From: 'Conceptual Modelling - ER'99',
18th Internation Conference on Conceptual Modelling,
J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais (Eds),
Springer Verlag LNCS 1728, Pages 96-113, 1999

Automatic migration and wrapping of database applications — a schema transformation approach

Peter McBrien[1] and Alexandra Poulovassilis[2]

[1] Dept. of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, pjm@doc.ic.ac.uk
[2] Dept. of Computer Science, Birkbeck College, University of London,
Malet Street, London WC1E 7HX, ap@dcs.bbk.ac.uk

Abstract.
Integration of heterogeneous databases requires that semantic differences between schemas are resolved by a process of schema transformation. Previously, we have developed a general framework to support the schema transformation process, consisting of a hypergraph-based common data model and a set of primitive schema transformations defined for this model. Higher-level common data models and primitive schema transformations for them can be defined in terms of this lower-level model.
In this paper, we show that a key feature of our framework is that both primitive and composite schema transformations are automatically *reversible*. We show how these transformations can be used to automatically migrate or wrap data, queries and updates between semantically equivalent schemas. We also show how to handle transformations between non-equivalent but overlapping schemas. We describe a prototype schema integration tool that supports this functionality. Finally, we briefly discuss how our approach can be extended to more sophisticated application logic such as constraints, deductive rules, and active rules.

1 Introduction

Common to many methods for integrating heterogeneous data sources is the requirement for **logical integration** [14, 6] of the data, due to variations in the design of data models for the same universe of discourse (UoD). Logical integration requires that we are able to transform models to equivalent ones w.r.t. the UoD, and also to translate data, queries or updates on such models. The work in this paper focuses on providing a practical, yet formal, approach to logical integration which directly supports this translation. Our previous work [9, 10, 12] has provided a general formalism to underpin logical schema transformation and integration. Here we extend this work by providing schema transformations that are automatically reversible. As we will see, this allows the automatic translation of data, queries and updates between equivalent schemas. We also extend our schema transformations to operate between schemas representing overlapping but non-equivalent UoDs. This means that should two databases vary in their coverage of the UoD we are still able translate data, queries and updates over their common part of the UoD.

Current implementations of database interoperation, such as InterViso [15], TSIMMIS [2] and Garlic [13, 5], are **query-processing oriented**. They focus on providing mechanisms by which a query can be submitted to the federated schema or mediator, and that query be translated to queries on the source databases. In contrast, our approach is **schema-equivalence oriented** in that we focus on providing mechanisms for defining the equivalence between database schemas, and on then using that equivalence to automatically perform whatever transformations on data, queries or updates are necessary. As we will see below, our approach has the further advantage of decomposing the transformation of schemas into a sequence of small steps, whereas the query-processing oriented approaches require that constructs in one schema are directly defined in terms of those in the other schema. We differ from other approaches to 'automatic wrapper generation' [16, 3] in directly using the basic schema transformation steps to migrate or transform both data, queries and updates — to give what we term application migration and transformation.

The remainder of this paper is as follows. In Section 2 we review the hypergraph data model that underpins our approach and the primitive transformations on schemas defined in this data model. We show how a small extension to our previous framework makes schema transformations reversible. We also extend our previous framework to handle non-equivalent schemas. In Section 3 we show how the reversible schema transformations that result from our framework can be used to automatically migrate or wrap database applications. Section 4 applies our approach to a simple binary ER model, describes a prototype ER schema integration tool, and gives several examples illustrating its use for transforming ER schemas and automatically translating queries, updates and data between them. Section 5 gives our concluding remarks and directions for further work.

2 The Schema Transformation Framework

2.1 Review of our previous work

This section gives a short review of our schema transformation framework, and more details of this material can be found in [12].

A **schema** in the **hypergraph data model** (HDM) is a triple $\langle Nodes, Edges, Constraints \rangle$. A **query** q **over a schema** $S = \langle Nodes, Edges, Constraints \rangle$ is an expression whose variables are members of $Nodes \cup Edges$[1]. $Nodes$ and $Edges$ define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes *and* other edges. $Constraints$ is

[1] Since what we provide is a framework, the query language is not fixed but will vary between different implementation architectures. In our examples in this paper, we assume that it is the relational calculus.

a set of boolean-valued queries over S. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them.

An **instance** I of a schema $S = \langle Nodes, Edges, Constraints\rangle$ is a set of sets satisfying the following:

(i) each construct $c \in Nodes \cup Edges$ has an extent, denoted by $Ext_{S,I}(c)$, that can be derived from I; [2]
(ii) conversely, each set in I can be derived from the set of extents $\{Ext_{S,I}(c) \mid c \in Nodes \cup Edges\}$;
(iii) for each $e \in Edge$, $Ext_{S,I}(e)$ contains only values that appear within the extents of the constructs linked by e (domain integrity);
(iv) the value of every constraint $c \in Constraints$ is true, the **value** of a query q being given by $q[c_1/Ext_{S,I}(c_1), \ldots, c_n/Ext_{S,I}(c_n)]$ where $c_1, \ldots, c_n$ are the constructs in $Nodes \cup Edges$.

We call the function $Ext_{S,I}$ an **extension mapping**. A **model** is a triple $\langle S, I, Ext_{S,I}\rangle$. Two schemas are **equivalent** if they have the same set of instances. Given a condition f, a schema S **conditionally subsumes** a schema S' w.r.t. f if any instance of S' satisfying f is also an instance of S. Two schemas S and S' are **conditionally equivalent** w.r.t f if they each conditionally subsume each other w.r.t. f.

We first developed these definitions of schemas, instances, and schema equivalence in the context of an ER common data model, in earlier work [9, 10]. A comparison with other approaches to schema equivalence and schema transformation can be found in [10], which also discusses how our framework can be applied to schema integration.

