Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach

AutoMed Technical Report 5, Version 1

Peter McBrien[1] and Alexandra Poulovassilis[2]
[1] Dept. of Computing, Imperial College, Univ. of London, pjm@doc.ic.ac.uk
[2] School of Computer Science and Information Systems, Birkbeck College, Univ. of London, ap@dcs.bbk.ac.uk

Abstract

This paper presents a new approach to schema evolution, which combines the activities of schema integration and schema evolution into one framework. 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 data model and a set of primitive schema transformations defined for this model. Higher-level 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 shows how this 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* [22, 11] of the data, due to variations in the design of data models for the same universe of discourse. Logical integration requires facilities for transforming and integrating a set of source schemas into a global schema, and for translating queries posed on the global schema to queries over the source schemas. In previous work [16, 17, 20, 19, 18] we have developed a general framework which provides these facilities, and we are currently implementing this functionality within the AutoMed project (see http://www.doc.ic.ac.uk/automed/). 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.

Other heterogeneous database systems, such as TSIMMIS [6], InterViso [23], IM [15] and Garlic [21], 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 [24, 7, 2, 10] and agent-based mediation [3] is also query-oriented. In contrast, our approach is *schema transformation-oriented*. What we provide are 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 transformation-oriented approach has several advantages over the query-oriented approach:

- 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;

- decomposing the transformation and integration of schemas into a sequence of small steps by providing a set of primitive transformations which can be incrementally composed into more complex ones;
- 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:
- 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 [13]. In contrast, we represent such higher-level data modelling languages in terms of an underlying hypergraph-based data model [20, 19]. Thus, the techniques for handling source schema evolutions that we propose in this paper can be applied to any of the common data modelling languages. In [14] it was argued that a uniform approach to schema evolution and schema integration is both desirable and possible, and this is our view also. The higher-order logic language *SchemaLog* was used to describe the relationship between schemas, contrasting with our approach which uses a simple set of schema transformation primitives augmented with a first-order query language. 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.

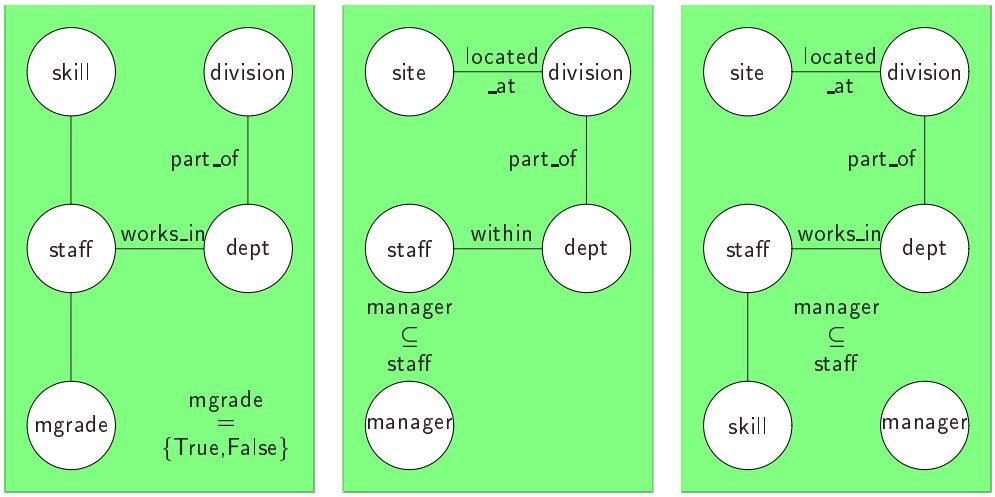
The remainder of this paper is structured as follows. In Section 2 we briefly review the hypergraph data model that underpins our approach and the primitive transformations on schemas defined in this data model. In Section 3 we show how global schemas and global query translation can be repaired in the face of source schema evolution, considering in particular the evolution of a source schema into a semantically equivalent, semantically contracted, or semantically expanded schema. Section 4 extends this approach to handling derived schema constructs. In Section 5 we show how the same approach can be used to repair global schemas defined using higher-level modelling languages. Section 6 gives our concluding remarks.

2 Review of Our Schema Transformation/Integration Framework

A **schema** in the **hypergraph data model** (HDM) is a triple $\langle Nodes, Edges, Constraints\rangle$. $Nodes$ 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 [20]). It is a directed hypergraph because edges link sequences of nodes or edges. A **query** q **over a schema** $S = \langle Nodes, Edges, Constraints\rangle$ is an expression whose variables are members of $Nodes \cup Edges$. $Constraints$ is a set of boolean-valued queries over S[1].

Nodes have unique names. Edges and constraints have an optional name associated with them. The nodes and edges of a schema are identified by their **scheme**. For a node this is of the form $\langle\langle nodeName\rangle\rangle$ and for an edge it is of the form $\langle\langle edgeName, scheme_1, scheme_2, \ldots, scheme_n\rangle\rangle$.

[1] Our framework is independent of the specific language adopted to define queries and constraints. In this paper we use a simple comprehension-based language to define queries and constraints. We do not consider here the issue of translating between this intermediate language and the query languages supported by the actual data sources, but refer the interested reader to [8] for a discussion of comprehension languages, and to [9] for a discussion of how ODMG OQL can be translated into a comprehension language similar to ours. More generally, the topic of inter-language translation is beyond the scope of this paper and is much-studied elsewhere in the literature.



