Using AutoMed Metadata in Data Warehousing Environments

Hao Fan, Alexandra Poulovassilis
School of Computer Science and Information Systems
Birkbeck College, University of London {hao, ap@dcs.bbk.ac.uk}

Abstract

Metadata management is an essential factor in data warehousing. In data warehousing environments, data is transformed and integrated into a single database from multiple autonomous and heterogeneous data sources. What kind of metadata can be used for expressing the multiplicity of data models and the data transformation and integration processes in such environments? How can this metadata be further used for supporting other data warehouse activities? We examine how these questions are addressed by AutoMed, a system for expressing data transformation and integration processes in heterogeneous database environments.

1 Introduction

Metadata has been a part of information processing since there have been programs and data. It captures the information necessary for supporting the management, querying, consistent use, and understanding of data. Metadata is particularly important in a data warehouse environment, since data warehouse activities such as data integration, data transformation, OLAP, data mining, and so on, are enabled by metadata. Due to the increasing complexity of data warehouses, metadata management has received increasing research focus [28, 21, 3, 8].

Typically, the metadata in a data warehouse includes information about both data and data processing. Information about data includes the schemas of the data sources, warehouse and data marts, ownership of the data, time information etc. Information about data processing includes rules for data extraction, cleansing and transformation, data refresh and purging policies, the lineage of migrated and transformed data etc.

Up to now, in order to transform and integrate data from heterogenous data sources, a conceptional data model (CDM) has been used. For example, [23, 27, 9] use a dimensional model; [7, 4, 35, 20] use an ER model, or extensions of it; [36] uses a multidimensional CDM called MAC providing modelling concepts familiar to OLAP users; [17] describes a framework for data warehouse design based on its Dimensional Fact Model; [38, 37] present a conceptual model and a set of abstract transformations for data extraction-transformation-loading (ETL); and [19] adopts the relational data model as the CDM.

All these approaches assume a single conceptual data model (CDM) for the data transformation/integration — see Figure 1(a). Each data source has a wrapper for translating its schema and data into the CDM. The integrated schema is then derived from these CDM schemas by means of view definitions, and is expressed in the same modelling language as them.

This traditional CDM framework has a number of drawbacks. Firstly, since they are both high-level conceptual data models, semantic mismatches may exist between the CDM and a source data model, and there may be a loss of information between them. Secondly, if a source schema changes, it is not straightforward to evolve the view definitions of the integrated schema constructs in terms of source schema constructs. Finally, the data transformation and integration metadata is tightly coupled with the CDM of the particular data warehouse. If the

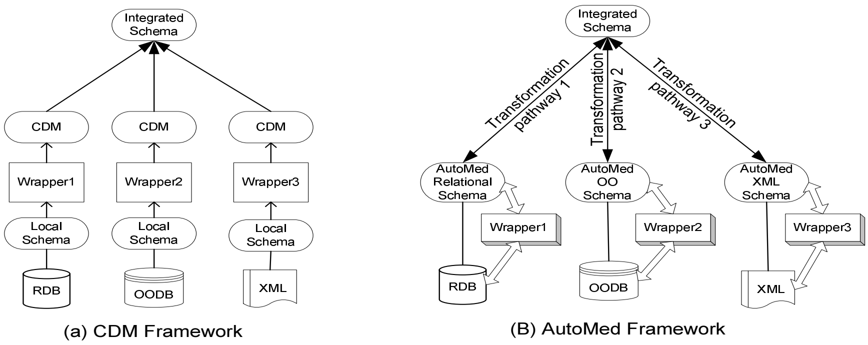

Figure 1: Frameworks of Data Integration

warehouse is to be redeployed on a platform with a different CDM, it is not easy to reuse the previous warehouse implementation.

AutoMed is a data transformation and integration system which adopts a low-level hypergraph-based data model (HDM) as its common data model[1]. So far, research has focused on using AutoMed for virtual data integration. This paper describes how AutoMed can also be used for materialized data integration, in particular for expressing the data transformation/integration metadata, and using this metadata to support warehouse activities such as data cleansing, populating the warehouse, incrementally maintaining the warehouse data after data source updates, and tracing the lineage of warehouse data.

Using AutoMed for materialised data integration, the data source wrappers translate the source schemas into their equivalent specification in terms of AutoMed's low-level HDM, without loss of information — see Figure 1(*b*). AutoMed's schema transformation facilities can then be used to incrementally transform and integrate the source schemas into an integrated schema. The integrated schema can be defined in any modelling language which has been specified in terms of AutoMed's HDM (see Section 2 below for more details). We will examine in this paper the benefits of this alternative approach to data transformation/integration in data warehousing environments.

An earlier paper [**?**] proposed a variation of the HDM as the common data model for both virtual and materialised integration, and a visual, hypergraph-based query language (HQL) for defining views of derived constructs in terms of source constructs. However, that paper did not focus on expressing data warehouse metadata, or on warehouse activities such as data cleansing or populating and maintaining the warehouse.

Outline of this paper: Section 2 gives an overview of the AutoMed framework, to the level of detail necessary for our purposes here. Section 3 shows how AutoMed metadata has enough expressivity to describe the data integration and transformation processes in a data warehouse. Section 4 discusses how the AutoMed metadata can be used for some data warehousing activities. Section 5 discusses the benefits of our approach. Section 6 gives our concluding remarks and directions of further work.

2 AutoMed

The basis of the AutoMed data integration system is a low-level *hypergraph-based data model (HDM)* [**?**, 24]. Facilities are provided for defining higher-level modelling languages in terms of this lower-level HDM. An HDM schema consists of a set of nodes, edges and constraints, and so each modelling construct of a higher-level modelling language is specified as some combination

[1] `http://www.doc.ic.ac.uk/automed`

of HDM nodes, edges and constraints. For any modelling language $\mathcal{M}$ specified in this way (via the API of AutoMed's Model Definitions Repository [5]), AutoMed automatically provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$. In particular, for every construct of $\mathcal{M}$ there is an add and a delete primitive transformation which (conceptually) add to, or delete from, the underlying HDM schema the corresponding set of nodes, edges and constraints. For those constructs of $\mathcal{M}$ which have textual names, there is also a rename primitive transformation.

