Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach

Peter McBrien
Dept. of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, pjm@doc.ic.ac.uk

Alexandra Poulovassilis
Dept. of Computer Science, Birkbeck College, University of London,
Malet Street, London WC1E 7HX, ap@dcs.bbk.ac.uk

Monday 27th March 2000

Abstract

In previous work we have a developed general framework to support schema transformation and integration in heterogeneous database architectures. The framework consists 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. A key feature of the framework is that both primitive and composite schema transformations are automatically reversible. We have shown in earlier work how this allows automatic query translation from a global schema to a set of source schemas. In this paper we show how our framework also readily supports *evolution of source schemas*, allowing the global schema and the query translation pathways to be easily repaired, as opposed to having to be regenerated, after changes to source schemas.

1 Introduction

Common to many methods for integrating heterogeneous data sources is the requirement for **logical integration** [19, 9] of the data, due to variations in the design of data models for the same universe of discourse. Logical integration requires that we are able to transform and integrate a set of source schemas into a global schema, and to translate queries posed on the global schema to queries posed on the source schemas. Our previous work [13, 14, 17, 15] has provided a general formalism to underpin schema transformation and integration, and automatic translation of queries posed on a global schema to queries posed on the source schemas. Our approach is applicable to all three main database interoperation architectures [9]: **federation**, **mediator** and **workflow**. In this paper we consider the problem of evolving the global schema and repairing the query translation pathways in the face of source schema evolution.

Current implementations of database interoperation, such as TSIMMIS [6], InterViso [20], IM [12] and Garlic [18], are what may be termed *query-oriented*. They provide mechanisms by which users define global schema constructs as views over source schema constructs (or vice versa in the case of IM) but do not focus on the semantics of the data sources. More recent work on automatic wrapper generation [21, 7, 2, 8] and agent-based mediation [3] is also query-oriented. In contrast, our approach is *schema-oriented* in that we provide mechanisms by which the user specifies transformations on schemas. These transformations are then used to automate the translation of queries between global and source schemas.

Our approach has several advantages over the query-oriented one: (i) focusing the human input to the integration process where it is most needed, namely on the semantics of the data

sources, rather than on the more automatable query processing aspects; (ii) decomposing the transformation/integration of schemas into a sequence of small steps by the provision of a set of primitive transformations which can be incrementally composed into more complex ones; (iii) using the transformation pathways between schemas to automatically translate queries posed on a global schema to queries posed on a set of source schemas; and, as we will see below, (iv) enabling the systematic repair of global schemas and global query translation in the face of evolving source schemas.

Much of the work on schema evolution has presented approaches in terms of just one data model e.g. ER [1], OO [4, 5] or workflow [10]. In contrast, our approach of representing higher-level data modelling languages in terms of an underlying hypergraph-based data model allows us to propose in this paper a method which can be applied to any of the common data modelling languages.

In [11] it was argued that a uniform approach to schema evolution and schema integration is both desirable and possible. The higher-order logic *SchemaLog* was used to describe the relationship between schemas, contrasting with our approach which uses a simple set of schema transformation primitives augmented with 'standard' first-order logic. A particular advantage of our approach is that we clearly distinguish between equivalent and non-equivalent constructs in different schemas, and hence are able to distinguish between queries that can and cannot be translated between the two schemas. This ability to specify *capacity-augmentations* is also present in the approach of [4], but this approach is specific to O2 and not readily transferable to other data models.

The remainder of this paper is structured as follows. In Section 2 we review the hypergraph data model (HDM) that underpins our approach and the primitive transformations on schemas defined in this data model. We also present a concrete version of the framework that adopts a specific query language (Datalog). In Section 3 we show how global schemas and global query translation can be repaired in the face of source schema evolution, considering in particular the evolution of a source schema into a semantically equivalent, semantically contracted or or semantically expanded schema. In Section 4 we show how the same approach applies to the repair of global schemas defined using higher-level modelling languages than the HDM. Section 5 gives our concluding remarks and directions for further work.

2 The Schema Transformation/Integration Framework

2.1 Review of our previous work

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 may link any number of both nodes and other edges (this facility is needed in order to support higher-level constructs such as composite attributes and attributes on relations [17]). It is a directed hypergraph because edges link sequences of nodes or edges. $Constraints$ is 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 $\{Ext_{S,I}(c) \mid c \in Nodes \cup Edges\}$;

[1] Since this is a framework, the query language is not a specific one.

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

(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 $Nodes \cup Edges$.

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 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. *addCons* c adds a new constraint c. Proviso: c evaluates to true.
4. *delCons* 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.
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 also 3-ary version which takes as an extra argument a condition which must be satisfied in order for the transformation to be successful. A key point to note is that the query q in transformations 5 and 7 ensures that the construct being added is semantically redundant by stating how it can be derived from the other schema constructs. Similarly, the query q in transformations 6 and 8 ensures that the construct being deleted is semantically redundant.

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.

We first developed these definitions of schemas, instances, and schema equivalence in the context of an ER common data model, in [13, 14]. A detailed comparison with other approaches to schema equivalence and schema transformation can be found in [14]. The Hypergraph Data Model and the primitive transformations for it were first introduced in [17] where we showed how

higher-level modelling languages and schema transformations for them can be defined in terms of this lower-level HDM.

In [16] we developed a generic method for defining the semantics of higher-level modelling languages in terms of the HDM, showing how the set of primitive schema transformations for such higher-level modelling languages can then be *automatically derived*. These transformations can be used to map between schemas expressed in the same or different modelling languages. The unifying underlying HDM allows constructs from different modelling languages to be mixed within the same schema, and it is also possible to define "inter-model" links between such constructs. This is particularly useful in integration situations where there is not a single common data model that can fully represent the constructs of all the data sources.

