Generating and Optimising Views from Both as View Data Integration Rules

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 the generation and logical optimisation of views in the AutoMed heterogeneous data integration framework, which is based on the use of reversible schema transformation sequences called both as view (BAV) rules. We show how views can be generated from such sequences, for global as view (GAV), local as view (LAV) and GLAV 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**. The two most common data integration approaches are **global as view** (**GAV**) (used in TSIMMIS [4], InterViso [19] and Garlic [18]), and **local as view** (**LAV**) (used in IM [9] and Agora [11]). 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. 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].

Both LAV and GAV lack a certain degree of expressiveness. GAV is unable to fully capture data integration semantics where a source schema construct can be defined by a non-reversible function over global schema constructs. For example, if source schema attribute money is the sum of global schema attributes coins and notes, neither coins nor notes in the global schema can be defined by views over the source schema. Thus a query on the global schema asking for the sum of coins and notes cannot be answered even though the answer (money) is present in the source schema. In LAV, the attribute money can be defined by a view as the sum of global schema attributes coins and notes. Reversing the presence of the attributes, so that coins and notes are in the local schema and money in the global schema, leads to a situation which GAV can express but LAV cannot.

GLAV [5] is a variation of LAV that allows the head of the view definition rules to contain conjunctions of relations from a source schema as a natural join, and is thus able to capture situations where a non-reversible function is a natural-join between attributes. In [10] GLAV was extended to allow any source schema query in the head of the rule,

and hence is able to express the case where a single source schema is used to define the global schema constructs referenced in the body of the rule.

We have developed a richer integration framework which is based on the use of reversible sequences of primitive schema transformations, called transformation **pathways**. In [15] we showed how these pathways incorporate the semantics of GAV rule definitions and LAV rule definitions, and hence termed 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`).

Since BAV integration 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, GAV or GLAV view definitions would be. However, in Section 5 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.

The outline of this paper is as follows. Section 2 gives a review and examples of the BAV integration approach, and compares it with the GAV, LAV and GLAV approaches. Section 3 shows how view definitions can be generated from BAV pathways for GAV, LAV or GLAV query processing. Section 4 presents techniques for optimising these generated views, and Section 5 gives techniques for optimising the BAV pathways themselves. Section 6 gives our concluding remarks and directions of further work.

2 The BAV Integration Approach

In previous work (see `http://www.doc.ic.uk/automed`) we have developed a 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 BAV, schemas are incrementally transformed by applying a sequence of primitive transformations $t_1, \ldots, t_r$, where each t_i adds, deletes or renames just one schema construct. Each add or delete transformation is accompanied by a query, expressed in the **intermediate query language** (**IQL**), specifying the extent of the new or deleted construct in terms of the rest of the constructs in the schema. All primitive transformations have an optional additional argument which specifies a constraint (also expressed in the IQL) on the current schema extension that must hold if the transformation is to be applied.

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 a variety of methodologies for performing data integration and hence forming a **network** of pathways joining schemas together. For example, Figure 1 illustrates the integration of n local schemas, $LS_1, \ldots, LS_n$, into a global schema GS.

In order to integrate these n local schemas, each LS_i is first 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 $US_i:c$ is used to denote construct c appearing in schema US_i. 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, outerjoins *etc*.

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 $LS_i \rightarrow US_i$. These extend and contract transformations behave in the same way as add and delete, respectively, except that they indicate that only partial information can be derived about the new or deleted construct. Rather than a single query, they take a pair of queries which specify a lower and upper bound on the extent of the new or deleted construct. The lower bound query may be the constant Void if no lower bound can be specified, and the upper bound query may be the constant Any if no upper bound can be specified.

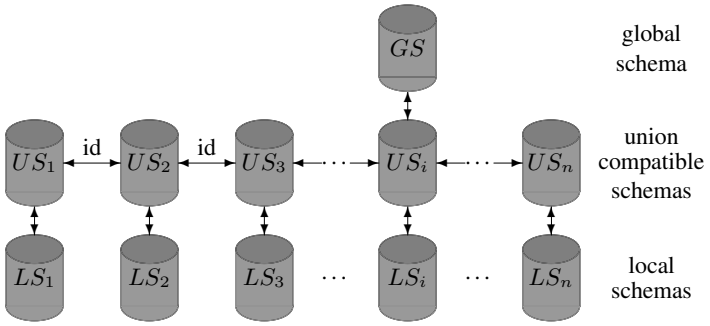

Fig. 1. A general AutoMed Integration

Each primitive transformation t has an **automatically derivable** reverse transformation $\overline{t}$. 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 [12] we show how this reversibility of schema transformations allows automatic data query translation between schemas.

In [13] we described how our framework can be applied to different high-level modelling languages such as relational, ER and UML, and more recently we have extended AutoMed to also support semi-structured data models (flat file, XML, RDF). For our examples in this paper we will use a simplified relational data model. However, we stress

that the techniques that we describe here are equally applicable to any data modelling language supported by AutoMed.

In our simple relational model, there are two kinds of schema construct: Rel and Att (see [13] for an encoding of a richer relational data model, including the modelling of constraints).

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 $\langle\langle \mathsf{student} \rangle\rangle$, and three Att constructs $\langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle$, $\langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle$ and $\langle\langle \mathsf{student}, \mathsf{dname} \rangle\rangle$.

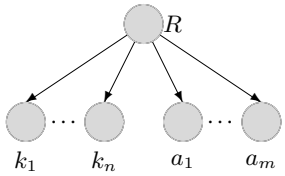

Fig. 2. A simple relational data model

Once the constructs of modelling language $\mathcal{M}$ have been defined in terms of the HDM (via the API of AutoMed's metadata repository [1]), a set of primitive schema transformations for $\mathcal{M}$ are automatically available. For the simple relational model above, these would be 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'.

There is also a set of extendRel, extendAtt, contractRel and contractAtt primitive transformations.

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 a question mark.