One advantage of using a low-level common data model such as the HDM is that semantic mismatches between high-level modelling constructs are avoided. Another advantage is that the HDM provides a unifying semantics for higher-level modelling constructs and hence a basis for automatically or semi-automatically generating the semantic links between them — this is ongoing work being undertaken by other members of the AutoMed project.

In AutoMed, schemas are incrementally transformed by applying to them a sequence of primitive transformations $t_1, \ldots, t_r$. Each primitive transformation t_i makes a 'delta' change to the schema by adding, deleting or renaming just one schema construct. Thus, intermediate schemas may contain constructs of more than one modelling language.

Each add or delete transformation is accompanied by a query specifying the extent of the new or deleted construct in terms of the rest of the constructs in the schema. This query is expressed in a functional intermediate query language, IQL [30][2]. Also available are contract and extend transformations which behave in the same way as add and delete except that they indicate that their accompanying query may only partially construct the extent of the new/removed schema construct. Moreover, their query may just be the constant Void, indicating that the extent of the new/removed construct cannot be derived even partially, in which case the query can be omitted.

We term a sequence of primitive schema transformations from one schema S_1 to another schema S_2 a transformation *pathway* from S_1 to S_2, denoted $S_1 \rightarrow S_2$.

All source schemas, intermediate schemas, integrated schemas, and the pathways between them are stored in AutoMed's Schemas & Transformations Repository [5]. The queries present within transformations that add or delete schema constructs mean that each primitive transformation has an automatically derivable *reverse transformation*. In particular, each add/extend transformation is reversed by a delete/contract transformation with the same arguments, while each rename transformation is reversed by another rename transformation with the two arguments swapped. We refer the reader to [25] for an extensive discussion of the AutoMed data integration approach.

If a set of source schemas $S_1, \ldots, S_n$ have been integrated into an integrated schema S by means of a set of pathways, we will see in Section 4.1 below how these pathways can be used for populating S if it is going to be a materialised schema.

If S is a virtual schema, then view definitions defining its constructs in terms of the constructs of $S_1, \ldots, S_n$ can automatically be derived from the pathways, and in particular, from the add, extend and rename steps within them. This is done by traversing the pathways $S_1 \rightarrow S, \ldots, S_n \rightarrow S$ backwards from S down to each S_i (see [?]). These view definitions can then be used for global query processing by substituting them into queries expressed over the integrated schema in order to reformulate them into queries expressed on the source schemas. This is the global-as-view approach to global query processing over distributed heterogeneous data sources and is what we currently support in AutoMed. More specifically, after a query expressed on an integrated schema has been reformulated and optimised, sub-queries of it are submitted to the appropriate data source Wrappers for translation into the data source query languages and evaluation at the data sources. The wrappers then translate sub-query results

[2]IQL is a comprehensions-based functional query language. Such languages subsume query languages such as SQL and OQL in expressiveness [?, ?].

back into the IQL type system and the IQL query processor undertakes any further necessary post-processing of the query [22].

The fine granularity of AutoMed's primitive schema transformations readily allows *evolution* of both the integrated and the source schemas, the latter including also the addition or removal of source schemas. This is because such schema evolutions can be expressed as extensions to the existing pathways, possibly followed by some automatic or semi-automatic simplification of the resulting pathways — see [**?**, 25] for details. New view definitions for the constructs of the integrated schema can then be regenerated from the evolved pathways. If the integrated schema is a virtual one, these new view definitions can now be used for global query processing. We discuss the case of a materialised integrated schema in Section 5 below.

2.1 Representing a Multidimensional Model in the AutoMed

Previous work has shown how conceptual modelling languages such as relational, ER, UML and XML can be represented in terms of the HDM [**?**, **?**]. Here we illustrate how a simple multidimensional data model can be specified in terms of the HDM.

A *schema* in the HDM data model is a triple $(Nodes, Edges, Constraints)$. $Nodes$ and $Edges$ define a labelled, directed, nested hypergraph. It may be 'nested' in the sense that edges can link any number of both nodes and other edges. A *query* q over a schema is an expression whose variables are members of $Nodes \cup Edges$. $Constraints$ is a set of boolean-valued queries over the schema which are satisfied by all instances of the schema. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them. Edges need not be uniquely named within an HDM schema but are uniquely identified by the combination of their name and the components they link.

The constructs of any higher-level modelling language $\mathcal{M}$ may be *nodal*, *linking*, *nodal-linking* or *constraint* constructs, or possibly a combination of these:

- *nodal* constructs may exist independently of any other constructs and are represented by an HDM node.
- *linking* constructs can only exist when certain other constructs exist. The extent of a linking construct is a subset of the Cartesian product of the extents of its dependent constructs. Linking constructs are represented by HDM edges.
- *nodal-linking* constructs are nodal constructs that can only exist when certain other constructs exist, and that are linked to these constructs (attributes in the relational model are an example). Nodal-linking constructs are represented by a combination of an HDM node and an HDM edge.
- *Constraint* constructs represent restrictions on the extents of the other three kinds of constructs.

The *scheme* of a construct (delimited by double chevrons) uniquely identifies it within a schema.

Our simple multi-dimensional data model has four basic modelling constructs: Fact, Dim (dimension), Att (non-key attribute) and Hierarchy; for simplicity, we model a measure as any other non-key attribute. Fact and Dim are nodal, Att is nodal-linking, and Hierarchy is a constraint. This specification is illustrated in Table 1.

We see that a fact or dimension table R with primary attributes $k_1, \ldots, k_n$ $(n \geq 1)$ is uniquely identified by the scheme $\ll R, k_1, \ldots, k_n \gg$. This translates in the HDM to a nodal construct $\ll R \gg$ the extent of which is the projection of the table R onto its primary key attributes $k_1, \ldots, k_n$. Each non-key attribute a of a fact or dimension table R is uniquely identified by the scheme $\ll R, a \gg$. This translates in the HDM into a nodal-linking construct

Dimensional Construct	HDM Representation
construct: `Fact` class: `nodal` scheme: $\ll R, k_1, \ldots, k_n \gg$	node: $\ll R \gg$
construct: `Dim` class: `nodal` scheme: $\ll R, k_1, \ldots, k_n \gg$	node: $\ll R \gg$
construct: `Att` class: `nodal-linking` scheme: $\ll R, a \gg$	node: $\ll R : a \gg$ edge: $\ll \_, R, R : a \gg$
construct: `Hierarchy` class: `constraint` scheme: $\ll R, R', k_i, k'_j \gg$	constraint: $[x_i \| (x_1, \ldots, x_n) \leftarrow \ll R, k_1, \ldots, k_n \gg] \subseteq$ $[y_j \| (y_1, \ldots, y_m) \leftarrow \ll R', k'_1, \ldots, k'_m \gg]$

Table 1: Definition of Dimensional Model Constructs

comprising a new node $\ll R : a \gg$ and an edge $\ll \_, R, R : a \gg$. The extent of the edge is the projection of table R onto $k_1, \ldots, k_n, a$.

