View Generation and Optimisation in the AutoMed Data Integration Framework

Edgar Jasper[1], Nerissa Tong[2], Peter McBrien[2], and Alexandra Poulovassilis[1]

[1] School of Computer Science and Information Systems, Birkbeck College, Univ. of London, {edgar,ap}@dcs.bbk.ac.uk
[2] Dept. of Computing, Imperial College, {nnyt98,pjm}@doc.ic.ac.uk

Abstract. This paper describes view generation and view optimisation in the AutoMed heterogeneous data integration framework. In AutoMed, schema integration is based on the use of reversible schema transformation sequences. We show how views can be generated from such sequences, either for global as view (GAV) query processing or for local as view (LAV) query processing. We also present techniques for optimising these generated views, firstly by optimising the transformation sequences, and secondly by optimising the view definitions generated from them.

1 Introduction

Data integration is a process by which several databases, with associated **local schemas**, are integrated to form a single virtual database with an associated **global schema**. Up to now, most data integration approaches have been either **global as view (GAV)** or **local as view (LAV)** [7].

In GAV, the constructs of a global schema are described as views over the local schemas. These view definitions are used to rewrite queries over a global schema into distributed queries over the local databases. Examples of the GAV approach are TSIMMIS [3], InterViso [20] and Garlic [19]. The principal disadvantage of GAV is that it does not readily support the evolution of the local schemas.

In LAV, the constructs of the local schemas are defined as views over the global schema, and processing queries over the global schema involves rewriting queries using views [8]. Examples of the LAV approach are IM [9] and Agora [10]. LAV isolates changes to local schemas to impact only on the derivation rules defined for that schema. However, it has problems if one needs to change the global schema, since all the rules for defining local schemas as views of the global schema will need to be reviewed. An in-depth comparison of GAV and LAV is given in [6].

In [16], we presented a unifying framework for GAV and LAV which is based on the use of reversible sequences of primitive schema transformations, called transformation **pathways**. An important feature of our approach is that it is possible to extract a definition of the global schema as a view over the local

schemas, and it is also possible to extract definitions of the local schemas as views over the global schema. Hence we term our approach **both as view (BAV)**.

We have implemented the BAV data integration approach within the AutoMed system (see `http://www.doc.ic.uk/automed`). BAV combines the benefits of both GAV and LAV in the sense that any reasoning or processing which is possible with the view definitions of GAV or LAV will also be possible with the BAV definition. Moreover, as discussed in [15, 16], one advantage of our BAV approach over GAV and LAV is that it readily supports the evolution of both global and local schemas, allowing pathways and schemas to be incrementally modified as opposed to having to be regenerated.

Since the BAV integration approach is based on sequences of primitive schema transformations, it could be argued that the pathways resulting from BAV are likely to be more costly to reason with and process (e.g. for global query processing) than the corresponding LAV or GAV view definitions would be. However, in Section 3 of this paper we show how BAV pathways are amenable to considerable simplification. Moreover, standard query optimisation techniques can be applied to the view definitions derived from BAV pathways, and we discuss these in Section 5 of this paper.

The outline of this paper is thus as follows. We begin with a brief review of the BAV integration approach in Section 2, and give an example of its use. We present techniques for optimising BAV pathways in Section 3. We then show how view definitions can be generated from BAV pathways, either for GAV query processing or for LAV query processing, in Section 4. We then present techniques for optimising these generated views in Section 5. Section 6 gives our concluding remarks and directions of further work.

2 The BAV Integration Approach

In previous work [18, 12] we have developed a general framework to support schema transformation and integration in heterogeneous database architectures. The framework consists of a low-level **hypergraph-based data model (HDM)** and a set of primitive schema transformations defined for this model. Higher-level data models and primitive schema transformations for them are defined in terms of this lower-level common data model.

In our framework, 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. 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**, whose basic collection type is the bag (multi-set) type. All primitive transformations have an optional additional argument which specifies a constraint on the current schema extension that must hold if the transformation is to be applied. Constraints are also expressed as IQL queries.

A **composite transformation** is a sequence of primitive transformations. We term the composite transformation defined for transforming schema S_1 to schema S_2 a transformation **pathway** $S_1 \rightarrow S_2$. All source schemas, intermediate schemas and global schemas, and the pathways between them are stored in AutoMed's metadata repository [1].

AutoMed supports many methodologies for performing data integration and hence forming a **network** of pathways joining schemas together. Here we assume a simple methodology based on forming union-compatible schemas, the general structure of which is illustrated in Figure 1. In order to integrate n local schemas, $LS_1, \ldots, LS_n$, each LS_i first needs to be transformed into a "union" schema US_i. These n union schemas are syntactically identical, and this is asserted by creating a sequence of id transformation steps between each pair US_i and US_{i+1}, of the form id $(US_i\!:\!c, US_{i+1}\!:\!c)$ for each schema construct (id is an additional type of primitive transformation, and the notation $US_i : c$ distinguishes each schema's c construct; an id transformation is reversed by swapping its two arguments). These id transformations are generated automatically by the AutoMed software. An arbitrary one of the US_i can then be selected for further transformation into a global schema GS. This is where constructs sourced from different local schemas can be combined together by unions, joins, outer-joins etc.

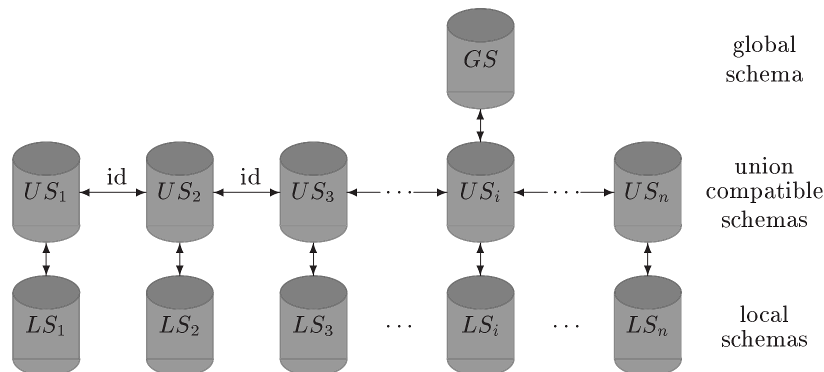

Fig. 1. A general AutoMed Integration

There may be information within a US_i which is not semantically derivable from the corresponding LS_i. This is asserted by means of extend transformation steps within the pathway $LS_i \rightarrow US_i$. Conversely, not all of the information within a local schema LS_i need be transferred into US_i and this is asserted by means of contract transformation steps within $LS_i \rightarrow US_i$. extend and contract transformations behave in the same way as add and delete, respectively, except that they indicate that their accompanying query may only partially construct the extent of the new/removed 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.