(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

Figure 1 illustrates three HDM schemas that we will use as a running example in this paper.

Schema S_1 contains information about staff, whether or not they are of manager grade, the skills of each member of staff, and the department in which they work. Each department belongs to a division. The node mgrade is constrained to have a two-valued extent $\{True, False\}$. Instances of ⟨⟨staff⟩⟩ who are managers are linked to $True$ by an instance of ⟨⟨_,staff,mgrade⟩⟩ while other staff are linked to $False$.

Schema S_2 is drawn from the same domain, and contains information about staff, their departments, and the division that each department belongs to. However, S_2 differs from S_1 in several ways:

- The edge between ⟨⟨staff⟩⟩ and ⟨⟨dept⟩⟩ has a different name in the two schemas, although its "real-world" semantics are the same.
- S_2 does not contain information about staff skills.
- S_2 contains information about the site that each division is located at.
- S_2 represents the information about managers in a different manner to S_1. In particular, S_2 uses a node ⟨⟨manager⟩⟩ to hold instances of staff which are managers, together with a constraint that states that all instances of manager are also instances of staff.

Schema S is an integration of S_1 and S_2, with the edge within of S_2 having been renamed to works_in, and the node manager of S_2 being used to record information about which staff members are managers, rather than the node mgrade of S_1. We will shortly see how S_1 and S_2 are integrated into S using our framework.

2.1 Transformations on HDM Schemas

The HDM is equipped with a set of **primitive transformations** on schemas. Each transformation has a proviso associated with it which states when the transformation is **successful**. Unsuccessful transformations return an "undefined" value, ϕ. Any transformation applied to ϕ returns ϕ:

1. renNode($\langle\langle fromName\rangle\rangle, \langle\langle toName\rangle\rangle$) renames a node. Proviso: $toName$ is not already the name of some node.
2. renEdge($\langle\langle fromName, s_1, \ldots, s_m\rangle\rangle, \langle\langle toName\rangle\rangle$) 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($\langle\langle name\rangle\rangle, q$) 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($\langle\langle name\rangle\rangle, q$) deletes a node. q is a query that states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs. Proviso: the node exists and participates in no edges.
7. addEdge($\langle\langle name, c_1, \ldots, c_m\rangle\rangle, q$) 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($\langle\langle name, c_1, \ldots, c_m\rangle\rangle, q$) 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 ternary 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 key point to note is that the query q in transformations 5 and 7 above ensures that the construct being added to the schema is semantically redundant by stating how it can be derived from the existing schema constructs. Similarly, the query q in transformations 6 and 8 ensures that the construct being deleted from the schema is semantically redundant.

The Hypergraph Data Model and the primitive transformations for it were first introduced in [20], where we showed how higher-level modelling languages and schema transformations for them can be defined in terms of this lower-level HDM. One advantage of using a low-level common data model such as the HDM is that semantic mismatches between modelling constructs are avoided. Another advantage is that it provides a unifying semantics for higher-level modelling constructs.

In [19] 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 generated. 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 [18] 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 the *extend* transformations the VOID query component indicates that the new construct cannot be derived from the existing schema constructs while for the *contract* transformations a VOID query component similarly indicates that the removed construct cannot be derived from the remaining schema constructs:

$$\begin{array}{lcl} \text{extendNode}(n) & = & \text{addNode}(n, \text{VOID}) \\ \text{extendEdge}(e) & = & \text{addEdge}(e, \text{VOID}) \\ \text{contractNode}(n) & = & \text{delNode}(n, \text{VOID}) \\ \text{contractEdge}(e) & = & \text{delEdge}(e, \text{VOID}) \end{array}$$

Example 2.1 Building the global schema S.

Schemas 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 query $\{x \mid x \in \langle\langle \text{staff} \rangle\rangle; (x, True) \in \langle\langle _, \text{staff}, \text{mgrade} \rangle\rangle\}$ 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 mgrade by an instance of the $\langle\langle _, \text{staff}, \text{mgrade} \rangle\rangle$ edge.

In step ③, the query $\{(x, True) \mid x \in \langle\langle \text{manager} \rangle\rangle\} \cup \{(x, False) \mid x \in \langle\langle \text{staff} \rangle\rangle - \langle\langle \text{manager} \rangle\rangle\}$ states how the extent of the deleted edge $\langle\langle _, \text{staff}, \text{mgrade} \rangle\rangle$ can be recovered from the remaining schema constructs (indicating that this edge is a redundant construct). In particular, an instance $(x, True)$ is created for each manager x and an instance $(x, False)$ is created for each staff member x who is not a manager.

