Database Schema Transformation Optimisation
Techniques for the AutoMed System

Nerissa Tong
Department of Computing, Imperial College,
180 Queen’s Gate, London SW7 2BZ, United Kingdom
nyt9@doc.ic.ac.uk

Abstract. AutoMed is a database integration system that is designed to support the integration of schemas expressed in a variety of high-level conceptual modelling languages. It is based on the idea of expressing transformations of schemas as a sequence of primitive transformation steps, each of which is a bi-directional mapping between schemas. To become an efficient schema integration system in practice, where the number and size of schemas involved in the integration may be very large, the amount of time spent on the evaluation of transformations must be reduced to a minimal level. It is also important that the integrity of a set of transformations is maintained during the process of transformation optimisation. This paper discusses a new representation of schema transformations which facilitates the verification of the well-formedness of transformation sequences, and the optimisation of transformation sequences.

1 Introduction

A major task in database integration is the generation of a global schema from a collection of local source schemas of existing databases. There are three main approaches to database integration, namely global as view (GAV), local as view (LAV), and both as view (BAV) [12].

In GAV, the constructs in the global schema are defined as views over source local schemas. The popularity of this approach, which is adopted by a number of database integration systems such as TSIMMIS [6], InterViso [15], and Garlic [14], can be attributed to its simplicity of implementation. Source schemas are integrated by a set of view definitions which contain predefined query plans that describe the location and retrieval method of the required data. Query planning is made simple and efficient in this approach, however it suffers from one major drawback – when new schemas are added to the system or existing schemas are modified, all corresponding query templates will have to be rewritten.

In LAV, local schema constructs in data sources are defined as views over the global schema. Some systems adopting the LAV approach include Infomaster [5], Information Manifold [8], and Agora [9]. Query plans are computed at the time queries are submitted to the system. This approach offers greater flexibility over the GAV approach in changes in the number or contents of local schemas because
in LAV, the changes can be handled without affecting existing view definitions. The drawbacks of the LAV approach are that, (1) query processing is much more complex than in GAV, and (2) if the contents of the global schema changes, modification is then required for all the views that contain in their definition the changed global schema constructs.

In BAV, bi-directional mappings between schemas are used for transforming schemas and thus it supports evolution of both global and local schemas [11]. It is also possible to automatically derive GAV and LAV views from BAV views. Section 2 discusses the AutoMed [13] framework which adopts the BAV approach. More detailed discussion on the conversion of GAV and LAV into BAV views and vice versa can be found in [12]. The flexibility of the BAV approach allows transformations to be manipulated for optimisation purposes. Section 3 describes new techniques we have developed for the optimisation of transformation sequences. Section 4 concludes the paper with some remarks on the possible extension in the applicability of our optimisation techniques.

2 The AutoMed Framework

The AutoMed framework supports the integration of schemas that are expressed in different data modelling languages. The use of a high-level data model as the Common Data Model (CDM) in the global schema makes it very complicated to map constructs of local schemas, which possibly use different data models, with one another. This is because, typically, high-level models provide a richer set of modelling constructs, and hence a concept may be represented in a number of ways. To avoid this complication, the AutoMed framework uses the Hypergraph Data Model (HDM) [13], a low-level hypergraph-based data model, as the CDM.

The constructs contained in the HDM are Node, Edge, and Constraint. An HDM schema $S$ is then a triple containing a set of Nodes, a set of Edges, and a set of Constraints — $S = \langle \text{Nodes, Edges, Constraints} \rangle$. Nodes and Edges have a scheme and Constraints are boolean-valued queries over $S$. The scheme of a node is $\langle N \rangle$, where $N$ is the name of the node. The scheme of an edge is $\langle E, N_1, \ldots, N_n \rangle$, where $E$ is the name of the edge and $N_1, \ldots, N_n$ are the nodes connected by $E$. A set of mappings between higher-level model constructs and HDM constructs is defined. A set of primitive transformations has been defined to transform HDM models. The operators of these transformations include add, delete and rename for semantically equivalent schemas, extend and contract for semantically overlapping (non-equivalent) schemas, and id for use only in the implementation of the AutoMed system. By using the mappings between constructs of different models, schemas and transformations can be translated.