Each primitive transformation t has an **automatically derivable** reverse transformation $\bar{t}$. In particular, each add or extend transformation is reversed by a delete or contract transformation with the same arguments, and vice versa, while each rename or id transformation is reversed by another rename or id transformation with the two arguments swapped. This holds for the primitive transformations of **any** modelling language defined in AutoMed.

In [13] we described how our framework can be applied to different high-level modelling languages such as relational, ER and UML. The approach was extended to encompass XML data sources in [14], formatted data files [5] and RDF [22]. We refer the reader to [16] for an extensive discussion of the AutoMed integration approach.

For our purposes in the present paper, we assume that all schemas are specified in the very simple relational data model that we define below. But we stress that the techniques that we describe here are equally applicable to any data modelling language supported by AutoMed.

Schemas in our simple relational model are constructed from primary key attributes, non-primary key attributes, and the relationships between them. Figure 2 illustrates the representation of a relation R with primary key attributes $k_1, ..., k_n$ and non-primary key attributes $a_1, ..., a_m$. There is a one-one correspondence between this representation and the underlying HDM graph. In our simple relational model, there are two kinds of schema construct: Rel and Att (for simplicity, we ignore here the constraints present in a relational schema but see [12] for an encoding of a richer relational data model).

The extent of a Rel construct $\langle\langle R\rangle\rangle$ is the projection of the relation R onto its primary key attributes $k_1, ..., k_n$. The extent of each Att construct $\langle\langle R, a\rangle\rangle$ where a is an attribute (key or non-key) is the projection of relation R onto $k_1, ..., k_n, a$. For example, a relation student(id,sex,dname) would be modelled by a Rel construct ⟨⟨student⟩⟩, and three Att constructs ⟨⟨student,id⟩⟩, ⟨⟨student,sex⟩⟩ and ⟨⟨student,dname⟩⟩.

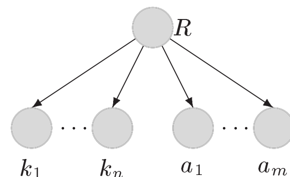

Fig. 2. A simple relational data model

The set of primitive transformations for schemas expressed in this simple relational data model is as follows:

- addRel($\langle\langle R\rangle\rangle, q$) adds to the schema a new relation R. The query q specifies the set of primary key values in the extent of R in terms of the already existing schema constructs.
- addAtt($\langle\langle R, a\rangle\rangle, q$) adds to the schema an attribute a (key or non-key) for relation R. The query q specifies the extent of the binary relationship between

the primary key attribute(s) of R and this new attribute a in terms of the already existing schema constructs.
- deleteRel($\langle\langle R\rangle\rangle, q$) deletes from the schema the relation R (provided all its attributes have first been deleted). The query q specifies how the extent of R can be restored from the remaining schema constructs.
- deleteAtt($\langle\langle R, a\rangle\rangle, q$) deletes from the schema attribute a of relation R. The query q specifies how the extent of the binary relationship between the primary key attribute(s) of R and a can be restored from the remaining schema constructs.
- renameRel($\langle\langle R\rangle\rangle, \langle\langle R'\rangle\rangle$) renames the relation R to R' in the schema.
- renameAtt($\langle\langle R, a\rangle\rangle, \langle\langle R, a'\rangle\rangle$) renames the attribute a of R to a'.

In [12] we show how the reversibility of schema transformations allows automatic query translation between schemas. In Section 4 we discuss how this translation scheme can be extended in order to derive a view definition for each global schema construct in terms of the source schema constructs. These derivations can then be substituted into any query posed on a global schema in order to obtain an equivalent query distributed over the local schemas: this GAV approach to global query processing is what we currently support within the AutoMed implementation. However, the same translation scheme could equivalently be applied to each of the constructs of a local schema in order to obtain its derivation from a global schema, as in the LAV approach, and we discuss this too in Section 4.

2.1 An Example Integration

Figure 3 gives some specific schemas to illustrate the integration approach of Figure 1. Primary key attributes are underlined, foreign key attributes are in italics and nullable attributes are suffixed by #.

For conciseness, we only list here in Example 1 the pathway $\mathsf{LS}_2 \rightarrow \mathsf{US}$ and in Example 2 the pathway $\mathsf{US} \rightarrow \mathsf{GS}$ (all the pathways are listed in [11]). In Example 1, transformations t_1–t_{10} use *extend* transformations to state that the tables student, college and degree in US cannot be derived from LS_2. Then t_{11}–t_{14} use the dname attribute of person to derive the dept table in US, and use *extend* transformations for the two attributes street and cname that cannot be derived from LS_2. Finally, in t_{15}–t_{19} the male and female relations of LS_2 are restructured into the single sex attribute of staff.

In IQL, ++ is the bag union operator and the construct $[e \mid Q_1; \ldots Q_n]$ is a **comprehension** [2]. The expressions Q_1 to Q_n are termed **qualifiers**, each qualifier being either a **filter** or a **generator**. A filter is a boolean-valued expression. A generator has syntax $p \leftarrow c$ where p is a **pattern** and c is a bag-valued expression. In IQL, the patterns p are restricted to be single variables or tuples of variables.