Hierarchy constructs reflect the relationship between a primary key attribute k_i in a fact table R and its referenced foreign key attribute k'_j in a dimension table R', or between a primary key attribute in a dimension table R and its referenced foreign key attribute in a sub-dimension table R'. A hierarchy construct maps to a constraint in the corresponding HDM schema, which asserts that the set of values of k_i in R are always contained in the set of values for k'_j in R'.

3 Expressing Data Warehouse Schemas and Transformations in AutoMed

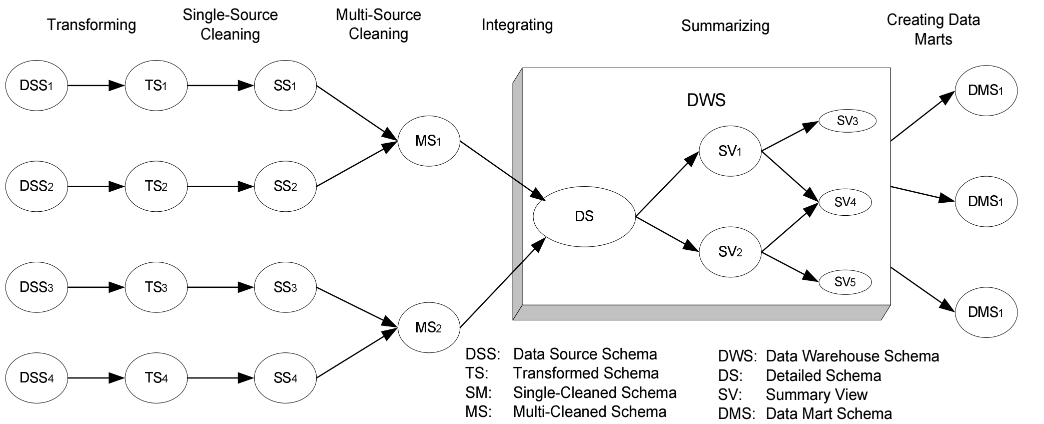

Figure 2: Data Transformation and Integration at the Schema Level

Figure 2 illustrates the data transformation and integration process in a typical data warehouse. Generally, the ETL process includes extracting and transforming data from the data sources, and loading the transformed data into the warehouse schema. In this paper we assume that data extraction has already happened i.e. that all the 'data sources' are local copies of data extracted from remote data sources. Thus, the data transformation/integration process is divided into the six stages shown in Figure 2: transforming, single-source cleansing, multi-source cleansing, integrating, summarizing, and creating data marts.

In Figure 2, the data source schemas (DSS_i) may be expressed in any modelling language that has been specified in AutoMed. The transforming process translates each DSS_i into a transformed schema TS_i which is ready for single-source data cleansing. Each TS_i may be defined in the same, or a different, modelling language as DSS_i and other TSs. The translation from a DSS_i to a TS_i is expressed as an AutoMed transformation pathway $\mathrm{DSS}_i \to \mathrm{TS}_i$. Such translation may not be necessary if the data cleansing tools to be employed can be applied directly to DSS_i, in which case TS_i and DSS_i are identical.

The single-source data cleansing process transforms each TS_i into a single-source-cleaned schema SS_i, which is defined in the same modelling language as TS_i but may be a different from it. The single-source cleansing process is expressed as an AutoMed transformation pathway $\mathrm{TS}_i \to \mathrm{SS}_i$ and we discuss it further in Section 3.3.1 below.

Multi-source data cleansing removes conflicts between sets of single-source-cleaned schemas and creates a multi-source-cleaned schema MS_i from them — we discuss it further in Section 3.3.2 below. Between the single-source-cleaned schemas and the detailed schema (DS) of the data warehouse there may be several stages of MSs, possibly represented in different modelling languages.

In general, if during multi-source data cleansing, n schemas $S_1, \ldots, S_n$ need to be transformed and integrated into one schema S, we can first automatically create a 'union' schema $S_1 \cup \ldots \cup S_n$ (after first undertaking any renaming of constructs necessary to avoid any naming ambiguities between constructs from different schemas). We can then express the transformation/integration process as a pathway $S_1 \cup \ldots \cup S_n \to S$. (There are also other schema integration approaches possible with AutoMed; with this approach, and in a data warehousing context, there is no need for `extend` transformation steps and we ignore them henceforth in this paper).

After mulit-source data cleansing, the resulting MSs are then transformed and integrated into a single detailed schema, DS, expressed in the data model supported by the data warehouse. The DS can then be enriched with summary views by means of a transformation pathway from DS to the final data warehouse schema DWS.

Data mart schemas (DMS) can subsequently be derived from the DWS and these may be expressed in the same, or a different, modelling language as the DWS. Again, the derivation is expressed as a transformation pathway DWS $\to$ DMS.

Using AutoMed, four steps are needed in order to create the metadata expressing the above schemas and transformation pathways:

1. **Create AutoMed repositories:** AutoMed metadata is stored in the Model Definitions Repository (MDR) and the Schemas & Transformations Repository (STR) [5]. The API to these repositories uses JDBC to access an underlying relational database. Thus, these repositories can be implemented using any DBMS supporting JDBC. If the DBMS of the data warehouse supports JDBC, then the AutoMed repositories can be part of the data warehouse itself.

2. **Specify data models:** All the data models that will be required for expressing the various schemas of Figure 2 need to be specified in terms of AutoMed's HDM, via the API of the MDR (if they are not already specified within the MDR).

3. **Extract data source schemas:** Each data source schema is automatically extracted and translated into its equivalent AutoMed representation using the appropriate wrapper for that data source.

4. **Define transformation pathways:** The remaining schemas of Figure 2 and the pathways between them can now be defined, via the API of the STR.