We now list the primitive transformations of the HDM. Each transformation is a function that when applied to a model returns a new model. Each transformation has a proviso associated with it which states when the transformation is **successful**. Unsuccessful transformations return an "undefined" model, denoted by ϕ. Any transformation applied to ϕ returns ϕ.

1. $renameNode$ $\langle fromName, toName\rangle$ renames a node. Proviso: $toName$ is not already the name of some node.
2. $renameEdge$ $\langle\langle fromName, c_1, \ldots, c_m\rangle, toName\rangle$ renames an edge. Proviso: $toName$ is not already the name of some edge.
3. $addConstraint$ c adds a new constraint c. Proviso: c evaluates to true.
4. $delConstraint$ c deletes a constraint. Proviso: c exists.
5. $addNode$ $\langle name, q\rangle$ adds a node named $name$ whose extent is given by the value of the query q. Proviso: a node of that name does not already exist.
6. $delNode$ $\langle name, q\rangle$ deletes a node. Here, q is a query that states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs (thus, not violating property (ii) of an instance). Proviso: the node exists and participates in no edges.

[2] Again, the language in which this derivation mapping is defined, and also that in point (ii), is not fixed by our framework.

7. *addEdge* $\langle\langle name, c_1, \ldots, c_m\rangle, q\rangle$ adds a new edge between a sequence of existing schema constructs $c_1, \ldots, c_m$. The extent of the edge is given by the value of the query q. Proviso: the edge does not already exist, $c_1, \ldots, c_m$ exist, and q satisfies the appropriate domain constraints.
8. *delEdge* $\langle\langle name, c_1, \ldots, c_m\rangle, q\rangle$ deletes an edge. q states how the extent of the deleted edge could be recovered from the extents of the remaining schema constructs. Proviso: the edge exists and participates in no edges.

For each of these transformations, there is a 3-ary version which takes as an extra argument a condition which must be satisfied in order for the transformation to be successful.

A **composite transformation** is a sequence of $n \geq 1$ primitive transformations. A transformation t is **schema-dependent (s-d)** w.r.t. a schema S if t does not return ϕ for any model of S, otherwise t is **instance-dependent (i-d)** w.r.t. S. It is easy to see that if a schema S can be transformed to a schema S' by means of a s-d transformation, and vice versa, then S and S' are equivalent. If S can be transformed to S' by means of an i-d transformation with proviso f, and vice versa, then S and S' are conditionally equivalent w.r.t f.

Example 1 Transforming between two schemas Consider two schemas $S_1 = \langle N_1, E_1, C_1\rangle$ and $S_2 = \langle N_2, E_2, C_2\rangle$ where
$N_1 = \{person, mathematician, compScientist\}$, $E_1 = \{\}$,
$C_1 = \{mathematician \subseteq person, compScientist \subseteq person\}$,
$N_2 = \{person, dept\}$, $E_2 = \{\langle Null, person, dept\rangle\}$, $C_2 = \{\}$.
The transformation T_{S_1,S_2} below transforms S_1 to S_2 and is s-d:

$$
\begin{array}{ll}
T_{S_1,S_2} = & \\
addNode & \langle dept, \{maths, compsci\}\rangle; \\
addEdge & \langle Null, person, dept, \{\langle x, maths\rangle \mid x \in mathematician\} \cup \\
 & \{\langle x, compsci\rangle \mid x \in compScientist\}\rangle; \\
delConstraint & (compScientist \subseteq person); \\
delConstraint & (mathematician \subseteq person); \\
delNode & \langle compScientist, \{x \mid \langle x, compsci\rangle \in \langle Null, person, dept\rangle\}\rangle \\
delNode & \langle mathematician, \{x \mid \langle x, maths\rangle \in \langle Null, person, dept\rangle\}\rangle;
\end{array}
$$

Conversely, the transformation T_{S_2,S_1} below transforms S_2 to S_1 and is i-d since there is a condition $dept = \{maths, compsci\}$ on the last-but-one step:

$$
\begin{array}{ll}
T_{S_2,S_1} = & \\
addNode & \langle mathematician, \{x \mid \langle x, maths\rangle \in \langle Null, person, dept\rangle\}\rangle; \\
addNode & \langle compScientist, \{x \mid \langle x, compsci\rangle \in \langle Null, person, dept\rangle\}\rangle; \\
addConstraint & (mathematician \subseteq person); \\
addConstraint & (compScientist \subseteq person); \\
delEdge & \langle Null, person, dept, \{\langle x, maths\rangle \mid x \in mathematician\} \cup \\
 & \{\langle x, compsci\rangle \mid x \in compScientist\}\rangle \quad (dept = \{maths, compsci\}); \\
delNode & \langle dept, \{maths, compsci\}\rangle
\end{array}
$$

Thus S_1 and S_2 are c-equivalent.

2.2 Reversibility of schema transformations

There are two minor but significant differences between the transformations listed in the previous section and those given in [12]. Firstly, the *delNode* and *delEdge* transformations now require a restoring query for the deleted node or edge to be specified. Secondly, addition and deletion transformations are only successful if the construct being added (deleted) does not exist (exists) in the schema, and similarly renamings of constructs must be to new names. These changes allow the reverse transformation to the original model to be automatically derivable. In particular, for every primitive transformation t such that $t(\langle S, I, Ext_{S,I}\rangle) \neq \phi$ there exists a **reverse primitive transformation** t^{-1} such that $t^{-1}(t(\langle S, I, Ext_{S,I}\rangle)) = \langle S, I, Ext_{S,I}\rangle$. We list the reverse transformation of each primitive transformation below. Notice that if t depends on a condition c, since t is successful c must necessarily hold and so need not be verified within t^{-1}:

Transformation (t)	Reverse Transformation (t^{-1})
$renameNode\ \langle from, to\rangle\ c$	$renameNode\ \langle to, from\rangle$
$renameEdge\ \langle\langle from, schemes\rangle, to\rangle\ c$	$renameEdge\ \langle\langle to, schemes\rangle, from\rangle$
$addConstraint\ q\ c$	$delConstraint\ q$
$delConstraint\ q\ c$	$addConstraint\ q$
$addNode\ \langle n, q\rangle\ c$	$delNode\ \langle n, q\rangle$
$delNode\ \langle n, q\rangle\ c$	$addNode\ \langle n, q\rangle$
$addEdge\ \langle e, q\rangle\ c$	$delEdge\ \langle e, q\rangle$
$delEdge\ \langle e, q\rangle\ c$	$addEdge\ \langle e, q\rangle$

The reversibility of primitive transformations generalises to any successful composite transformation: given any such composite transformation $t_1; \ldots; t_n$ its reverse composite transformation is $t_n^{-1}; \ldots; t_1^{-1}$. Thus, in Example 1, $T^{-1}_{S_2,S_1} = T_{S_1,S_2}$ and $T^{-1}_{S_1,S_2}$ is T_{S_2,S_1} but without the condition $dept = \{maths, compsci\}$ in the last-but-one step.

2.3 Transforming between non-equivalent schemas

Even when two databases are designed to hold the same information, it is likely that they will differ slightly and will not be precisely equivalent. We define four more low-level transformations to deal with such situations. If a schema S_2 contains a construct (*i.e.* node or edge in the low-level framework) which cannot be derived from a schema S_1, we say that S_2 is formed by **extending** S_1 with that construct, giving the value `void` as the query that creates the extent of the new construct. The reverse transformation involves **contracting** S_2 to obtain S_1, and the query is again `void` indicating that the deleted construct cannot be restored from S_1. The four new low-level transformations can be defined in terms of the existing ones as follows:

$$\begin{array}{llll} \mathit{extendNode}\, n &= \mathit{addNode}\,\langle n, \mathtt{void}\rangle & \mathit{extendEdge}\, e &= \mathit{addEdge}\,\langle e, \mathtt{void}\rangle \\ \mathit{contractNode}\, n &= \mathit{delNode}\,\langle n, \mathtt{void}\rangle & \mathit{contractEdge}\, e &= \mathit{delEdge}\,\langle e, \mathtt{void}\rangle \end{array}$$

If a compound transformation t such that $S_2 = t(S_1)$ contains a transformation of the form *contractNode* n or *contractEdge* e then S_1 can be regarded as

extending S_2 **with respect to** n **or** e, in the sense that S_1 holds information on n or e that cannot be held in S_2.

Similarly, if a compound transformation t such that $S_2 = t(S_1)$ contains a transformations of the form *extendNode* n or *extendEdge* e then S_2 can be regarded as **extending** S_1 **with respect to** n **or** e, in the sense that S_2 holds information in n or e that cannot be held in S_1.

Clearly both of the above may apply for any compound transformation t, giving rise to a pair of schemas which are **overlapping**. For such pairs of schemas there will be some queries that can only be answered by performing a join between the two schemas.

Suppose that pre-integration transformations have been applied to a set of source schemas so that all conflicts between them (both structural and constraint-based) have been removed and the schemas can be integrated by forming a union of their *Nodes*, *Edges* and *Constraints* components. Then the resulting **federated schema**, S_f, has the properties that (i) there is no node or edge with respect to which S_f extends all of the source schemas (**minimality**), and (ii) there is no node or edge with respect to which a source schema extends S_f (**completeness**). In practice, property (ii) is often not implemented when a partial federated schema is constructed for some specific application.

3 Migrating and Wrapping Databases

If a schema S_1 is transformed into an equivalent one S_2, we may want to migrate (*i.e.* re-engineer) an existing database application from S_1 to S_2. The old schema and extension become obsolete and the new schema and new extension become the operational database. In this case, we will see below how the transformation from S_1 to S_2 can be used to automatically:

(i) migrate the database extension associated with S_1 to populate a new database extension associated with S_2;
(ii) migrate queries posed on S_1 to queries posed on S_2;
(iii) migrate updates posed on S_1 to updates posed on S_2.

An alternative scenario is that an existing database is being "wrapped" by a new, equivalent schema. The old schema and extension carry on being operational but clients are able to interact with them via the new schema. In this case, we will see below how the transformation from S_1 to S_2 can be used to automatically:

(iv) translate queries posed on S_2 to queries posed on S_1;
(v) translate updates posed on S_2 to updates posed on S_1.

3.1 Data migration

Suppose a schema S_1 is transformed into an equivalent one S_2 and we wish to automatically migrate the current extension of S_1 to S_2. Consider first the case that S_1 is transformed to S_2 by a single primitive transformation. The only cases we need to consider are the addition of a new node or edge:

– *addNode* $\langle n, q\rangle$: the new node n is populated by evaluating q on the extension of S_1.
– *addEdge* $\langle e, q\rangle$: the new edge e is similarly populated by evaluating q on the extension of S_1.

For composite transformations, a new extent is generated each time an $addNode$ or *addEdge* transformation is applied.

For example, if S_1 is transformed to S_2 by means of T_{S_1,S_2} in Example 1 then the queries specified in the first and second steps of T_{S_1,S_2} can be used to automatically derive $Ext_{S_2,I}$:

$$\begin{aligned} Ext_{S_2,I}(dept) &= \{maths, compsci\} \\ Ext_{S_2,I}(\langle Null, person, dept\rangle) &= \{\langle x, maths\rangle \mid x \in Ext_{S_1,I}(mathematician)\} \cup \\ &\quad \{\langle x, compsci\rangle \mid x \in Ext_{S_1,I}(compScientist)\} \end{aligned}$$

Notice that $Ext_{S_2,I}$ is precisely as defined in Example 1, except that it has now been automatically derived from the schema transformation steps.