1 It is optional to give an edge a name: where an edge is not given a name, its scheme will be $\langle E, N_1, \ldots, N_n \rangle$.
2 The id transformations are special transformations that are used only in the implementation of the AutoMed system. They are used for mapping Java object references that point to two semantically equivalent constructs. More details on id transformations can be found in [3].
from one modelling language to another. Table 1 shows some of the primitive transformations available for transforming ER models and their corresponding transformations expressed in the HDM.

<table>
<thead>
<tr>
<th>ER transformations</th>
<th>HDM transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>addEnt($\langle N \rangle, q \rangle$)</td>
<td>addNode($\langle N \rangle, q \rangle$)</td>
</tr>
<tr>
<td>addAtt($\langle N, A \rangle, q \rangle$)</td>
<td>addNode($\langle N : A \rangle$, ${ Y</td>
</tr>
<tr>
<td>addRel($\langle R, N_1, \ldots, N_n \rangle, q \rangle$)</td>
<td>addEdge($\langle R, N_1, \ldots, N_n \rangle, q \rangle$)</td>
</tr>
<tr>
<td>addGen($\langle \langle G, N, N_1, \ldots, N_n \rangle \rangle \rangle$)</td>
<td>addCons($N_1 \subseteq N \rangle$, $\ldots$, addCons($N_n \subseteq N \rangle$)</td>
</tr>
</tbody>
</table>

**Table 1.** Example primitive transformations

In AutoMed [1] two schemas $S_1$ and $S_2$ are transformed into each other by incrementally applying to them a set of primitive transformations. This set of transformations forms the pathway between $S_1$ and $S_2$. A distinguishing feature of the AutoMed approach is that transformations are automatically reversible, i.e., transformations are bi-directional, thus pathways are also bi-directional. This is achieved by embedding in each transformation the extent of the construct created or removed by the transformation. The extent is expressed as a query $q$, as shown in Table 1, which defines how the data associated with the new/removed construct can be derived from other existing constructs in the original schema. Note that some transformations do not contain $q$. This means that the new/removed construct cannot be derived from existing constructs in the original schema. The reader is referred to [10] for a more detailed discussion on the AutoMed transformations and its current state of implementation [3, 2]. Table 2 shows some example ER transformations $t$ and their reversed form $t^\dagger$. The reversibility of transformations enables automatic translation of queries posed on any schema into appropriate queries on a particular target schema, as long as there exists a pathway between the schemas. To illustrate how schemas

\[
\begin{align*}
t : S_x & \rightarrow S_y \\
t^\dagger : S_y & \rightarrow S_x \\
\text{addEnt}(\langle N \rangle, q \rangle) & \rightarrow \text{deleteEnt}(\langle N \rangle, q \rangle) \\
\text{addAtt}(\langle N, A \rangle, q) & \rightarrow \text{deleteAtt}(\langle N, A \rangle, q) \\
\text{deleteEnt}(\langle N \rangle, q \rangle) & \rightarrow \text{addEnt}(\langle N \rangle, q \rangle) \\
\text{deleteAtt}(\langle N, A \rangle, q) & \rightarrow \text{addAtt}(\langle N, A \rangle, q) \\
\end{align*}
\]

**Table 2.** Reversibility of ER transformations

are transformed, Figure 1 shows three source ER schemas $S_1$, $S_2$ and $S_3$, and their global schema $S_g$. In the figure, rectangular boxes, circles, diamonds and hexagons respectively denote entities, attributes, relationships and generalisation hierarchies; key attributes are underlined and nullable attributes are suffixed by #.
Fig. 1. Example ER schemas
The pathway from $S_1$ to $S_g$, denoted $TP_{S_1 \rightarrow S_g}$, is shown below.\(^3\) The last value in the scheme of attributes is one of key, null and notnull, which respectively represents primary key, nullable and non-nullable attributes.

$TP_{S_1 \rightarrow S_g}$:

1. $addEnt([\text{male}], \{X \mid (X, \text{'m'}) \in \langle \text{person,sex,notnull} \rangle \})$
2. $addEnt([\text{female}], \{X \mid (X, \text{'f'}) \in \langle \text{person,sex,notnull} \rangle \})$
3. $addGen([\langle \text{sex,person,male,female} \rangle])$
4. $deleteAtt([\langle \text{person,sex,notnull} \rangle], \{X, Y \mid X \in \langle \text{male} \rangle \land Y = 'm' \lor X \in \langle \text{female} \rangle \land Y = 'f' \})$
5. $extendAtt([\langle \text{dept,site,null} \rangle])$

Reversing each of the transformations and their order in $TP_{S_1 \rightarrow S_g}$ gives us the pathway from $S_g$ back to $S_1$.