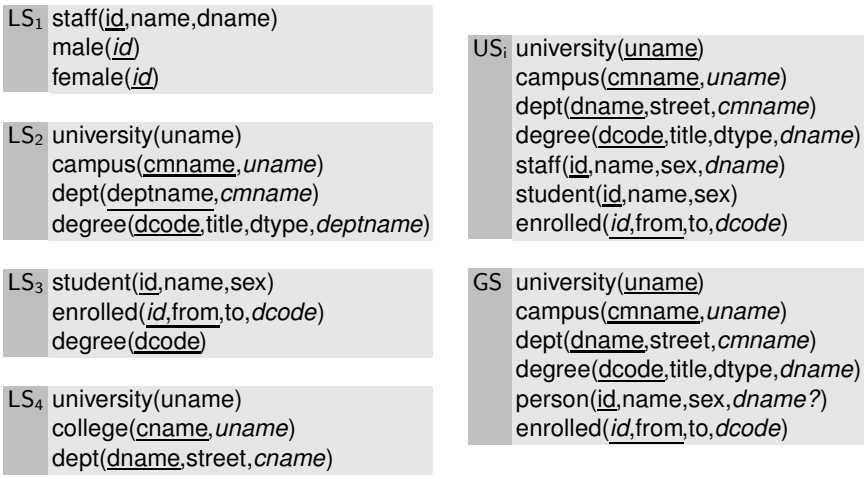

Fig. 3. Example schemas

In Example 1, transformations t_1–t_5 use a composite transformation extendTable to state that the tables student, enrolled, university, campus and degree in US_1 cannot be derived from LS_1. The definition of extendTable is:

$$\mathsf{extendTable}(\langle\langle R, a_1, \ldots, a_n\rangle\rangle) = \mathsf{extendRel}(\langle\langle R\rangle\rangle, \mathsf{Void}, \mathsf{Any});$$
$$\mathsf{extendAtt}(\langle\langle R, a_1\rangle\rangle, \mathsf{Void}, \mathsf{Any}); \ldots; \mathsf{extendAtt}(\langle\langle R, a_n\rangle\rangle, \mathsf{Void}, \mathsf{Any})$$

Then transformations t_6–t_9 use the dname attribute of staff to derive the dept table in US_1, and use extend transformations for the two attributes street and uname that cannot be derived from LS_1. Finally, in t_{10}–t_{14} the male and female relations of LS_1 are restructured into the single sex attribute of staff.

The queries accompanying the add and delete transformations are expressed in our IQL intermediate query language. 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.

Example 1 Pathway $\mathsf{LS}_1 \rightarrow \mathsf{US}_1$

t_1 $\mathsf{extendTable}(\langle\langle \mathsf{student, id, name, sex}\rangle\rangle)$
t_2 $\mathsf{extendTable}(\langle\langle \mathsf{university, uname}\rangle\rangle)$
t_3 $\mathsf{extendTable}(\langle\langle \mathsf{campus, cmname, uname}\rangle\rangle)$
t_4 $\mathsf{extendTable}(\langle\langle \mathsf{degree, dcode, title, dtype, dname}\rangle\rangle)$
t_5 $\mathsf{extendTable}(\langle\langle \mathsf{enrolled, id, from, to, dcode}\rangle\rangle)$
t_6 $\mathsf{addRel}(\langle\langle \mathsf{dept}\rangle\rangle, [\mathsf{x} \mid (\mathsf{y}, \mathsf{x}) \leftarrow \langle\langle \mathsf{staff, dname}\rangle\rangle])$
t_7 $\mathsf{addAtt}(\langle\langle \mathsf{dept, dname}\rangle\rangle, [(\mathsf{x}, \mathsf{x}) \mid \mathsf{x} \leftarrow \langle\langle \mathsf{dept}\rangle\rangle])$
t_8 $\mathsf{extendAtt}(\langle\langle \mathsf{dept, street}\rangle\rangle, \mathsf{Void}, \mathsf{Any})$
t_9 $\mathsf{extendAtt}(\langle\langle \mathsf{dept, uname}\rangle\rangle, \mathsf{Void}, \mathsf{Any})$

t_{10} addAtt(⟨⟨staff, sex⟩⟩, [(x, 'M') | x ← ⟨⟨male⟩⟩] ++ [(x, 'F') | x ← ⟨⟨female⟩⟩])
t_{11} deleteAtt(⟨⟨male, id⟩⟩, [(x, x) | x ← ⟨⟨male⟩⟩])
t_{12} deleteRel(⟨⟨male⟩⟩, [x | (x, 'M') ← ⟨⟨staff, sex⟩⟩])
t_{13} deleteAtt(⟨⟨female, id⟩⟩, [(x, x) | x ← ⟨⟨female⟩⟩])
t_{14} deleteRel(⟨⟨female⟩⟩, [x | (x, 'F') ← ⟨⟨staff, sex⟩⟩])

The pathway $LS_2 \rightarrow US_2$ contains extend steps to add the missing staff, student, and enrolled tables. It then renames deptname, and adds the missing attributes of dept:

Example 2 Pathway $LS_2 \rightarrow US_2$
t_{15} extendTable(⟨⟨student, id, name, sex⟩⟩)
t_{16} extendTable(⟨⟨staff, id, name, sex, dname⟩⟩)
t_{17} extendTable(⟨⟨enrolled, id, from, to, dcode⟩⟩)
t_{18} renameAtt(⟨⟨dept, deptname⟩⟩, ⟨⟨dept, dname⟩⟩)
t_{19} renameAtt(⟨⟨degree, deptname⟩⟩, ⟨⟨degree, dname⟩⟩)
t_{20} extendAtt(⟨⟨dept, street⟩⟩, Void, Any)
t_{21} extendAtt(⟨⟨dept, uname⟩⟩, Void, Any)

The pathway $LS_3 \rightarrow US_3$ contains a sequence of extend steps for its missing information. The pathway $LS_4 \rightarrow US_4$ creates in t_{22} a new attribute ⟨⟨dept, uname⟩⟩ by joining the dept and college relations, and then deletes in t_{23}–t_{25} the college table that can be recovered from the remaining ⟨⟨dept, cname⟩⟩ attribute. Transformation t_{26} is unable to put any restriction on the values of ⟨⟨dept, cname⟩⟩, since that association cannot be recovered from the global schema. Transformations t_{27}–t_{31} then perform the logical inverse of t_{22}–t_{26} to partially extract the campus table from the direct association between departments and universities represented by ⟨⟨dept, uname⟩⟩.

