangle\rangle(e,dob(e),im,r) \wedge \langle\langle \mathsf{root} \rangle\rangle(r) \wedge r = \&0$$

The only transformation we need in the growth phase of this pathway is the one to add $\mathsf{column}{:}\langle\langle \mathsf{Emp}, \mathsf{isMarried} \rangle\rangle$.

(135) add(column:⟨⟨Emp, isMarried, bool, null⟩⟩, column:⟨⟨root_FinEmp, isMarried⟩⟩)

The shrinking phase removes $\mathsf{table}{:}\langle\langle \mathsf{root\_FinEmp} \rangle\rangle$ and $\mathsf{table}{:}\langle\langle \mathsf{root} \rangle\rangle$.

(136) delete(foreign_key:⟨⟨root_fk, root_FinEmp, ⟨⟨root_FinEmp, root⟩⟩, root, ⟨⟨root, root⟩⟩⟩⟩)
(137) delete(primary_key:⟨⟨root_FinEmpeid_pk, root_FinEmp, ⟨⟨root_FinEmp, eid⟩⟩⟩⟩)
(138) delete(column:⟨⟨root_FinEmp, root⟩⟩, [{e, r} | {e, e} ← column:⟨⟨Emp, eid⟩⟩; r ← &0])
(139) delete(column:⟨⟨root_FinEmp, isMarried⟩⟩, column:⟨⟨Emp, isMarried⟩⟩)
(140) delete(column:⟨⟨root_FinEmp, dob⟩⟩, [{e, dob} | {e, e} ← column:⟨⟨Emp, eid⟩⟩;
dob ← generateGID($S'_{newFinEmp-xml}$, e, [e], 'dob')])
(141) delete(column:⟨⟨root_FinEmp, eid⟩⟩, column:⟨⟨Emp, eid⟩⟩)
(142) delete(table:⟨⟨root_FinEmp⟩⟩, table:⟨⟨Emp⟩⟩)
(143) delete(primary_key:⟨⟨rootroot_pk, root, ⟨⟨root, eid⟩⟩⟩⟩)
(144) delete(column:⟨⟨root, root⟩⟩, [{r, r} | r ← &0])
(145) delete(table:⟨⟨root⟩⟩, [{r} | r ← &0])

In Step 14

$$map_{S'_{finEmp-xml},S_{newEmp}} = \mathsf{Invert}(map_{S_{merge},S'_{finEmp-xml}}) \circ map_{S_{merge},S_{newEmp}}$$

we compose the results of Steps 12 and 13. The only transformation in the growth phase of $p_{S_{merge},S_{newEmp}}$ is Transformation (135). We rewrite the query over $S'_{finEmp-xml}$ objects. The other transformations are from $\mathsf{Invert}(p_{S_{merge},S'_{finEmp-xml}})$, *i.e.* the shrink phase of the confluence mapping.

The transformation primitives for all the non-constraint objects in

Invert($p_{S_{merge},S'_{finEmp-xml}}$) are all extends so we use this primitive in our compose pathway. The growth phase of $p_{S'_{finEmp-xml},S_{newEmp}}$ is thus:

(146) extend(table:⟨⟨Emp⟩⟩, Range [{e} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩] Any)
(147) extend(column:⟨⟨Emp, eid, int, notnull⟩⟩,
Range simpleElement:⟨⟨FinEmp, eid⟩⟩ Any)
(148) extend(column:⟨⟨Emp, name, varchar, notnull⟩⟩,
Range simpleElement:⟨⟨FinEmp, name⟩⟩ Any)
(149) extend(column:⟨⟨Emp, isMarried, bool, null⟩⟩,
Range simpleElement:⟨⟨FinEmp, isMarried⟩⟩ Any)
(150) extend(column:⟨⟨Emp, dept, int, notnull⟩⟩,
Range [{e, d} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(151) extend(table:⟨⟨Dept⟩⟩, Range [{d} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(152) extend(column:⟨⟨Dept, did, int, notnull⟩⟩,
Range [{d, d} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did')] Any)
(153) extend(column:⟨⟨Dept, dname, varchar, notnull⟩⟩,
Range [{d, dn} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'd');
dn ← generateGID($S'_{finEmp-xml}$, d, [d], 'dname')] Any)
(154) extend(column:⟨⟨Dept, numEmps, int, null⟩⟩,
Range [{d, ne} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
d ← generateGID($S'_{finEmp-xml}$, e, [e], 'did');
ne ← generateGID($S'_{finEmp-xml}$, d, [d], 'numEmps')] Any)
(155) add(primary_key:⟨⟨Emp_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
(156) add(primary_key:⟨⟨Dept_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(157) add(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)