$TP_{S_g \rightarrow S_1}$:

1. $contractAtt([\langle \text{dept,site,null} \rangle])$
2. $addAtt([\langle \text{person,sex,notnull} \rangle], \{X, Y \mid X \in \langle \text{male} \rangle \land Y = 'm' \lor X \in \langle \text{female} \rangle \land Y = 'f' \})$
3. $deleteGen([\langle \text{sex,person,male,female} \rangle])$
4. $deleteEnt([\langle \text{female} \rangle], \{X \mid (X, \text{'f'}) \in \langle \text{person,sex,notnull} \rangle \})$
5. $deleteEnt([\langle \text{male} \rangle], \{X \mid (X, \text{'m'}) \in \langle \text{person,sex,notnull} \rangle \})$
6. $addRel([\langle \text{works in, person, dept.1:1:1:N} \rangle], \{X, Y \mid X \in \langle \text{person, dname, notnull} \rangle \})$
7. $deleteAtt([\langle \text{person, dname, notnull} \rangle], \{X, Y \mid (X, Y) \in \langle \text{works in, person, dept.1:1:1:N} \rangle \})$

Paths $TP_{S_g \rightarrow S_d}$ and $TP_{S_d \rightarrow S_g}$ are shown below. Their reverse, i.e., $TP_{S_g \rightarrow S_d}$ and $TP_{S_d \rightarrow S_g}$ are derived in a similar fashion as for $TP_{S_1 \rightarrow S_g}$.

$TP_{S_g \rightarrow S_d}$:

1. $addAtt([\langle \text{dept} \rangle], \{X \mid (\_ X) \in \langle \text{person, dname, notnull} \rangle \})$
2. $addAtt([\langle \text{dept, dname, key} \rangle], \{X, X \mid (\_ X) \in \langle \text{person, dname, notnull} \rangle \})$
3. $extendAtt([\langle \text{dept, site, null} \rangle])$
4. $addRel([\langle \text{works in, person, dept.1:1:1:N} \rangle], \{X, Y \mid X \in \langle \text{person, dname, notnull} \rangle \})$
5. $deleteAtt([\langle \text{person, dname, notnull} \rangle], \{X, Y \mid (X, Y) \in \langle \text{works in, person, dept.1:1:1:N} \rangle \})$

$TP_{S_d \rightarrow S_g}$:

1. $renameAtt([\langle \text{employee} \rangle], \{\langle \text{person} \rangle \})$
2. $renameAtt([\langle \text{dept, location, null} \rangle], \{\langle \text{dept, site, null} \rangle \})$
3. $addAtt([\langle \text{male} \rangle], \{X \mid (X, \text{'m'}) \in \langle \text{person, sex, notnull} \rangle \})$
4. $addAtt([\langle \text{female} \rangle], \{X \mid (X, \text{'f'}) \in \langle \text{person, sex, notnull} \rangle \})$
5. $addGen([\langle \text{sex, person, male, female} \rangle])$
6. $deleteAtt([\langle \text{person, sex, notnull} \rangle], \{X, Y \mid X \in \langle \text{male} \rangle \land Y = 'm' \lor X \in \langle \text{female} \rangle \land Y = 'f' \})$

\(^3\) Note that for transformation $t_3$, because an ER generalization is translated down into a constraint in the HDM, and constraints do not have an extent, so a query is not required for the addGen transformation. More details can be found in [10, pg. 104].
3 Optimising Transformation Pathways

The transformations in Section 2 are specific to the ER model. In this section, the focus is on the general operation types of transformations. For example, an add transformation in this section refers to all the add-type transformations including addEnt, addRel, etc., for the ER model, addNode and addEdge, etc., for the HDM, and all other addX for other data models, where X is a construct of a particular data model.