In step ⑤, the query $\{True, False\}$ states how the extent of the deleted node mgrade can be recovered, in this case by a simple enumeration of its two values. The rest of the steps are straight-forward:

transformation $S_1 \rightarrow S$
① *addNode* $\langle\langle \text{manager} \rangle\rangle$ $\{x \mid (x, True) \in \langle\langle _, \text{staff}, \text{mgrade} \rangle\rangle\}$
② *addCons* $\langle\langle \text{manager} \rangle\rangle \subseteq \langle\langle \text{staff} \rangle\rangle$
③ *delEdge* $\langle\langle _, \text{staff}, \text{mgrade} \rangle\rangle$ $\{(x, True) \mid x \in \langle\langle \text{manager} \rangle\rangle\} \cup \{(x, False) \mid x \in \langle\langle \text{staff} \rangle\rangle - \langle\langle \text{manager} \rangle\rangle\}$
④ *delCons* $\langle\langle \text{mgrade} \rangle\rangle = \{True, False\}$
⑤ *delNode* $\langle\langle \text{mgrade} \rangle\rangle$ $\{True, False\}$
⑥ *extendNode* $\langle\langle \text{site} \rangle\rangle$
⑦ *extendEdge* $\langle\langle \text{located_at}, \text{division}, \text{site} \rangle\rangle$

transformation $S_2 \rightarrow S$
⑧ *renEdge* $\langle\langle \text{within}, \text{staff}, \text{dept} \rangle\rangle$ works_in
⑨ *extendNode* $\langle\langle \text{skill} \rangle\rangle$
⑩ *extendEdge* $\langle\langle _, \text{staff}, \text{skill} \rangle\rangle$

□

2.2 Reversibility of transformations

In [18] we developed a second distinguishing feature of our framework, namely that schema transformations defined on the HDM, or on higher-level modelling languages defined in terms of it, are *automatically reversible*. In particular, for every primitive transformation t such that $t(S) \neq \phi$ there exists a transformation $\overline{t}$ such that $\overline{t}(t(S)) = S$, 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 ($\overline{t}$)
renNode($\langle\langle n \rangle\rangle, \langle\langle n' \rangle\rangle, c$)	renNode($\langle\langle n' \rangle\rangle, \langle\langle n \rangle\rangle, c$)
renEdge($\langle\langle n, c_1, \ldots, c_m \rangle\rangle, \langle\langle n' \rangle\rangle, c$)	renEdge($\langle\langle n', c_1, \ldots, c_m \rangle\rangle, \langle\langle n \rangle\rangle$)
addCons(q, c)	delCons(q)
delCons(q, c)	addCons(q)
addNode($\langle\langle n \rangle\rangle, q, c$)	delNode($\langle\langle n \rangle\rangle, q$)
delNode($\langle\langle n \rangle\rangle, q$) c	addNode($\langle\langle n \rangle\rangle, q$)
addEdge(e, q, c)	delEdge(e, q)
delEdge(e, q, c)	addEdge(e, q)

This 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 $\overline{T} = \overline{t_n}; \ldots; \overline{t_1}$.

Example 2.2 Generation of Reverse Transformations.
Using the above rules, the reverse transformations from S to S_1 and from S to S_2 are automatically generated from the two transformations given in Example 1 and are as follows:

transformation $S \rightarrow S_1$
⑦̄ *contractEdge* ⟨⟨located_at,division,site⟩⟩
⑥̄ *contractNode* ⟨⟨site⟩⟩
⑤̄ *addNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$
④̄ *addCons* ⟨⟨mgrade⟩⟩$= \{True, False\}$
③̄ *addEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$
②̄ *delCons* ⟨⟨manager⟩⟩ $\subseteq$ ⟨⟨staff⟩⟩
①̄ *delNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$

transformation $S \rightarrow S_2$
⑩̄ *contractEdge* ⟨⟨_,staff,skill⟩⟩
⑨̄ *contractNode* ⟨⟨skill⟩⟩
⑧̄ *renEdge* ⟨⟨within,staff,dept⟩⟩ works_in

□

Due to their reversibility, schema transformations set up a two-way transformation pathway between pairs of schemas. In [18] we showed 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 a schema A is transformed to a schema B by a single primitive transformation, the only cases that need be considered in order to translate a query q_A posed on A to an equivalent query q_B posed on B are to apply renamings, and to substitute occurrences of a deleted node or edge by their restoring query:

q_A *on* A	*equivalent query* q_B *on* B
$renNode\ (from, to)$	$[from/to]q_A$
$renEdge\ (\langle\langle from, schemes\rangle\rangle, to)$	$[\langle\langle from,schemes\rangle\rangle/\langle\langle to,schemes\rangle\rangle]q_A$
$delNode\ (n, q)$	$[n/q]q_A$
$delEdge\ (e, q)$	$[e/q]q_A$

For composite transformations, these substitutions are successively applied in order to obtain the final translated query q_B.

This translation scheme can be applied to each of the constructs of a global schema in order to obtain the possible derivations of each construct from the set of source schemas. These derivations can then be substituted into any query over the global schema in order to obtain an equivalent query over the source schemas [12]. To illustrate, consider the following query on schema S, which finds the skills of staff members working within divisions based at the London site:

$\{sk \mid (s, sk) \in$ ⟨⟨_,staff,skill⟩⟩;
$(s, dep) \in$ ⟨⟨works_in,staff,dept⟩⟩;
$(dep, div) \in$ ⟨⟨part_of,dept,division⟩⟩;
$(div, "London") \in$ ⟨⟨located_at,division,site⟩⟩$\}$

Applying the above translation scheme to this query, gives the following query over the constructs of S_1 and S_2, where we distinguish the constructs of these schemas by the suffixing them by 1 or 2, respectively:

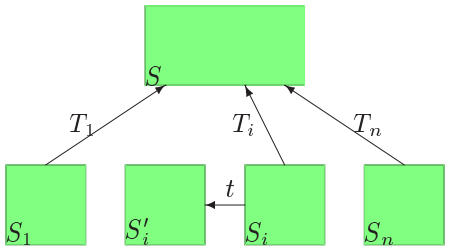


Figure 2: Evolution of a source schema S_i

$\{sk \quad | \quad (s, sk) \in \langle\langle _,\text{staff,skill} \rangle\rangle_1 \cup \text{VOID};$
$(s, dep) \in \langle\langle \text{works_in,staff,dept} \rangle\rangle_1 \cup \langle\langle \text{within,staff,dept} \rangle\rangle_2;$
$(dep, div) \in \langle\langle \text{part_of,dept,division} \rangle\rangle_1 \cup \langle\langle \text{part_of,dept,division} \rangle\rangle_2;$
$(div, \text{"}London\text{"}) \in \text{VOID} \cup \langle\langle \text{located_at,division,site} \rangle\rangle_2\}$

The first line of the query indicates that (s, sk) may only be retrieved from the $\langle\langle _,\text{staff,skill} \rangle\rangle$ edge in S_1, the second line indicates that (s, dep) may be retrieved from either $\langle\langle \text{works_in,staff,dept} \rangle\rangle$ in S_1 or from $\langle\langle \text{within,staff,dept} \rangle\rangle$ in S_2, the third line that (dep, div) may be retrieved from $\langle\langle \text{part_of,dept,division} \rangle\rangle$ in S_1 or S_2, and the last line that $(div, \text{"}London\text{"})$ should be retrieved from $\langle\langle \text{located_at,division,site} \rangle\rangle$ in S_2. Logical and physical optimisation techniques can now be applied to this translated query, in order to generate a reasonable query plan for execution (discussion of these techniques is beyond the scope of this paper).

3 Handling Evolution of Source Schemas

We turn now to the main theme of this paper, namely 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 queries/constraints expressed in a comprehension language, the treatment is fully general and applies to higher-level schema constructs and other query formalisms. We discuss this issue further in Section 5, by illustrating the approach applied to UML models.

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 generated 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?

The first step is to describe the transformation from S_i to S'_i by a transformation — in effect we 'integrate' the old and new versions of the schema. The second step is analyse what the new transformation contains. Without loss of generality, we need to consider only changes on S_i that consist of a *single* primitive transformation step t being applied to give S'_i, since changes that are composite transformations can be handled as a sequence of primitive transformations. A new transformation pathway T'_i from S'_i to S can be automatically generated:

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

as can a new transformation pathway $\overline{T'_i}$ from S to S'_i:

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

How we handle the new T_i depends on what type of transformation t was, which we divide into three classes:

1. If t was an add, delete or rename transformation, then the new schema S_i' will be equivalent S_i. In this case, all information in S that was derivable from S_i may now be derived from S_i', and no changes are required to S. Section 3.1 illustrates such an equivalence preserving transformation for our example. The only issue is that T_i might be optimised, which is considered in Section 3.2.

2. If t was a contract transformation, then there will be some information that used to be present in S_i that will no longer be available from the evolved database. Section 3.3 describes the circumstances in which this will require S to also be contracted.

3. If t was an expand transformation, then the relationship of the new construct with what is already present in S needs to be investigated, and we discuss this in Section 3.4.

A final point to note is that this treatment also covers the case where a new schema S_{n+1} is added to the set of source schemas from which S is derived, in that the original schema S_{n+1} can be regarded as an empty schema which is successively expanded with new constructs. Similarly, our treatment covers the removal of a source schema S_i, since its constructs can be successively contracted to leave an empty schema.

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:

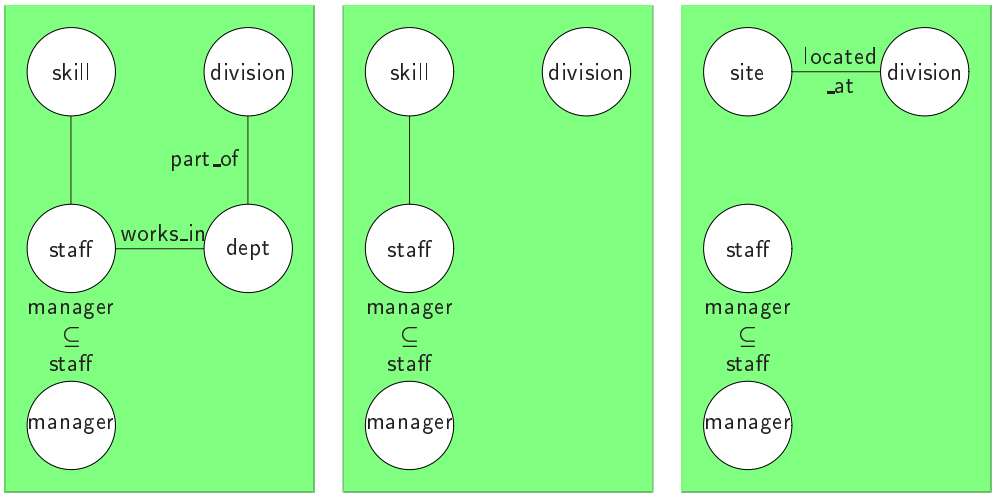
transformation $S_1 \rightarrow S_1^a$
⑪ *addNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$
⑫ *addCons* ⟨⟨manager⟩⟩ $\subseteq$ ⟨⟨staff⟩⟩
⑬ *delEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$
⑭ *delCons* ⟨⟨mgrade⟩⟩$= \{True, False\}$
⑮ *delNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$