LS$_1$
dept(dname)
staff(id,name,sex,*dname*)

LS$_2$
staff(id,name,dname)
male(*id*)
female(*id*)

LS$_3$
dept(deptname)
degree(dcode,title,*dname*)
person(id,*dname*)
male(*id*)
female(*id*)

LS$_4$
dept(dname)
student(id,sex,*dname*)
degree(dcode,*dname*)

LS$_5$
university(uname)
college(cname,*uname*)
dept(dname,street,*cname*)
staff(id,name,sex,*dname*)

US
college(cname)
dept(dname,street,*cname*)
degree(dcode,title,*dname*)
staff(id,name,sex,*dname*)
student(id,sex,*dname*)

GS
college(cname)
dept(dname,street,*cname*)
degree(dcode,title,*dname*)
person(id,name#,sex,*dname*)

Fig. 3. Example schemas

Example 1 Pathway LS$_2$ → US

t_1 extendRel(⟨⟨student⟩⟩)
t_2 extendAtt(⟨⟨student,id⟩⟩)
t_3 extendAtt(⟨⟨student,sex⟩⟩)
t_4 extendAtt(⟨⟨student,dname⟩⟩)
t_5 extendRel(⟨⟨college⟩⟩)
t_6 extendAtt(⟨⟨college,cname⟩⟩)
t_7 extendRel(⟨⟨degree⟩⟩)
t_8 extendAtt(⟨⟨degree,dcode⟩⟩)
t_9 extendAtt(⟨⟨degree,title⟩⟩)
t_{10} extendAtt(⟨⟨degree,dname⟩⟩)
t_{11} addRel(⟨⟨dept⟩⟩, [x | (y, x) ← ⟨⟨staff,dname⟩⟩])
t_{12} addAtt(⟨⟨dept,dname⟩⟩, [(x, x) | x ← ⟨⟨dept⟩⟩])
t_{13} extendAtt(⟨⟨dept,street⟩⟩)
t_{14} extendAtt(⟨⟨dept,cname⟩⟩)
t_{15} addAtt(⟨⟨staff,sex⟩⟩, [(x,′ M′) | x ← ⟨⟨male⟩⟩] ++ [(x,′ F′) | x ← ⟨⟨female⟩⟩])
t_{16} deleteAtt(⟨⟨male,id⟩⟩, [(x, x) | x ← ⟨⟨male⟩⟩])
t_{17} deleteRel(⟨⟨male⟩⟩, [x | (x,′ M′) ← ⟨⟨staff,sex⟩⟩])
t_{18} deleteAtt(⟨⟨female,id⟩⟩, [(x, x) | x ← ⟨⟨female⟩⟩])
t_{19} deleteRel(⟨⟨female⟩⟩, [x | (x,′ F′) ← ⟨⟨staff,sex⟩⟩])

□

The pathway LS$_1$ → US involves just the *extend* transformations from LS$_2$ → US. The pathway LS$_3$ → US contains *extend* steps t_{20},t_{21},t_{22},t_{23},t_{24},t_{25} to add the missing student and college tables, which are textually the same as t_1–t_6. It then renames deptname, adds the missing attributes of dept, renames person to staff, and adds the missing name attribute:

t_{26} renameAtt(⟪dept,deptname⟫, ⟪dept,dname⟫)
t_{27} extendAtt(⟪dept,street⟫)
t_{28} extendAtt(⟪dept,cname⟫)
t_{29} renameRel(⟪person⟫, ⟪staff⟫)
t_{30} extendAtt(⟪staff,name⟫)

Finally, in steps t_{31},t_{32},t_{33},t_{34},t_{35} it does the same restructuring as steps t_{15}–t_{19} of $\mathsf{LS}_2 \rightarrow \mathsf{US}$, converting the male and female relations into the single sex attribute of staff.

The pathway $\mathsf{LS}_4 \rightarrow \mathsf{US}$ contains a sequence of *extend* steps for its missing information. The pathway $\mathsf{LS}_5 \rightarrow \mathsf{US}$ contains a sequence of *extend* steps for its missing information and also three *contract* steps to remove the university relation and its attributes. Finally, we list below the pathway from the union schema US to the global schema GS:

Example 2 Pathway US → GS
t_{36} addRel(⟪person⟫, ⟪staff⟫ ++ ⟪student⟫)
t_{37} addAtt(⟪person,id⟫, ⟪staff,id⟫ ++ ⟪student,id⟫)
t_{38} addAtt(⟪person,name⟫, ⟪staff,name⟫)
t_{39} addAtt(⟪person,sex⟫, ⟪staff,sex⟫ ++ ⟪student,sex⟫)
t_{40} addAtt(⟪person,dname⟫, ⟪staff,dname⟫ ++ ⟪student,dname⟫)
t_{41} deleteAtt(⟪student,id⟫, [(x, y) | x ← ⟪student⟫; (x, y) ← ⟪person,id⟫])
t_{42} deleteAtt(⟪student,sex⟫, [(x, y) | x ← ⟪student⟫; (x, y) ← ⟪person,sex⟫])
t_{43} deleteAtt(⟪student,dname⟫, [(x, y) | x ← ⟪student⟫; (x, y) ← ⟪person,dname⟫])
t_{44} deleteRel(⟪student⟫), [x | x ← ⟪person⟫; not(member⟪staff⟫x)])
t_{45} deleteAtt(⟪staff,id⟫, [(x, y) | x ← ⟪staff⟫; (x, y) ← ⟪person,id⟫])
t_{46} deleteAtt(⟪staff,name⟫, [(x, y) | x ← ⟪staff⟫; (x, y) ← ⟪person,name⟫])
t_{47} deleteAtt(⟪staff,sex⟫, [(x, y) | x ← ⟪staff⟫; (x, y) ← ⟪person,sex⟫])
t_{48} deleteAtt(⟪staff,dname⟫, [(x, y) | x ← ⟪staff⟫; (x, y) ← ⟪person,dname⟫])
t_{49} deleteRel(⟪staff⟫, [x | x ← ⟪person⟫; (x, y) ← ⟪person,name⟫])

□

3 Optimising AutoMed Transformation Pathways

One important feature of the AutoMed approach is that once a set of schemas have been joined in a network of pathways, data and queries may be translated or migrated between any pair of schemas in the network. For example, once $\mathsf{LS}_2 \rightarrow \mathsf{US}$ and $\mathsf{LS}_3 \rightarrow \mathsf{US}$ are defined, we can derive $\mathsf{LS}_2 \rightarrow \mathsf{LS}_3$ as $t_1, \ldots, t_{19}, \overline{t_{35}}, \ldots, \overline{t_{20}}$ and $\mathsf{LS}_3 \rightarrow \mathsf{LS}_2$ as $t_{20}, \ldots, t_{35}, \overline{t_{19}}, \ldots, \overline{t_1}$.

Since such networks may be complex to analyse, we need to support automated verification that a network is well-formed. We also need to support automated simplification of the pathways between schemas, since they may contain redundant transformations. Taking $\mathsf{LS}_2 \rightarrow \mathsf{LS}_3$ as an example, the two relations male and female in LS_2 are first converted to an attribute sex in US, and then converted back to relations in LS_3, where clearly there could instead have been a direct mapping between the relations in the two local schemas.