A pathway may contain redundancy as the number and size of schemas grow in a network of schemas interconnected by pathways. The aim of developing transformation optimisation techniques [11] is to detect such redundancy, and rebuild the pathway with the redundant transformations removed, so as to make the evaluation of transformations, and hence the materialization of intentional schemas, more efficient.

We have developed a formal representation of transformation called the Transformation Manipulation Language (TML) that can be used for detecting any redundancy in pathways, as well as validating their well-formedness.

3.1 Semantics of Transformations and a Transformation Manipulation Language

The TML is designed to represent transformations in a form suitable for the analysis of the schema constructs that are created, deleted or are required to be present or absent for the transformation to be correct. In the definitions that follow, we require a function sc which, given a query or a schema construct, determines all the schema constructs that must exist for the query or schema construct to be valid.

The function sc(P), where P is a schema construct, is a recursive function that returns the union of P itself, plus sc(p1) ∪ sc(p2) ∪ ... ∪ sc(πn), where pi are the constructs in the scheme of P.

\[ sc(\langle p, p_1, p_2, \ldots, p_n \rangle) = \langle p, p_1, p_2, \ldots, p_n \rangle \cup sc(p_1) \cup sc(p_2) \cup \ldots \cup sc(\pi_n) \]

For example, sc(\langle w.p.d \rangle) = \{ \langle w.p.d \rangle, \langle p \rangle, \langle d \rangle \}. Table 3 shows the properties of the sc(P) function.

The TML notation formalises a transformation ti transforming a schema Si to a schema Si+1 as having four conditions a_i^+, b_i^−, c_i^+, and d_i^−:
\[
\text{sc}(P \cup \ldots \cup P_j) = \text{sc}(P_1) \cup \ldots \cup \text{sc}(P_j)
\]
\[
\text{sc}(\emptyset) = \emptyset
\]

Table 3. Properties of the \(\text{sc}(P)\) function

- The positive precondition \(a^+_i\) is the set of constructs that \(t_i\) implies must be present in \(S_i\). It comprises those constructs that are present in the query of the transformation (given by \(\text{sc}(q)\)) together with any constructs implied as being present by the construct \(c\):
  \[ t_i \in \{\text{add}(c, q), \text{extend}(c, q)\} \rightarrow a^+_i = (\text{sc}(c) - c) \cup \text{sc}(q) \]
  \[ t_i \in \{\text{delete}(c, q), \text{contract}(c, q), \text{rename}(c, c'), \text{id}(c, c')\} \rightarrow a^+_i = \text{sc}(c) \cup \text{sc}(q) \]

- The negative precondition \(b^-_i\) is the set of constructs that \(t_i\) implies must not be present in \(S_i\). It comprises those constructs which the transformation will add to the schema, and thus must not already be present:
  \[ t_i \in \{\text{add}(c, q), \text{extend}(c, q), \text{rename}(c', c), \text{id}(c', c)\} \rightarrow b^-_i = c \]
  \[ t_i \in \{\text{delete}(c, q), \text{contract}(c, q)\} \rightarrow b^-_i = \emptyset \]

- The positive postcondition \(c^+_i\) is the set of constructs that \(t_i\) implies must be present in \(S_{i+1}\), and is derived in the same way as \(a^+_i\) (i.e. the positive precondition of \(t_i\)):
  \[ t_i \in \{\text{add}(c, q), \text{extend}(c, q), \text{rename}(c', c), \text{id}(c', c)\} \rightarrow c^+_i = \text{sc}(c) \cup \text{sc}(q) \]
  \[ t_i \in \{\text{delete}(c, q), \text{contract}(c, q)\} \rightarrow c^+_i = (\text{sc}(c) - c) \cup \text{sc}(q) \]

- The negative postcondition \(d^-_i\) is the set of constructs that \(t_i\) implies must not be present in \(S_{i+1}\), and is derived in the same way as \(b^-_i\):
  \[ t_i \in \{\text{delete}(c, q), \text{contract}(c, q), \text{rename}(c, c'), \text{id}(c, c')\} \rightarrow d^-_i = c \]
  \[ t_i \in \{\text{add}(c, q), \text{extend}(c, q)\} \rightarrow d^-_i = \emptyset \]

Example 1 shows the add and extend transformations and their corresponding TML representation. To save space, the constructs in Figure 1 are abbreviated as shown in Table 4.