3.2 Query and Update migration

Suppose a schema S_1 is transformed into an equivalent one S_2 and the extension of S_1 has been migrated to S_2. We can also automatically migrate queries posed on S_1 to ones posed on S_2.

Consider first the case that S_1 is transformed to S_2 by a single primitive transformation. The only cases we need to consider in order to migrate a query q_1 posed on S_1 to an equivalent query q_2 posed on S_2 are to apply renamings and to substitute occurrences of a deleted node or edge by their restoring query:

– $renameNode$ $\langle from, to\rangle$: $q_2 = [from/to]q_1$.
– $renameEdge$ $\langle\langle from, schemes\rangle, to\rangle$: $q_2 = [\langle from, schems\rangle/\langle to, schemes\rangle]q_1$.
– $delNode$ $\langle n, q\rangle$: $q_2 = [n/q]q_1$.
– $delEdge$ $\langle e, q\rangle$: $q_2 = [e/q]q_1$.

For composite transformations, the above substitutions are successively applied in order to obtain the final migrated query q_2.

Consider now an update u_1 posed on S_1, taking one of the following general forms, where q is a query, n a node of S_1 and e an edge of S_1:

$$insert\ q\ n,\ insert\ q\ e,\ delete\ q\ n,\ delete\ q\ e$$

Then exactly the same substitutions as for queries above can be applied to u_1 in order to obtain an equivalent update u_2 posed on S_2. Of course u_2 will be unambiguous only if S_1 is an updateable view of S_2, otherwise all the usual problems associated with view updates [4, 7] will need to be addressed.

To illustrate, the following query on S_1 returns people who work in either the Computer Science and or the Maths department:

$$compScientist \cup mathematician$$

The transformation T_{S_1,S_2} results in the following substitution:

$$[mathematician/\{x \mid \langle x, maths\rangle \in \langle Null, person, dept\rangle\},$$
$$compScientist/\{x \mid \langle x, compsci\rangle \in \langle Null, person, dept\rangle\}]$$

which when applied to the query results in the following equivalent query on S_2:

$$\{x \mid \langle x, compsci\rangle \in \langle Null, person, dept\rangle\} \cup \{x \mid \langle x, maths\rangle \in \langle Null, person, dept\rangle\}$$

Similarly, this update on S_1 adds Bob to the extent of $mathematician$:

$$insert\ Bob\ mathematician$$

Applying the above substitution results in the following update on S_2:

$$insert\ Bob\ \{x \mid \langle x, maths\rangle \in \langle Null, person, dept\rangle\}$$

and this update has the unambiguous meaning that the tuple $\langle Bob, maths\rangle$ should be added to the extent of $\langle Null, person, dept\rangle$.

3.3 Query and Update translation

Suppose now a schema S_1 is transformed into an equivalent one S_2 but that the extension of S_1 is not migrated to S_2. Then we can automatically translate queries and updates posed on S_2 to ones posed on S_1, as follows.

We first automatically derive the reverse transformation from S_2 to S_1, as shown in Section 2.2. In order to translate queries posed on S_2 to queries posed on S_1 we then use exactly the same method as for migrating queries from S_1 to S_2 in Section 3.2 above, except that now it is with reference to the reverse transformation. Updates posed on S_2 are similarly be translated to updates posed on S_1 and will be unambiguous if S_2 is an updateable view of S_1.

4 A Prototype ER Schema Transformation Tool

In [12] we show how to use the HDM to define transformations in a rich ER modelling language supporting n-ary relations, attributes on relations, generalisation, and complex attributes. That paper, together with [10], show the semantic richness of our framework by defining, and thereby formalising, the main ER schema equivalences that have appeared in literature. We do not repeat this work here and instead focus on the reversibility of ER schema transformations, and the fact that they can be used to migrate or wrap databases whose schemas are defined using an ER data model. We consider a simple ER modelling language supporting binary relationships and generalisation hierarchies. The primitive transformations for this language are: $rename_E$, $rename_A$, $rename_R$, for renaming an entity class, attribute and relationship, respectively; add_E, add_A, add_R, add_G for adding an entity class, attribute, relationship or generalisation hierarchy; and $del_E/A/R/G$ for deleting such constructs. The definitions of these transformations in terms of the lower-level HDM primitive transformations can be found in [12].

The notions of instances, models, schema equivalence, and s-d/i-d transformations extend naturally to ER schemas. Thus, if an ER schema S can be transformed to an ER schema S' by means of an s-d transformation, and vice versa, then S and S' are u-equivalent. Similarly, if S can be transformed to S' by means of an i-d transformation with proviso f, and vice versa, then S and S' are c-equivalent w.r.t f.

Migration and wrapping are handled in the same way at this higher semantic level as in the low-level framework. For data migration, new constructs created by add_E, add_A or add_R transformations are populated by evaluating the supplied query (add_G transformations simply add a number of constraints to the schema and do not have an associated extent). For query and update migration from the original to the resulting schema, we apply renamings and substitute occurrences of a deleted entity, attribute or relationship by their restoring query. For query and update translation from the resulting to the original schema, we use the same technique but with reference to the reverse transformation. Several examples of migrating and wrapping at this higher semantic level are given in the next section, where we describe a prototype schema integration tool.