The shrinking phase is made up of copies of the transformations in the shrinking phase of Invert($p_{S_{merge},S'_{finEmp-xml}}$) rewritten over objects in S_{newEmp}. There are a number of objects in S_{merge} that are not in S_{newEmp} so we do not need all the transformations from the shrinking phase of Invert($p_{S_{merge},S'_{finEmp-xml}}$). The transformations we need are:

(158) delete(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)
(159) delete(simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩,
[{e, im} | {e, d} ← column:⟨⟨Emp, dept⟩⟩; {e, im} ← column:⟨⟨root_FinEmp, isMarried⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(160) delete(simpleElement:⟨⟨root/FinEmp, dob⟩⟩, [{e, dob} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{e, dob} ← column:⟨⟨root_FinEmp, dob⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩;
dn ← 'Finance'])
(161) delete(simpleElement:⟨⟨root/FinEmp, name⟩⟩, [{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩;
{e, d} ← column:⟨⟨Emp, dept⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(162) delete(complexElement:⟨⟨root, FinEmp⟩⟩, [{r, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩; r ← &0
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(163) delete(simpleElement:⟨⟨root/FinEmp, eid⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; dn ← 'Finance'])
(164) delete(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | r ← &0])

In Step 15

$$\langle S'_{newFinEmp-xml}, map_{S'_{finEmp-xml}, S'_{newFinEmp-xml}} \rangle = \mathsf{Extract}(S'_{finEmp-xml}, map_{S'_{finEmp-xml}, S_{newEmp}})$$

we use Extract to create a new version of $S'_{finEmp-xml}$ that only has those objects that are in S_{newEmp} and thus correctly reflects the structure of the new database. Looking at the growth phase of $p_{S'_{finEmp-xml}, S_{newEmp}}$ we see that the only object in $S'_{finEmp-xml}$ not used in a query is simpleElement:⟨⟨root/FinEmp, dob⟩⟩. It is also not referenced by any objects we do need to include in the extract schema so we can leave it out of $S'_{newFinEmp-xml}$. None of the queries in the pathway above that involve the objects we are to add to the result schema have filters in them so we use all the instances of the objects we need. This means we do not need an add phase in the result pathway. We create the extract schema by simply removing the object we do not want, in this case simpleElement:⟨⟨root/FinEmp, dob⟩⟩. The resultant schema is shown in Figure 7.13 and the pathway is:

(165) delete(simpleElement:⟨⟨root/FinEmp, dob⟩⟩, [{e, dob} | {e, e} ← simpleElement:⟨⟨FinEmp, eid⟩⟩;
dob ← generateGID($S'_{newfinEmp-xml}$, e, [e], 'dob')])

Finally, in Step 16

$$map_{S_{newEmp}, S'_{newFinEmp-xml}} = \mathsf{Invert}(map_{S'_{finEmp-xml}, S_{newEmp}}) \circ map_{S'_{finEmp-xml}, S'_{newFinEmp-xml}}$$

we create a pathway between the new SQL database and the updated XML schema by composing the pathways created in steps 14 and 15. The final pathway in the script is:

```
<xsd:complexType name = "emp_type">
 <xsd:all>
  <xsd:element name = "name" type = "xsd:string" />
  <xsd:element name = "isMarried" type = "xsd:boolean" />
 </xsd:all>
 <xsd:attribute name = "eid" type = "xsd:int" use = "required"/>
</xsd:complexType>
<xsd:element name = "root">
 <xsd:complexType>
  <xsd:all>
    <xsd:element name = "FinEmp" type = "emp_type"
            minOccurs = "0" maxOccurs = "unbounded" />
  </xsd:all>
 </xsd:complexType>
 <xsd:key name = "FinEmpKey">
  <xsd:selector xpath =  "root/FinEmp" />
  <xsd:field xpath =  "@eid" />
 </xsd:key>
</xsd:element>
```

Figure 7.13: $S'_{newFinEmp-xml}$

(166) add(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
(167) add(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'; $r \leftarrow$ &0])
(168) add(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(169) add(simpleElement:⟨⟨root/FinEmp, name, 1, 1, string⟩⟩, [{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩;
{e, d} ← column:⟨⟨Emp, dept⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(170) add(simpleElement:⟨⟨root/FinEmp, isMarried, 0, 1, boolean⟩⟩,
[{e, im} | {e, im} ← column:⟨⟨Emp, isMarried⟩⟩; {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
(171) add(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)

The shrinking phase is as follows:

(172) delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
(173) delete(primary_key:⟨⟨Deptdid_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(174) delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
(175) contract(column:⟨⟨Dept, numEmps⟩⟩,
Range [{d, ne} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{newFinEmp-xml}$, e, [e], 'did');
$ne \leftarrow$ generateGID($S'_{newFinEmp-xml}$, d, [d], 'numEmps')] Any)
(176) contract(column:⟨⟨Dept, dname⟩⟩,
Range [{d, dn} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{newFinEmp-xml}$, e, [e], 'did');
$dn \leftarrow$ generateGID($S'_{newFinEmp-xml}$, d, [d], 'dname')] Any)

(177) contract(column:⟨⟨Dept, did⟩⟩,
Range [{d, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{newFinEmp-xml}$, e, [e], 'did')] Any)
(178) contract(table:⟨⟨Dept⟩⟩,
Range [{d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{newFinEmp-xml}$, e, [e], 'did')] Any)
(179) contract(column:⟨⟨Emp, dept⟩⟩,
Range [{e, d} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
d ← generateGID($S'_{newFinEmp-xml}$, e, [e], 'did')] Any)
(180) contract(column:⟨⟨Emp, isMarried⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩ Any)
(181) contract(column:⟨⟨Emp, name⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, name⟩⟩ Any)
(182) contract(column:⟨⟨Emp, eid⟩⟩, Range simpleElement:⟨⟨root/FinEmp, eid⟩⟩ Any)
(183) contract(table:⟨⟨Emp⟩⟩, Range [{e} | {e, e} ← simpleElement:⟨⟨root/FinEmp, eid⟩⟩] Any)

We can now remove the transformations created in Steps 14 and 15 to leave us with a final pathway made up simply of Transformations (166) to (183) describing the mapping we need between the new SQL database and the new XML schema.

Chapter 8

Conclusion

MM is a way of raising the level of abstraction in data management applications. The key idea behind MM is the development a set of operators that perform common data management tasks in a DDL independent way which can then be used together in a script to solve a wide range of data management problems in the field. In this thesis we have presented what we believe to be the most complete implementation of a MMS that supports instance-based mappings and operators.

The corner stone of any MMS is the framework in which the operators are implemented. It must be flexible enough to process schemas and mappings from a wide range of DDLs and yet enable efficient implementations of a diverse set of operators. In particular it must provide a uniform way of processing the constructs of all the DDLs supported by the system, using a mapping language that is closed under the operations we wish to implement. There must also be a way of generating transformations from the mappings and extracting the instances of the resultant schemas. Finally the framework must allow the operators to be combined together into a script.

The CDM, mapping language and query language of the AutoMed system on which we have based our framework, all have features that are advantageous in a MMS.

- Our CDM is flexible enough to express schemas from a wide range of DDLs. In particular it has a DDL independent way of expressing constraints, something that other MMS prototypes lack and has been identified as a failing in current systems [BM07].

- Our mapping language allows us to write executable, bidirectional mappings between schemas expressed in any of the DDLs supported by AutoMed. The schema transformation technique underlying our mapping language allows us to specify the extents of our target objects unambiguously and makes analysing the mappings during the operator implementation relatively simple. Another advantage of the schema transformation technique is that in the operator implementations we create the mapping from source to target as we create the target schema. Other mapping approaches require two separate steps, one to create the target schema and a second to create the mapping. This is of benefit in implementing the Merge, Diff and Extract operators which require mappings and a schema as output.

- We use a DDL-independent functional query language that supports second order functions which is necessary for our mapping language to be closed under composition [FKPT05].

8.1 Summary of Thesis Achievements

In this thesis we have described what we believe to be the first full implementation of a MMS for schema based DDLs that supports the instance-based semantics for all the MM operators [BHP00, BM07], excluding Match. Match has been independently implemented in the AutoMed framework [RM05, MRMM05].

We have shown how we can use the HDM to express schemas from a wide range of DDLs. In particular we have described how two new DDLs, XML Schema and RDFS, can be expressed in terms of the HDM. We have also implemented a function that translates a schema expressed in a high-level DDL into the HDM, returning a BAV transformation pathway that describes how we translate the high-level schema objects into the HDM, as output.

We have described how BAV pathways equivalent to SO tgds can be created that allow us to execute these declarative mappings and thereby transform schemas based on these mappings. We have made use of the generateGID function in AutoMed and the higher-order capability of IQL to create a flexible and powerful mapping language.