Example 3 Pathway $LS_4 \rightarrow US_4$
t_{22} addAtt(⟨⟨dept, uname⟩⟩,
    [(x, y) | (x, z) ← ⟨⟨dept, cname⟩⟩; (z, y) ← ⟨⟨college, uname⟩⟩])
t_{23} deleteAtt(⟨⟨college, uname⟩⟩,
    [(x, y) | (z, x) ← ⟨⟨dept, cname⟩⟩; (z, y) ← ⟨⟨dept, uname⟩⟩])
t_{24} deleteAtt(⟨⟨college, cname⟩⟩, [(x, x) | x ← ⟨⟨college⟩⟩])
t_{25} deleteRel(⟨⟨college⟩⟩, [y | (x, y) ← ⟨⟨dept, cname⟩⟩])
t_{26} contractAtt(⟨⟨dept, cname⟩⟩, Void, Any)
t_{27} extendAtt(⟨⟨dept, cmname⟩⟩, Void, Any)
t_{28} addRel(⟨⟨campus⟩⟩, [y | (x, y) ← ⟨⟨dept, cmname⟩⟩])
t_{29} addAtt(⟨⟨campus, cmname⟩⟩, [(x, x) | x ← ⟨⟨campus⟩⟩])
t_{30} addAtt(⟨⟨campus, uname⟩⟩,
    [(x, y) | (z, x) ← ⟨⟨dept, cmname⟩⟩; (z, y) ← ⟨⟨dept, uname⟩⟩])
t_{31} delAtt(⟨⟨dept, uname⟩⟩,
    [(x, y) | (x, z) ← ⟨⟨dept, cmname⟩⟩; (z, y) ← ⟨⟨campus, uname⟩⟩])
t_{32} extendTable(⟨⟨student, id, name, sex⟩⟩)
t_{33} extendTable(⟨⟨staff, id, name, sex, dname⟩⟩)
t_{34} extendTable(⟨⟨enrolled, id, from, to, dcode⟩⟩)

Finally, we list in Example 4 the pathway from the union schema US_1 to the global schema GS. The pathway from US_2, US_3 or US_4 would be identical.

Example 4 Pathway $\mathsf{US}_1 \rightarrow \mathsf{GS}$

t_{35} addRel(⟨⟨person⟩⟩, ⟨⟨staff⟩⟩ ++ [x | x ← ⟨⟨student⟩⟩; not (member x ⟨⟨staff⟩⟩)])
t_{36} addAtt(⟨⟨person, id⟩⟩, ⟨⟨staff, id⟩⟩ ++
[(x, y) | (x, y) ← ⟨⟨student, id⟩⟩; not (member x ⟨⟨staff⟩⟩)])
t_{37} addAtt(⟨⟨person, name⟩⟩, ⟨⟨staff, name⟩⟩ ++
[(x, y) | (x, y) ← ⟨⟨student, name⟩⟩; not (member x ⟨⟨staff⟩⟩)])
t_{38} addAtt(⟨⟨person, sex⟩⟩, ⟨⟨staff, sex⟩⟩++
[(x, y) | (x, y) ← ⟨⟨student, sex⟩⟩; not (member x ⟨⟨staff⟩⟩)])
t_{39} addAtt(⟨⟨person, dname⟩⟩, ⟨⟨staff, dname⟩⟩)
t_{40} contractAtt(⟨⟨student, id⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, id⟩⟩;
not (member x ⟨⟨staff⟩⟩)], [(x, y) | (x, y) ← ⟨⟨person, id⟩⟩])
t_{41} contractAtt(⟨⟨student, name⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, name⟩⟩;
not (member x ⟨⟨staff⟩⟩)], [(x, y) | (x, y) ← ⟨⟨person, name⟩⟩])
t_{42} contractAtt(⟨⟨student, sex⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, sex⟩⟩;
not (member x ⟨⟨staff⟩⟩)], [(x, y) | (x, y) ← ⟨⟨person, sex⟩⟩])
t_{43} contractRel(⟨⟨student⟩⟩), [x | x ← ⟨⟨person⟩⟩; not (member x ⟨⟨staff⟩⟩)],
[x | x ← ⟨⟨person⟩⟩])
t_{44} deleteAtt(⟨⟨staff, id⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, id⟩⟩; member x ⟨⟨staff⟩⟩])
t_{45} deleteAtt(⟨⟨staff, name⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, name⟩⟩; member x ⟨⟨staff⟩⟩])
t_{46} deleteAtt(⟨⟨staff, sex⟩⟩, [(x, y) | (x, y) ← ⟨⟨person, sex⟩⟩; member x ⟨⟨staff⟩⟩])
t_{47} deleteAtt(⟨⟨staff, dname⟩⟩, ⟨⟨person, dname⟩⟩)
t_{48} deleteRel(⟨⟨staff⟩⟩, [x | (x, y) ← ⟨⟨person, dname⟩⟩])

We assume in this example integration that a person may be both a member of staff and a student. For such people, their information in the staff table is preferred for propagation to the global person table in steps t_{35}–t_{38} above. Thus, there is not sufficient information in the global schema to totally derive the student table, and only contract statements can be given in steps t_{40}–t_{43}, where as a lower bound we know all persons not in the staff table are students, but as an upper bound know that all persons could be in student (if it were the case that all staff members were former students). Conversely, there is sufficient information to totally derive the staff table.

2.2 Comparison of BAV with GAV, LAV and GLAV

We see from the above example that the add and extend steps in the transformation pathways from the local schemas to the global schema correspond to GAV, since it is these steps that are incrementally defining global constructs in terms of local ones. Similarly, it is the delete and contract steps in the transformation pathways from the local schemas to the global schema that correspond to LAV, since it is these steps that are incrementally defining local constructs in terms of global ones. We will see in Section 3 how these pathways can be traversed to derive GAV and LAV views.

If a GAV view is derived from solely add steps it will be *exact* in the terminology of [7]. If, in addition, it is derived from one or more extend steps using their lower-bound

(upper-bound) queries, then the GAV view will be *sound* (*complete*) in the terminology of [7]. Similarly, if a LAV view is derived solely from delete steps it will be exact. If, in addition, it is derived from one or more contract steps using their lower-bound (upper-bound) queries, then the LAV view will be *complete* (*sound*) in the terminology of [7]. For example, in pathway $\mathsf{US}_1 \rightarrow \mathsf{GS}$ above, we could enhance t_{43} above to:
$\mathsf{contractRel}(\langle\langle\mathsf{student}\rangle\rangle, [\mathsf{x} \mid \mathsf{x} \leftarrow \langle\langle\mathsf{person}\rangle\rangle; \mathsf{not}\ (\mathsf{member}\ \mathsf{x}\ \langle\langle\mathsf{staff}\rangle\rangle)], \langle\langle\mathsf{person}\rangle\rangle])$
asserting that $\langle\langle\mathsf{student}\rangle\rangle$ contains the set of people who are not staff (completeness) and is contained by the whole set of people (soundness).

As we discussed in the introduction, BAV is a more expressive data integration language than LAV, GAV or GLAV, since it allows for the expression of mappings in both directions, and since it is not limited on how many source schemas are associated by a mapping. Indeed, in the context of peer-to-peer integration, [3] has suggested using GLAV rules in both directions in a similar manner to BAV, in order to overcome weaknesses of using GLAV alone.

As discussed in [14, 15], a further advantage of BAV over GAV and LAV is that it readily supports the evolution of both global and local schemas, by allowing pathways and schemas to be incrementally modified as opposed to having to be regenerated.