After any primitive transformation is applied to a schema, a new schema results. By default, this will be an 'intentional' schema within the STR i.e. it is not stored but its definition can be derived by traversing the pathway from its nearest ancestor 'extensional' schema. The data source schemas are, by definition, extensional schemas i.e their full definition is stored within the STR. It is also possible to request that any other schema becomes an extensional one e.g. the successive stages of schemas identified in Figure 2.

After any add(c,q) transformation step, it is possible to materialise the new construct c by creating, externally to AutoMed, a new local data source whose local schema includes c and populating this data source by the result of evaluating the query q (we discuss this process in more detail in Section 4.1 below).

Thus, in general, a schema may be a materialised schema (all its constructs are materialised) or a virtual schema (none of its constructs are materialised) or partially materialised (some of its constructs are materialised, some not).

In the following sections, we discuss in more detail how AutoMed transformation pathways can be used for describing the six stages of the data transformation/integration process illustrated in Figure 2. We first give a simple example illustrating data transformation/integration, assuming that no data cleansing is necessary.

3.1 An Example

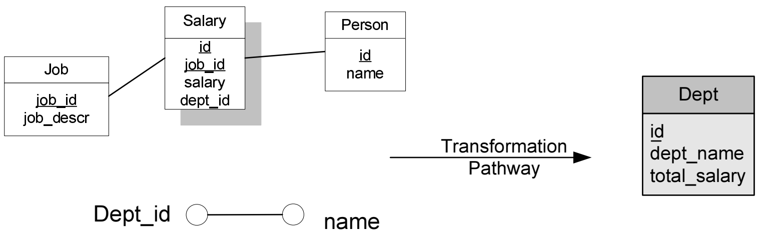

Figure 3: An Example of Data Integration/Transformation

Figure 3 shows a multi-dimensional schema consisting of a fact table Salary and two dimension tables Person and Job; an HDM schema consisting of two nodes Dept_id and name and an (un-named) edge between them; and a relational schema consisting of a single table Dept into which the other two schemas need to be transformed and integrated.

We discussed in Section 2.1 how a multi-dimensional model can be represented in AutoMed. For the HDM itself, the modelling constructs are Node, Edge and Constaint. We assume here a simple relational model which is represented similarly to our multi-dimensional model: each relation R with key attributes $k_1, \ldots, k_n$ and non-key attributes $a_1, \ldots, a_m$ is represented by a Rel construct $\ll R, k_1, \ldots, k_n \gg$ whose extent is the projection of R onto $k_1, \ldots, k_n$, plus a set of Att constructs $\ll R, a_1 \gg$, $\ldots$, $\ll R, a_m \gg$, where the extent of each $\ll R, a_i \gg$ is the projection of R onto $k_1, \ldots, k_n, a_i$ (we refer the reader to [**?**] for an encoding of the full relational model in the HDM).

In order to integrate the two source schemas into the target schema we first form their union schema. The following three primitive transformations are then applied to this schema in order to add the Dept relation to it, defining the extent of its id key attribute to be the same as the extent of the Dept_id HDM node, the extent of its dept_name attribute to be the same as that of the HDM edge from Dept_id to name, and the extent of its total_salary attribute to be obtained by summing the salaries for each department in the Salary table:

```
addRel (<<Dept,id>>,<<Dept_id>>);
addAtt (<<Dept,dept_name>>,<<_,Dept_id,name>>);
addAtt (<<Dept,total_salary>>,gc sum [(d,s)|(i,j,s)<-<<Salary,salary>>;
                                            (i',j',d)<-<<Salary,dept_id>>;
                                             i=i'; j=j']);
```

In the last step above, the IQL function gc is a higher-order function that takes as its first argument an aggregation function and as its second argument a bag of pairs; it groups the pairs on their first component, and then applies the aggregation function to each bag of values formed from the second components.

The following three transformations can then be applied to the current schema to remove the HDM constructs from it — note how the queries show how the extents of these constructs could be reconstructed from the remaining schema constructs:

```
delEdge (<<_,Dept_id,name>>,<<Dept,dept_name>>);
delNode (<<name>>,[n|(d,n)<-<<Dept,dept_name>>);
delNode (<<Dept_id>>,(<<Dept,id>>);
```

Finally, the following sequence of transformations remove the multi-dimensional schema constructs — note that contract rather than delete transformations are used since their extents cannot be reconstructed from the remaining schema constructs:

```
contractHierarchy(<<Salary,Person,id,id>>);
contractHierarchy(<<Salary,Job,job_id,job_id>>);
contractAtt      (<<Salary,salary>>);
contractAtt      (<<Salary,dept_id>>);
contractFact     (<<Salary,id,job_id>>);
contractAtt      (<<Job,job_descr>>);
contractDim      (<<Job,job_id>>);
contractAtt      (<<Person,name>>);
contractDim      (<<Person,id>>);
```

The final schema consists of the Dept relation and its attributes, as required.

3.2 Transforming

The above example illustrates how a schema expressed in one data model can be transformed into a schema expressed in another. The general approach is to first add the new schema constructs in the target data model (relational in the above example) and then to delete or contract the schema constructs expressed in the original data model(s) (multi-dimensional and HDM in the above example).

3.3 Data Cleansing

Data cleansing, also called data *cleaning* or *scrubbing*, deals with detecting and removing errors and inconsistencies from data in order to improve its quality, and is typically required before loading the transformed data into the data warehouse. In [31], the data cleansing problem is classified into two aspects, *single-source* and *multi-source*. For each of these, there are two levels of problems, *schema-level* and *instance-level*. Schema-level problems can be addressed by evolving the schema as necessary. Instance-level problems refer to errors and inconsistencies in the actual data which are not visible at the schema level. In this section, we describe how AutoMed metadata can be used for expressing the data cleansing process, for both single and multiple data sources, and for both schema-level and instance-level problems.

3.3.1 Single-Source Cleansing

Schema-level single-source problems may arise within a transformed schema TS_i in Figure 2 and they can be resolved by means of an AutoMed transformation pathway that evolves TS_i as necessary. Single-source instance-level problems include value, attribute and record problems. Value problems occur within a single value and include problems such as a missing value, a mis-spelled value, a mis-fielded value (e.g. putting a city name in a country attribute), embedded values (putting multiple values into one attribute value), using an abbreviation, or a mis-expressed value (e.g. using the wrong order of first name and family name within a name attribute). Attribute problems relate to multiple attributes in one record and include problems such as dependence violation (e.g. between city and zip, or between birth-date and age). Record problems relate to multiple records in the data source, and include problems such as duplicate records or contradictory records.