We have presented a data level implementation of ModelGen based on the composition of CTs, that can be applied to any DDL supported by AutoMed. These CTs

are chosen automatically using a novel approach based on matching the structure of the HDM schema to the objects created by translating objects from the target schema into the HDM. The choice of CT at each step in the process is limited by placing preconditions on the execution of each CT. As part of this work we introduced an extension to the HDM that allows us to translate primitive data type information between different DDLs using a common type hierarchy, an approach that has not been used before. The type hierarchy allows us to perform some type checking and identify where inter-DDL mappings may cause type casting errors.

We have described implementations of the instance-based semantics of all the MM operators, excluding Match, within our framework. These implementations are DDL-independent and make use of the detailed information contained in the transformation pathways we use as our mappings. There is an implementation of Match [RM05, MRMM05] in AutoMed that has been done independently of the work in this thesis as part of another PhD. At the moment the technique only works for the relational model and the output of Match is used to create a merged schema rather than providing a pathway between the input schemas as required by Match in MM. When the current work on Match is complete we aim to expand the matching technique developed to work on HDM schemas. This should allow us to apply it to any DDL supported by AutoMed we also aim to extend Match to output a pathway between the input schemas as well as between the input schemas and the merged schema and thereby integrate it with the system described here.

To demonstrate our system we have extended AutoMed by adding a MM API that allows us to write MM programs that can be used to execute MM scripts.

In summary, we have described what we believe to be the first implementation of a MMS that supports:

- a wide range of DDLs,
- instance based mappings,
- all the operators proposed by Bernstein excluding Match,
- a programming interface that allows the writing of MM programs.

8.2 Future Work

The vision for MM described in [BHP00] is that it should be applicable across a number of application domains. As well as the data management domain that we have discussed in this thesis, other areas that could benefit from the MM approach include workflow systems, message translation in middleware, interface specification systems and knowledge bases, among others. In common with the prototype described here, other existing prototypes also focus on the data management domain [BM07].

The fine-grained transformation based approach to implementing the operators described in this thesis could usefully be applied in new research investigating MMS that work across all these domains. The ability to 'divide and conquer' the problems by transforming a single, simple schema object at a time reduces the complexity of the individual steps making an overall solution easier to find. The major challenges are representing the various models in our CDM and 'querying' their extents with IQL. We have recently started to investigate how we can extend our MMS to process knowledge models, *i.e.* schema based DDLs from whose schemas new facts can be inferred using a reasoner. Specifically we have been looking at the ontology language OWL-DL [Mik04]. The focus of our work in this area has been how to simulate the reasoning ability of OWL-DL in SQL using triggers. We were successful in this, but how the work could be extended to allow us to translate an OWL-DL into the other DDLs in the AutoMed MMS and then infer new facts from the translated schema is the subject of ongoing work.

Looking further into the future, if we wish to add support for non schema based models like workflow systems we need to add constructs to the HDM to represent control flow and other concepts not found in a data model. Even more challenging is how to query the extent of a workflow and also to decide exactly what this extent should be. This would in turn impact the implementation of ModelGen. New CTs would need to be defined to transform the new constructs as well as new preconditions.

Another potential area for future research is on incremental updates to the data in the schemas taking part in a MM script. Our system generates executable mappings that are used for query processing. Any updates to the data sources will be automatically reflected in any virtual schemas generated by the script, however, schemas that have been materialised will need to be updated by hand or recreated. For example, in the case study in Chapter 7 we materialise an XML view of the original

SQL database. Any changes to the database will necessitate updates being made on the materialised XML view. At the moment the only way we can do this in our system is to re-execute the script to recreate the view if data is added to the source database. This is not a very efficient approach. This particular problem is one of those addressed in ADO.NET from Microsoft [MAB08]. Queries and updates are translated between an entity data model [BCMN06] and a relational database. The relationship between the application data and the persistent storage is specified using a declarative mapping, which is compiled into bidirectional views that drive the data transformation engine. View maintenance algorithms that incrementally update the entity data model are used for update translation. This does not represent a full solution to the problem in a MMS as only two DDLs are involved, however, the approach of generating executable *updates* based on the existing mappings in the script is one we could adopt in our system. Specifically we would need to develop algorithms that are executed when data updates are made to the source schemas in a MM script. These algorithms would identify what the updates are and execute methods in the wrapper of the DDL of any materialised target schemas to generate the required code to materialise the updates.

In a current project we are investigating how best to add provenance meta data to data sources and transformations in AutoMed. This information may make a useful addition to a MMS script. In our initial work we have added meta data describing the owner of the data source and the user responsible for creating a transformation, as well as functions that allow us to retrieve this information from a given transformation pathway. Adding this functionality to our MMS would involve adding the meta data to any source schemas in a script and possibly also providing some way of recording who created the script.