A further difference between BAV and GAV, LAV or GLAV (including the approach of using GLAV in each direction of [3]) is that statements about the relationships between global and local schemas are made at a finer level of detail, *i.e.* at the level of individual attributes as opposed to entire tables. So we can assert exact knowledge about some attributes of a table, and sound or complete knowledge about other attributes. We are also able to introduce intermediate constructs in the mapping, such as in $\mathsf{LS}_4 \rightarrow \mathsf{US}_4$.

3 Generating Views

We now present a general technique for generating GAV, LAV and GLAV view definitions from a BAV pathway. This ability to generate any of these kinds of view definitions from a single BAV pathway means that we can select a query processing technique that can vary between queries as appropriate.

To define a construct c in S_x in terms of the constructs in schema S_y, we consider in turn the transformations of $S_x \rightarrow S_y$. The only transformations that are significant are those that delete, contract or rename a construct[1]. These transformations are significant because the current view definitions may query constructs that no longer exist after such a transformation. Each of these types of transformation is handled as follows if it is encountered during the traversal of $S_x \rightarrow S_y$:

- delete: 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.
- contract: Any occurrence of the contracted construct within the current view definitions is replaced by either the lower-bound or the upper-bound query accompa-

[1] Note that this is equivalent to considering the add, extend and rename steps in the reverse $S_y \rightarrow S_x$

nying this transformation step, depending on whether sound or complete views are required.
- rename: All references to the old construct in the current view definitions are replaced by references to the new construct.

3.1 Generating GAV Views

To generate the set of GAV views for a global schema, the pathways from it 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 may be seen from the tree structure of Figure 1. Each node of this transformations tree is a schema (global, intermediate or local) linked to its neighbours by a single transformation step. View definitions for each global schema construct are derived by traversing the tree from top to bottom. Initially, each construct's view definition is just the construct itself. Each node in the tree is then visited in a downwards direction, and delete, contract and rename transformations are handled as described above. In particular, if a contract transformation step is encountered, any occurrence of the contracted construct within the current GAV view definitions is replaced by the lower-bound query accompanying this transformation step (so that sound GAV views will be generated).

At some points the tree may branch. When this happens, constructs of the parent schema are semantically identical to constructs that have the same scheme within the child schemas. The possibility of using all paths is retained within the view definitions by replacing each construct of the parent schema by a disjunction (OR) of the corresponding constructs in the child schemas.

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 $\mathsf{GS} : \langle\langle \mathsf{person}, \mathsf{sex} \rangle\rangle$ in the global schema. The pathway $\mathsf{GS} \rightarrow \mathsf{US}_1$ would be processed first (*i.e.* the reverse of the pathway $\mathsf{US}_1 \rightarrow \mathsf{GS}$ listed in Section 2.1). The only significant transformation is $\overline{t_{38}}$ that deletes $\langle\langle \mathsf{person}, \mathsf{sex} \rangle\rangle$, resulting in an intermediate view definition:

$\mathsf{GS} : \langle\langle \mathsf{person}, \mathsf{sex} \rangle\rangle$:- $\mathsf{US}_1 : \langle\langle \mathsf{staff}, \mathsf{sex} \rangle\rangle$ ++
$[(\mathsf{x}, \mathsf{y}) \mid (\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{US}_1 : \langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle; \mathsf{not}\ (\mathsf{member}\ \mathsf{x}\ \mathsf{US}_1 : \langle\langle \mathsf{staff} \rangle\rangle)]$

at one copy, US_1, of the four union schemas. Traversing the pathways $\mathsf{US}_1 \rightarrow \mathsf{LS}_1$ and $\mathsf{US}_1 \rightarrow \mathsf{US}_2$, we replace the body of the view definition with:

$([(\mathsf{x}, \text{'M'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{male} \rangle\rangle] ++ [(\mathsf{x}, \text{'F'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{female} \rangle\rangle]\ \mathsf{OR}$
$\mathsf{US}_2 : \langle\langle \mathsf{staff}, \mathsf{sex} \rangle\rangle)$
$++ ([(\mathsf{x}, \mathsf{y}) \mid (\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Void}\ \mathsf{OR}\ \mathsf{US}_2 : \langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle;$
$\mathsf{not}\ (\mathsf{member}\ \mathsf{x}\ (\mathsf{LS}_1 : \langle\langle \mathsf{staff} \rangle\rangle\ \mathsf{OR}\ \mathsf{US}_2 : \langle\langle \mathsf{staff} \rangle\rangle))])$

Traversing next $\mathsf{US}_2 \rightarrow \mathsf{LS}_2$ and $\mathsf{US}_2 \rightarrow \mathsf{US}_3$, we get:

$([(\mathsf{x}, \text{'M'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{male} \rangle\rangle] ++ [(\mathsf{x}, \text{'F'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{female} \rangle\rangle])\ \mathsf{OR}$
$\mathsf{Void}\ \mathsf{OR}\ \mathsf{US}_3 : \langle\langle \mathsf{staff}, \mathsf{sex} \rangle\rangle)$
$++ ([(\mathsf{x}, \mathsf{y}) \mid (\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Void}\ \mathsf{OR}\ \mathsf{Void}\ \mathsf{OR}\ \mathsf{US}_3 : \langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle;$
$\mathsf{not}\ (\mathsf{member}\ \mathsf{x}\ (\mathsf{LS}_1 : \langle\langle \mathsf{staff} \rangle\rangle\ \mathsf{OR}\ \mathsf{Void}\ \mathsf{OR}\ \mathsf{US}_3 : \langle\langle \mathsf{staff} \rangle\rangle))])$

Continuing with $\mathsf{US}_3 \rightarrow \mathsf{LS}_3$, $\mathsf{US}_3 \rightarrow \mathsf{US}_4$ and finally $\mathsf{US}_4 \rightarrow \mathsf{LS}_4$, we obtain the view definition:

$GS:\langle\langle person, sex\rangle\rangle$:-
([(x, 'M') | x ← $LS_1:\langle\langle male\rangle\rangle$] ++ [(x, 'F') | x ← $LS_1:\langle\langle female\rangle\rangle$])OR
Void OR Void OR Void)
++ ([(x, y) | (x, y) ← Void OR Void OR $LS_3:\langle\langle student, sex\rangle\rangle$ OR Void;
not (member x ($LS_1:\langle\langle staff\rangle\rangle$ OR Void OR Void OR Void))])

Such view derivations can be substituted into any query posed on a global schema in order to obtain an equivalent query distributed over the local schemas — this is the GAV approach to global query processing, which is what the AutoMed implementation currently supports. Section 4 will justify how this view definition can be simplified further.