Example 1
\[
\text{TML}(t_r) = t_r : [\langle d \rangle \langle p \rangle \langle (p, dn) \rangle^+, \langle (d, dn) \rangle^- , \langle d \rangle \langle p \rangle \langle (p, dn) \rangle^+, \emptyset ]
\]
\[
\text{TML}(t_s) = t_s : [\langle d \rangle^+, \langle (d, s) \rangle^- , \langle d \rangle \langle (d, s) \rangle^+, \emptyset ]
\]

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle p \rangle_1)</td>
<td>(\langle \text{person} \rangle)</td>
</tr>
<tr>
<td>(\langle p, dn \rangle)</td>
<td>(\langle \text{person}, \text{dname}, \text{notnull} \rangle)</td>
</tr>
<tr>
<td>(\langle p, s \rangle)</td>
<td>(\langle \text{person}, \text{sex}, \text{notnull} \rangle)</td>
</tr>
<tr>
<td>(\langle m \rangle)</td>
<td>(\langle \text{male} \rangle)</td>
</tr>
<tr>
<td>(\langle f \rangle)</td>
<td>(\langle \text{female} \rangle)</td>
</tr>
<tr>
<td>(\langle e \rangle)</td>
<td>(\langle \text{employee} \rangle)</td>
</tr>
<tr>
<td>(\langle w, p, d \rangle)</td>
<td>(\langle \text{worksIn}, \text{person}, \text{dept.1:1}, \text{1:N} \rangle)</td>
</tr>
<tr>
<td>(\langle s, p, m, f \rangle)</td>
<td>(\langle \text{sex}, \text{person}, \text{male}, \text{female} \rangle)</td>
</tr>
<tr>
<td>(\langle d \rangle)</td>
<td>(\langle \text{dept} \rangle)</td>
</tr>
<tr>
<td>(\langle d, dn \rangle)</td>
<td>(\langle \text{dept}, \text{dname}, \text{key} \rangle)</td>
</tr>
<tr>
<td>(\langle d, s \rangle)</td>
<td>(\langle \text{dept}, \text{site}, \text{null} \rangle)</td>
</tr>
<tr>
<td>(\langle d, \rangle)</td>
<td>(\langle \text{dept}, \text{location}, \text{null} \rangle)</td>
</tr>
</tbody>
</table>

Table 4. Abbreviations used for the scheme of constructs in examples

Example 2 shows the delete and contract transformations and their corresponding TML representation and Example 3 shows the rename transformation and its corresponding TML representation.
Example 2

\[
TML(t_0) = \tau_0 : [\langle d \rangle \langle p \rangle \langle p, d_n \rangle, \emptyset, \langle p \rangle \langle p, d_n \rangle, \langle d \rangle] \\
TML(t_2) = \tau_2 : [\langle d \rangle \langle d, s \rangle, \emptyset, \langle d \rangle, \langle d, s \rangle] \\
\]

Example 3

\[
TML(t_{11}) = t_{11} : [\langle e \rangle, \langle p \rangle, \langle p \rangle, \langle e \rangle] \\
\]

3.2 Properties of the TML

There are three types of transformations, namely insertion-only, removal-only and insertion-removal transformations. add and extend are insertion-only transformations as they insert a single construct into a schema. The delete and contract transformations are removal-only as they remove a single construct from a schema. rename and id are insertion-removal transformations where they insert a construct into a schema and at the same time remove another construct from that schema. In the TML, a transformation \( t_i \) can be deduced as an insertion-only transformation if \( d_i^- = \emptyset \) because insertion-only transformations do not require in their postconditions the absence of any constructs. Similarly, \( t_i \) is a removal-only transformation if \( b_i^- = \emptyset \) because removal-only transformations do not require in their preconditions the absence of any constructs. An insertion-removal transformation will have the property \( (b_i^- \neq \emptyset \land d_i^- \neq \emptyset) \). The construct inserted by a transformation \( t_i \) can be found in \( b_i^- \) and the construct removed by \( t_i \) can be found in \( d_i^- \).