On an implementation level we could make it easier to specify our mappings. As we described in Section 6.7, we create the initial mappings in our system by translating a string describing a set of SO tgds into a BAV pathway. This provides an accurate way of specifying the mapping but the string can be very long if the target schema is big. In some cases it would be much more convenient to be able to specify the mappings graphically. This is the approach adopted in *GeromeSUITE* and the IBM data exchange system Clio [MHH00]. This would be particularly useful if objects in the source schema are mapped directly to objects in the target schema. We hope to extend the AutoMed GUI to allow a user to graphically specify mappings between a source and target schema in the future.

Appendix A

Pathways

A.1 Confluence Pathway from Chapter 7

The pathway generated by step 12

$$map_{S_{merge},S'_{finEmp}} = (map_{S_{merge},S_{emp}} \circ map_{S_{emp},S'_{finEmp-xml}}) \oplus (map_{S_{merge},S_{diff}} \circ map_{S_{diff},S'_{finEmp-xml}})$$

in the script in Chapter 7 is

① add(complexElement:⟨⟨null, root, 1, 1⟩⟩, [{r} | $r \leftarrow$ &0])
② add(complexElement:⟨⟨root, FinEmp, 0, unbounded⟩⟩, [{r, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
$r \leftarrow$ &0; {d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
③ add(simpleElement:⟨⟨root/FinEmp, eid, 1, 1, int⟩⟩, [{e, e} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
④ add(simpleElement:⟨⟨root/FinEmp, name, 1, 1, string⟩⟩, [{e, n} | {e, n} ← column:⟨⟨Emp, name⟩⟩;
{e, d} ← column:⟨⟨Emp, dept⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩; $dn \leftarrow$ 'Finance'])
⑤ add(simpleElement:⟨⟨root/FinEmp, dob, 0, 1, string⟩⟩, [{e, dob} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{e, dob} ← column:⟨⟨root_FinEmp, dob⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩;
$dn \leftarrow$ 'Finance'])
⑥ add(simpleElement:⟨⟨root/FinEmp, isMarried, 0, 1, string⟩⟩,
[{e, im} | {e, d} ← column:⟨⟨Emp, dept⟩⟩;
{e, im} ← column:⟨⟨root_FinEmp, isMarried⟩⟩; {d, dn} ← column:⟨⟨Dept, dname⟩⟩;
$dn \leftarrow$ 'Finance'])
⑦ add(key:⟨⟨FinEmpkey, ⟨⟨root, FinEmp⟩⟩, ⟨⟨root/FinEmp, eid⟩⟩⟩⟩)

The shrinking phase is:

⑧ delete(foreign_key:⟨⟨root_fk, root_FinEmp, ⟨⟨root_FinEmp, root⟩⟩, root, ⟨⟨root, root⟩⟩⟩⟩)
⑨ delete(foreign_key:⟨⟨Dept_fk, Emp, ⟨⟨Emp, dept⟩⟩, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
⑩ delete(primary_key:⟨⟨Empeid_pk, Emp, ⟨⟨Emp, eid⟩⟩⟩⟩)
⑪ delete(primary_key:⟨⟨Deptdid_pk, Dept, ⟨⟨Dept, did⟩⟩⟩⟩)
⑫ contract(column:⟨⟨Dept, numEmps⟩⟩,
Range $[\{d, ne\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{finEmp}, e, [e]$, 'did');
$ne \leftarrow$ generateGID($S'_{finEmp}, d, [d]$, 'numEmps')] Any)
⑬ contract(column:⟨⟨Dept, dname⟩⟩,
Range $[\{d, dn\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{finEmp}, e, [e]$, 'did'); $dn \leftarrow$ generateGID($S'_{finEmp}, d, [d]$, 'dname')] Any)
⑭ contract(column:⟨⟨Dept, did⟩⟩,
Range $[\{d, d\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{finEmp}, e, [e]$, 'did')] Any)
⑮ contract(table:⟨⟨Dept⟩⟩,
Range $[\{d\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{finEmp}, e, [e]$, 'did')] Any)
⑯ delete(primary_key:⟨⟨rootroot_pk, root, ⟨⟨root, root⟩⟩⟩⟩)
⑰ contract(column:⟨⟨root, root⟩⟩, $[\{r, r\} \mid \{r\} \leftarrow$ complexElement:⟨⟨null, root, 1, 1⟩⟩])
⑱ contract(table:⟨⟨root⟩⟩, complexElement:⟨⟨null, root, 1, 1⟩⟩)
⑲ delete(primary_key:⟨⟨root_FinEmpeid_pk, root, ⟨⟨root_FinEmp, eid⟩⟩⟩⟩)
⑳ contract(column:⟨⟨root_FinEmp, root⟩⟩,
Range $[\{e, r\} \mid \{r, e\} \leftarrow$ complexElement:⟨⟨root, FinEmp⟩⟩] Any)
㉑ contract(column:⟨⟨root_FinEmp, isMarried⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, isMarried⟩⟩ Any)
㉒ contract(column:⟨⟨root_FinEmp, dob⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, dob⟩⟩ Any)
㉓ contract(column:⟨⟨root_FinEmp, eid⟩⟩,
Range simpleElement:⟨⟨root/FinEmp, eid⟩⟩ Any)
㉔ contract(table:⟨⟨root_FinEmp⟩⟩,
Range $[\{e\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩] Any)
㉕ contract(column:⟨⟨Emp, dept⟩⟩,
Range $[\{e, d\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩;
$d \leftarrow$ generateGID($S'_{finEmp}, e, [e]$, 'did')] Any)
㉖ contract(column:⟨⟨Emp, name⟩⟩, Range simpleElement:⟨⟨root/FinEmp, name⟩⟩ Any)
㉗ contract(column:⟨⟨Emp, eid⟩⟩, Range simpleElement:⟨⟨root/FinEmp, eid⟩⟩ Any)
㉘ contract(table:⟨⟨Emp⟩⟩, Range $[\{e\} \mid \{e, e\} \leftarrow$ simpleElement:⟨⟨root/FinEmp, eid⟩⟩] Any)