In [15] we developed a second distinguishing feature of our framework, namely that schema transformations defined on the HDM, or on higher-level constructs defined in terms of it, are *automatically reversible*. In particular, for every primitive transformation t such that $t(\langle S, I, Ext_{S,I}\rangle) \neq \phi$ there exists a transformation $\overline{t}$ such that $\overline{t}(t(\langle S, I, Ext_{S,I}\rangle)) = \langle S, I, Ext_{S,I}\rangle$, as shown 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 $\overline{t}$:

Transformation (t)	Reverse Transformation ($\overline{t}$)
$renameNode\ \langle from, to\rangle\ c$	$renameNode\ \langle to, from\rangle$
$renameEdge\ \langle\langle from, schemes\rangle, to\rangle\ c$	$renameEdge\ \langle\langle to, schemes\rangle, from\rangle$
$addCons\ q\ c$	$delCons\ q$
$delCons\ q\ c$	$addCons\ q$
$addNode\ \langle n, q\rangle\ c$	$delNode\ \langle n, q\rangle$
$delNode\ \langle n, q\rangle\ c$	$addNode\ \langle n, q\rangle$
$addEdge\ \langle e, q\rangle\ c$	$delEdge\ \langle e, q\rangle$
$delEdge\ \langle e, q\rangle\ c$	$addEdge\ \langle e, q\rangle$

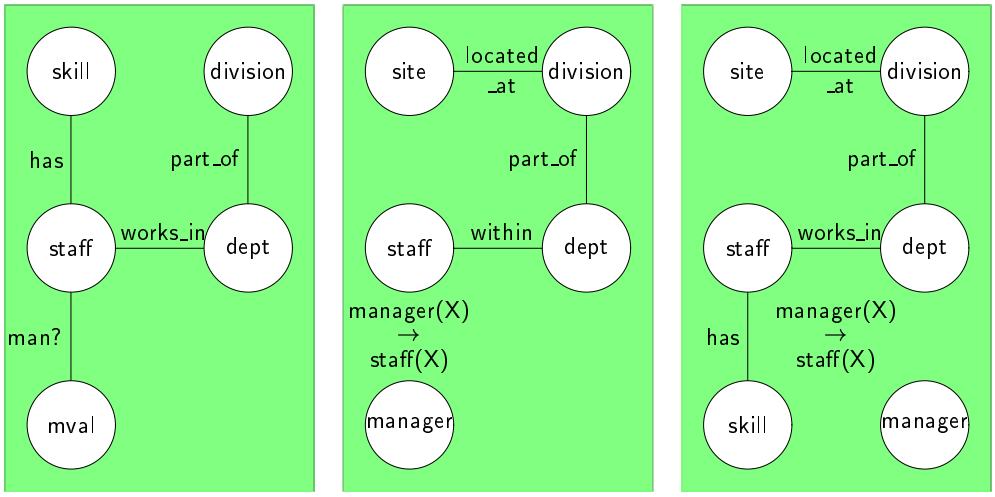
In [15] we defined four more low-level transformations in order to also allow transformations between overlapping schemas rather than just between equivalent schemas e.g. between a source schema and a global schema. These new transformations are defined in terms of the existing transformations as follows, where for *extend* transformations a VOID query indicates that the new construct cannot be derived from the existing schema constructs while for *contract* transformations a VOID query similarly indicates that the removed construct cannot be derived from the remaining schema constructs:

$$\begin{array}{lcllcl} extendNode\ n & = & addNode\ \langle n, \text{VOID}\rangle & extendEdge\ e & = & addEdge\ \langle e, \text{VOID}\rangle \\ contractNode\ n & = & delNode\ \langle n, \text{VOID}\rangle & contractEdge\ e & = & delEdge\ \langle e, \text{VOID}\rangle \end{array}$$

The reversibility of primitive transformations generalises to any successful composite transformation: for any such composite transformation $T = t_1; \ldots; t_n$ its reverse composite transformation is $\overline{T} = \overline{t_n}; \ldots; \overline{t_1}$. Thus, our schema transformations set up a *two-way transformation pathway* between pairs of schemas. In [15] we show how these pathways can be used to automatically translate queries in either direction between a pair of semantically equivalent or overlapping schemas. In particular, if S_1 is transformed to S_2 by a single primitive transformation, the only cases we need to consider in order to translate 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, these substitutions are successively applied in order to obtain the final translated query q_2.



(a) S_1: staff skills

(b) S_2: staff locations

(c) S: global schema, integration of S_1 and S_2

Figure 1: Two HDM source schemas, and a global schema

2.2 A Concrete Schema Transformation/Integration Language

In order to illustrate the schema transformation/integration framework reviewed above, and to more easily illustrate our approach to handing schema evolution later in the paper, henceforth we will use a concrete version of the framework which assumes that all schemas are expressed at the low level of the HDM and that all queries and constraints are expressed in Datalog. We illustrate this concrete schema transformation/integration language by means of two examples below which show how it can be used to transform and integrate source schemas into a global schema, and how query translation between the global schema and the source schemas is automatically supported.

Our examples use the source schemas S_1 and S_2 and the global schema S illustrated in Figure 1. S_1 contains information about staff and whether or not they are of manager grade, skills, departments and divisions. (The node mval has a two-valued extent {true,false}. Instances of staff who are managers are linked to true by an instance of man? while other staff are linked to false.) S_2 contains information about staff, managers (who are constrained to be a sub-class of people by the Datalog rule manager(X) $\rightarrow$ staff(X)), departments, divisions, and sites. S is an integration of this information, with within of S_2 having been renamed to works_in, and the class manager of S_2 being used rather than the attribute mval of S_1.

Example 2.1 Building the global schema

S_1 and S_2 are transformed into S by the two composite transformations listed below. We have labelled the primitive transformation steps of these transformations as we will be referring to them later in the paper. In step 1, the statement manager(X) $\vdash$ man?(X,true) states how to populate the extent of the new node manager from the extents of the existing schema constructs (indicating that manager adds no new information to the schema). In particular, the extent of manager is populated by those instances of staff which are linked to the value true of mval by an instance of man?. In step 3, the statement staff#man?#mval identifies the edge being deleted and the statement man?(X,Y) $\vdash$ manager(X),Y=true; staff(X),not manager(X),Y=false states how the extent of the deleted edge man? can be recovered from the remaining schema constructs (indicating that man? is a redundant construct). In particular, an instance (m,true) is created for each manager m and an instance (s,false) is created for each staff member s who is not a manager.