3.2 Generating LAV Views

LAV views are derived similarly: the pathway from a local schema to the global schema is again retrieved from the metadata repository and 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. Also, if a contract transformation step is encountered, any occurrence of the contracted construct within the current LAV view definitions is replaced by the upper-bound query accompanying this transformation step (so that sound LAV views will be generated).

For example, to generate a LAV definition of $LS_1:\langle\langle male\rangle\rangle$, we inspect the pathway t_1,..,t_{14},t_{35},..,t_{48}. The transformation t_{12} deletes $\langle\langle male\rangle\rangle$, and therefore we have an intermediate view definition on US_1:
$LS_1:\langle\langle male\rangle\rangle$:- [x | (x, 'M') ← $US_1:\langle\langle staff, sex\rangle\rangle$]

Then $\langle\langle staff, sex\rangle\rangle$ construct is deleted by t_{46}, which substitutes$(x, y) \leftarrow \langle\langle staff, sex\rangle\rangle$ with $(x, y) \leftarrow \langle\langle person, sex\rangle\rangle$; member x $\langle\langle staff\rangle\rangle$, and the $\langle\langle staff\rangle\rangle$ construct in this query is deleted by t_{40} giving a final LAV rule:
$LS_1:\langle\langle male\rangle\rangle$:-
[x | (x, 'M') ← $GS:\langle\langle person, sex\rangle\rangle$; member x [x | (x, y) ← $GS:\langle\langle person, dname\rangle\rangle$]]

3.3 Generating GLAV Views

First, it should be noted that GLAV view definitions will include all the LAV view definitions, and all the GAV view definitions where the body of the rule is a query that matches the conditions required for the GLAV query processing system in use (which in [10] would be queries over a single source). In addition, we inspect now all the add and extend transformations of the pathway that would be ignored by LAV view generation, and for each one use the query to form the head of a new GLAV rule.

For example, in $LS_4 \rightarrow US_4$, the query in t_{22} gives a new view rule head:
$[(x, y) \mid (x, z) \leftarrow LS_4:\langle\langle dept, cname\rangle\rangle; (z, y) \leftarrow LS_4:\langle\langle college, uname\rangle\rangle]$ which is defined by $\langle\langle dept, uname\rangle\rangle$ at this stage. We then use our standard algorithm on construct $\langle\langle dept, uname\rangle\rangle$, detect that it is deleted in t_{31}, and hence replace it with the query from t_{31} to result in the GLAV rule:
$[(x, y) \mid (x, z) \leftarrow LS_4:\langle\langle dept, cname\rangle\rangle; (z, y) \leftarrow LS_4:\langle\langle college, uname\rangle\rangle]$:-
$[(x, z) \mid GS:\langle\langle dept, cmname\rangle\rangle; (z, y) \leftarrow GS:\langle\langle campus, uname\rangle\rangle]$

Note however that the BAV integration would still hold if LS_4 were fragmented, with campus and departments held on separate sources, whereas GLAV would cease to be valid in this situation.

4 Logical Optimisation of the Generated Views

The view definitions generated by the process described above can be simplified by a process of logical optimisation, where redundant parts of the query are removed. This saves later work for the query optimiser, when these definitions are substituted into specific global queries for query processing. It also generates views that are similar to the views that would have been specified directly in a GAV, LAV or GLAV framework.

4.1 The OR Operator and Void

The Void value represents a construct that is unobtainable from a data source. We thus define $\mathsf{e}\ \mathsf{OR}\ \mathsf{Void} = \mathsf{Void}\ \mathsf{OR}\ \mathsf{e} = \mathsf{e}$ for any IQL expression e. Applying this simplification to the GAV view definition derived in Section 3.1 results in:

$\mathsf{GS} : \langle\langle \mathsf{person}, \mathsf{sex} \rangle\rangle$:-
($[(\mathsf{x}, \text{'M'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{male} \rangle\rangle] \mathbin{++} [(\mathsf{x}, \text{'F'}) \mid \mathsf{x} \leftarrow \mathsf{LS}_1 : \langle\langle \mathsf{female} \rangle\rangle]$))
++ ($[(\mathsf{x}, \mathsf{y}) \mid (\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{LS}_3 : \langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle; \mathsf{not}\ (\mathsf{member}\ \mathsf{x}\ (\mathsf{LS}_1 : \langle\langle \mathsf{staff} \rangle\rangle))]$)

It may be the case that two data sources supply information for a single schema construct. For example, the global schema attribute $\langle\langle \mathsf{university}, \mathsf{uname} \rangle\rangle$ has the GAV view definition:

$\mathsf{GS} : \langle\langle \mathsf{university}, \mathsf{uname} \rangle\rangle$:- $\mathsf{LS}_2 : \langle\langle \mathsf{university}, \mathsf{uname} \rangle\rangle\ \mathsf{OR}\ \mathsf{LS}_4 : \langle\langle \mathsf{university}, \mathsf{uname} \rangle\rangle$

which expresses the fact that either LS_2 or LS_4 can be used to extract information about university names. This leads to several possibilities for operational semantics that may be used for the OR operator:

1. **ident** semantics would choose one of the expressions to evaluate, since the integration rules specify that they are the same. This may be defined by the rule $\mathsf{e1}\ \mathsf{OR}\ \mathsf{e2} = \mathsf{e1} = \mathsf{e2}$.
2. **intersect** semantics would determine that a value should be returned only if it is present in all data sources, defined by $\mathsf{e1}\ \mathsf{OR}\ \mathsf{e2} = \mathsf{intersect}\ \mathsf{e1}\ \mathsf{e2}$.
3. **append** semantics would determine that all values in all data sources should be returned, defined by $\mathsf{e1}\ \mathsf{OR}\ \mathsf{e2} = \mathsf{e1} \mathbin{++} \mathsf{e2}$.
4. **union** semantics would determine that one copy of a value should be returned if present in any data source, defined by $\mathsf{e1}\ \mathsf{OR}\ \mathsf{e2} = \mathsf{distinct}\ (\mathsf{e1} \mathbin{++} \mathsf{e2})$.

Option (1) is that which should be used if it is known that the data sources obey the semantics specified by the data integration rules *i.e.* that their extents are identical and there are no distributed data integrity violations. In this circumstance, the OR operator may also be used during distributed data integrity checking, where both expressions are evaluated, and the results compared to determine if the data sources contain consistent data.

Options (2)–(4) provide different mechanisms for handling situations where the data sources are possibly inconsistent, and thus may not share information that they should

share. Option (3) provides a result that may be used to derive Options (2) and (4), and therefore is the default semantics provided by the AutoMed's view generation algorithm. Note also that Option (4) gives the same result as Option (1) if the data sources are identical.