Bibliography

[AAB+05] Serge Abiteboul, Rakesh Agrawal, Philip A. Bernstein, Michael J. Carey, Stefano Ceri, W. Bruce Croft, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Dieter Gawlick, Jim Gray, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Martin L. Kersten, Michael J. Pazzani, Michael Lesk, David Maier, Jeffrey F. Naughton, Hans-Jörg Schek, Timos K. Sellis, Avi Silberschatz, Michael Stonebraker, Richard T. Snodgrass, Jeffrey D. Ullman, Gerhard Weikum, Jennifer Widom, and Stanley B. Zdonik. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[AB87] Malcolm P. Atkinson and Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–190, 1987.

[AB01] Suad Alagic and Philip A. Bernstein. A model theory for generic schema management. In *DBPL*, pages 228–246, 2001.

[ACB05] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Modelgen: Model independent schema translation. In *ICDE*, pages 1111–1112, 2005.

[ACB06] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Model-independent schema and data translation. In *EDBT*, pages 368–385, 2006.

[ACM02] Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and translation for heterogeneous data. *Theor. Comput. Sci.*, 275(1-2):179–213, 2002.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. The universality of data retrieval languages. In *POPL*, pages 110–120, 1979.

[BBC+98] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[BCF+03] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3c.org/TR/xquery, November 2003.

[BCMN06] José A. Blakeley, David Campbell, S. Muralidhar, and Anil Nori. The ado.net entity framework: making the conceptual level real. *SIGMOD Record*, 35(4):32–39, 2006.

[BD03] Shawn Bowers and Lois M. L. Delcambre. The uni-level description: A uniform framework for representing information in multiple data models. In *ER*, pages 45–58, 2003.

[BDD+89] Philip A. Bernstein, Umeshwar Dayal, David J. DeWitt, Dieter Gawlick, Jim Gray, Matthias Jarke, Bruce G. Lindsay, Peter C. Lockemann, David Maier, Erich J. Neuhold, Andreas Reuter, Lawrence A. Rowe, Hans-Jörg Schek, Joachim W. Schmidt, Michael Schrefl, and Michael Stonebraker. Future directions in dbms research - the laguna beach participants. *SIGMOD Record*, 18(1):17–26, 1989.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[BFH+03] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Bridging the xml relational divide with legodb. In *ICDE*, pages 759–760, 2003.

[BG04] Dan Brickely and R.V Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004.

[BGMN06] Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing mapping composition. In *VLDB*, pages 55–66, 2006.

[BGMN08] Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing mapping composition. *VLDB J.*, 17(2):333–353, 2008.

[BH07] Philip A. Bernstein and Howard Ho. Model management and schema mappings: Theory and practice. In *VLDB*, pages 1439–1440, 2007.

[BHP00] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[BKL+04] Michael Boyd, Sasivimol Kittivoravitkul, Charalambos Lazanitis, Peter McBrien, and Nikos Rizopoulos. Automed: A bav data integration system for heterogeneous data sources. In *CAiSE*, pages 82–97, 2004.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[BM04] Paul Biron and Ashok Malhotra. XML Schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2, 2004.