In step 4, the statement mval(X) ⊢ X=true;X=false states how the extent of the deleted node mval can be recovered, in this case by a simple enumeration of its two values. The rest of the syntax is straight-forward.

transformation $S_1 \rightarrow S$
begin
① *addNode* manager(X) ⊢ man?(X,true)
② *addCons* (manager(X) → staff(X))
③ *delEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
④ *delNode* mval(X) ⊢ X=true;X=false
⑤ *extendNode* site
⑥ *extendEdge* division#located_at#site
end

transformation $S_2 \rightarrow S$
begin
⑦ *renameEdge* within works_in
⑧ *extendNode* skill
⑨ *extendEdge* staff#has#skill
end

The reverse transformations from S to S_1 and from S to S_2 are automatically derivable from the above two transformations, as discussed in Section 2.1, and are as follows:

transformation $S \rightarrow S_1$
begin
⑥̄ *contractEdge* division#located_at#site
⑤̄ *contractNode* site
④̄ *addNode* mval(X) ⊢ X=true;X=false
③̄ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
②̄ *delCons* (manager(X) → staff(X))
①̄ *delNode* manager(X) ⊢ man?(X,true)
end

transformation $S \rightarrow S_2$
begin
⑨̄ *contractEdge* staff#has#skill
⑧̄ *contractNode* skill
⑦̄ *renameEdge* works_in within
end

□

Example 2.2 Query translation

The transformations $S \rightarrow S_1$ and $S \rightarrow S_2$ can be used to automatically translate queries posed on S to queries over S_1 and S_2. For example, the following query on S asks for the staff, skills, departments and divisions of all staff based in London:

has(X,Y), works_in(X,V), part_of(V,W), located_at(W,'London')

The translation of each conjunct of this query onto S_1 and S_2 is shown below, together with the transformation step, if any, which is significant. Notice that the first conjunct can be answered from S_1 only, the fourth conjunct from S_2 only, and the other two conjuncts from both source schemas.

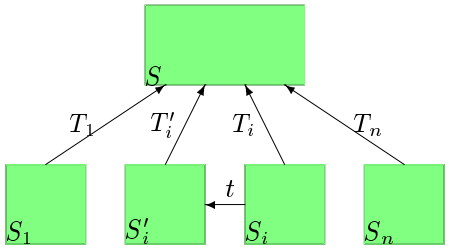


Figure 2: Evolution of a source schema S_i

Translation	has(X,Y)	,	works_in(X,V)	,	part_of(V,W)	,	VOID
to S_1	↑	↑	↑	↑	↑	↑	↑ ⑥
Global query	has(X,Y)	,	works_in(X,V)	,	part_of(V,W)	,	located_at(W,'London')
Translation	↓ ⑨	↓	↓ ⑦	↓	↓	↓	↓
to S_2	VOID	,	within(X,V)	,	part_of(V,W)	,	located_at(W,'London')

We may use this translation to build the global query plan listed below, where the construct ask(SchemaList,Q) means that query Q can be executed on any of the schemas in SchemaList:

ask([S_1],has(X,Y)),
(ask([S_1],works_in(X,V));ask([S_2],within(X,V))),
ask([S_1,S_2],part_of(V,W)),
ask([S_2],located_at(W,'London'))

□

3 Handling Evolution of Source Schemas

We now consider how global schemas can be repaired (as opposed to regenerated) in order to reflect changes in source schemas, and how query translation operates over the repaired global schema. Although our examples assume HDM schemas and Datalog queries/constraints, our treatment is fully general and applies to higher-level schema constructs and other query formalisms. We will see this in Section 4 where this is discussed further.

Let us suppose then that there are n source schemas S_1, ..., S_n which have been transformed and integrated into a global schema S. There are thus available n transformations $T_1 : S_1 \rightarrow S$, ..., $T_n : S_n \rightarrow S$. From these, the reverse transformations $\overline{T_1} : S \rightarrow S_1$, ..., $\overline{T_n} : S \rightarrow S_n$ are automatically derivable and can be used to translate queries posed on S to queries on $S_1, ..., S_n$.

The source schema evolution problem that we consider is illustrated in Figure 2 and is as follows: if some source schema S_i evolves, to S'_i say, how should S be repaired to reflect this change and how should queries on the repaired S now be translated in order to operate on S'_i rather than on S_i ?

Without loss of generality, we need to consider only changes on S_i that consist of a *single* primitive transformation step, since changes that are composite transformations can be handled as a sequence of primitive transformations. There are three classes of primitive transformations to consider:

- those that result in a new schema S'_i which is equivalent S_i,
- those where S'_i is a contraction of S_i, and
- those where S'_i is an extension of S_i.

In each of these cases, suppose that S_i is transformed to S_i' by the application of a primitive transformation t. Then we can automatically derive a new transformation pathway T_i' from S_i' to S to be

$$T_i' = \overline{t}; T_i$$

and a new transformation pathway $\overline{T_i'}$ from S to S_i' to be

$$\overline{T_i'} = \overline{T_i}; t$$

As we will see in the rest of this section, if t is an equivalence-preserving transformation, we are done, apart from perhaps some simplification of the resulting T_i' and $\overline{T_i'}$. However, in the case of contractions and extensions some more work needs to be done.

3.1 Equivalence-preserving transformations

Suppose that t is an equivalence-preserving transformation, so that S_i' is equivalent to S_i. Then T_i' and $\overline{T_i'}$ as defined above provide a new automatic translation pathway between S and the new source schema S_i'.

Example 3.1

Suppose that it has been decided to evolve schema S_1 in Figure 1 to a new equivalent schema S_1^a which models the notion of a manager in the same way as S_2 (see Figure 3(a)). This can be achieved by the following composite transformation:

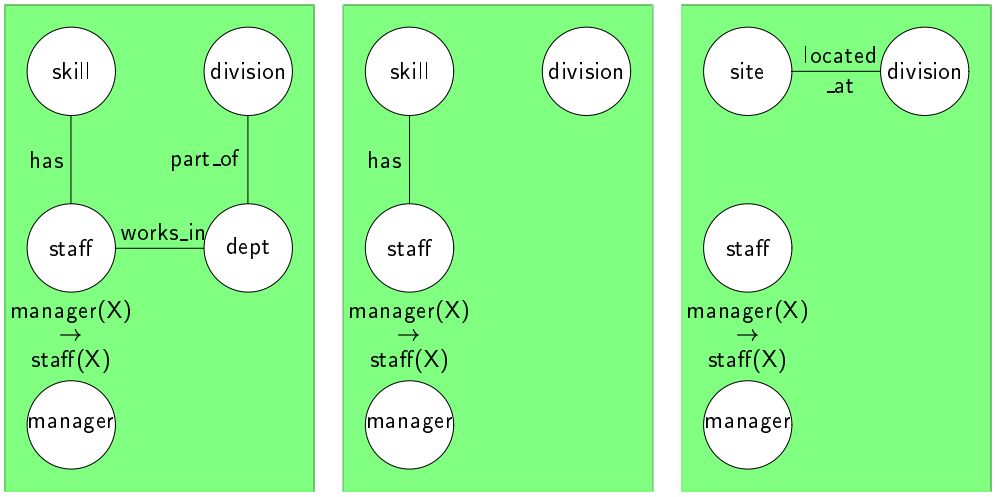
transformation $S_1 \rightarrow S_1^a$
begin
⑩ *addNode* manager(X) ⊢ man?(X,true)
⑪ *addCons* (manager(X) → staff(X))
⑫ *delEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
⑬ *delNode* mval(X) ⊢ X=true;X=false
end

The reverse transformation from S_1^a to S_1 can be derived to be:

transformation $S_1^a \rightarrow S_1$
begin
$\overline{⑬}$ *addNode* mval(X) ⊢ X=true;X=false
$\overline{⑫}$ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
$\overline{⑪}$ *delCons* (manager(X) → staff(X))
$\overline{⑩}$ *delNode* manager(X) ⊢ man?(X,true)
end

The new transformation from S_1^a to S is obtained by prefixing the transformation steps $\overline{13}$-$\overline{10}$ to the transformation $S_1 \rightarrow S$ of Example 2.1:

transformation $S_1^a \rightarrow S$
begin
$\overline{⑬}$ *addNode* mval(X) ⊢ X=true;X=false
$\overline{⑫}$ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
$\overline{⑪}$ *delCons* (manager(X) → staff(X))
$\overline{⑩}$ *delNode* manager(X) ⊢ man?(X,true)
① *addNode* manager(X) ⊢ man?(X,true)
② *addCons* (manager(X) → staff(X))
③ *delEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false



(a) S_1^a (b) S_1^b (c) S_2^a

Figure 3: Evolution of source schemas

④ *delNode* mval(X) ⊢ X=true;X=false
⑤ *extendNode* site
⑥ *extendEdge* division#located_at#site
end

Conversely, the new transformation from S to S_1^a is obtained by appending the transformation steps 10-13 to the transformation $S \rightarrow S_1$ of Example 2.1:

transformation $S \rightarrow S_1^a$
begin
⑥̄ *contractEdge* division#located_at#site
⑤̄ *contractNode* site
④̄ *addNode* mval(X) ⊢ X=true;X=false
③ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
②̄ *delCons* (manager(X) → staff(X))
①̄ *delNode* manager(X) ⊢ man?(X,true)
⑩ *addNode* manager(X) ⊢ man?(X,true)
⑪ *addCons* (manager(X) → staff(X))
⑫ *delEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
⑬ *delNode* mval(X) ⊢ X=true;X=false
end

This new transformation can now be used to automatically translate queries posed on S to queries on S_1^a rather than on S_1.

□

3.2 Removing redundant transformation steps

In composite transformations such as those above there may be pairs of primitive transformation steps which are inverses of each other and which can be removed without altering the overall effect

of the transformation. In particular, a composite transformation

$$T; t; T'; \overline{t}; T''$$

where T, T', T'' are arbitrary sequences of primitive transformations, t is primitive transformation and $\overline{t}$ is its inverse, can be simplified to

$$T; T'; T''$$

provided that there are no references within T' to the construct being renamed, added or deleted by t.

For example, in the transformation $S_1^a \rightarrow S$, steps $\overline{10}$ and 1 can be removed, followed by $\overline{11}$ and 2, followed by $\overline{12}$ and 3, followed by $\overline{13}$ and 4, obtaining the following expected simplified transformation:

transformation $S_1^a \rightarrow S$
begin
⑤ *extendNode* site
⑥ *extendEdge* division#located_at#site
end

Similarly, in the transformation $S \rightarrow S_1^a$, steps $\overline{1}$ and 10 can be removed, followed by $\overline{2}$ and 11, followed by $\overline{3}$ and 12, followed by $\overline{4}$ and 13, obtaining the following expected simplified transformation:

transformation $S \rightarrow S_1^a$
begin
$\overline{⑥}$ *contractEdge* division#located_at#site
$\overline{⑤}$ *contractNode* site
end

3.3 Contraction transformations

Suppose S_i is transformed to S_i' by a primitive transformation t of the form *contract* c. The new transformation pathway from S to S_i' is $\overline{T_i}$; *contract* c. Any sub-queries over S that translate to sub-queries involving construct c of S_i will now correctly be replaced by the value VOID over S_i'.

However, after a series of contractions on source schemas, the global schema S may eventually contain constructs that are no longer supported by any source schema. How can S be repaired so that it no longer contains such unsupported constructs ? One way is by *dynamic repair* during query processing: if a sub-query posed on a construct of S returns VOID for all possible local sub-queries, then that construct can be removed from S. Note that if this construct is a node, removal of the node from S must be preceded by removal from S of any edges that it participates in.

Another way to repair S if it contains constructs that are no longer supported by any source schema is by *static repair*. With this approach, we can first use T_i to trace how the removed construct c of S_i is represented in S — call this global representation $global(c)$. We can then use the transformations $\overline{T_j}$ $j \neq i$ to trace how $global(c)$ is represented in all the other source schemas S_j, $j \neq i$. If all of these source constructs have VOID extents, then we can remove $global(c)$ from S (again taking care to precede the removal of a node by removal of any edges that it participates in).

Example 3.2 illustrates how the removal of a construct from one source schema may still allow the construct to be derived from another source schema, or may require the removal of the construct from the global schema.

Example 3.2 Contractions of source schemas

Suppose the owner of schema S_1^a has decided not to export information about departments. We can derive the schema S_1^b illustrated in Figure 3(b) from S_1^a as follows:

transformation $S_1^a \rightarrow S_1^b$
begin
⑭ *contractEdge* staff#works_in#dept
⑮ *contractEdge* dept#part_of#division
⑯ *contractNode* dept
end

Adopting a static repair approach to repairing S, if necessary, we would check the transformation paths of the contracted constructs dept, works_in and part_of from S to S_1^b and S_2, and would discover that all of them still map to a non-VOID extent in S_2. Thus, S would not be changed.