The reverse transformation from S_1^a to S_1 is automatically generated:

transformation $S_1^a \rightarrow S_1$
⑮ *addNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$
⑭ *addCons* ⟨⟨mgrade⟩⟩$= \{True, False\}$
⑬ *addEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$
⑫ *delCons* ⟨⟨manager⟩⟩ $\subseteq$ ⟨⟨staff⟩⟩
⑪ *delNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$

The new transformation from S_1^a to S is automatically obtained by prefixing the transformation steps ⑮–⑪ to the transformation $S_1 \rightarrow S$ of Example 2.1:

transformation $S_1^a \rightarrow S$
⑮ *addNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$
⑭ *addCons* ⟨⟨mgrade⟩⟩$= \{True, False\}$
⑬ *addEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$



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

Figure 3: Evolution of source schemas

⑫ *delCons* ⟨⟨manager⟩⟩ ⊆ ⟨⟨staff⟩⟩
⑪ *delNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$
① *addNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$
② *addCons* ⟨⟨manager⟩⟩ ⊆ ⟨⟨staff⟩⟩
③ *delEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$
④ *delCons* ⟨⟨mgrade⟩⟩ $= \{True, False\}$
⑤ *delNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$
⑥ *extendNode* ⟨⟨site⟩⟩
⑦ *extendEdge* ⟨⟨located_at,division,site⟩⟩

Conversely, the new transformation from S to S_1^a is automatically obtained by appending the transformation steps ⑪–⑮ to the transformation $S \to S_1$ of Example 2.1. 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 $\widehat{\overline{11}}$ and (1) can be removed, followed by $\widehat{\overline{12}}$ and (2), followed by $\widehat{\overline{13}}$ and (3), followed by $\widehat{\overline{14}}$ and (4), followed by $\widehat{\overline{15}}$ and (5), obtaining the following expected simplified transformation:

transformation $S_1^a \rightarrow S$
(6) *extendNode* ⟨⟨site⟩⟩
(7) *extendEdge* ⟨⟨located_at,division,site⟩⟩

Similarly, in the transformation $S \rightarrow S_1^a$, steps $\widehat{\overline{1}}$ and (11) can be removed, followed by $\widehat{\overline{2}}$ and (12), followed by $\widehat{\overline{3}}$ and (13), followed by $\widehat{\overline{4}}$ and (14), followed by $\widehat{\overline{5}}$ and (15), obtaining the following expected simplified transformation:

transformation $S \rightarrow S_1^a$
$\widehat{\overline{7}}$ *contractEdge* ⟨⟨located_at,division,site⟩⟩
$\widehat{\overline{6}}$ *contractNode* site

Renaming transformations may also be redundant, and can also be removed. For example, the following simplifications can be applied to transformation sequences:
$ren\ c\ c'; del\ c' \equiv del\ c$
$ren\ c\ c'; contract\ c' \equiv contract\ c$
$add\ c'; ren\ c'\ c \equiv add\ c$
$extend\ c'; ren\ c'\ c \equiv extend\ c$
$ren\ c'\ c''; ren\ c''\ c \equiv ren\ c'\ c$

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 the 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 participates in any edges in S, then these must be removed first (if the construct has already been removed from all the source schemas, then so must any edges that it participated in there, and so such edges will also be redundant in S).

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 $global(c)$. The transformations $\overline{T_j}$ $j \neq i$ can be used 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 $global(c)$ can be removed from S (again taking care to precede the removal of a construct by removal of any edges that it participates in).

Example 3.2 illustrates how the removal of a construct from one source schema may or may not still allow the construct to be derived from another source schema.

Example 3.2 Contractions of source schemas.

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

transformation $S_1^a \rightarrow S_1^b$
(16) *contractEdge* ⟨⟨within,staff,dept⟩⟩
(17) *contractEdge* ⟨⟨part_of,dept,division⟩⟩

⑱ *contractNode* ⟨⟨dept⟩⟩

Adopting a static repair approach to repairing S involves checking 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. With a dynamic repair approach, it would be found that queries on S over dept, works_in or part_of can still be posed on S_2 and so S would again not be changed.

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

transformation $S_2 \rightarrow S_2^a$
⑲ *contractEdge* ⟨⟨within,staff,dept⟩⟩
⑳ *contractEdge* ⟨⟨part_of,dept,division⟩⟩
㉑ *contractNode* ⟨⟨dept⟩⟩

At this stage the transformations from S to S_1^b and S_2^a are as follows:

transformation $S \rightarrow S_1^b$
⑦̄ *contractEdge* ⟨⟨located_at,division,site⟩⟩
⑥̄ *contractNode* ⟨⟨site⟩⟩
⑯ *contractEdge* ⟨⟨within,staff,dept⟩⟩
⑰ *contractEdge* ⟨⟨part_of,dept,division⟩⟩
⑱ *contractNode* ⟨⟨dept⟩⟩