[BM05] Michael Boyd and Peter McBrien. Comparing and transforming between data models via an intermediate hypergraph data model. *J. Data Semantics IV*, pages 69–109, 2005.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12, 2007.

[BS81] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[CHS02] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *VLDB J.*, 11(3):216–237, 2002.

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *VLDB*, pages 646–648, 2000.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[CP84] Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of relational views. *J. ACM*, 31(4):742–760, 1984.

[CR03] Kajal T. Claypool and Elke A. Rundensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *ICEIS (1)*, pages 219–224, 2003.

[Dat95] C. J. Date. *An Introduction to Database Systems, 6th Edition.* Addison-Wesley, 1995.

[DK97] Susan B. Davidson and Anthony Kosky. Wol: A language for database transformations and constraints. In *ICDE*, pages 55–65, 1997.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql: 2003 has been published. *SIGMOD Record*, 33(1):119–126, 2004.

[FBJV05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying generic model management to data mapping. In *BDA*, 2005.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[FKP05] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[FKPT05] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[FLM99] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *AAAI/IAAI*, pages 67–73, 1999.

[FP04] Hao Fan and Alexandra Poulovassilis. Schema evolution in data warehousing environments - a schema transformation-based approach. In *ER*, pages 639–653, 2004.

[FTS00] Mary Ferández, Wang-Chiew Tan, and Dan Suciu. Silkroute: Trading between relations and XML. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[GBM08] Michael N. Gubanov, Philip A. Bernstein, and Alexander Moshchuk. Model management engine for data integration with reverse-engineering support. In *ICDE*, pages 1319–1321, 2008.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[HFG87] D. I. Howells, N. J. Fiddian, and W. A. Gray. A source-to-source meta-translation system for relational query languages. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB*, pages 227–234, 1987.

[IYEP95] Song Il-Yeol, Mary Evans, and E. K. Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer & Software Engineering*, 3(4):427–459, 1995.

[JJNQ09] Matthias Jarke, Manfred A. Jeusfeld, Hans W. Nissen, and Christoph Quix. Heterogeneity in model management: A meta modeling approach. In *Conceptual Modeling: Foundations and Applications*, pages 237–253, 2009.

[JTMP03] Edgar Jasper, Nerissa Tong, Peter McBrien, and Alexandra Poulovassilis. View generation and optimisation in the automed data integration framework. In *CAiSE Short Paper Proceedings*, 2003.

[JTMP04] Edgar Jasper, Nerissa Tong, Peter McBrien, and Alexandra Poulovassilis. Generating and optimising views from both as view data integration rules. In *DBIS*, 2004.

[KM05] Sasivimol Kittivoravitkul and Peter McBrien. Integrating unnormalised semi-structured data sources. In *CAiSE*, pages 460–474, 2005.

[Kol05] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[KQCJ07] David Kensche, Christoph Quix, Mohamed Amine Chatti, and Matthias Jarke. Gerome: A generic role based metamodel for model management. *J. Data Semantics*, 8:82–117, 2007.

[KQLJ07] David Kensche, Christoph Quix, Yong Li, and Matthias Jarke. Generic schema mappings. In *ER*, pages 132–148, 2007.

[KQLL07] David Kensche, Christoph Quix, Xiang Li, and Yong Li. Geromesuite: A system for holistic generic model management. In *VLDB*, pages 1322–1325, 2007.

[LBU01] Chen Li, Mayank Bawa, and Jeffrey D. Ullman. Minimizing view sets without losing query-answering power. In *ICDT*, pages 99–113, 2001.

[Len02] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[LSM07] Duc Minh Le, Andrew Smith, and Peter McBrien. Inter model data integration in a p2p environment. In *DBISP2P*, pages 189–193, 2007.

[LSM08] Duc Minh Le, Andrew Smith, and Peter McBrien. Robust data exchange for unreliable p2p networks. In *DEXA Workshops*, pages 352–356, 2008.

[LV03] Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, 2003.

[LVLG03] Chengfei Liu, Millist W. Vincent, Jixue Liu, and Minyi Guo. A virtual xml database engine for relational databases. In *Xsym*, pages 37–51, 2003.

[MAB08] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4), 2008.

[MBHR05] Sergey Melnik, Philip A. Bernstein, Alon Y. Halevy, and Erhard Rahm. Supporting executable mappings in model management. In *SIGMOD Conference*, pages 167–178, 2005.

[MBM07] Peter Mork, Philip A. Bernstein, and Sergey Melnik. Teaching a schema translator to produce o/r views. In *ER*, pages 102–119, 2007.