Some instance-level problems do not require the schema to be evolved, only the extent of one or more schema constructs to be corrected. In general, suppose that the extent of a schema construct c needs to be replaced by a new, cleaned, extent. We can do this using an AutoMed pathway by following these steps:

1. Add a new temporary construct *temp* to the schema, whose extent consists of the 'clean' data that is needed to generate the new extent of c. This clean data is derived from the extents of the existing schema constructs. This derivation may be expressed as an IQL query, or as a call to an 'external' function or, more generally, as an IQL query with embedded calls to external functions.

   (The IQL interpreter is easily extensible with new built-in functions, implemented in Java, and these may themselves call out to other external functions. If the extent of a new schema construct depends on calls to one or more external functions, then the new construct must be materialised. Otherwise, if the extent of a new construct is defined purely in terms of IQL and its own built-in functions then the new construct need not be materialised.)

2. Contract the construct c from the schema.

3. Add a new construct c whose extent is derived from *temp*.

4. Delete or contract the *temp* construct.

To illustrate, suppose we have available a built-in function toolCall which allows a specified external data cleansing tool to be invoked with specified input data. Then, we can invoke the QuickAddress Batch tool[3] to correct the zip and address attributes of a table Person(<u>id</u>,name,address,zip,city,country,phoneAndFax,maritalStatus) by regenerating these attributes given the combination of address, zip and city information:

```
addRel      (<<Temp,id,address,zip>>,
             toolCall 'QuickAddress Batch' '<<Person,address>>'
                      '<<Person,zip>' '<<Person,city>>');
contractAtt(<<Person,zip>>);
contractAtt(<<Person,address>>);
addAtt      (<<Person,zip>>,[(i,z)|(i,a,z)<-<<Temp,id,address,zip>>]);
addAtt      (<<Person,address>>,[(i,a)|(i,a,z)<-<<Temp,id,address,zip>>]);
deleteRel   (<<Temp,id,address,zip>>,[(i,a,z)|(i,a) <- <<Person,address>>;
                                          (i',z) <- <<Person,zip>>;
                                           i = i'])
```

[3] http://www.techie.techieindex.com/cug/qas/product/search/search.jsp

Some instance-level problems will also require the schema to be evolved. For example, if we have available a built-in function split_phone_fax which slits a string comprising a phone number followed by one or more spaces followed by a fax number into a pair of numbers, then the following AutoMed pathway converts the attribute phoneAndFax of the Person table above into two new attributes phone and fax:

```
addRel (<<Temp,id,phone,fax>>,[(i,p,f)|(i,pf)<-<<Person,phoneAndFax>>;
                                (p,f)<-split_phone_fax pf]);
addAtt (<<Person,phone>>,[(i,p)|(i,p,f)<-<<Temp,id,phone,fax>>]);
addAtt (<<Person,fax>>,  [(i,f)|(i,p,f)<-<<Temp,id,phone,fax>>]);
contractAtt (<<Person,phoneAndFax>>);
delRel (<<Temp,id,phone,fax>>,[(i,p,f)|(i,p)<-<<Person,phone>>;
                                (i',f)<-<<Person,fax>>;
                                i = i'])
```

3.3.2 Multi-Source Cleansing

After single-source data cleansing, there may still exist conflicts between different single-source cleaned schemas in Figure 2, leading to the process of multi-source cleansing.

Schema-level problems in multi-source cleansing include attribute and structure conflicts. Attribute conflicts arise when different sources use the same name for different constructs (homonyms) or different names for the same construct (synonyms), and they can be resolved by applying appropriate rename transformations to one of the schemas. Structure conflicts arise when the same information is modelled in different ways in different schemas, and they can be resolved by evolving one or more of the schemas using appropriate AutoMed pathways.

Instance-level problems in multi-source cleansing include attribute, record, reference, and data source problems. Attribute problems include different representations of the same attribute in different schemas (e.g. for a maritalStatus attribute) or a different interpretations of the values of an attribute in different schemas (e.g. US Dollar vs Euro in a currency attribute).

Such problems can be resolved by generating a new extent for the attribute in one of the schemas by applying an appropriate conversion function to each its values. In general, suppose we wish to convert each of the values within the extent of a construct c in a schema S by applying a function f to it. First a new construct *c_new* is added to S, whose extent is populated by iterating over the extent of c and applying f to each of its values. Then, the old construct c is deleted or contracted from the schema, and finally *c_new* is renamed to c. For example, the following pathway converts a 'M'/'S' representation for the maritalStatus attribute in the above Person table into a 'Y'/'N' representation, assuming the availability of a built-in function convertMS which maps 'M' to 'Y' and 'S' to 'N':

```
addAtt      (<<Person,maritalStatus_new>>,
             [(i,convertMS s)|(i,s)<-<<Person,maritalStatus>>]);
contractAtt (<<Person,maritalStatus>>);
renameAtt   (<<Person,maritalStatus_new>>, <<Person,maritalStatus>>);
```

Note that if there is also available an inverse function convertMSinv which maps 'Y' to 'M' and 'N' to 'S', then a delete transformation could have been used in the second step above instead of a contract:

```
deleteAtt   (<<Person,maritalStatus>>,
             [(i,convertMSinv s)|(i,s)<-<<Person,maritalStatus_new>>]);
```

Record problems include duplicate records or contradictory records among different data sources. For duplicate records, suppose that constructs c and c' from different schemas are to be integrated into a single construct within some multi-source cleaned schema. Then, prior to the integration, we can create a new extent for c comprising only those values not present in the extent of c':

```
add      (c_new, [v | v <- c; not (member c’ v)];
contract (c);
rename   (c_new, c)
```