4.1 A Prototype ER Schema Integration Tool

We have implemented a prototype ER schema integration tool that supports the functionality described above, with some minor departures. The tool provides the database integrator with the following set of primitive ER schema transformations: [3]

```
addEntity(Ss,Sn,Scheme,CSs,CSn,Query)
delEntity(Sn,Ss,Scheme,CSn,CSs,Query)
addRelationship(Ss,Sn,Scheme,Card,CSs,CSn,Query)
delRelationship(Sn,Ss,Scheme,Card,CSn,CSs,Query)
addAttribute(Ss,Sn,Scheme,Card,CSs,CSn,Query)
delAttribute(Sn,Ss,Scheme,Card,CSn,CSs,Query)
addGeneralisation(Ss,Sn,Scheme,CSn)
delGeneralisation(Sn,Ss,CSn,Scheme)
```

The tool also also provides a set of renaming transformations, but we will not be using these for our examples here and refer the reader to [8] for details of them. The parameters of the above primitive transformations have the following meaning:

`Scheme` identifies the **subject construct**. This is the construct being added or deleted by the transformation. `Scheme` includes variable(s) used to instantiate instances of the construct and it takes one of the following forms:

[3] The tool is implemented in Edinburgh Prolog, in which identifiers beginning with an upper case letter are variables, identifiers beginning with a lower case letter or enclosed in single quotes are constants, and the underscore character is an anonymous variable.

- [ent,Name,E] represents an entity type called Name, and E may be instantiated with instances of Name.
- [att,EName,AName,E,A] represents an attribute AName of entity type EName, where A may be instantiated with a value of the attribute associated with the instance E of EName.
- [rel,E1Name,RName,E2Name,E1,E2] represents a relationship RName between entities E1Name and E2Name. E1 and E2 may be instantiated with pairs of entity instances involved in the relationship.
- [gen,GenType,Super,SubList] represents a generalisation between a super-entity type Super and the list of sub-entity types SubList. GenType may be either total or partial.

When anonymous free variables appear in a Scheme e.g. `[att,person,name,_,_]`, we may abbreviate the scheme to exclude those variables e.g. [att,person,name]. Also, when a scheme is passed to a transformation its first component may be omitted as it can be inferred e.g. we need only write [person,E] instead of [ent,person,E].

Ss is what we term the **subject schema** of the transformation and Sn is the **non-subject schema** of the transformation. For addition transformations Sn is the input schema of the transformation (the one not containing the subject construct) and Ss is the output schema. Conversely, for deletion transformations Ss is the input schema (the one containing the subject construct) and Sn is the output schema.

Card is a pair of cardinality constraints [C1,C2], where C1 and C2 are each one of [1,1], [1,1], [1,n], or [1,n], with their usual meaning. Hence a scheme [rel,person,worksin,dept,P,D] and cardinality constraints [[1,1],[1,n]] indicates that each person instance P is associated with exactly one dept instance D and that each dept instance is associated with one or more person instances.

CSn and CSs give constraints, placed on Sn and Ss respectively, which must be satisfied for the transformation to be successful. The tool here extends the theoretical treatment given in previous sections by allowing the user to provide constraints on *both* the input and the output schema, as opposed to just the former. This makes the transformations between schemas bi-directional, as opposed to uni-directional, and there is really no distinction between the input and the output schemas.

Query is a query expression over the non-subject schema which defines the extent of the subject construct. The syntax of a Query is:

```
Query ::= Scheme | Predicate | [or,Query,Query] |
          [and,Query,Query] | [implies,Query,Query] | [not,Query]
Predicate ::= [eq,Atom,Atom] | [less,Atom,Atom]
```

4.2 Transformations between equivalent schemas

A sequence of transformations made between two schemas will draw an equivalence between those schemas when none of the steps involves a contract or