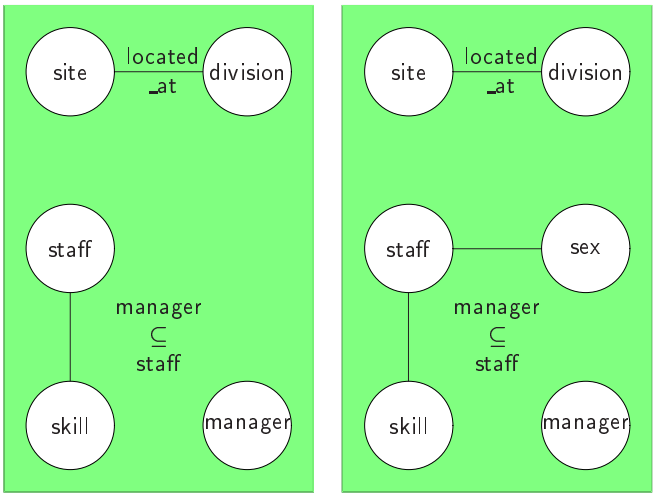
transformation $S \rightarrow S_2^a$
⑩̄ *contractEdge* ⟨⟨_,staff,skill⟩⟩
⑨̄ *contractNode* ⟨⟨skill⟩⟩
⑧̄ *renEdge* ⟨⟨within,staff,dept⟩⟩ works_in
⑲ *contractEdge* ⟨⟨within,staff,dept⟩⟩
⑳ *contractEdge* ⟨⟨part_of,dept,division⟩⟩
㉑ *contractNode* ⟨⟨dept⟩⟩

Adopting a static repair approach again means checking 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 all three of them map to a VOID extent in both source schemas. Thus, these constructs are removed from S, obtaining the schema S^a illustrated in Figure 4. The corresponding *contract* steps are also removed, as are any prior renamings of these constructs, from the transformations from S^a to the source schemas S_1^b and S_2^a (in the reverse transformations the corresponding *extend* steps would be removed). With a dynamic repair approach, it would be found that queries over dept, works_in and part_of on S translate to VOID on all source schemas, and the same actions would be taken.

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

transformation $S^a \rightarrow S_1^b$
⑦̄ *contractEdge* ⟨⟨located_at,division,site⟩⟩
⑥̄ *contractNode* ⟨⟨site⟩⟩

transformation $S^a \rightarrow S_2^a$
⑩̄ *contractEdge* ⟨⟨_,staff,skill⟩⟩
⑨̄ *contractNode* ⟨⟨skill⟩⟩



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

Figure 4: Evolution of global schemas

□

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 would merely translate to VOID.

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 $\overline{T_i}; \mathit{extend}\ c$. However, this may be incorrect and there are four alternatives to be considered:

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 must 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 must 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'; *ren* $c'c$, and the situation reverts to case 2 above.

In 1 to 4 above, determining whether the new construct c can or cannot be derived from the existing constructs of S requires domain knowledge (as does specifying the transformation T in 1) e.g. from a human expert or a domain ontology. After this is determined, the repair steps on S and the transformation pathways can be performed automatically.

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 (Figure 3(c)) has decided to extend it into a new schema S_2^b containing information about staff members' skills and their sex. This can be achieved by the following transformation:

transformation $S_2^a \rightarrow S_2^b$
㉒ *extendNode* ⟨⟨skill⟩⟩
㉓ *extendEdge* ⟨⟨_,staff,skill⟩⟩
㉔ *extendNode* ⟨⟨sex⟩⟩
㉕ *extendEdge* ⟨⟨_,staff,sex⟩⟩

Comparing S_2^b with S^a (Figure 4(a)), it is apparent that the constructs introduced by ㉒ and ㉓ already appear in S^a. For these two constructs a choice must be made 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 $S_2^a \rightarrow S_2^b$, the redundant pair $\overline{⑨}$ and ㉒ can be eliminated, as can the redundant pair $\overline{⑩}$ and ㉓. This results in the following transformation from S^a to S_2^b:

transformation $S^a \rightarrow S_2^b$
㉔ *extendNode* ⟨⟨sex⟩⟩
㉕ *extendEdge* ⟨⟨_,staff,sex⟩⟩

Suppose now that the constructs introduced by ㉔ and ㉕ are entirely new ones, so that case 2 above applies. Then S^a is extended to S^b using these same two extend steps. The two redundant contract/extend pairs are then removed from the transformation $S^b \rightarrow S_2^b$, giving the expected identity transformation from S_2^b and S^b.

□

4 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 whose extent is derived from the extents of other constructs e.g. views in relational databases.

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 ⟨⟨employed_by,staff,division⟩⟩
$\{(s, div) \mid (s, d) \in$ ⟨⟨works_in,staff,dept⟩⟩; $(d, div) \in$ ⟨⟨part_of,dept,division⟩⟩$\}$

If the contractions of steps ⑯–⑱ in Example 3.2 were applied, there would be no method of populating the extent of the derived edge. Thus, generally contracting a source schema construct that some other intensional construct depends on is not possible.