For contradictory records, we can similarly create a new extent for c comprising only those values which do not contradict values in the extent of c'. For example, suppose we have tables Person and Employee in different schemas, both with key id, and the attributes << Person,maritalStatus >> and << Emp,maritalStatus >> are going to be integrated into a single attribute of a single table within some multi-source cleaned schema. Then the following transformation removes values from << Person,maritalStatus >> which contradict values in << Emp,maritalStatus >> (assuming that the latter is the more reliable source — the opposite choice would also of course be possible);

```
addAtt      (<<Person,maritalStatus_new>>,
             <<Person,maritalStatus>> --
              [(i,s)|(i,s)<-<<Person,maritalStatus>>;
                     (i’,s’)<-<<Emp,maritalStatus>>;
                     i = i’; not (s = s’)]);
contractAtt (<<Person,maritalStatus>>);
renameAtt   (<<Person,maritalStatus_new>>,<<Person,maritalStatus>>);
```

Reference problems occur when a referenced value does not exist in the target schema construct and can be resolved by removing the dangling references. For example, if an attribute << Emp,dept_id >> references a table << Dept >> with key dept_id, then the following transformation removes values from << Emp,dept_id >> for which there is no corresponding dept_id value in << Dept >> :

```
addAtt      (<<Emp,dept_id_new>>,[(i,d)|(i,d)<-<<Emp,dept_id>>;
                                        member <<Dept>> d]);
contractAtt (<<Emp,dept_id>>);
renameAtt   (<<Emp,dept_id_new>>,<<Person,dept_id>>);
```

Finally, data source problems relate to whole data sources, for example, aggregation at different levels of detail in different data sources (e.g. sales may be recorded per product in one data source and per product category in another data source). Such conflicts can be resolved either by retaining both sets of source data within the target multi-source schema MS_i (with appropriate renaming of schema constructs as necessary) or by selecting the ‘coarser’ aggregation and creating a view over the more detailed data which summarises this data at the coarser level, ready for integration with the more coarsely aggregated data from the other data source.

3.3.3 Summary

In this subsection we have shown how AutoMed metadata has enough expressivity to express the data cleansing process in a data warehouse environment. For all categories of data cleansing problems, the general approach is to add new constructs to the current schema and to populate them by ‘clean’ data generated from the extents of the existing schema constructs by means of IQL queries and/or or calls to external functions. The old, ‘dirty’, schema constructs are then contracted from the schema.

3.4 Integrating

After data cleansing, the resulting multi-source-cleaned schemas $\mathrm{MS}_1, \ldots, \mathrm{MS}_n$ are ready to be transformed and integrated into the detailed schema, DS. First, a union schema $\mathrm{MS}_1 \cup \ldots \cup \mathrm{MS}_n$ is automatically generated. The transformation/integration process is then expressed as a pathway $\mathrm{MS}_1 \cup \ldots \cup \mathrm{MS}_n \rightarrow \mathrm{DS}$. Section 3.1 above illustrated this process.

3.5 Summarizing

Data summarization builds either virtual or materialized views over the detailed data. This can be expressed by means of a transformation pathway from DS to the final data warehouse schema DWS, consisting of a series of `add` steps defining the new summarised constructs as views over the constructs of DS.

3.6 Creating Data Marts

Data mart schemas (DMS) can subsequently be derived from the DWS, again by means of a transformation pathway DWS $\rightarrow$ DMS. Unlike the previous, summarizing, step the target schema may be expressed in a different modelling language to the DWS. In fact, this step can be regarded as a separate instance of Figure 2 where the DWS now plays the role of the (single) data source and the DMS plays the role of the target warehouse schema. The scenario is a simplification of Figure 2 since there is only one data source, and there are no single-source or multi-source cleaned schemas.

4 Using the AutoMed Transformation Pathways

In the previous section we showed how AutoMed metadata can be used for expressing the ETL process in a data warehouse. In this section, we discuss how the resulting transformation pathways can be used for some key data warehousing activities: populating the data warehouse, incrementally maintaining the warehouse data after data source updates, and tracing the lineage of warehouse data.

4.1 Populating the Data Warehouse

In order to use the AutoMed transformation pathways for populating the data warehouse, a wrapper is required for each kind of data store from which data will be extracted or in which data will be stored. AutoMed's wrappers are implemented at two levels. A *high level wrapper* converts between AutoMed queries and data and the standard representation for a class of data sources e.g. the SQL92Wrapper converts between IQL and SQL92. A *low level wrapper* deals with differences between the class standard and a particular data source e.g. the *PostgresSQLWrapper* converts between SQL92 and Postgres databases.

In order to populate a construct c of a schema S, we need to generate a view definition for each construct of S in terms of its nearest ancestor materialised constructs within the pathways from the data source schemas $\mathrm{DSS}_1, \ldots, \mathrm{DSS}_n$ to S. This can be done using a modification of the view generation algorithm described in [**?**]. This algorithm traverses the pathway from S to each DSS_i backwards, all the way to DSS_i. The modified algorithm stops whenever a materialised construct is encountered in a pathway. The result is a view definition of the construct c in terms of already materialised constructs. This view definition is an IQL query which can be evaluated, and the resulting data can be inserted into the data store linked with c, via a series of update requests to that data store's wrapper.

4.2 Incrementally Maintaining the Warehouse Data

In order to incrementally maintain materialised warehouse data, we need to use incremental view maintainence techniques. If a materialised construct c is defined by an IQL query q over other materialised constructs, [13] gives formulae for incrementally maintaining c if one its ancestor constructs c^a has new data inserted into it (an increment) or data deleted from it (a decrement). We actually do not use the whole view definition q generated for c, but instead track the changes from c^a through each step of the pathway. In particular, at each add or rename step we use the set of increments and decrements computed so far to compute the increment and decrement for the schema constructed being generated by this step of the pathway.

4.3 Tracing the Lineage of Data Warehouse Data

The *lineage* of a data item t in the extent of a materialised construct c of a schema S is a set of source data items from which t was derived. The fundamental definitions regarding data lineage were developed in [11], including the concept of a *derivation pool* for tracing the data lineage of a tuple in a materialised view. Another fundamental concept was addressed in [6], namely the difference between 'why-provenance' and 'where-provenance'. Why-provenance refers to the source data that had some influence on the existence of the integrated data. Where-provenance refers to the actual data in the sources from which the integrated data was extracted. The problem of why-provenance has been studied for relational databases in [11, 39, 10].