To support such analysis of pathways for being well-formed and for simplification, we have developed the **Transformation Manipulation Language (TML)** [21], which is designed to represent each transformation step in a form suited to analysis of the schema constructs that are created, deleted or are required to be present for the transformation step to be correct. Our definitions below require a function *sc* that given a query determines all schema constructs that must exist if the query is valid. For the IQL language constructs used in our earlier examples, *sc* is defined as:

$$sc(\langle\langle \mathsf{r} \rangle\rangle) = \langle\langle \mathsf{r} \rangle\rangle$$
$$sc(\langle\langle \mathsf{r,a} \rangle\rangle) = \{\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r,a} \rangle\rangle\}$$
$$sc([q_1, \ldots, q_n]) = sc(q_1) \cup \ldots \cup sc(q_n)$$
$$sc(q_1 \mathbin{++} q_2) = sc(q_1) \cup sc(q_2)$$
$$sc([q \mid q_1, \ldots, q_n]) = sc(q) \cup sc(q_1) \cup \ldots \cup sc(q_n)$$

TML formalises each transformation step t_n transforming a schema S_x into a schema S_y as having four **conditions** $a_n^+, b_n^-, c_n^+, d_n^-$:

- The positive precondition a_n^+ is the set of constructs that t_n implies must be present in S_x. It comprises those constructs that are present in the query of the transformation (given by $sc(q)$) together with any constructs implied as being present by the construct c:
  $t_n \in \{add(c,q), extend(c,q)\} \rightarrow a_n^+ = (sc(c) - c) \cup sc(q)$
  $t_n \in \{delete(c,q), contract(c,q), rename(c,c'), id(c,c')\} \rightarrow a_n^+ = sc(c) \cup sc(q)$
- The negative precondition b_n^- is the set of constructs that t_n implies must not be present in S_x. It comprises those constructs which the transformation will add to the schema, and thus must not already be present:
  $t_n \in \{add(c,q), extend(c,q), rename(c',c), id(c',c)\} \rightarrow b_n^- = c$
  $t_n \in \{delete(c,q), contract(c,q)\} \rightarrow b_n^- = \emptyset$
- The positive postcondition c_n^+ is the set of constructs that t_n implies must be present in S_y, and is derived in the same way as $\overline{a_n^+}$ (*i.e.* the positive precondition of the $\overline{t_n}$):
  $t_n \in \{add(c,q), extend(c,q), rename(c',c), id(c',c)\} \rightarrow c_n^+ = sc(c) \cup sc(q)$
  $t_n \in \{delete(c,q), contract(c,q)\} \rightarrow c_n^+ = (sc(c) - c) \cup sc(q)$
- The negative postcondition d_n^- is the set of constructs that t_n implies must not be present in S_y, and is derived in the same way as $\overline{b_n^-}$:
  $t_n \in \{delete(c,q), contract(c,q), rename(c,c'), id(c,c')\} \rightarrow d_n^- = c,$
  $t_n \in \{add(c,q), extend(c,q)\} \rightarrow d_n^- = \emptyset$

Thus we can express $\mathsf{LS}_2 \rightarrow \mathsf{US}$ in TML as follows:

$t_1 : [\emptyset, \{\langle\langle \mathsf{student} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle\}, \emptyset]$
$t_2 : [\emptyset, \{\langle\langle \mathsf{student,id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student,id} \rangle\rangle\}, \emptyset]$
$t_3 : [\emptyset, \{\langle\langle \mathsf{student,id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student,sex} \rangle\rangle\}, \emptyset]$
$t_4 : [\emptyset, \{\langle\langle \mathsf{student,id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student,dname} \rangle\rangle\}, \emptyset]$
$\vdots$
$t_{11} : [\{\langle\langle \mathsf{staff} \rangle\rangle, \langle\langle \mathsf{staff,dname} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle, \langle\langle \mathsf{staff} \rangle\rangle, \langle\langle \mathsf{staff,dname} \rangle\rangle\}, \emptyset]$
$t_{12} : [\{\langle\langle \mathsf{dept} \rangle\rangle\}, \{\langle\langle \mathsf{dept,dname} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle, \langle\langle \mathsf{dept,dname} \rangle\rangle\}, \emptyset]$

$t_{13} : [\{\langle\langle \mathsf{dept} \rangle\rangle\}, \{\langle\langle \mathsf{dept,street} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle, \langle\langle \mathsf{dept,street} \rangle\rangle\}, \emptyset]$
$t_{14} : [\{\langle\langle \mathsf{dept} \rangle\rangle\}, \{\langle\langle \mathsf{dept,cname} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle, \langle\langle \mathsf{dept,cname} \rangle\rangle\}, \emptyset]$
$t_{15} : [\{\langle\langle \mathsf{staff} \rangle\rangle\}, \{\langle\langle \mathsf{staff,sex} \rangle\rangle\}, \{\langle\langle \mathsf{staff} \rangle\rangle, \langle\langle \mathsf{staff,sex} \rangle\rangle\}, \emptyset]$
$t_{16} : [\{\langle\langle \mathsf{male} \rangle\rangle, \langle\langle \mathsf{male,id} \rangle\rangle\}, \emptyset, \{\langle\langle \mathsf{male} \rangle\rangle\}, \{\langle\langle \mathsf{male,id} \rangle\rangle\}]$
$t_{17} : [\{\langle\langle \mathsf{male} \rangle\rangle\}, \emptyset, \emptyset, \{\langle\langle \mathsf{male} \rangle\rangle\}]$
$t_{18} : [\{\langle\langle \mathsf{female} \rangle\rangle, \langle\langle \mathsf{female,id} \rangle\rangle\}, \emptyset, \{\langle\langle \mathsf{female} \rangle\rangle\}, \{\langle\langle \mathsf{female,id} \rangle\rangle\}]$
$t_{19} : [\{\langle\langle \mathsf{female} \rangle\rangle\}, \emptyset, \emptyset, \{\langle\langle \mathsf{female} \rangle\rangle\}]$

3.1 Well-formed Transformation Pathways

A pathway T is said to be **well-formed** if for each transformation step $t_n : S_x \to S_y$ within it:

- The only difference between the schema constructs in S_y and S_x is those constructs specifically changed by transformation t_n, implying that $S_y = (S_x \cup c_n^+) - d_n^-$ and $S_x = (S_y \cup a_n^+) - b_n^-$
- The constructs required by t_n are in the schemas, implying that $a_n^+ \subseteq S_x$, $b_n^- \cap S_x = \emptyset$, $c_n^+ \subseteq S_y$ and $d_n^- \cap S_y = \emptyset$

The above definition leads to the recursive definition of a well-formed pathway, wf, given below. The first rule applies each transformation step in turn, and the second rule ensures that the schema that results from applying all the transformation steps is equal to the schema at the end of the pathway (equal both in terms of the schema constructs found in each schema and the extent of the schemas).

$$\begin{aligned} wf(S_m, S_n, [t_m, t_{m+1} \ldots, t_n]) &\leftarrow a_m^+ \subseteq S_m \wedge b_m^- \cap S_m = \emptyset \wedge \\ &\quad wf((S_m \cup c_m^+) - d_m^-, S_n, [t_{m+1} \ldots, t_n]) \\ wf(S_m, S_n, []) &\leftarrow S_m = S_n \wedge Ext(S_m) = Ext(S_n) \end{aligned}$$

3.2 Reordering of Transformations

Certain transformations may be performed in any order, whilst others must be performed in a specific order. For example, in $\mathsf{LS}_2 \to \mathsf{US}$, t_{16} must be performed before t_{17}, since the attribute $\langle\langle \mathsf{male,id} \rangle\rangle$ must be deleted before the $\langle\langle \mathsf{male} \rangle\rangle$ relation is deleted. However the sub-pathway t_{16},t_{17} could be performed before or after the sub-pathway t_{18},t_{19} since it does not matter which of the $\langle\langle \mathsf{male} \rangle\rangle$ or $\langle\langle \mathsf{female} \rangle\rangle$ relations is deleted first.

In the TML, this intuition is expressed by stating that transformations may be swapped provided the pathway remains well-formed. This may be verified by inspecting the conditions of each transformation. In particular, a pair of transformations t_n,t_{n+1} may be reordered to t_{n+1},t_n provided:

1. t_n does not add a construct that is required by t_{n+1}, and $\overline{t_{n+1}}$ does not add a construct that is required by $\overline{t_n}$, *i.e.* $(c_n^+ - a_n^+) \cap a_{n+1}^+ = \emptyset$ and $(a_{n+1}^+ - c_{n+1}^+) \cap c_n^+ = \emptyset$

2. t_n does not delete a construct required not to be present by t_{n+1}, and $\overline{t_{n+1}}$ does not delete a construct required not to be present by $\overline{t_n}$, *i.e.* $d_n^+ \cap b_{n+1}^+ = \emptyset$

We can now formalise the two examples given above from $\mathsf{LS}_2 \to \mathsf{US}$. For t_{16},t_{17}, (1) is broken, and hence they may not be swapped. The changing of t_{16},t_{17},t_{18},t_{19} to t_{18},t_{19},t_{16},t_{17} may be performed by iteratively swapping pairs of transformations. Considering first t_{17},t_{18}, we find neither rule is broken, and they may be reordered to t_{18},t_{17}. Then t_{17},t_{19} breaks neither rule, and may be reordered to t_{19},t_{17}. This leaves a sub-pathway t_{16},t_{18},t_{19},t_{17}, and a similar argument allows t_{16} swap with t_{18} and then t_{19}, to give the sub-pathway t_{18},t_{19},t_{16},t_{17}.

3.3 Redundant Transformations

Two transformations t_x and t_y in a well-formed pathway T are **redundant** if T may be reordered such that t_x and t_y become consequtive within it, and T remains well-formed if they are then removed. If we inspect the path $\mathsf{LS}_2 \to \mathsf{LS}_3$, it may be reordered to contain the sub-pathway:

t_{16} deleteAtt(⟪male,id⟫, [(x, x) | x ← ⟪male⟫])
t_{17} deleteRel(⟪male⟫, [x | (x,′ M′) ← ⟪staff,sex⟫])
$\overline{t_{33}}$ addRel(⟪male⟫, [x | (x,′ M′) ← ⟪staff,sex⟫])
$\overline{t_{32}}$ addAtt(⟪male,id⟫, [(x, x) | x ← ⟪male⟫])

Clearly t_{17},$\overline{t_{33}}$ forms a redundant pair, because we are adding and deleting the same construct *with the same extent* since the query is the same. Once this has been performed t_{16},$\overline{t_{32}}$ may be removed for the same reason.

Using the TML, we can identify redundant transformations as satisfying:

$$(a_x^+ = c_y^+) \land (b_x^- = d_y^-) \land (c_x^+ = a_y^+) \land (d_x^- = b_y^-) \land \\ Ext(c_x^+ \oplus a_x^+) = Ext(c_y^+ \oplus a_y^+)$$

where $(x \oplus y) = (x - y) \cup (y - x)$, and thus serves to find all the constructs being added or deleted by the pair of transformations. In practice, this rule means that any pair of transformations which add/extend and then delete/contract (in either order) the same construct are redundant, providing the query can be demonstrated to result in the same extent.

Example 3 Redundant transformation removal from $\mathsf{LS}_2 \to \mathsf{LS}_3$

t_7 extendRel(⟪degree⟫)
t_8 extendAtt(⟪degree,dcode⟫)
t_9 extendAtt(⟪degree,title⟫)
t_{10} extendAtt(⟪degree,dname⟫)
t_{11} addRel(⟪dept⟫, [x | (y, x) ← ⟪staff,dname⟫])
t_{12} addAtt(⟪dept,dname⟫, [(x, x) | x ← ⟪dept⟫])
$\overline{t_{30}}$ contractAtt(⟪staff,name⟫)
$\overline{t_{29}}$ renameRel(⟪staff⟫, ⟪person⟫)
$\overline{t_{26}}$ renameAtt(⟪dept,dname⟫, ⟪dept,deptname⟫)

□

3.4 Partially Redundant Transformations

Two transformations t_x and t_y in a well-formed pathway T are **partially redundant** if T may be reordered to make t_x and t_y consequtive, and T remains well-formed if they are then replaced by a single transformation t_{xy}.