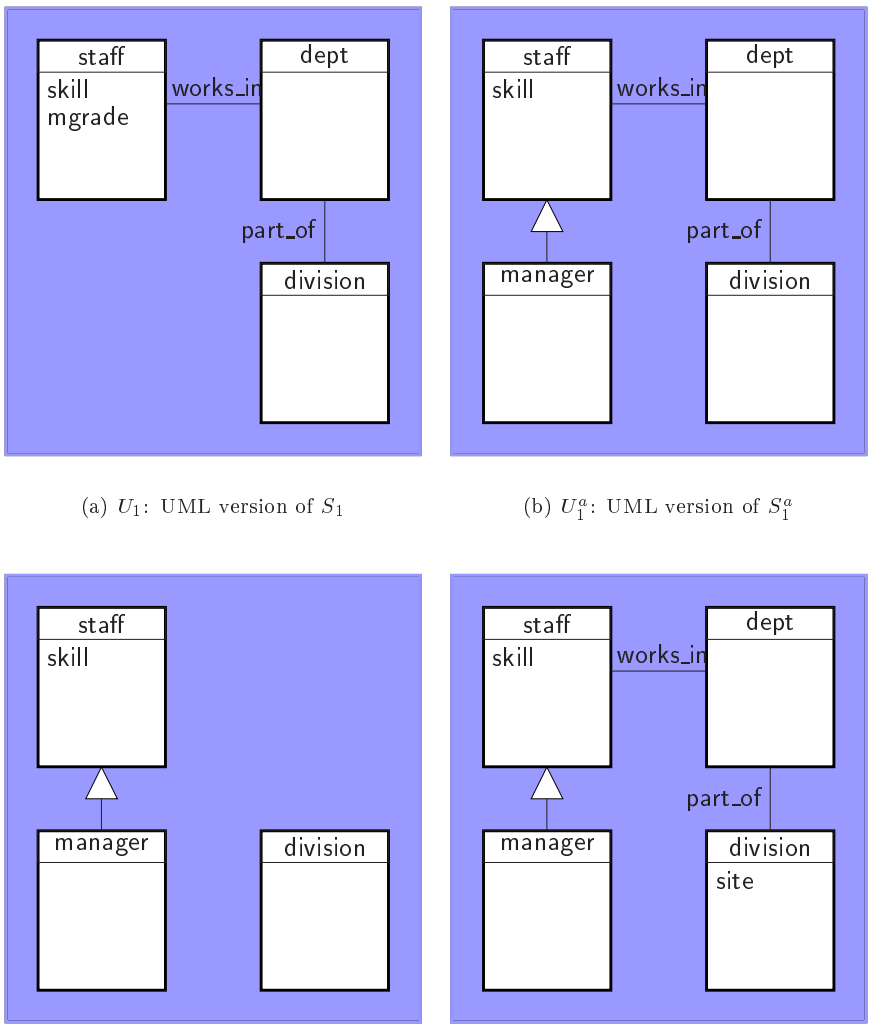
If such a change to S_1 is necessary, then the add step should instead be expressed as an extend step, indicating that employed_by is a new extensional construct. The overall transformation is as follows (its reverse $S_1' \rightarrow S_1$ is straightforwardly derived):

transformation $S_1 \rightarrow S_1'$
㉖ *extendEdge* ⟨⟨employed_by,staff,division⟩⟩
㉗ *contractEdge* ⟨⟨works_in,staff,dept⟩⟩
㉘ *contractEdge* ⟨⟨part_of,dept,div⟩⟩
㉙ *contractNode* ⟨⟨dept⟩⟩

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 $S \rightarrow S_1'$
$\overline{⑦}$ *contractEdge* ⟨⟨located_at,division,site⟩⟩
$\overline{⑥}$ *contractNode* ⟨⟨site⟩⟩
$\overline{⑤}$ *addNode* ⟨⟨mgrade⟩⟩ $\{True, False\}$
$\overline{④}$ *addCons* ⟨⟨mgrade⟩⟩$= \{True, False\}$
$\overline{③}$ *addEdge* ⟨⟨_,staff,mgrade⟩⟩ $\{(x, True) \mid x \in$ ⟨⟨manager⟩⟩$\} \cup \{(x, False) \mid x \in$ ⟨⟨staff⟩⟩ $-$ ⟨⟨manager⟩⟩$\}$
$\overline{②}$ *delCons* ⟨⟨manager⟩⟩ $\subseteq$ ⟨⟨staff⟩⟩
$\overline{①}$ *delNode* ⟨⟨manager⟩⟩ $\{x \mid (x, True) \in$ ⟨⟨_,staff,mgrade⟩⟩$\}$
㉖ *extendEdge* ⟨⟨employed_by,staff,division⟩⟩
㉗ *contractEdge* ⟨⟨works_in,staff,dept⟩⟩
㉘ *contractEdge* ⟨⟨part_of,dept,div⟩⟩
㉙ *contractNode* ⟨⟨dept⟩⟩

The techniques of Sections 3.3 and 3.4 can now be applied 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 derived from the works_in and part_of constructs of S i.e. case 1 of Section 3.4 applies. We thus convert S to a new global schema S' by appending the step
addEdge ⟨⟨employed_by,staff,division⟩⟩
$\{(s, div) \mid (s, d) \in$ ⟨⟨works_in,staff,dept⟩⟩; $(d, div) \in$ ⟨⟨part_of,dept,division⟩⟩$\}$ to the transformations $S_1' \rightarrow S$ and $S_2 \rightarrow S$, and prefixing its inverse 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 $S' \rightarrow S_1' =$ $\overline{⑦}$, $\overline{⑥}$, $\overline{⑤}$, $\overline{④}$, $\overline{③}$, $\overline{②}$, $\overline{①}$, ㉗, ㉘, ㉙. The last three contraction steps ㉗–㉙ can now be handled as in Section 3.3.