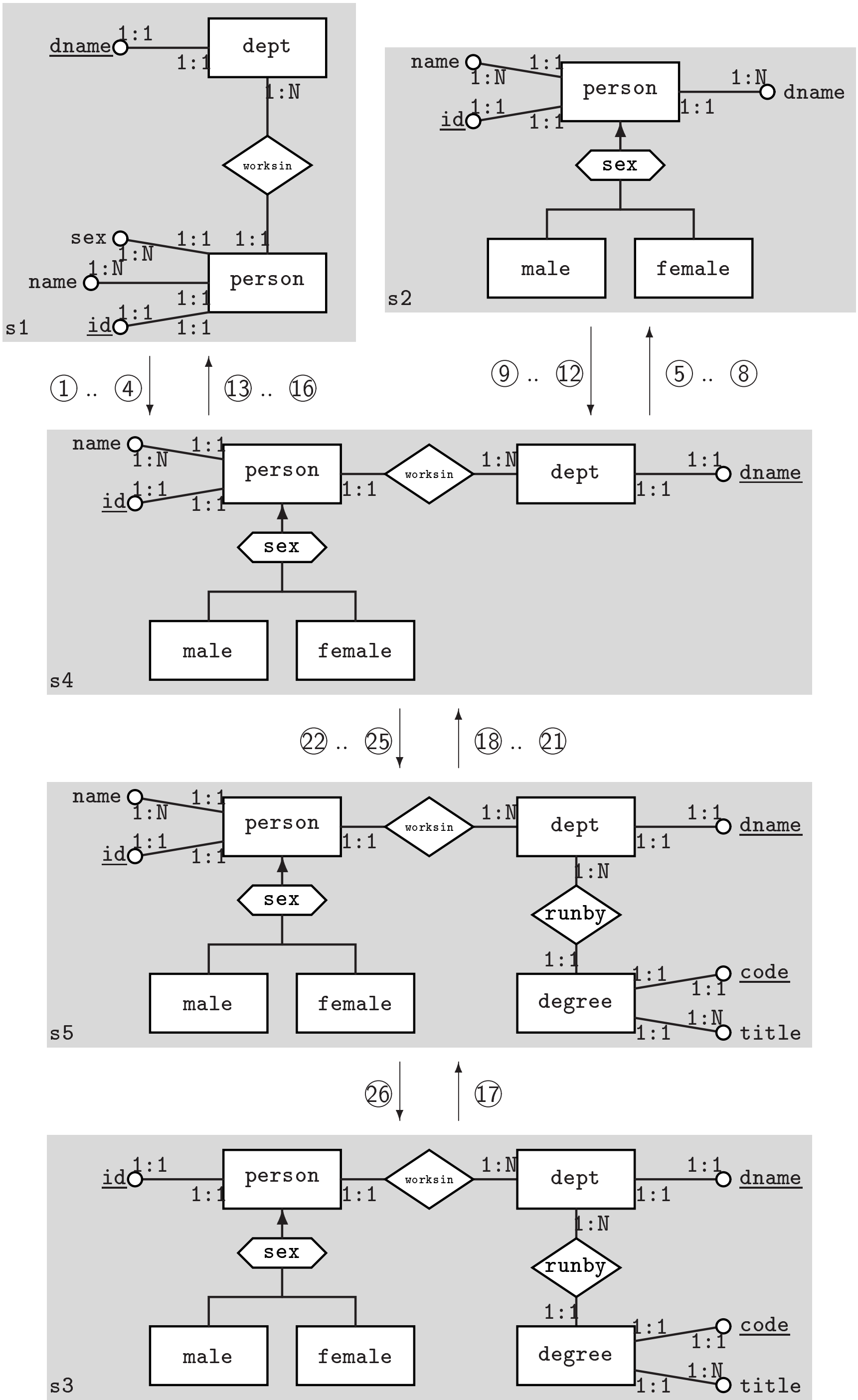

Fig. 1. Example ER schemas, and the transformations that relate them

extend transformation. We illustrate in Example 2 one such sequence for the pair of schemas `s1` and `s2` illustrated in Figure 1. We will later show in Section 4.5 how we may transform these schemas to a non-equivalent `s3`.

Example 2 Transforming `s1` to `s2`

① Add a new entity type `male` to schema `s1`, to construct a new schema `s1a`. The instances of the new entity type (the variable `EV`) are obtained by taking those instances of `person` which have value `m` for their `sex` attribute. There is no condition on the transformation (represented by `[]`). In the forward direction (*i.e.* transforming `s1` to `s1a`), any value other than `m` for the `sex` attribute will mean that a person will not be classified as `male`. In the reverse direction (*i.e.* `s1a` to `s1`), any instance of `male` will result in the `sex` attribute of that person taking the value `m`.
```
addEntity(s1a,s1,[male,EV],[],[],[att,person,sex,EV,m]),
```
② The `female` entity type is added similarly to `s1a` to construct a new schema `s1b`:
```
addEntity(s1b,s1a,[female,EV],[],[],[att,person,sex,EV,f]),
```
③ Add that `person` is a generalisation of `male` and `female`:
```
addGeneralisation(s1c,s1b,[total,person,[male,female]],
    [implies,[att,person,sex,_,AV],[or,[eq,AV,m],[eq,AV,f]]]),
```
④ Remove the now redundant `sex` attribute from `person`:
```
delAttribute(s4,s1c,[person,sex,EV,AV],[[1,1],[1,n]],[],[],
    [or,[and,[ent,male,EV],[eq,AV,m]],
       [and,[ent,female,EV],[eq,AV,f]]]),
```
At this stage we have transformed `s1` into the intermediate schema `s4` shown in Figure 1. Notice that the four steps ①- ④ implement the well-known equivalence between attributes and generalisation hierarchies [1].

⑤ Add the attribute `dname` to `person`:
```
addAttribute(s2c,s4,[person,dname,X,Y],[[1,1],[1,n]],[],[],
    [rel,person,worksin,dept,X,Y]),
```
⑥ Delete the now redundant `worksin` relationship:
```
delRelationship(s2b,s2c,[person,worksin,dept,X,Y],
    [[1,1],[1,n]],[],[],[att,person,dname,X,Y]),
```
⑦ Delete the redundant `dname` attribute on `dept`:
```
delAttribute(s2a,s2b,[dept,dname,X,Y],[[1,1],[1,n]],
    [implies,[att,dept,dname,X,Y],[eq,X,Y]],[],
    [att,person,dname,X,Y]),
```
⑧ Delete `dept`:
```
delEntity(s2,s2a,[dept,Y],[],[],[att,person,dname,_,Y]).
```

The four steps ⑤- ⑧ taken as a compound transformation implement the well-known equivalence between attributes and entity classes [1].

Example 3 below gives the reverse transformation of Example 2 which is automatically derived by the tool. Conversely, the database integrator could have specified Example 3 and the tool would then automatically derive Example 2.

Example 3 Transforming s2 to s1

```
9  addEntity(s2a,s2,[dept,Y],[],[],[att,person,dname,_,Y]),
10 addAttribute(s2b,s2a,[dept,dname,X,Y],[[1,1],[1,1]],
       [],[implies,[att,dept,dname,X,Y],[eq,X,Y]],
       [att,person,dname,X,Y]),
11 addRelationship(s2c,s2b,[person,worksin,dept,X,Y],
       [[1,1],[1,n]],[],[],[att,person,dname,X,Y]),
12 delAttribute(s4,s2c,[person,dname,X,Y],[[1,1],[1,n]],[],[],
       [rel,person,worksin,dept,X,Y]),
13 addAttribute(s1c,s4,[person,sex,EV,AV],[[1,1],[1,n]],[],[],
       [or,[and,[ent,male,EV],[eq,AV,m]],
          [and,[ent,female,EV],[eq,AV,f]]]),
14 delGeneralisation(s1b,s1c,[total,person,[male,female]],
       [implies,[att,person,sex,_,AV],[or,[eq,AV,m],[eq,AV,f]]]),
15 delEntity(s1a,s1b,[female,EV],[],[],[att,person,sex,EV,f]),
16 delEntity(s1,s1a,[male,EV],[],[],[att,person,sex,EV,m]).
```

4.3 Data migration

Example 4 below shows how the extent of each construct in s2 is defined in terms of the extents of constructs in s1, together with which step (if any) of the transformation from s1 to s2 is used. Example 5 similarly shows how the extent of each construct in s1 is defined in terms of the extents of constructs in s2, together with which step of the transformation from s2 to s1 is used.