3.3 Rules for Optimisation

We can verify whether or not a pathway is well-formed by expressing the transformation steps in the TML. Provided that the pathway is well-formed, we can determine when the order of two transformations can be rearranged, when they can be simplified, and when they are redundant and hence can be removed from the pathway. In this section, \( TP \) refers to the pathway containing transformations \( t_m \) to \( t_n \), denoted \( TP_{m,n} \), as shown below.

\[
TP_{m,n} = [t_m : [a^+_m, b^-_m, c^+_m, d^-_m], t_{m+1} : [a^+_m, b^-_{m+1}, c^+_m, d^-_{m+1}], \ldots, t_n : [a^+_n, b^-_n, c^+_n, d^-_n]] \\
\]

The set of rules discussed include the well-formedness rules (for verifying whether or not \( TP \) is well-formed), the reordering rules (for checking whether or not two transformations can be reordered), and the optimisation rules (for detecting redundant and partially redundant transformations). A \( TP \) must be verified as well-formed before any optimisation rules can be applied and its well-formedness is maintained after the application of any optimisation rules.

**Well-Formed Transformation Pathways** A pathway \( TP \) from schema \( S_m \) to \( S_n \) is said to be well-formed if for each transformation \( t_i : S_i \rightarrow S_{i+1} \) within it:
- The only difference between the schema constructs in $S_{i+1}$ and $S_i$ is those constructs specifically changed by transformation $t_i$, implying that $S_{i+1} = (S_i \cup c_i^+) - d_i^-$ and $S_i = (S_{i+1} \cup a_i^+) - b_i^-$.
- The constructs required by $t_i$ are in the schemas, implying that $a_i^+ \subseteq S_i$, $\overline{b_i} \cap S_i = \emptyset$, $c_i^+ \subseteq S_{i+1}$ and $d_i^- \cap S_{i+1} = \emptyset$.

The rule for verifying the well-formedness of a pathway, $wf$, which captures the definition discussed above, is given below. The first $wf$ rule applies recursively to each transformation in the pathway. When there is no more transformation, the second $wf$ rule is used to verify that applying all the transformations in the pathway to $S_m$ results in a schema that is equal to $S_n$, both in terms of the content of the schema constructs in each schema and the extent of the schemas. Note that the $wf$ rule may be used in two different ways. Firstly, given a schema $S_m$ representing a data source and a pathway $TP$, we can derive the structure and the extent of the resultant schema $S_n$. Secondly, if both $S_m$ and $S_n$ are existing schemas representing two data sources, the $wf$ rule may be used to verify that $TP$ contains the transformations that correctly transforms $S_m$ into $S_n$.

$$
wf(S_m, S_n, [t_m, t_{m+1}, \ldots, t_{n-1}]) \iff a_m^+ \subseteq S_m \land b_m^- \cap S_m = \emptyset \land \neg \exists t_i \land \neg \exists t_j \land (t_i \not\subseteq S_{t_j})
$$

Reordering Transformations

Because the rules for detecting redundant and partially redundant transformations only apply to adjacent transformations, the order of transformations in a pathway may need to be altered during the detection of any possible redundancy, so that a transformation may be moved and paired up with any other transformations in the pathway. Moving a transformation $t_i$ to pair up with $t_j$ in $TP$ involves recursively reordering $t_i$ with the next transformation in $TP$ until the target index is reached. For example, moving $t_i$ in $TP$ so that it precedes $t_j$ involves reordering $t_i$ with $t_{i+1}$, if successful, then $t_i$ with $t_{i+2}$, etc., until the new index of $t_i$ in $TP$ is one less than the index of $t_j$.

To rearrange the order of two adjacent transformations $t_i$ and $t_{i+1}$ in a well-formed $TP = [t_m, \ldots, t_i, t_{i+1}, \ldots, t_n]$, we must first ensure that (i) $t_{i+1}$ does not contain in its preconditions a constraint that is satisfied by the postconditions of $t_i$. That is, if $t_{i+1}$ requires construct $P$ to exist, i.e., $P \in a_{i+1}^+$, then $P$ must not have been inserted by $t_i$, i.e., $P \not\in b_i^-$. If $t_{i+1}$ requires construct $P$ not to exist, i.e., $P \in b_{i+1}^-$, then $P$ must not have been removed by $t_i$, i.e., $P \not\in d_i^-$. Assuming the reordering has taken place, $TP$ would now look like $TP' = [t_m, \ldots, t_{i-1}, t_{i+1}, t_i, t_{i+2}, \ldots, t_n]$. For $TP'$ to be well-formed, the conditions that (ii) the postconditions of $t_{i+1}$ do not conflict with the preconditions of $t_i$ must hold. That is, if $P \in c_{i+1}^+$, then $P \not\in b_i^-$ must hold, and if $P \in d_{i+1}^-$, it must be true that $P \not\in a_i^+$. Also, (iii) the postconditions of $t_{i-1}$ must not conflict with the preconditions of $t_{i+1}$, which is now positioned next to $t_{i-1}$. Similarly, (iv)
the postconditions of $t_i$ must not conflict with the preconditions of $t_{i+2}$. All the reordering rules are listed below, in the order they were described.