Adopting a dynamic repair approach, attempting to pose on S_1^b queries over the dept, works_in or part_of constructs gives a VOID result. However, such query fragments can still be posed on S_2 (see for example the query in Example 2.2) and so S is not changed.

Suppose now that S_2 is also transformed in a similar manner, resulting in S_2^a illustrated in Figure 3(c):

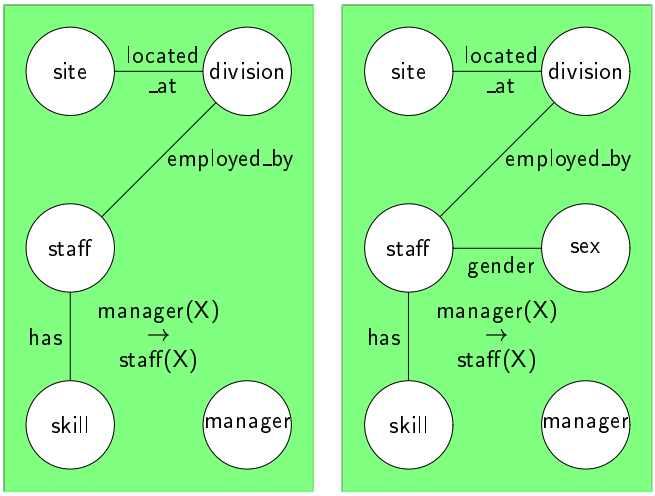
transformation $S_2 \rightarrow S_2^a$
begin
⑰ *contractEdge* staff#within#dept
⑱ *contractEdge* dept#part_of#division
⑲ *contractNode* dept
end

At this stage the transformations from S to S_1^b and S_2^a are as follows (the reverse transformations are straight-forward so we don't list them):

transformation $S \rightarrow S_1^b$
begin
$\overline{⑥}$ *contractEdge* division#located_at#site
$\overline{⑤}$ *contractNode* site
⑭ *contractEdge* staff#works_in#dept
⑮ *contractEdge* dept#part_of#division
⑯ *contractNode* dept
end

transformation $S \rightarrow S_2^a$
begin
$\overline{⑨}$ *contractEdge* staff#has#skill
$\overline{⑧}$ *contractNode* skill
$\overline{⑦}$ *renameEdge* works_in within
⑰ *contractEdge* staff#within#dept
⑱ *contractEdge* dept#part_of#division
⑲ *contractNode* dept
end

Adopting a static repair approach we would again check the transformation paths of the contracted constructs dept, works_in and part_of from S to S_1^b and S_2^a. In this case we see that all three of them map to a VOID extent in both source schemas. We can thus remove these constructs from S, obtaining the schema S^a illustrated in Figure 4. We also need to remove the corresponding *contract* steps, and any prior renamings of these constructs, from the transformations from S^a to the source schemas S_1^b and S_2^a (we similarly remove the corresponding *extend* steps from the



(a) S^a (b) S^b

Figure 4: Evolution of global schemas

reverse transformations).

With a dynamic repair approach we would instead wait until a query such as that of Example 2.2 is asked, find that some fragments of it translate to VOID on all source schemas, and remove those corresponding constructs from S and from the transformation pathways between it and the source schemas.

With both approaches, the resulting transformations from S^a to S_1^b and S_2^a are:

transformation $S^a \rightarrow S_1^b$
begin
$\overline{6}$ *contractEdge* division#located_at#site
$\overline{5}$ *contractNode* site
end

transformation $S^a \rightarrow S_2^a$
begin
$\overline{9}$ *contractEdge* staff#has#skill
$\overline{8}$ *contractNode* skill
end

□

Notice that it is not actually *wrong* in our framework for a construct from a global schema to map to a VOID extent in all source schemas (for example, a new source schema may later be added that does support an extent for this construct, and this is likely to be a common situation in mediator architectures). Sub-queries over such constructs merely translate to VOID. So the repair steps we have described above are simply a matter of "tidying up" the global schema and are optional.

3.4 Extension transformations

Suppose S_i is transformed to S_i' by a primitive transformation t of the form *extend* c, meaning that a new construct c is now supported by S_i' that is not derivable from S_i. Naively, the new

transformation pathway from S to S'_i is $\overline{T_i}; extend\ c$. However, this may be incorrect and there are generally four cases that we need to be consider:

1. The construct c does not appear in S but can be derived from existing constructs of S by some transformation T.

   In this case, S is transformed to a new global schema S' that contains c by appending T to the transformation from each local schema to the original S. The transformation pathway from S' to S'_i then simplifies to just $\overline{T_i}$ i.e. $\overline{T}$ and *extend* c are inverses of each other and can be removed.

2. c does not appear in S and cannot be derived from the existing constructs of S.

   In this case, S is transformed to a new global schema S' that contains c by appending the step *extend* c to the transformation from each local schema to the original S. The reverse transformation from S' to S'_i thus consists of an initial *contract* c step. This matches up with the newly appended *extend* c step. This pair of steps should be removed in order for the new extent of c in S'_i to be usable by queries posed on S'.

3. c already appears in S and has the same semantics as the newly added c in S'_i.

   In this case there must be a transformation step *contract* c in the original transformation from S to S_i. This matches up with the newly appended *extend* c in the transformation from S to S'_i. This pair of steps should be removed in order for the new extent of c in S'_i to be usable by queries posed on S.

4. c already appears in S but has different semantics to the newly added c in S'_i.

   In this case there must again be a transformation step *contract* c in the original transformation from S to S_i. Now, the new construct c in S'_i needs to be renamed to some name that does not appear in S, c', say. The resulting transformation from S to S'_i is $\overline{T_i}; extend\ c'; rename\ c'c$, and the situation reverts to case 2 above.

Notice that, by analogy to our remark at the end of Section 3.3 that it is not compulsory to repair the global schema after a series of contractions have left a global schema construct unsupported by any source schema, it is similarly not compulsory to extend the global schema after a new construct is added to a source schema in cases 2 and 4 above. If this is the choice, then the *extend* c step is not appended to the transformations from the source schemas to the global schema, and the final *extend* c step remains in the transformation from S to S'_i.