The pathway in Example 3 has a pair of such partially redundant transformations, since it can be reordered to obtain the sub-pathway:

t_{12} addAtt(⟨⟨dept,dname⟩⟩, [(x, x) | x ← ⟨⟨dept⟩⟩])
$\overline{t_{26}}$ renameAtt(⟨⟨dept,dname⟩⟩, ⟨⟨dept,deptname⟩⟩)

This may be replaced by the new transformation given below, which leaves a fully optimised pathway $\mathsf{LS}_2 \rightarrow \mathsf{LS}_3$.

t_{50} addAtt(⟨⟨dept,deptname⟩⟩, [(x, x) | x ← ⟨⟨dept⟩⟩])

Using the TML, we can identify partially redundant transformations as satisfying:

$(a_x^+ = c_y^+ \oplus b_x^- = d_y^-) \wedge d_x^- \cap b_y^- = \emptyset \wedge d_x^- \neq \emptyset \wedge b_y^- \neq \emptyset$

The simplifications for removing partially redundant and fully redundant transformations are summarised in the table below. The table shows what simplifications may be applied where a pair of transformations is found to operate on the same construct c. NWF denotes 'not well-founded' and [ ] denotes the removal of the pair. The table would remain correct if *extend* were to replace *add*, *contract* replace *delete*, and *id* replace *rename*. Further details of redundant and partially redundant transformations may be found in [21].

		t_y		
		$add(c,q)$	$delete(c,q)$	$rename(c,c')$
	$add(c,q)$	NWF	[ ]	$add(c',q)$
t_x	$delete(c,q)$	[ ]	NWF	NWF
	$rename(c',c)$	NWF	$delete(c',q)$	[ ]
	$rename(c'',c)$	NWF	$delete(c'',q)$	$rename(c'',c')$

4 Generating Views

One of the strengths of the Automed architecture is that a number of different varieties of view can be generated automatically once the pathways are in place. All primitive transformations are automatically reversible. Thus, for any two schemas linked by primitive transformations, there exists a pathway between them, and it does not matter in which direction the pathway was originally created. Thus from the one set of pathways linking a set of schemas together, we can derive GAV views, LAV views, and indeed peer-to-peer views.

For GAV views, the pathways from the global schema to each local schema are retrieved from AutoMed's metadata repository. For some part of the start of their length these pathways may be the same as each other, as may be seen from the tree structure of Figure 1 and from our example schema integration in Section 2.1. Note that each node of this tree is a schema (local, global, or intermediate) which is linked to its neighbours by a single transformation step.

View definitions for each global schema construct are derived incrementally by traversing the tree. Initially, each construct's view definition is just the construct itself. Each node in the tree is then visited in a downwards direction. The only transformations that need to be considered are those that contract, delete or rename an extensional construct. These are relevant because the current view definitions may query extensional constructs that no longer exist after such a transformation. Each type of transformation is handled as follows if it is encountered during the tree traversal:

- A delete transformation: This has an associated query which shows how to reconstruct the extent of the construct being deleted. Any occurrence of the deleted construct within the current view definitions is replaced by this query.
- A sf contract transformation: If it has an associated reconstructing query, it is treated in the same way as a delete. Otherwise, any occurrence of the contracted construct within the current view definitions is replaced by Void.
- A rename transformation: All references to the old construct in the current view definitions are replaced by references to the new construct.

As this is not a linear pathway but a tree, at some points it will branch. When the tree branches, constructs of the parent schema at the branching point are semantically identical to constructs with the same scheme within the two child schemas. However, one (or both) of a pair of child constructs may later be made Void by contract transformations further down the tree. Thus, the possibility of using either path is retained within the view definitions by replacing each construct of the parent schema by a disjunction (OR) of the two corresponding constructs in the child schemas. For example, in the integration of Section 2.1, US_3 branches into US_2 and another intermediate schema (call it IS_3) on the pathway to LS_3. At this point, all occurrences of the construct $US_3 : \langle\langle staff \rangle\rangle$ within the current view definitions would be replaced by the query $US_2 : \langle\langle staff \rangle\rangle$ OR $IS_3 : \langle\langle staff \rangle\rangle$.

The tree is traversed in this fashion from the root to the leaves until all the nodes are visited. The resulting view definitions are the GAV definitions for the global schema constructs over the local schemas. Referring again to the example of Section 2.1, consider the construct $GS : \langle\langle person,sex \rangle\rangle$ in the global schema. The pathway $GS \rightarrow US$ would be processed first (*i.e.* the reverse of the pathway $US \rightarrow GS$ listed above). The only significant transformation is the one that deletes $\langle\langle person,sex \rangle\rangle$, resulting in an intermediate view definition $US : \langle\langle staff,sex \rangle\rangle ++ US : \langle\langle student,sex \rangle\rangle$ at one copy, US, of the five union schemas. Suppose that US is US_1. Traversing the pathways $US_1 \rightarrow LS_1$ and $US_1 \rightarrow US_2$, we get the new intermediate view definition:

$$(LS_1 : \langle\langle staff,sex \rangle\rangle \text{ OR } US_2 : \langle\langle staff,sex \rangle\rangle) \\ ++ (Void \text{ OR } US_2 : \langle\langle student,sex \rangle\rangle)$$

Traversing next $US_2 \rightarrow LS_2$ and $US_2 \rightarrow US_3$, we get:

$$
\begin{array}{l}
\quad (\ \mathsf{LS}_1 : \langle\langle \mathsf{staff,sex} \rangle\rangle \ \texttt{OR} \\
\quad\ ([(x, \text{`}M\text{'}) \mid x \leftarrow \mathsf{LS}_2 : \langle\langle \mathsf{male} \rangle\rangle] ++ [(x, \text{`}F\text{'}) \mid x \leftarrow \mathsf{LS}_2 : \langle\langle \mathsf{female} \rangle\rangle]) \ \texttt{OR} \\
\quad\ \mathsf{US}_3 : \langle\langle \mathsf{staff,sex} \rangle\rangle\,) \\
++ \ (\ \mathsf{Void} \ \texttt{OR} \ \mathsf{Void} \ \texttt{OR} \ \mathsf{US}_3 : \langle\langle \mathsf{student,sex} \rangle\rangle\,)
\end{array}
$$