Example 4 Migrating data from s1 to s2

s2	s1	Step
[ent,person,X]	[ent,person,X]	-
[att,person,id,X,Y]	[att,person,id,X,Y]	-
[att,person,name,X,Y]	[att,person,name,X,Y]	-
[att,person,dname,X,Y]	[rel,person,worksin,dept,X,Y]	5
[ent,male,X]	[att,person,sex,EV,m]	1
[ent,female,X]	[att,person,sex,EV,f]	2

Example 5 Migrating data from s2 to s1

s1	s2	Step
[ent,person,X]	[ent,person,X]	-
[att,person,id,X,Y]	[att,person,id,X,Y]	-
[att,person,name,X,Y]	[att,person,name,X,Y]	-
[att,person,sex,X,Y]	[or,[and,[ent,male,X],[eq,Y,m]], [and,[ent,female,X],[eq,Y,f]]]	13
[rel,person,worksin,dept]	[att,person,dname,X,Y]	11
[ent,dept,Y]	[att,person,dname,_,Y]	9
[att,dept,dname,X,Y]	[att,person,dname,X,Y]	10

4.4 Query and Update Translation

Since the tool handles bi-directional transformations, there is no distinction between migration and translation of queries and updates, so we use the latter term. For any query on `s1`, the table in Example 5 can be used to translate constructs of `s1` into ones of `s2`, resulting in a query on `s2`. Conversely, for any query on `s2`, the table in Example 4 can be used to translate constructs of `s2` into ones of `s1`, resulting in a query on `s1`.

Example 6 Translation of queries from `s1` to `s2` "Find the `id`s of all females that work in Computing." (Since `id` is the key attribute of `person`, we assume that person entities are identical to their `id` attributes.)

```
Query on s1                Step Query on s2
[and,                        -  [and,
 [att,person,sex,Id,f],     ⑬   [or,[and,[ent,male,Id],[eq,f,m]],
                                     [and,[ent,female,Id],[eq,f,f]]],
 [rel,person,worksin,       ⑪   [att,person,dname,Id,maths]]
    dept,Id,computing]]
```

The same method as for query translation can be used to translate updates:

Example 7 Translation of inserts from `s2` to `s1` An insertion on `s2` of a male person with `id` 1000 and `name 'Peter'`, can be translated into an insertion on `s1` of a person with with `id` 1000, `name 'Peter'` and `sex m`:

```
Insert into s2          Step Insert into s1
[and,                     -  [and,
 [ent,person,1000],       -   [ent,person,1000],
 [and,                    -   [and,
  [ent,male,1000],        ①    [att,person,sex,1000,m],
  [att,person,name,       -    [att,person,name,
     1000,'Peter']]]      -       1000,'Peter']]]
```

4.5 Transformations between non-equivalent schemas

Higher-level transformations such as *extendEntity*, *contractEntity*, *extendRelationship*, *contractRelationship*, and so forth are straightforwardly defined in terms of the low-level *extend* and *contract* ones. To illustrate their use, Example 8 below shows how we may transform schema `s3` in Figure 1 to schema `s4`.

Example 8 Transforming `s3` to `s4`

```
⑰ expandAttribute(s5,s3,[person,name,_,_]).
⑱ contractAttribute(s5a,s5,[degree,code,_,_]),
⑲ contractAttribute(s5b,s5a,[degree,title,_,_]),
⑳ contractRelationship(s3c,s3b,[degree,runby,dept,_,_]),
㉑ contractEntity(s4,s5c,[degree,_]),
```

s4 is to one of the intermediate schemas produced in Examples 2 and 3. Thus, we can use steps 17-21 above followed by steps 13-16 in Example 3 to transform s3 to s1 and steps 17-21 followed by steps 5-8 in Example 2 to transform s3 to s2. As before, the reverse steps 22-26 below may be derived from 17-21, to transform s4 to s3:

Example 9 Transforming s4 to s3

```
22 expandEntity(s4,s5c,[degree,_]),
23 expandRelationship(s5c,s3b,[degree,runby,dept,_,_]),
24 expandAttribute(s5b,s5a,[degree,title,_,_]),
25 expandAttribute(s5a,s5,[degree,code,_,_]),
26 contractAttribute(s5,s3,[person,name,_,_]).
```

Data, updates and queries can be translated between pairs of non-equivalent schemas so long as this is confined to their common constructs. If a query uses some construct which has no derivation in the target schema, as for example in Example 10 below, then the resulting query will contain a void term, which our tool will return as invalid (in a production version of the tool it may be possible to give partial answers, using those parts of the query that were successfully translated).

Example 10 Invalid translation of queries from s3 to s1 "Find the id's of the persons and the dname of departments which are all involved with the degree programme with code='G500'."

Query on s3	Step	Query on s1
[and,	-	[and,
[rel,person,	-	[rel,person,
worksin,dept,Id,DName],		worksin,dept,Id,DName],
[rel,degree,runby,dept,'G500',F]]	20	[void]]

4.6 Global Schemas and Queries

In Figure 1, schema s5 may be regarded as a federated schema which contains the union of the information of the three source schemas s1, s2 and s3. Since s5 extends (as defined in Section 2.3) all three source schemas, there will be some global queries on s5 which will result in void queries whichever source schema they are mapped onto.

To illustrate, Example 11 below first shows how a global query on s5 would be translated to each of the three source schemas. It then shows how these three individual translations can be amalgamated into an overall global query plan. The construct [ask,schema,query] identifies that a sub-query can be directed to a particular database. If more than one database contains the information, the various alternative queries are placed in construct [plan,ask1,...,askn], indicating that some further query planning is required in order to choose one of the queries for execution. At this stage, consideration of the query processing capabilities of the various databases needs to be taken into account.