(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 5: Transformations on UML class diagrams

5 Handling Higher-Level Modelling Languages

In [19] we showed how higher-level modelling languages L can be expressed using the HDM, representing each construct c in L as a set of constructs $c_1, \ldots, c_n$ in the HDM. We then showed how *add*, *del* and *ren* primitive transformations on c can be automatically derived as sequences of primitive HDM transformations on $c_1, \ldots, c_n$. In particular, we showed how primitive transformations for UML schemas such as *addClass, delClass, addAttribute, delAttribute, addAssociation, delAssociation, addGeneralisation, delGeneralisation*, are defined in terms of the set of primitive transformations on HDM schemas presented in Section 2. Thus, the schema evolution methodology presented in Section 3 transfers directly to this higher semantic level.

To illustrate this, Figure 5(a) shows a UML class diagram U_1 which is semantically equivalent to the HDM schema S_1 of Figure 1(a). staff, dept and division are represented as UML classes, and mgrade and skill as attributes of staff. The works_in and part_of relationships are represented as UML associations. Figure 5(d) shows 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 $\subseteq$ staff in S. Transforming U_1 to U may be achieved by the following steps:

transformation $U_1 \rightarrow U$
(U1) *addClass* manager $\{x \mid x \in \text{staff}; (x, True) \in \text{staff.mgrade}\}$
(U2) *addGeneralisation* (staff,manager)
(U3) *delAttribute* staff.mgrade $\{(x, True) \mid x \in \text{manager}\} \cup \{(x, False) \mid x \in \text{staff}; not\ (x \in \text{manager})\}$
(U4) *extendAttribute* division.site

Note that each of the steps in $U_1 \rightarrow U$ equates with one or more of the steps in $S_1 \rightarrow S$. In particular, (U1) adding UML class manager is equivalent to (1) (adding the manager node), (U2) adding the UML generalisation is equivalent to (2) (adding a constraint), (U3) deleting a UML attribute is equivalent to (3) and (4) (deleting an edge and node), and (U4) extending the UML schema with an attribute is equivalent to (6) and (7) (extending the HDM schema with a node and an edge).

Figure 5(b) shows 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, the steps in this transformation can be equated with those in $S_1 \rightarrow S_1^a$):

transformation $U_1 \rightarrow U_1^a$
(U5) *addClass* manager $\{x \mid x \in \text{staff}; (x, True) \in \text{staff.mgrade}\}$
(U6) *addGeneralisation* (staff,manager)
(U7) *delAttribute* staff.mgrade $\{(x, True) \mid x \in \text{manager}\} \cup \{(x, False) \mid x \in \text{staff}; not\ (x \in \text{manager})\}$

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 $U \rightarrow U_1; U_1 \rightarrow U_1^a = \widehat{\text{U4}}, \widehat{\text{U3}}, \widehat{\text{U2}}, \widehat{\text{U1}}$, U5, U6, U7. Applying the removal of redundant pairs of transformations in Section 3.2 to the higher level of UML simplifies this to:

transformation $U \rightarrow U_1^a$
$\widehat{\text{U4}}$ *contractAttribute* division.site

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

transformation $U_1^a \rightarrow U_1^b$
(U8) *contractAssociation* works_in
(U9) *contractAssociation* part_of
(U10) *contractClass* dept

Thus, after U_1^a has evolved to U_1^b, the same analysis as in Section 3.3 applies, 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 way.

6 Concluding Remarks

In this paper we have shown how our framework for schema transformation provides a 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 same set of primitive transformations can be used to specify the evolution of a source schema into a new schema. We have shown how the transformations between the source schemas and the global schema can be used to systematically repair the global schema and the query translation pathways as source schemas evolve, considering in particular the evolution of a source schema into a semantically equivalent, semantically contracted, or semantically expanded schema. The first two cases can be handled totally automatically, and the third case semi-automatically.

Our framework is based on a low-level hypergraph-based data model whose primitive constructs are nodes, edges, and constraints. In previous work we have shown how the HDM supports the representation and transformation of a wide variety of higher-level data modelling languages. Our use of the relatively simple HDM means that our approach to schema evolution is straightforward to analyse while at the same time being applicable to real-world modelling situations using more complex data models for database schemas. We are currently implementing the schema transformation, integration and evolution functionality described here within the AutoMed project (`http://www.doc.ic.ac.uk/automed/`).

References

[1] J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *Proceedings of VLDB'91*. Morgan-Kaufman, 1991.

[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 *Proceedings of the 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 *Proceedings of the 5th International Workshop on Knowledge Represenation Meets Databases (KRDB '98)*, volume 10. CEUR Workshop Proceedings, 1998.

[8] P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[9] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *Proceedings of ACM SIGMOD*, pages 47–58, 1995.

[10] 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.

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

[12] E. Jasper. Query translation in heterogeneous database environments. MSc thesis, Birkbeck College, September 2001.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proceedings of ER99*, volume 1728 of *LNCS*, pages 96–113. Springer-Verlag, 1999.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.