4.2 Other IQL Operators

The AutoMed intermediate query language IQL supports several primitive operators for manipulating lists. The list **append** operator, $++$, concatenates two lists together. The **distinct** operator removes duplicates from a list. The **monus** operator $--$ subtracts each instance of the second list from the first. For example, $[1,\ 2,\ 3,\ 2,\ 4] -- [4,\ 4,\ 2,\ 1] = [3,\ 2]$. The **fold** operator applies a given function f to each element of a list and then 'folds' a binary operator op into the resulting values, and is defined as follows:

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)

Other IQL list manipulation operators may be defined using fold together with the usual set of built-in operators and also the support of lambda abstractions. For example, the IQL functions sum and count are equivalent to the SQL SUM and COUNT aggregation functions and are defined respectively as sum = fold (id) $(+)$ 0 and count = fold (lambda x.1) $(+)$ 0.

The function flatmap applies a list-valued function f to each member of a list b and is defined as flatmap f b = fold f $(++)$ [] b. flatmap can in turn be used to define selection, projection, join and, more generally, the comprehension syntax used in the view definitions of the previous section. For example, the list comprehension [x | x $\leftarrow$ $\langle\langle$student$\rangle\rangle$; not (member x $\langle\langle$staff$\rangle\rangle$)] translates to:

flatmap (lambda x.if (not (member x $\langle\langle$staff$\rangle\rangle$)) then [x] else []) $\langle\langle$student$\rangle\rangle$

Optimisations for fold apply to all the operators defined in terms of it. Regarding the view definitions generated from BAV pathways there are two particular optimisations that can be applied to them. First, any instances of fold applied to Void can be simplied by treating Void as identical to the empty bag, so that fold f op e Void = e for any f, op, e. Second, due to the step-wise specification of our schema transformations, **loop fusion** may be applicable. This 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 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 $\langle\langle I\rangle\rangle$, and then projects onto the a and b attributes, creating a relation $\langle\langle V\rangle\rangle$, and finally deletes $\langle\langle I\rangle\rangle$:

addRel($\langle\langle I\rangle\rangle$, $[(x,y,z) \mid (x,y) \leftarrow \langle\langle R,a\rangle\rangle; (x,z) \leftarrow \langle\langle R,b\rangle\rangle]$)

addRel($\langle\langle V\rangle\rangle$, $[(y,z) \mid (x,y,z) \leftarrow \langle\langle I\rangle\rangle]$)

deleteRel($\langle\langle I\rangle\rangle$, $[(x,y,z) \mid (x,y) \leftarrow \langle\langle R,a\rangle\rangle; (x,z) \leftarrow \langle\langle R,b\rangle\rangle]$)

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

$$[(y,z) \mid (x,y,z) \leftarrow [(x,y,z) \mid (x,y) \leftarrow \langle\langle R,a\rangle\rangle; (x,z) \leftarrow \langle\langle R,b\rangle\rangle]]$$

and the generator $(x,y,z) \leftarrow$ in the outer comprehension can be fused with the head

expression of the inner comprehension, giving:

$$[(y, z) \mid (x, y) \leftarrow \langle\langle R, a\rangle\rangle; (x, z) \leftarrow \langle\langle R, b\rangle\rangle]$$

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.

We finally note that, although IQL is list-based, if the ordering of elements within lists is ignored then its operators are faithful to the expected bag semantics. Moreover, use of the distinct operator can be used to obtain set semantics as needed. We refer the reader to [17, 6] for more details of IQL and for references to work on fold-based functional query languages and optimisation techniques for such languages.

5 Validating and Optimising 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. Such networks may be complex to analyse, so we need to support automated validation that a network is well-formed. We also need to support automated optimisation of the pathways between schemas, since they may contain redundant transformations.

To support such validation and optimisation of pathways, we have developed the **Transformation Manipulation Language** (**TML**) [20, 21], which represents each transformation in a form suited to analysis of the schema constructs that are created, deleted or are required to be present for the transformation to be correct. Our definitions below require two functions sc and rc. Given a query q on schema S containing n number of constructs, sc determines all schema constructs that must exist in S if the query is valid, rc determines all schema constructs in S referencing the constructs in q. For the IQL language constructs used in our earlier examples, sc and rc are defined as:

$$\begin{aligned}
sc(\langle\langle \mathsf{r}\rangle\rangle) &= \langle\langle \mathsf{r}\rangle\rangle \\
sc(\langle\langle \mathsf{r}, \mathsf{a}\rangle\rangle) &= \{\langle\langle \mathsf{r}\rangle\rangle, \langle\langle \mathsf{r}, \mathsf{a}\rangle\rangle\} \\
sc([q_1, \ldots, q_n]) &= sc(q_1) \cup \ldots \cup sc(q_n) \\
sc(q_1 \texttt{++} q_2) &= sc(q_1) \cup sc(q_2) \\
sc([q \mid q_1, \ldots, q_n]) &= sc(q) \cup sc(q_1) \cup \ldots \cup sc(q_n) \\
rc(\langle\langle \mathsf{r}\rangle\rangle) &= \bigcup_{1 \leq i \leq n} c_i(c_i \in S \wedge \langle\langle \mathsf{r}\rangle\rangle \in sc(c_i))
\end{aligned}$$

Note that as a shorthand, we will write the pair of queries q_l, q_u in extend or contract as just q, with the semantics in such cases that $sc(q_l, q_u) = sc(q_l) \cup sc(q_u)$. The TML formalises each transformation t_i of schema S_i into schema S_{i+1} as having four **conditions** $a_i^+, b_i^-, c_i^+, d_i^-$:

- The positive precondition a_i^+ is the set of constructs that t_i implies must be present in S_i. 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:

$t_i \in \{\mathsf{add}(c,q), \mathsf{extend}(c,q)\} \rightarrow a_i^+ = (sc(c) - c) \cup sc(q)$
$t_i \in \{\mathsf{delete}(c,q), \mathsf{contract}(c,q), \mathsf{id}(c,c')\} \rightarrow a_i^+ = sc(c) \cup sc(q)$
$t_i = \mathsf{rename}(c,c') \rightarrow a_i^+ = rc(c)$