Continuing with $\mathsf{US}_3 \rightarrow \mathsf{LS}_3$ and $\mathsf{US}_3 \rightarrow \mathsf{US}_4$, then $\mathsf{US}_4 \rightarrow \mathsf{LS}_4$ and $\mathsf{US}_4 \rightarrow \mathsf{US}_5$, and finally $\mathsf{US}_5 \rightarrow \mathsf{LS}_5$, we obtain a view definition for the global construct $\mathsf{GS} : \langle\langle \mathsf{person,sex} \rangle\rangle$ as:

$$
\begin{array}{l}
\quad (\ \mathsf{LS}_1 : \langle\langle \mathsf{staff,sex} \rangle\rangle \ \texttt{OR} \\
\quad\ ([(x, \text{`}M\text{'}) \mid x \leftarrow \mathsf{LS}_2 : \langle\langle \mathsf{male} \rangle\rangle] ++ [(x, \text{`}F\text{'}) \mid x \leftarrow \mathsf{LS}_2 : \langle\langle \mathsf{female} \rangle\rangle]) \ \texttt{OR} \\
\quad\ ([(x, \text{`}M\text{'}) \mid x \leftarrow \mathsf{LS}_3 : \langle\langle \mathsf{male} \rangle\rangle] ++ [(x, \text{`}F\text{'}) \mid x \leftarrow \mathsf{LS}_3 : \langle\langle \mathsf{female} \rangle\rangle]) \ \texttt{OR} \\
\quad\ \mathsf{Void} \ \texttt{OR} \ \mathsf{LS}_5 : \langle\langle \mathsf{staff,sex} \rangle\rangle\,) \\
++ \ (\ \mathsf{Void} \ \texttt{OR} \ \mathsf{Void} \ \texttt{OR} \ \mathsf{Void} \ \texttt{OR} \ \mathsf{LS}_4 : \langle\langle \mathsf{student,sex} \rangle\rangle \ \texttt{OR} \ \mathsf{Void})
\end{array}
$$

We will discuss shortly how this view definition can be simplified.

LAV views can be derived in a similar way. The pathway from a local schema to the global schema is again retrieved from the metadata repository. This pathway is processed as above to derive the view definitions, except that it is the local schema end of the pathway that is now taken as the root of the tree. The derivation of LAV views is simpler because there is now only a single pathway being processed, with no branching.

More generally, views can be derived for any schema in terms of any set of other schemas provided that pathways linking all the schemas exist. For example, we can derive views for the constructs of one local schema in terms of the constructs of another local schema. Such **peer-to-peer views** could be used for example to check that constructs which have been identified as having the same extent in the two local schemas do indeed contain the same data. For example, processing the pathway $\mathsf{LS}_2 \rightarrow \mathsf{LS}_1$ going via the two union schemas US_2 and US_1 gives the following view definition for $\mathsf{LS}_2 : \langle\langle \mathsf{male} \rangle\rangle$ in terms of the constructs of LS_1 as $[x \mid (x, \text{`}M\text{'}) \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{staff,sex} \rangle\rangle]$.

We note that in our particular example integration of Section 2.1, the different local schemas either contain different representations of the same information, or they contain non-overlapping information. Thus, to generate peer-to-peer views it is sufficient to go via the shortest pathway between two local schemas. In more complex integration scenarios where there may be overlapping data between different data sources, it would be necessary to traverse also the pathway from the union schema to the global schema.

5 Optimising the Generated Views

The view definitions generated by the process described above can be simplified after they have been derived. This saves later work for the query optimiser when these definitions are substituted into specific global queries for GAV query processing (which is what AutoMed supports). It also means that our generated views end up looking much like the views that would have been specified directly in a GAV or LAV framework.

The AutoMed intermediate query language IQL supports two kinds of operator for manipulating bags: the bag append operator, ++, and also a family of

operators which are all derived from a single `fold` function. `fold` applies a given function `f` to each element of a bag and then 'folds' a binary operator `op` into the resulting values; it is defined as follows, where `[]` is the empty bag, `[x]` is a singleton bag containing one element `x`, and `b1 ++ b2` is the union of two bags `b1` and `b2`:

```
fold f op e []          = e
fold f op e [x]         = f x
fold f op e (b1 ++ b2) = (fold f op e b1) op (fold f op e b2)
```

For example, `sum = fold (id) (+) 0` and `count = fold (lambda x.1) (+) 0`. The other common grouping and aggregation operators can also be defined in terms of `fold`, as can a function `flatmap`:

```
flatmap f b = fold f (++) [] b
```

`flatmap` can in turn be used to define selection, projection, and join operators. The comprehension syntax mentioned earlier also translates into successive applications of `flatmap`.

Optimisations for `fold` apply to all operators that can be defined in terms of it. We refer the reader to [17] for a discussion of IQL and for references to relevant work on fold-based functional query languages and optimisation techniques for them.

Regarding the view definitions generated from BAV pathways as described in the previous section, there are two particular optimisations that can be applied to them. Firstly, instances of Void can be removed. For the purposes of query processing, Void is regarded as being equal to the empty bag. Thus, from the above definitions of `fold` and `flatmap`,

```
fold f op e Void = fold f op e [] = e
flatmap f Void   = flatmap f []   = []
```

We also have that

```
Void ++ e = [] ++ e = e ++ [] = e ++ Void = e
Void OR e           = e OR Void           = e
```

Applying these simplifications to the view definition we derived for the global construct GS:⟨⟨person,sex⟩⟩ results in:

(LS_1:⟨⟨staff,sex⟩⟩ OR
([$(x, `M')$ | $x \leftarrow \mathsf{LS}_2$:⟨⟨male⟩⟩] ++ [$(x, `F')$ | $x \leftarrow \mathsf{LS}_2$:⟨⟨female⟩⟩]) OR
([$(x, `M')$ | $x \leftarrow \mathsf{LS}_3$:⟨⟨male⟩⟩] ++ [$(x, `F')$ | $x \leftarrow \mathsf{LS}_3$:⟨⟨female⟩⟩]) OR
LS_5:⟨⟨staff,sex⟩⟩)
++ LS_4:⟨⟨student,sex⟩⟩

Due to the step-wise specification of our schema transformations, there is a second major optimisation which may be applicable. This is known as **loop fusion** and it replaces two successive iterations over a collection by one iteration provided the operators in question satisfy certain algebraic properties.

A simple instance of loop fusion is the standard relational query optimisation $\pi_A(\pi_B(R)) = \pi_{A,B}(R)$.