Example 3.3 Extensions of source schemas

Suppose that the owner of S_2^a has decided to extend it with information about staff members' skills and their sex. This can be achieved by the following transformation:

transformation $S_2^a \rightarrow S_2^b$
begin
⑳ *extendNode* skill
㉑ *extendEdge* staff#has#skill
㉒ *extendNode* sex
㉓ *extendEdge* staff#gender#sex
end

Comparing S_2^b with S^a, we see that the constructs introduced by ⑳ and ㉑ already appear in S^a. Thus for these two constructs we need to choose between cases 3 and 4 above. Let us suppose that both constructs have the same semantics in S^a and S_2^b, so that case 3 holds. Examining the transformations $S^a \rightarrow S_2^a$ and $S_2^a \rightarrow S_2^b$, we eliminate the redundant pair $\overline{⑧}$ and ⑳, and the redundant pair $\overline{⑨}$ and ㉑. This results in the following transformation from S^a to S_2^b:

transformation $S^a \rightarrow S_2^b$
begin
㉒ *extendNode* sex
㉓ *extendEdge* staff#gender#sex
end

Suppose now that the constructs introduced by ㉒ and ㉓ are entirely new ones, so that case 2 above applies. We can extend S^a to S^b using these same two extend steps. We then remove the two redundant contract/extend pairs from the transformation from S^b to S_2^b, giving the expected identity transformation from S_2^b and S^b.

□

3.5 Handling derived source schema constructs

So far we have considered only extensional schema constructs in the source schemas. However, source schemas may also contain intentional constructs. For example, let us return to the original schemas of Figure 1 and suppose S_1 evolves to include a derived edge employed_by which is populated by the join of works_in and dept:

addEdge staff#employed_by#division employed_by(X,Z) ⊢ works_in(X,Y),part_of(Y,Z)

If we now attempted to apply the contractions of steps ⑭ - ⑯ in Example 3.2, this would leave us with no way of populating the extent of the derived edge. Thus, generally contracting a source schema construct that some other intensional construct depends on is not permitted.

If we want to make the above change to S_1 then the add step should be expressed as an extend step, indicating that employed_by is a new extensional construct. The overall transformation is as follows (its reverse $S_1' \rightarrow S_1$ is straightforwardly derived):

transformation $S_1 \rightarrow S_1'$
begin
㉔ *extendEdge* staff#employed_by#division
㉕ *contractEdge* staff#works_in#dept
㉖ *contractEdge* dept#part_of#division
㉗ *contractNode* dept
end

The overall transformation from the global schema S to the new source schema S_1' is as follows (its reverse $S_1' \rightarrow S$ is again straightforwardly derived):

transformation $S \rightarrow S_1'$
begin
⑥̄ *contractEdge* division#located_at#site
⑤̄ *contractNode* site
④̄ *addNode* mval(X) ⊢ X=true;X=false
③̄ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
②̄ *delCons* (manager(X) → staff(X))
①̄ *delNode* manager(X) ⊢ man?(X,true)
㉔ *extendEdge* staff#employed_by#division
㉕ *contractEdge* staff#works_in#dept
㉖ *contractEdge* dept#part_of#division
㉗ *contractNode* dept
end

We can now apply the treatment of Sections 3.3 and 3.4 to repair the global schema S. Firstly comparing S_1' with S we determine that although employed_by does not appear in S it can be

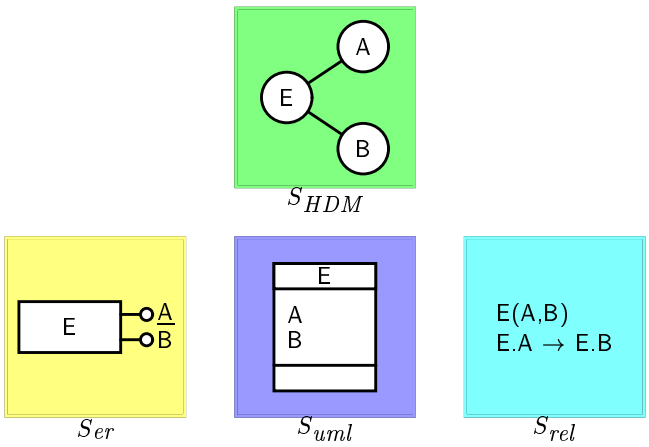


Figure 5: Multiple models based on the HDM

derived from the works_in and part_of constructs of S i.e. case 1 of Section 3.4 pertains. We thus convert S to a new global schema S' by appending the step

addEdge staff#employed_by#division employed_by(X,Z) ⊢ works_in(X,Y),part_of(Y,Z)

to the transformations $S'_1 \rightarrow S$ and $S_2 \rightarrow S$, and prefixing the reverse step

delEdge staff#employed_by#division employed_by(X,Z) ⊢ works_in(X,Y),part_of(Y,Z)

to the transformations $S' \rightarrow S'_1$ and $S' \rightarrow S_2$. This results in an inverse pair of delete/extend steps in the transformation $S' \rightarrow S'_1$ (and similarly of add/contract steps in $S'_1 \rightarrow S$) which can be removed, giving:

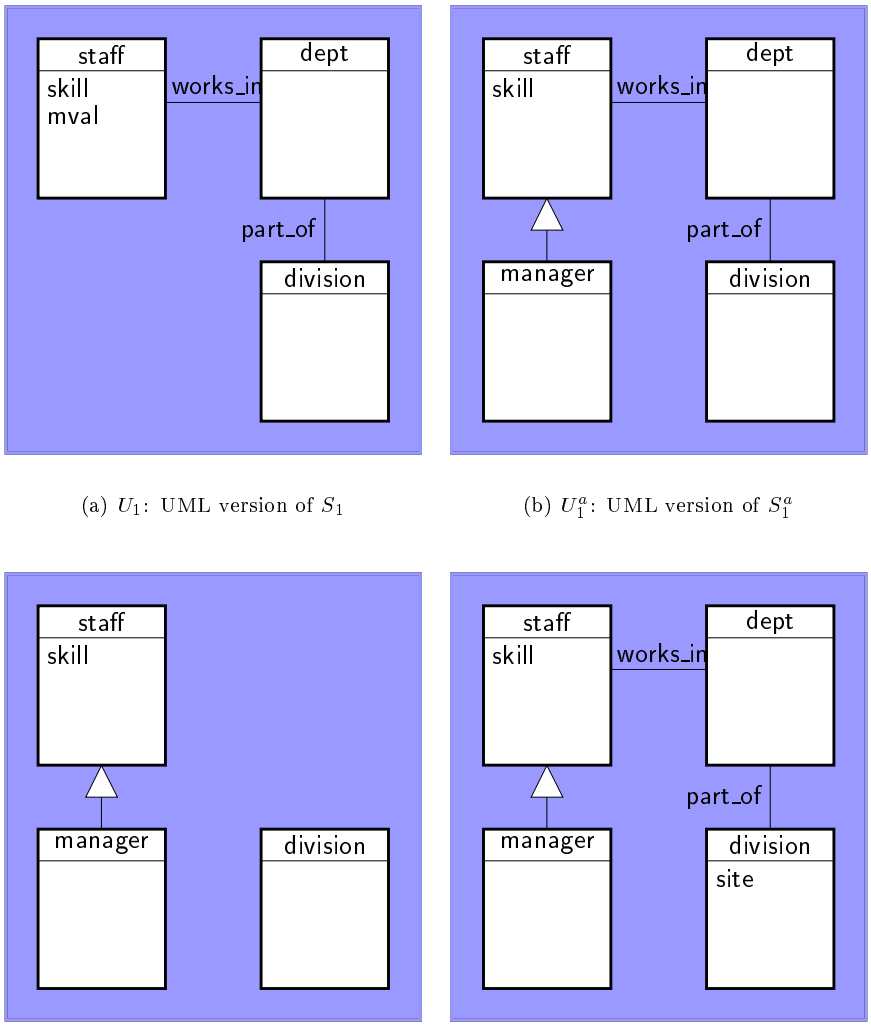
transformation $S' \rightarrow S'_1$
begin
⑥̄ *contractEdge* division#located_at#site
⑤̄ *contractNode* site
④̄ *addNode* mval(X) ⊢ X=true;X=false
③̄ *addEdge* staff#man?#mval man?(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
②̄ *delCons* (manager(X) → staff(X))
①̄ *delNode* manager(X) ⊢ man?(X,true)
㉕ *contractEdge* staff#works_in#dept
㉖ *contractEdge* dept#part_of#division
㉗ *contractNode* dept
end

We can now handle the last three contraction steps ㉕-㉗ as in Section 3.3.

4 Schemas using Higher Level Modelling Constructs

In [16] we showed how higher-level modelling languages can be expressed using the low-level constructs of the HDM. This is illustrated in Figure 5 which shows three higher-level schemas and an HDM schema. The constructs of the three higher-level modelling languages (ER, UML and relational) are represented by nodes and edges in the underlying HDM. All three higher-level schemas illustrated have a common HDM representation as the graph with three nodes and two edges shown in S_{HDM}. There is also an (unillustrated) constraint in S_{uml} and in S_{HDM} which captures the fact that each instance of A is associated with at most one instance of E.

In [16] we showed how that once a construct c in a higher level modelling language L has been defined as a set of constructs $c_1, \ldots, c_n$ in the HDM, then *add*, *del* and *rename* primitive transformations on c can be automatically derived as sequences of primitive HDM transformations on $c_1, \ldots, c_n$. In particular, we show how primitive transformations for UML schemas such



(a) U_1: UML version of S_1

(b) U_1^a: UML version of S_1^a

(c) U_1^b: UML version of S_1^b

(d) U: UML version of S

Figure 6: Transformations on UML class diagrams

as *addClass, delClass, addAttribute, delAttribute, addAssociation, delAssociation, addGeneralisation, delGeneralisation*, are defined in terms of the lower-level primitive transformations on HDM schemas presented in Section 2.1. We now illustrate how this technique can be applied to the schema evolution methodology described in this paper.

Figure 6(a) illustrates a UML class diagram U_1 which is semantically equivalent to the HDM schema S_1 of Figure 1(a). We have represented staff, dept and division as UML classes, and mval and skill as attributes of staff. The works_in and part_of are relationships are represented as UML associations.

Figure 6(d) illustrates a UML class diagram U which is semantically equivalent to the HDM schema S of Figure 1(c). manager is represented as a UML class and the generalisation hierarchy between staff and manager is the counterpart of the constraint manager(X) $\rightarrow$ staff(X) in S.

Transforming U_1 to U may be achieved by the following steps:

transformation $U_1 \rightarrow U$
begin

Ⓤ1 *addClass* manager(X) ⊢ staff#mval(X,true)
Ⓤ2 *addGeneralisation* staff#manager
Ⓤ3 *delAttribute* staff#mval(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
Ⓤ4 *extendAttribute* division#site
end

Note that each of the steps in $U_1 \rightarrow U$ may be equated with one or more of the steps in $S_1 \rightarrow S$. In particular, Ⓤ1 adding UML class manager is equivalent to ① (adding the manager node), Ⓤ2 adding the UML generalisation is equivalent to ② (adding a constraint), Ⓤ3 deleting a UML attribute is equivalent to ③ and ④ (deleting an edge and node), and Ⓤ4 extending the UML schema with an attribute is equivalent to ⑤ and ⑥ (extending the HDM schema with a node and an edge).

Figure 6(b) illustrates a schema U_1^a which is semantically equivalent to S_1^a. The following is an equivalence-preserving transformation from U_1 to U_1^a (again, we may draw an equivalence between the steps in this transformation with those in $S_1 \rightarrow S_1^a$):

transformation $U_1 \rightarrow U_1^a$
begin
Ⓤ5 *addClass* manager(X) ⊢ staff#mval(X,true)
Ⓤ6 *addGeneralisation* staff#manager
Ⓤ7 *delAttribute* staff#mval(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
end