(i) $b_i^- \cap a_{i+1}^+ = \emptyset$

(ii) $c_{i+1}^- \cap b_i^+ = \emptyset$

(iii) $c_{i-1}^- \cap b_{i+1}^- = \emptyset$

(iv) $c_i^+ \cap b_{i+2}^- = \emptyset$

if $i > m$

if $i < n - 1$

Example 4 Determining whether or not the order of transformations $t_8$ and $t_9$ in $TP_{S_2 \rightarrow S_2}$ can be swapped:

$TML(t_8, t_9) = t_8 : [\langle d \rangle^+, \langle d.s \rangle^-], \langle d \rangle \langle d.s \rangle^+, \emptyset],$

t_9 : [\langle d \rangle \langle d \rangle \langle p.d.n \rangle^-], \langle d \rangle \langle d \rangle \langle p.d, d \rangle^-], \langle d \rangle \langle d \rangle \langle p.d.n \rangle \langle w.p.d \rangle^+, \emptyset]$

Because all the rules for order rearrangement evaluate to $\emptyset$, we can conclude that the order of $t_8$ and $t_9$ can be reversed without affecting the overall result of all the transformations in the pathway. The reader is referred to [16] for details of the evaluation of these rules.

Detecting Redundant Transformations Two transformations $t_i$ and $t_{i+1}$, that are adjacent to each other in a well-formed $TP$, are redundant if $t_i$ is the reverse of $t_{i+1}$, i.e., $t_i = \overline{t_{i+1}}$ and vice versa, and the constructs being transformed by $t_i$ and $t_{i+1}$ have the same extent. In this case, the state of the resultant schema after applying all the transformations in $TP$ is the same whether or not both $t_i$ and $t_{i+1}$ are applied. In the TML terms, two transformations $t_i$ and $t_{i+1}$ are redundant if the following holds:

$$(a_i^+ = c_{i+1}^-) \land (b_i^- = a_{i+1}^+) \land (c_i^+ = a_{i+1}^-) \land (d_i^- = b_{i+1}^-) \land Ext(c_i^+ \oplus a_i^+) = Ext(c_{i+1}^- \oplus a_{i+1}^-)$$

where $(x \oplus y) = (x - y) \cup (y - x)$, which serves to determine all the constructs added or deleted by the pair of transformations. This rule qualifies two transformations as redundant if they add/extend and then delete/contract (in either order) the same construct, providing their associated queries result in the same extent. In fact, the check on the extent is unnecessary if the transformations are a pair of add/delete in either order because add and delete imply the insertion and removal of all the data instances associated with the construct of the transformation. As for cases where an extend or contract is one of the transformations in the pair, a check on the extent of the construct must be carried out to ensure the transformations are indeed dealing with the same construct.

Example 5 Determining whether or not $t_2$ and $t_{14}$ are redundant (assuming verification has already been done that $t_2$ and $t_{14}$ can be reordered so that they are adjacent to each other):

$TML(t_2, t_{14}) = t_2 : [\langle d \rangle \langle d \rangle \langle d \rangle \langle p.d.n \rangle^+, \langle f \rangle^-], \langle p \rangle \langle p.s \rangle \langle f \rangle^+, \emptyset],$

t_{14} : [\langle f \rangle \langle d \rangle \langle p.d.n \rangle^-], \emptyset, \langle p \rangle \langle p.d.n \rangle^-, \langle f \rangle^-]$
Because all the conditions for redundant transformations are satisfied, we can conclude that \( t_2 \) and \( \overline{t_{12}} \) are redundant.