[MBR01] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[McG59] William C. McGee. Generalization: Key to successful electronic data processing. *J. ACM*, 6(1):1–23, 1959.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*, volume 2967 of *Lecture Notes in Computer Science*. Springer, 2004.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[Mik04] Mike Dean and Guus Schreiber. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, 2004.

[MIR94] Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. Schema equivalence in heterogeneous systems: bridging theory and practice. *Inf. Syst.*, 19(1):3–31, 1994.

[MP98] Peter McBrien and Alexandra Poulovassilis. A general formal framework for schema transformation. In *Data and Knowledge Engineering*, volume 28, pages 47–71, 1998.

[MP99] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering*, volume 1626 of *LNCS*, pages 333–348. Springer Verlag, 1999.

[MP01] Peter McBrien and Alexandra Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Advanced Information Systems Engineering*, volume 2068 of *LNCS*, pages 330–345. Springer Verlag, 2001.

[MP02] Peter McBrien and Alexandra Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *CAiSE*, pages 484–499, 2002.

[MP03] Peter McBrien and Alexandra Poulovassilis. Data integration by bi-directional schema transformation rules. In *ICDE*, pages 227–238, 2003.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD Conference*, pages 193–204, 2003.

[MRMM05] Matteo Magnani, Nikos Rizopoulos, Peter McBrien, and Danilo Montesi. Schema integration based on uncertain semantic mappings. In *ER'05*, LNCS, pages 31–46. Springer, 2005.

[MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, pages 122–133, 1998.

[Obj] Object Management Group. OMG homepage. http://www.omg.org/.

[Pat04] Susanne Patig. Measuring expressiveness in conceptual modeling. In *CAiSE*, pages 127–141, 2004.

[PB94] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Commun. ACM*, 37(5):42–49, 134, 1994.

[PB03] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.

[PB08] Rachel Pottinger and Philip A. Bernstein. Schema merging and mapping creation for relational sources. In *EDBT*, pages 73–84, 2008.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[Pou01] Alexandra Poulovassilis. The AutoMed Intermediate Query Language. Technical report, AutoMed Project, http://www.doc.ic.ac.uk/automed, 2001.

[PS93] Alexandra Poulovassilis and Carol Small. A domain-theoretic approach to integrating functional and logic database languages. In *VLDB*, pages 416–428, 1993.

[RAH+96] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William F. Cody, Ronald Fagin, Peter M. Schwarz, Joachim Thomas II, and Edward L. Wimmers. The garlic project. In *SIGMOD Conference*, page 557, 1996.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[RM05] Nikolaos Rizopoulos and Peter McBrien. A general approach to the generation of conceptual model transformations. In *CAiSE*, pages 326–341, 2005.

[SKZ06] Guang-Lei Song, Jun Kong, and Kang Zhang. Autogen: Easing model management through two levels of abstraction. *J. Vis. Lang. Comput.*, 17(6):508–527, 2006.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[SM06] Andrew Smith and Peter McBrien. Inter model data exchange of type information via a common type hierarchy. In *DISWEB*, 2006.

[SM08a] Andrew Smith and Peter McBrien. Automodelgen:a generic data level implementation of modelgen. In *CAiSE Forum*, pages 65–68, 2008.

[SM08b] Andrew Smith and Peter McBrien. A generic data level implementation of modelgen. In *BNCOD*, pages 63–74, 2008.

[SMD03] Arijit Sengupta, Sriram Mohan, and Rahul Doshi. XER - Extensible Entity Relationship Modeling. In J. Harnad et al., editor, *Proceedings of the XML 2003 Conference*, Philadelphia, PA, USA, December 2003.

[SRM08] Andrew Smith, Nikos Rizopoulos, and Peter McBrien. Automed model management. In *ER*, pages 542–543, 2008.

[SRM09] Andrew Smith, Nikos Rizopoulos, and Peter McBrien. RoKEx: Robust Knowledge Exchange. In *SEAS DTC*, pages 82–89, 2009.

[SSB+01] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as xml documents. *VLDB J.*, 10(2-3):133–154, 2001.

[SSK+01] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A general techniques for querying xml documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.

[Sto75] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD Conference*, pages 65–78, 1975.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, 1999.

[TBMM01] Henry Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures. http://www.w3.org/TR/xmlschema-1, 2001.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.

[Zam08] Lucas Zamboulis. The AutoMed Intermediate Query Language ver.1.2. Technical report, AutoMed Project, http://www.doc.ic.ac.uk/automed, 2008.

[ZZKS05] Xiaoqin Zeng, Kang Zhang, Jun Kong, and Guang-Lei Song. Rgg+: An enhancement to the reserved graph grammar formalism. In *VL/HCC*, pages 272–274, 2005.