Since all UML schema transformations are equivalent to some HDM schema transformation, our analysis from Section 3 can be applied to the evolution of source schemas expressed in UML. For example, since $U_1 \rightarrow U_1^a$ is an equivalence-preserving transformation, then the steps of $U_1 \rightarrow U_1^a$ allow any query on U that used to execute on U_1 to instead execute on U_1^a using the transformation pathway $U \rightarrow U_1; U_1 \rightarrow U_1^a$:

transformation $U \rightarrow U_1^a$
begin
$\widehat{\text{Ⓤ4}}$ *contractAttribute* division#site
$\widehat{\text{Ⓤ3}}$ *addAttribute* staff#mval(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
$\widehat{\text{Ⓤ2}}$ *delGeneralisation* staff#manager
$\widehat{\text{Ⓤ1}}$ *delClass* manager(X) ⊢ staff#mval(X,true)
Ⓤ5 *addClass* manager(X) ⊢ staff#mval(X,true)
Ⓤ6 *addGeneralisation* staff#manager
Ⓤ7 *delAttribute* staff#mval(X,Y) ⊢ manager(X),Y=true; staff(X),not manager(X),Y=false
end

Applying the removal of redundant pairs of transformations in Section 3.2 to the higher-level of UML level simplifies this to:

transformation $U \rightarrow U_1^a$
begin
Ⓤ4 *contractAttribute* division#site
end

Our analysis of contraction and extension of HDM source schemas can also be applied to the UML level in the obvious way. For example, the following UML contraction transformation from U_1^a to U_1^b is equivalent to the HDM contraction transformation $S_1^a \rightarrow S_1^b$:

transformation $U_1^a \rightarrow U_1^b$
begin
Ⓤ8 *contractAssociation* works_in

Ⓤ9 *contractAssociation* part_of
Ⓤ10 *contractClass* dept
end

Thus, after we have performed $U_1^a \rightarrow U_1^b$, we have the same analysis as in Section 3.3 where dept, works_in and part_of in U will map to VOID in U_1^b, but still map to an non-VOID extent in U_2 (the unillustrated UML equivalent of S_2).

The remaining examples of Sections 3.3 and 3.4 transfer to the higher-level of UML in a similar straightforward way.

5 Summary and Conclusions

In this paper we have shown how our framework for schema transformation can provide a comprehensive uniform approach to handling both schema integration and schema evolution in heterogeneous database architectures. Source schemas are integrated into a global schema by applying a sequence of primitive transformations to them. The evolution of a source schema into a new schema can be described using the same set of primitive transformations. The transformations between the source schemas and the global schema can be used to systematically repair the global schema and the global query translation pathways after changes to source schemas.

Our framework is based on a low-level hypergraph-based data model (HDM) whose primitive constructs are nodes, edges and constraints. In previous work we have shown how the HDM supports the representation of a wide range of higher-level data modelling languages [16] and how use of our set of primitive schema transformations enables automatic query translation between semantically equivalent or overlapping schemas [15].

Our use of the relatively simple HDM means that our approach to schema evolution is straightforward to analyse, but is also applicable to real-world modelling situations using more complex data models for database schemas. Moreover, our use of the HDM as a common underlying representation permits transformations to be defined which derive constructs of one data modelling language from constructs of another data modelling language [16]. This means that we can integrate and evolve source schemas that are expressed in different data modelling languages without first having to translate them into one common data model. For example, an administrator of a relational database may specify the evolution of that database's schema using the primitive schema transformations of the relational model even if that source schema was transformed into a UML global schema.

For future work we plan to extend our schema transformation/integration framework to handle dynamic aspects of databases such as ECA rules, in addition to static aspects. The condition and action parts of ECA rules are relatively straightforward to transform (barring for the usual problems of view updates). However, the translation of the event part of ECA rules, and the interaction between translated actions and translated events, needs more careful study.

References

[1] J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *Proceedings of VLDB'91*. Morgan-Kaufman.

[2] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, December 1997.

[3] R.J. Bayardo *et al.* InfoSleuth: Agent-based semantic integration of information in open and dynamic environments. *SIGMOD Record*, 26(2):195–206, June 1997.

[4] Z. Bellahsene. View mechanism for schema evolution in object-oriented dbms. In *Advances in Databases: 14th British National Conference on Databases, BNCOD14*, volume 1094, pages 18–35. Springer-Verlag, 1996.

[5] B. Benatallah. A unified framework for supporting dynamic schema evolution in object databases. In *Proceedings of ER99*, LNCS, pages 16–30. Springer-Verlag, 1999.

[6] 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 *Proc. 10th Meeting of the Information Processing Society of Japan*, pages 7–18, October 1994.

[7] T. Critchlow, M. Ganesh, and R. Musick. Automatic generation of warehouse mediators using an ontology engine. In *Proc. 5th International Workshop on Knowledge Represenation Meets Databases (KRDB '98)*, volume 10. CEUR Workshop Proceedings, 1998.

[8] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system. *SIGMOD Record*, 26(2):532–535, June 1997.

[9] R. Hull. Managing sematic heterogeneity in databases: A theoretical perspective. In *Proceedings of PODS*, 1997.

[10] M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proceedings of CoopIS 1999*, pages 104–114, 1999.

[11] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proceedings of DOOD'93*, pages 81–100, Phoenix, AZ, December 1993.

[12] A. Levy, A. Rajamaran, and J.Ordille. Querying heterogeneous information sources using source description. In *Proc 22nd VLDB*, pages 252–262, 1996.

[13] P.J. McBrien and A. Poulovassilis. A formal framework for ER schema transformation. In *Proc. ER'97*, volume 1331 of *LNCS*, pages 408–421. Springer-Verlag, 1997.

[14] P.J. McBrien and A. Poulovassilis. A formalisation of semantic schema integration. *Information Systems*, 23(5):307–334, 1998.

[15] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proc. ER'99*, volume 1728 of *LNCS*, pages 96–113. Springer-Verlag, 1999.

[16] 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.

[17] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

[18] M.T. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for data sources. In *Proc. 23rd VLDB Conference*, pages 266–275, Athens, Greece, 1997.

[19] A. Sheth and J. Larson. Federated database systems. *ACM Computing Surveys*, 22(3):183–236, 1990.

[20] M. Templeton, H.Henley, E.Maros, and D.J. Van Buer. InterViso: Dealing with the complexity of federated database access. *The VLDB Journal*, 4(2):287–317, 1995.

[21] M.E. Vidal, L. Raschid, and J-R. Gruser. A meta-wrapper for scaling up to multiple autonomous distributed information sources. In *Proc. 3rd IFCIS Int. Conf. on Cooperative Information Systems (CoopIS98)*, pages 148–157. IEEE-CS Press, 1998.