We have developed definitions for data lineage in AutoMed based on both why-provenance and where-provenance, which we term *affect-pool* and *origin-pool*. In [15] we give formulae for deriving the affect-pool and origin-pool of a data item t in the extent of a materialised construct c created by a transformation step of the form add(c,q) applied to a schema S. These formulae generate *derivation tracing queries* [11] $q_S^{AP}(t)$ and $q_S^{OP}(t)$ which can be applied to S in order to respectively obtain the affect-pool and origin-pool of the data item t.

In [15] we give an algorithm for tracing the affect-pool and origin-pool of a materialised data item t all the way back to the data sources by using the AutoMed pathways from the data source schemas $\mathrm{DSS}_1, \ldots, \mathrm{DSS}_n$ to the warehouse schema. This algorithm traverses a pathway backwards, and incrementally computes new affect- and origin-pools whenever an add or rename step is encountered, finally ending with the required affect- and origin-pools for t from within $\mathrm{DSS}_1, \ldots, \mathrm{DSS}_n$.

5 Discussion

We have shown how AutoMed metadata can be used to express the ETL process in a data warehouse and how the resulting transformation pathways can be used for some key warehouse activities. There are three main differences between this approach and the traditional data warehousing approach based on a single conceptual data model (CDM):

1. In the CDM approach, each data source wrapper translates the data source model into the CDM. Since both are likely to be high-level conceptual models, semantic mismatches may exist between the CDM and the source data model, and there may be a loss of information between them. In contrast, with the AutoMed approach the data source wrappers translate each data source schema into its equivalent AutoMed representation, without loss of information. Any necessary inter-model translation then happens explicitly within the AutoMed transformation pathways, under the control of the data warehouse designer.

2. In the CDM approach, the data transformation and integration metadata is tightly coupled with the CDM of the particular data warehouse. If the data warehouse is to be

redeployed on a platform with a different CDM, it is not easy to reuse the previous data transformation and implementation effort. In contrast, with the AutoMed approach it is possible to extend the existing pathways from the data source schemas $\mathrm{DSS}_1, \ldots, \mathrm{DSS}_n$ to the current detailed data warehouse schema, DS, with extra transformation steps that evolve DS into a new schema DS^{new}, expressed in the data model of the new data warehouse implementation. The pathway $\mathrm{DS} \rightarrow \mathrm{DS}^{new}$ can be used to populate the detailed schema of the new data warehouse from the current warehouse. The downstream schemas from DS^{new} (i.e. the summary views and the data mart schemas) do still have to be defined again by means of new pathways from DS^{new}, but all the upstream data transformation/integration metadata can be reused. In particular, the pathways from $\mathrm{DSS}_1, \ldots, \mathrm{DSS}_n$ to DS^{new} (via DS) can be used to maintain the new data warehouse.

3. In the CDM approach, if a data source schema changes it is not straightforward to evolve the view definitions of the data warehouse constructs. With the AutoMed approach, a change of a data source schema DSS_i into a new schema DSS_i^{new} can be expressed as a transformation pathway $\mathrm{DSS}_i \rightarrow \mathrm{DSS}_i^{new}$. The (automatically derivable) reverse pathway $\mathrm{DSS}_i^{new} \rightarrow \mathrm{DSS}_i$ can then be prefixed to the original pathway $\mathrm{DSS}_i \rightarrow \mathrm{TS}_i$ to give a pathway $\mathrm{DSS}_i^{new} \rightarrow \mathrm{TS}_i$, thus extending the transformation network of Figure 2 to encompass the new schema.

   Let us examine the impact of this extension. Suppose that the pathway $\mathrm{DSS}_i \rightarrow \mathrm{DSS}_i^{new}$ consists of a single primitive transformation t (longer pathways can be treated as a sequence of single-step pathways). There are three cases to consider for t, the first two of which can be handled totally automatically and the third semi-automatically:

   (i) If t is an add, delete or rename transformation, then DSS_i^{new} is semantically equivalent to DSS_i and no further change to the transformation network is needed. When new data is extracted from DSS_i^{new}, the new pathway $\mathrm{DSS}_i^{new} \rightarrow \mathrm{TS}_i$ can be used to transmit the new data to TS_i.

   (ii) If t is of the form contract(c) then construct c will no longer be available from DSS_i^{new}. The transformation network needs to be modified to remove all downstream constructs directly or indirectly dependent on c (and their underlying extents, if they have been materialised). This is done by first removing the initial extend(c) step from DSS_i^{new} to DSS_i (and as a result removing also DSS_i) and then examining the query accompanying each subsequent add step and the constructs referenced in each subsequent rename step.

   (iii) If t is of the form extend(c) then there will be new data available from DSS_i^{new} that was not available before. If the transformation network remains as it is, then the first step from DSS_i^{new} to DDS_i is contract(c) which removes c. Thus, the transformation network is still consistent, but it does not utilise the new data.

   It may be the case that we want to evolve the warehouse detailed schema, DS, to include this new data. In this case, we can simply remove the contract(c) step (and hence also DSS_i) from the transformation network. This has the effect of automatically propagating the construct c to all the downstream schemas in the transformation network. Some further manual modification of the network may also be necessary e.g. removing c from the data mart schemas by inserting a contract(c) step in the pathway from the DWS to a data mart schema; 'cleaning' c by inserting extra transformation steps before SS_i and/or before the appropriate multi-source cleaned schemas; semantically integrating c with existing data by inserting extra transformation steps before the detailed schema, DS; utilising c in new view definitions etc.

6 Concluding Remarks

In this paper we have discussed the use of AutoMed metadata in data warehousing environments. We have shown how AutoMed metadata can be used to express the data schemas, and the data cleansing, transformation, and integration processes. We have shown how this metadata can then be used for populating the data warehouse, incrementally maintaining the warehouse data after data source updates, and tracing the lineage of warehouse data.

In contrast to the traditional data warehousing approach which adopts a single conceptual data model to transform and integrate data from multiple heterogeneous data sources, we use a low-level common data model, the HDM. Data source wrappers first translate the source schemas into their equivalent specification in terms of the HDM. AutoMed's transformation pathways are then used to incrementally transform and integrate the source schemas into an integrated schema. The integrated schema can be defined in any modelling language which has been specified in terms of AutoMed's HDM. We discussed in Section 5 how several benefits result: no semantic mismatch between the data source schemas and their representation in the HDM; support of evolution of the data source schemas; and reuse of much of the transformation/integration effort if the data warehouse is redeployed on a platform supporting a different data model.