Loop fusion does not arise in the simple schema integration example of Section 2.1 but consider the following fragment of an AutoMed pathway. This first joins two schemes $\langle\langle R, a\rangle\rangle$ and $\langle\langle R, b\rangle\rangle$, creating an intermediate relation $\langle\langle I_1\rangle\rangle$, then projects onto the a and b attributes, creating a second intermediate relation $\langle\langle I_2\rangle\rangle$, then groups $\langle\langle I_2\rangle\rangle$ on a, creating a relation $\langle\langle V\rangle\rangle$, and finally deletes $\langle\langle I_1\rangle\rangle$ and $\langle\langle I_2\rangle\rangle$ (these kinds of transformations are likely to arise if AutoMed is applied in a data warehousing environment [4]).

$\mathsf{addRel}(\langle\langle I_1\rangle\rangle, [(x, y, z) \mid (x, y) \leftarrow \langle\langle R, a\rangle\rangle; (x, z) \leftarrow \langle\langle R, b\rangle\rangle])$
$\mathsf{addRel}(\langle\langle I_2\rangle\rangle, \mathsf{map}\ (\mathsf{lambda}(x, y, z).(y, z))\ \langle\langle I_1\rangle\rangle)$
$\mathsf{addRel}(\langle\langle V\rangle\rangle, \mathsf{group}\ \langle\langle I_2\rangle\rangle)$
$\mathsf{deleteRel}(\langle\langle I_2\rangle\rangle, \mathsf{map}\ (\mathsf{lambda}(x, y, z).(y, z))\ \langle\langle I_1\rangle\rangle)$
$\mathsf{deleteRel}(\langle\langle I_1\rangle\rangle, [(x, y, z) \mid (x, y) \leftarrow \langle\langle R, a\rangle\rangle; (x, z) \leftarrow \langle\langle R, b\rangle\rangle])$

The view definition generated for $\langle\langle V\rangle\rangle$ would be

$\mathsf{group}\ (\mathsf{map}\ (\mathsf{lambda}(x, y, z).(y, z))\ [(x, y, z) \mid (x, y) \leftarrow \langle\langle R, a\rangle\rangle; (x, z) \leftarrow \langle\langle R, b\rangle\rangle])$

and the projection operation map can be fused with the head expression of the comprehension, giving $\mathsf{group}\ ([(y, z) \mid (x, y) \leftarrow \langle\langle R, a\rangle\rangle; (x, z) \leftarrow \langle\langle R, b\rangle\rangle])$.

There are a range of other standard algebraic optimisations that could be performed on the view definitions e.g. pushing down selections and projections. However, these kinds of optimisations will also be applied later, when a specific global query is reformulated by substituting into it the view definitions. Further optimisations and rewrites will be applied at this stage e.g. to bring constructs from the same local schemas together into sub-queries which can be posed entirely on one local schema and it is these sub-queries (appropriately translated) that will be sent to local data sources for evaluation. Given the large amount of rewriting that will take place at this stage, for now we just perform Void-elimination and loop fusion at the view generation stage.

6 Concluding Remarks

In this paper we have described view generation and view optimisation in the AutoMed heterogeneous database integration framework. We have shown how the AutoMed schema pathways and views generated from them are amenable to considerable simplification, resulting in view definitions that look much like the views that would have been specified directly in a GAV or LAV framework.

Since BAV integration is based on sequences of primitive schema transformations, it could be argued that data integration using it is more complex than with GAV or LAV. However, the integration process can be greatly simplified by specifying well-known schema equivalences as higher-level composite transformations, as discussed in [16]. Moreover, we are working on techniques for semi-automatically generating transformation pathways to convert a source schema expressed in one modelling language into an equivalent target schema expressed in another modelling language, based on well known schema equivalences. We

are also investigating schema matching techniques to automatically or semi-automatically integrate two specific schemas.

Other directions of current work include: optimising the simplification algorithms of Section 3; implementing the global query optimiser mentioned in Section 5; and investigating the use of AutoMed for materialised data integration.

References

1. M. Boyd, P.J. McBrien, and N. Tong. The automed schema integration repository. In *Proc. BNCOD02, LNCS 2405*, pages 42–45, 2002.
2. P. Buneman *et al.* Comprehension syntax. *ACM SIGMOD Record*, 23(1):87–96, 1994.
3. S.S. Chawathe *et al.* The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. 10th Meeting of the Information Processing Society of Japan*, pages 7–18, October 1994.
4. H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. In B.Omelayenko and M.Klein, editors, *Knowledge Transformation for the Semantic Web (to appear)*. IOS Press, 2003.
5. S. Kittivoravitkul. Transformation-based approach for integrating semi-structured data. Technical report, AutoMed Project, 2003.
6. M. Lenzerini. Data integration: A theorectical perspective. In *Proc. PODS02*, pages 247–258, 2002.
7. A.Y. Levy. Logic-based techniques in data integration. In J. Minker, editor, *Logic Based Artificial Intelligence*. Kluwer Academic Publishers, 2000.
8. A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS'95*, pages 95–104. ACM Press, May 1995.
9. A.Y. Levy, A. Rajamaran, and J.Ordille. Querying heterogeneous information sources using source description. In *Proc. VLDB'96*, pages 252–262, 1996.
10. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. VLDB'01*, pages 241–250, 2001.
11. P.J. McBrien. The university database integration example. Technical report, AutoMed Project, http://www.doc.ic.ac.uk/automed/, 2002.
12. P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proc. ER'99, LNCS 1728*, pages 96–113, 1999.
13. P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99, LNCS 1626*, pages 333–348, 1999.
14. P.J. McBrien and A. Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Proc. CAiSE'01, LNCS 2068*, pages 330–345, 2001.
15. P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02, LNCS 2348*, pages 484–499, 2002.
16. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03 (to appear)*, 2003.
17. A. Poulovassilis. The AutoMed Intermediate Query Language. Technical report, AutoMed Project, 2001.
18. A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

19. M.T. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for data sources. In *Proc. VLDB'97*, pages 266–275, Athens, Greece, 1997.
20. M. Templeton, H.Henley, E.Maros, and D.J. Van Buer. InterViso: Dealing with the complexity of federated database access. *The VLDB Journal*, 4(2):287–317, 1995.
21. N. Tong. Database schema transformation optimisation techniques for the AutoMed system. Technical report, AutoMed Project, 2002.
22. D. Williams. Representing RDF and RDF Schema in the HDM. Technical report, Automed Project, 2002.