- The negative precondition b_i^- is the set of constructs that t_i implies must not be present in S_i. It comprises those constructs which the transformation will add to the schema, and thus must not already be present:
  $t_i \in \{\mathsf{add}(c,q), \mathsf{extend}(c,q), \mathsf{id}(c',c)\} \rightarrow b_i^- = c$
  $t_i \in \{\mathsf{delete}(c,q), \mathsf{contract}(c,q)\} \rightarrow b_i^- = \emptyset$
  $t_i = \mathsf{rename}(c,c') \rightarrow b_i^- = \{c/c'\}rc(c)$
- The positive postcondition c_i^+ is the set of constructs that t_i implies must be present in S_{i+1}, and is derived in the same way as $\overline{a_i^+}$ (*i.e.* the positive precondition of the $\overline{t_i}$):
  $t_i \in \{\mathsf{add}(c,q), \mathsf{extend}(c,q), \mathsf{id}(c',c)\} \rightarrow c_i^+ = sc(c) \cup sc(q)$
  $t_i \in \{\mathsf{delete}(c,q), \mathsf{contract}(c,q)\} \rightarrow c_i^+ = (sc(c) - c) \cup sc(q)$
  $t_i = \mathsf{rename}(c,c') \rightarrow c_i^+ = \{c/c'\}rc(c)$
- The negative postcondition d_i^- is the set of constructs that t_i implies must not be present in S_{i+1}, and is derived in the same way as $\overline{b_i^-}$:
  $t_i \in \{\mathsf{delete}(c,q), \mathsf{contract}(c,q), \mathsf{id}(c,c')\} \rightarrow d_i^- = c$,
  $t_i \in \{\mathsf{add}(c,q), \mathsf{extend}(c,q)\} \rightarrow d_i^- = \emptyset$
  $t_i = \mathsf{rename}(c,c') \rightarrow d_i^- = rc(c)$

Below we show how the compounded transformation t_1 and the primitive transformation t_6 are represented in the TML.

$t_{1.1} : [\emptyset, \{\langle\langle \mathsf{student} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle\}, \emptyset]$
$t_{1.2} : [\emptyset, \{\langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle\}, \emptyset]$
$t_{1.3} : [\emptyset, \{\langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student}, \mathsf{sex} \rangle\rangle\}, \emptyset]$
$t_{1.4} : [\emptyset, \{\langle\langle \mathsf{student}, \mathsf{id} \rangle\rangle\}, \{\langle\langle \mathsf{student} \rangle\rangle, \langle\langle \mathsf{student}, \mathsf{dname} \rangle\rangle\}, \emptyset]$
$t_6 : [\{\langle\langle \mathsf{staff} \rangle\rangle, \langle\langle \mathsf{staff}, \mathsf{dname} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle\}, \{\langle\langle \mathsf{dept} \rangle\rangle, \langle\langle \mathsf{staff} \rangle\rangle, \langle\langle \mathsf{staff}, \mathsf{dname} \rangle\rangle\}, \emptyset]$

5.1 Well-formed Transformation Pathways

A pathway T from schema S_m to S_n is said to be **well-formed** if for each transformation step $t_i : S_i \rightarrow S_{i+1}$ within it:

- The only difference between the schema constructs in S_{i+1} and S_i is those constructs specifically changed by transformation t_i, implying that $S_{i+1} = (S_i \cup c_i^+) - d_i^-$ and $S_i = (S_{i+1} \cup a_i^+) - b_i^-$
- The constructs required by t_i are in the schemas, implying that $a_i^+ \subseteq S_i$, $b_i^- \cap S_i = \emptyset$, $c_i^+ \subseteq S_{i+1}$ and $d_i^- \cap S_{i+1} = \emptyset$

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). Note that any implementation

may use these rules in two ways. Firstly, given a schema S_m representing a data source, and pathway P, a new data source schema S_n and its extent can be derived. Secondly, if S_n exists as a data source already, a check can be made to verify that P correctly derives its schema and extent from that of S_m.

$$wf(S_m, S_n, [t_m, t_{m+1}, \ldots, t_{n-1}]) \leftarrow a_m^+ \subseteq S_m \wedge b_m^- \cap S_m = \emptyset \wedge wf((S_m \cup c_m^+) - d_m^-, S_n, [t_{m+1}, \ldots, t_{n-1}])$$

$$wf(S_m, S_n, []) \leftarrow S_m = S_n \wedge Ext(S_m) = Ext(S_n)$$

5.2 Reordering of Transformations

Certain transformations may be performed in any order, whilst others must be performed in a specific order. For example, in $\mathsf{LS}_1 \rightarrow \mathsf{US}_1$, t_{11} must be performed before t_{12}, since the attribute $\langle\langle \mathsf{male}, \mathsf{id} \rangle\rangle$ must be deleted before the $\langle\langle \mathsf{male} \rangle\rangle$ relation is deleted. However the sub-pathway t_{11},t_{12} could be performed before or after the sub-pathway t_{13},t_{14} 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_i,t_{i+1} may be reordered to t_{i+1},t_i provided:

1. t_i does not add a construct required by t_{i+1}, and $\overline{t_{i+1}}$ does not add a construct required by $\overline{t_i}$, *i.e.* $(c_i^+ - a_i^+) \cap a_{i+1}^+ = \emptyset$ and $(a_{i+1}^+ - c_{i+1}^+) \cap c_i^+ = \emptyset$
2. t_i does not delete a construct required not to be present by t_{i+1}, and $\overline{t_{i+1}}$ does not delete a construct required not to be present by $\overline{t_i}$, *i.e.* $d_i^+ \cap b_{i+1}^+ = \emptyset$
3. if t_i is preceded by t_{i-1}, the preconditions of t_{i+1} do not conflict with the postconditions of t_{i-1}, *i.e.* $c_{i-1}^+ \cap b_{i+1}^- = \emptyset$ and $d_{i-1}^- \cap a_{i+1}^+ = \emptyset$
4. if t_{i+1} is followd by t_{i+2}, the preconditions of t_{i+2} do not conflict with the postconditions of t_i, *i.e.* $c_i^+ \cap b_{i+2}^- = \emptyset$ and $d_i^- \cap a_{i+2}^+ = \emptyset$

We can now formalise the two examples given above from $\mathsf{LS}_1 \rightarrow \mathsf{US}_1$. For t_{11},t_{12}, (1) is broken, and hence they may not be swapped. The changing of t_{11},t_{12},t_{13},t_{14} to t_{13},t_{14},t_{11},t_{12} may be performed by iteratively swapping pairs of transformations. Considering first t_{12},t_{13}, we find neither rule is broken, and they may be reordered to t_{13},t_{12}. Then t_{12},t_{14} breaks neither rule, and may be reordered to t_{14},t_{12}. This leaves a sub-pathway t_{11},t_{13},t_{14},t_{12}, and a similar argument allows t_{11} swap with t_{13} and then t_{14}, to give the sub-pathway t_{13},t_{14},t_{11},t_{12}.