So far, our incremental view maintenance and data lineage tracing algorithms have used only the add and rename transformation steps from the data sources to the integrated schema. We are now looking at how to also make use of the information imparted by the queries wthin delete and contract transformation steps. We are also looking at the impact of calls to external functions on our incremental view maintenance and data lineage tracing algorithms.

Clearly, not all data warehouse metadata can be captured by AutoMed e.g. information about physical organisation of the data, ownership of the data, access control, temporal information, and data refresh and purging policies. Thus, in practice, we envisage AutoMed being used alongside an existing DBMS supporting such facilities. We are currently investigating this interaction of AutoMed with existing data warehousing functionality in the context of a data warehousing project in the bioinformatics domain. This project is creating an integrated data warehouse in Oracle from the CATH database[4], the MSD database[5] and other specialist data sources. It is also extracting specialist data marts from this warehouse, tailored for individual researchers' needs and deployed in lighter-weight DBMSs such as Postgres or MySQL. The data sources are evolving over time, as are the researchers' requirements for their data marts. Thus, this application is well-suited to the functionality that AutoMed offers.

References

[1] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Simeon, and S. Zohar. Tools for data translation and integration. *IEEE Data Engineering Bulletin*, 22(1):3–8, 1999.

[2] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[3] N. T. Binh, A. M. Tjoa, and O. Mangisengi. Meta Cube-X: An XML metadata foundation for interoperability search among web data warehouses. In *DMDW'01, Switzerland*, page 8, 2001.

[4] Lars Bkgaard. Event-entity-relationship modeling in data warehouse environments. In *DOLAP '99, USA*, pages 9–14. ACM, 1999.

[4] `http://www.biochem.ucl.ac.uk/bsm/cath_new/`

[5] `http://www.ebi.ac.uk/msd/`

[5] M. Boyd, P. McBrien, and N. Tong. The AutoMed schema integration repository. In *Proc. of BNCOD'02, Springer Verlag LNCS 2405, Sheffied*, pages 42–45, 2002.

[6] P. Buneman, S. Khanna, and W.C. Tan. Why and Where: A characterization of data provenance. In *Database Theory - ICDT'01, UK*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[7] D. Calvanese, G. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. A principled approach to data integration and reconciliation in data warehousing. In *DMDW'99, Germany*, page 16, 1999.

[8] L. Carneiro and A. Brayner. X-META: A methodology for data warehouse design with metadata management. In *DMDW'02*, pages 13–22, 2002.

[9] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[10] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB'01, Italy*, pages 471–480. Morgan Kaufmann, 2001.

[11] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

[12] G. Dong. Incremental maintenance of recursive views: A survey. In A Gupta and I. S. Mumick, editors, *Materialized Views: Techniques, Implementations, and Applications*, pages 159–162. The MIT Press, London, 1999.

[13] H. Fan. Incremental view maintenance and data lineage tracing in heterogeneous database environments. In *Proc. BNCOD'02 PhD Summer School, Sheffied*, pages 14–21, 2002.

[14] H. Fan. Tracing data lineage using AutoMed transformation pathways. In *BNCOD'2002, Springer Verlag LNCS 2405*, pages 50–53, 2002.

[15] H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. In *Knowledge Transformation for the Semantic Web*, pages 64–79. IOS Press, 2003.

[16] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: An extensible data cleaning tool. In *ACM SIGMOD*, page 590. ACM, 2000.

[17] M. Golfarelli and S. Rizzi. A methodological framework for data warehouse design. In *DOLAP '98, USA,*, pages 3–9. ACM, 1998.

[18] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering*, 18(2):3–18, 1995.

[19] Holger Hinrichs and Thomas Aden. An ISO 9001: 2000 compliant quality management system for data integration in data warehouse systems. In *DMDW'01, Switzerland*, page 1, 2001.

[20] B. Hsemann, J. Lechtenbrger, and G. Vossen. Conceptual data warehouse modeling. In *DMDW'00, Sweden*, page 6, 2000.

[21] T. N. Huynh, O. Mangisengi, and A. M. Tjoa. Metadata for Object-Relational data warehouse. In *DMDW'00, Sweden*, page 3, 2000.

[22] E. Jasper. Global query processing in the AutoMed heterogeneous database environment. In *Proc. of BNCOD'02, Springer Verlag LNCS 2405, Sheffied*, pages 46–49, 2002.

[23] R. Kimball, L. Reeves, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Depolying Data Warehousees.* John Wiley & Sons, Inc., 1998.

[24] P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *CAiSE'99, Germany*, volume 1626 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 1999.

[25] P. McBrien and A. Poulovassilis. Data integration by Bi-directional schema transformation rules. In *ICDE'03*, 2003.

[26] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB'98, USA*, pages 122–133. Morgan Kaufmann, 1998.

[27] D. Moody and M. Kortink. From enterprise models to dimensional models: a methodology for data warehouse and data mart design. In *DMDW'00, Sweden*, page 5, 2000.

[28] R. Müller, T. Stöhr, and E. Rahm. An integrative and uniform model for metadata management in data warehousing environments. In *DMDW'99, Heidelberg, Germany*, page 12, 1999.

[29] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE'95*, pages 251–260. IEEE Computer Society, 1995.

[30] A. Poulovassilis. The AutoMed Intermediate Query Language. Technical report, Automed Project, 2001.

[31] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[32] L. Seligman, A. Rosenthal, P. Lehner, and A. Smith. Data integration: Where does the time go? *IEEE Data Engineering Bulletin*, 25(3):3–10, 2002.

[33] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[34] M. Staudt, A. Vaduva, and T. Vetterli. Metadata management and data warehousing. Technical Report ifi-99.04, Department of Information Technology, University of Zurich, 1999.

[35] N. Tryfona, F. Busborg, and J. Christiansen. starER: A conceptual model for data warehouse design. In *DOLAP '99, USA*, pages 3–8. ACM, 1999.

[36] A. Tsois, N. Karayannidis, and T. Sellis. MAC: Conceptual data modeling for OLAP. In *DMDW'01, Switzerland*, page 5, 2001.

[37] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP'02*, pages 14–21. ACM, 2002.

[38] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Modeling ETL activities as graphs. In *DMDW'02, Canada*, pages 52–61, 2002.

[39] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ACDE'97, UK*, pages 91–102. IEEE Computer Society, 1997.