Example 11 Query decomposition "Find the `names` and `ids` of the persons and the `dname` of departments which are all involved with the degree programme with `code='G500'`."

```
Query on s5                     Step Query on s1
[and,                            -   [and,
 [att,person,name,Id,Name],      -     [att,person,name,Id,Name],
 [and,                           -    [and,
  [rel,person,worksin,           -     [rel,person,worksin,
     dept,Id,DName],                      dept,Id,DName],
  [rel,degree,runby,             20    [void]
     dept,'G500',DName]]]            ]]
```

```
Query on s5                     Step Query on s2
[and,                            -   [and,
 [att,person,name,Id,Name],      -     [att,person,name,Id,Name],
 [and,                           -    [and,
  [rel,person,worksin            11    [att,person,
     dept,Id,DName],                      dname,Id,DName],
  [rel,degree,runby              20    [void]
     dept,'G500',DName]]]            ]]
```

```
Query on s5                      Step Query on s3
[and,                             -   [and,
 [att,person,name,Id,Name],       26   [void],
 [and,                            -    [and,
  [rel,person,                    -     [rel,person,
     worksin,dept,Id,DName],      -        worksin,dept,Id,DName],
  [rel,degree,runby,              -     [rel,degree,runby,
     dept,'G500',DName]]]         -        dept,'G500',DName]]]
```

```
Query on s5                Global Query Plan
[and,                      [and,
 [att,person,name,          [plan,
     Id,Name],               [ask,s1,[att,person,name,Id,Name]],
                             [ask,s2,[att,person,name,Id,Name]]
                            ],
 [and,                      [and,
  [rel,person,worksin,       [plan,
     dept,Id,DName],          [ask,s1,
                                [rel,person,worksin,dept,Id,DName]],
                              [ask,s2,
                                [att,person,dname,Id,DName]],
                              [ask,s3,
                                [rel,person,worksin,dept,Id,DName]]
                             ],
  [rel,degree,runby,         [ask,s3,[rel,degree,runby,
     dept,'G500',DName]         dept,'G500',DName]]
]]                         ]]
```

5 Conclusions

This paper has described a formal framework for schema transformation which allows the reverse of a schema transformation to be derived automatically. This reversibility allows data, queries and updates to be automatically migrated or translated in either direction between two equivalent schemas S_1 and S_2 provided that a transformation between them has been defined. We have demonstrated how the approach can also be used when S_1 and S_2 are not equivalent, and we have illustrated the approach in use in a simple prototype schema integration tool. The approach can readily be extended to translate more sophisticated application logic such as constraints, deductive rules and active rules.

Our approach is readily applicable within all the main database interoperation architectures [6]. For example, in mediator architectures, it can be used to automatically generate those parts of wrappers that handle the translation between different semantic models. It can also be used to help handle changes in source databases: if a change of a database schema S to a schema S' is represented by a transformation from S to S' then any query that previously was used on S can be automatically translated to a query on S'.

In more recent work [11], we have developed a generic framework for defining the semantics of higher-level modelling language constructs in terms of the HDM. The definitions of the higher-level primitive transformations are then automatically derived. In that paper we also show how these transformations can be used to map between schemas expressed in different modelling languages. In combination with the work described here, this allows us to automatically transform queries between schemas defined in different modelling languages. Also, it is possible to define inter-model links, which support the development of stronger coupling between different modelling languages than is provided by current approaches.

Our future work will follow two main directions:

- *Embedding the Primitive Transformations into a Programming Language*: The primitive transformations can be extended into a full language for writing schema transformation programs e.g. by adding an iteration construct, a conditional branching construct, and procedures. This would allow the definition of more flexible general-purpose transformations.
- *A Graphical Tool*: The prototype tool we have described here can be used as the basis for a more sophisticated tool, for example one supporting the graphical display and manipulation of schemas, and predefined templates for the common schema transformations.

References

1. C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
2. S.S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J.D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous

information sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, pages 7–18, October 1994.

3. T. Critchlow, M. Ganesh, and R. Musick. Automatic generation of warehouse mediators using an ontology engine. In *Proceedings of the 5th International Workshop on Knowledge Represenation Meets Databases (KRDB '98)*, volume 10. CEUR Workshop Proceedings, 1998.
4. U. Dayal and P. Bernstein. On the updatability of relational views. In *Proc. 5th International Conference on Very Large Data Bases (VLDB 79)*, Rio de Janeiro, Brazil, October 1979.
5. L.M. Haas, D. Kossmann, E.L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd VLDB Conference*, pages 276–285, Athens, Greece, 1997.
6. R. Hull. Managing sematic heterogeneity in databases: A theoretical perspective. In *Proceedings of PODS*, 1997.
7. R. Langerak. View updates in relational databases with independent scheme. *ACM Transactions on Database Systems*, 15(1), March 1990.
8. P.J. McBrien. A reference implementation for a formal approach to schema transformation. Technical Report TR98-09, King's College London, 1998.
9. P.J. McBrien and A. Poulovassilis. A formal framework for ER schema transformation. In *Proceedings of ER'97*, volume 1331 of *LNCS*, pages 408–421, 1997.
10. P.J. McBrien and A. Poulovassilis. A formalisation of semantic schema integration. *Information Systems*, 23(5):307–334, 1998.
11. P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer-Verlag, 1999.
12. A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.
13. M.T. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for data sources. In *Proceedings of the 23rd VLDB Conference*, pages 266–275, Athens, Greece, 1997.
14. A. Sheth and J. Larson. Federated database systems. *ACM Computing Surveys*, 22(3):183–236, 1990.
15. 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, April 1995.
16. M.E. Vidal, L. Raschid, and J-R. Gruser. A meta-wrapper for scaling up to multiple autonomous distributed information sources. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS98)*, pages 148–157. IEEE-CS Press, 1998.