5.3 Redundant anad Partially 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 consecutive within it, and T remains well-formed if they are then removed. Such redundant transformations will occur if a source schema evolves to model information in the same way as the global schema when previously it modelled the information in a different way. For example, suppose LS_1 is evolved by transformations t_{49},t_{50},t_{51},t_{52},t_{53}, textually identical to transformations t_{10},t_{11},t_{12},t_{13},t_{14}, to model the gender of staff as a single sex attribute in a new version

of the schema LS'_1. By reversing these transformation steps we can derive the pathway from the new to the old schema $\mathsf{LS}'_1 \rightarrow \mathsf{LS}_1$:

Example 5 Pathway $\mathsf{LS}'_1 \rightarrow \mathsf{LS}_1$
$\overline{t_{53}}$ addRel(⟨⟨female⟩⟩, [x | (x, 'F') ← ⟨⟨staff, sex⟩⟩])
$\overline{t_{52}}$ addAtt(⟨⟨female, id⟩⟩, [(x, x) | x ← ⟨⟨female⟩⟩])
$\overline{t_{51}}$ addRel(⟨⟨male⟩⟩, [x | (x, 'M') ← ⟨⟨staff, sex⟩⟩])
$\overline{t_{50}}$ addAtt(⟨⟨male, id⟩⟩, [(x, x) | x ← ⟨⟨male⟩⟩])
$\overline{t_{49}}$ deleteAtt(⟨⟨staff, sex⟩⟩, [(x, 'M') | x ← ⟨⟨male⟩⟩] ++ [(x, 'F') | x ← ⟨⟨female⟩⟩])

If we inspect the entire path $\mathsf{LS}'_1 \rightarrow \mathsf{US}_1$, consisting of $\mathsf{LS}'_1 \rightarrow \mathsf{LS}_1$ followed by $\mathsf{LS}_1 \rightarrow \mathsf{US}_1$, it may be reordered to contain the sub-pathway:
$\overline{t_{51}}$ addRel(⟨⟨male⟩⟩, [x | (x, 'M') ← ⟨⟨staff, sex⟩⟩])
$\overline{t_{50}}$ addAtt(⟨⟨male, id⟩⟩, [(x, x) | x ← ⟨⟨male⟩⟩])
$\overline{t_{49}}$ deleteAtt(⟨⟨staff, sex⟩⟩, [(x, 'M') | x ← ⟨⟨male⟩⟩] ++ [(x, 'F') | x ← ⟨⟨female⟩⟩])
t_{10} addAtt(⟨⟨staff, sex⟩⟩, [(x, 'M') | x ← ⟨⟨male⟩⟩] ++ [(x, 'F') | x ← ⟨⟨female⟩⟩])
t_{11} deleteAtt(⟨⟨male, id⟩⟩, [(x, x) | x ← ⟨⟨male⟩⟩])
t_{12} deleteRel(⟨⟨male⟩⟩, [x | (x, 'M') ← ⟨⟨staff, sex⟩⟩])

Clearly $\overline{t_{49}}$,t_{10} 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 $\overline{t_{50}}$,t_{11} may be removed for the same reason, and then $\overline{t_{51}}$,t_{12}. Once all other redundant pairs have been removed, $\mathsf{LS}'_1 \rightarrow \mathsf{US}_1$ would comprise of just t_1–t_9.

Using the TML, we can identify redundant transformations as satisfying:
$(a_x^+ = c_y^+) \wedge (b_x^- = d_y^-) \wedge (c_x^+ = a_y^+) \wedge (d_x^- = b_y^-) \wedge\ Ext(c_x^+ \oplus a_x^+) = Ext(c_y^+ \oplus a_y^+)$
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.

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 consecutive, and T remains well-formed if they are then replaced by a single transformation t_{xy}.

The pathway $\mathsf{LS}_1 \rightarrow \mathsf{LS}_2$ has a pair of such partially redundant transformations, since it can be reordered to obtain the sub-pathway:
t_7 addAtt(⟨⟨dept, dname⟩⟩, [(x, x) | x ← ⟨⟨dept⟩⟩])
$\overline{t_{18}}$ renameAtt(⟨⟨dept, dname⟩⟩, ⟨⟨dept, deptname⟩⟩)

This may be replaced by the new transformation given below, which leaves a fully optimised pathway $\mathsf{LS}_1 \rightarrow \mathsf{LS}_2$.
t_{54} addAtt(⟨⟨dept, deptname⟩⟩, [(x, x) | x ← ⟨⟨dept⟩⟩])

Using the TML, we can identify partially redundant transformations as satisfying the following rules, where $\oplus$ indicates the exclusive-or operator:
$((a_x^+ = c_y^+) \wedge a_x^+ \neq \emptyset \oplus (b_x^- = d_y^-) \wedge b_x^- \neq \emptyset \wedge d_x^- \cap b_y^- = \emptyset \wedge d_x^- \neq \emptyset \wedge b_y^- \neq \emptyset$

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 [20, 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')

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, LAV or GLAV 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, LAV or GLAV. However, the integration process can be greatly simplified by specifying well-known schema equivalences as higher-level composite transformations. We gave such an example, extendTable, in Section 2.1 above, and further examples are given in [15]. 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.

Finally, it should be noted that BAV is well-suited to peer-to-peer data integration (see [16]) since it lacks the directionality inherent in LAV, GAV and GLAV, all of which are tied to the concept of there being a global schema which may not always be the case in peer-to-peer environments.

References

1. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE2004*, 2004.
2. P. Buneman *et al*. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
3. D. Calvanese, E. Damagio, G. De Giacomo, M. Lenzerini, and R. Rosati. Semantic data integration in P2P systems. In *Proc. DBISP2P*, Berlin, Germany, 2003.
4. 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.
5. M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proc.16th National Conf. on AI*, pages 67–73. AAAI Press, 1999.
6. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL Queries and Migrating Data in the AutoMed toolkit. Technical Report No. 20, AutoMed, 2003.

7. M. Lenzerini. Data integration: A theorectical perspective. In *Proc. PODS02*, pages 247–258, 2002.
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. J. Madhavan and A.Y. Halevy. Composing mappings among data sources. In *Proc. 29th Conference on VLDB*, pages 572–583, 2003.
11. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. VLDB'01*, pages 241–250, 2001.
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. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02, LNCS 2348*, pages 484–499, 2002.
15. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, 2003.
16. P.J. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P*, Berlin, Germany, 2003.
17. A. Poulovassilis. The AutoMed Intermediate Query Language. Technical report, AutoMed Project, 2001.
18. 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.
19. M. Templeton, H.Henley, E.Maros, and D.J. Van Buer. InterViso: Dealing with the complexity of federated database access. *VLDB Journal*, 4(2):287–317, 1995.
20. N. Tong. Database schema transformation optimisation techniques for the AutoMed system. Technical report, AutoMed Project, 2002.
21. N. Tong. Database schema transformation optimisation techniques for the AutoMed system. In *Proc. BNCOD'03*, volume 2712 of *LNCS*, pages 157–171. Springer, 2003.