**Detecting Partially Redundant Transformations** Two adjacent transformations, \( t_i \) and \( t_{i+1} \), are **partially redundant** if they satisfy the condition that (i) either the positive precondition of \( t_i \) is the same as the positive postcondition of \( t_{i+1} \), or the negative precondition of \( t_i \) is the same as the negative postcondition of \( t_{i+1} \). If either of these conditions is met, it is obvious that there is a certain level of overlap or redundancy in the effects of \( t_i \) and \( t_{i+1} \). Partially redundant transformations must also satisfy the condition that (ii) what \( t_i \) removes is not what \( t_{i+1} \) requires to be absent in its preconditions. This is because the construct \( c \) inserted by \( t_{i+1} \) may not have the same semantics as the construct \( c \) removed by \( t_i \). Therefore, we cannot treat them as the same construct. On the other hand, if \( t_i \) inserts a construct \( c \) which is required to be present in the positive precondition of \( t_{i+1} \), because of the adjacency of \( t_i \) and \( t_{i+1} \), \( c \) refers to the same construct and hence the operation on \( c \) in \( t_{i+1} \) may be simplified with that in \( t_i \). However, if \( t_{i+1} \) is a remove-only type transformation and removes \( c \), we cannot optimise \( t_i \) and \( t_{i+1} \) because they are not redundant transformations (refuted by rule (i)). Thus, (iii) partially redundant transformations are also required not to be a pair of insert-only transformation followed by a remove-only transformation. These three rules for partially redundant transformations are shown below.

\[
\begin{align*}
(i) \quad a_i^+ &= c_i^{+1} \oplus b_i^- = d_{i+1}^-,
\quad \text{where} \oplus \text{is the exclusive-or operator} \\
(ii) \quad d_i^- \cap b_{i+1}^- &= \emptyset \\
(iii) \quad \neg(d_i^- = \emptyset \land b_{i+1}^- = \emptyset)
\end{align*}
\]

The simplified transformation of two partially redundant transformations \( t_i \) and \( t_{i+1} \) can be derived by evaluating the transformation that represents the combined effect of \( t_i \) and \( t_{i+1} \). Example 6 shows the optimisation of a pair of partially redundant transformations \( t_5 \) and \( \overline{t_{12}} \).

**Example 6** Optimising partially redundant transformations \( t_5 \) and \( \overline{t_{12}} \) (assuming verification has already been done that \( t_5 \) and \( \overline{t_{12}} \) can be reordered so that they are adjacent to each other):

\[
TML(t_5, \overline{t_{12}}) = t_5 : [(\langle d \rangle)^+, \langle (d,s) \rangle^-, \langle (d,d) \rangle^+, \emptyset], \\
\overline{t_{12}} : [(\langle d \rangle \langle (d,s) \rangle^-, \langle (d,d) \rangle^{-}, \langle (d,d) \rangle^+, \langle (d,s) \rangle^-)]
\]

\( t_5 \) and \( \overline{t_{12}} \) can be optimised because they satisfy the three rules for partially redundant transformations. Evaluating the effects of \( t_5 \) and \( \overline{t_{12}} \) results in \( t_{sim} : [(\langle d \rangle)^+, \langle (d,d) \rangle^-, \langle (d,d) \rangle^+, \emptyset] \), which represents the primitive transformation \text{extendAtt}(\langle (d,d) \rangle) \). The reader is referred to [16] for full details of the evaluation.

Table 5 shows all possible transformation pairs, \( t_x \) followed by \( t_y \), that can be optimised using the techniques discussed in this section. By replacing add
with extend and delete with contract, this table also applies to the extend and contract transformations.

<table>
<thead>
<tr>
<th>add(c, q)</th>
<th>delete(c, q)</th>
<th>rename(c, c')</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_x</td>
<td>NWF</td>
<td>add(c', q)</td>
</tr>
<tr>
<td>delete(c, q)</td>
<td>NWF</td>
<td>NWF</td>
</tr>
<tr>
<td>rename(c', c)</td>
<td>NWF</td>
<td>delete(c', q)</td>
</tr>
<tr>
<td>rename(c', c)</td>
<td>NWF</td>
<td>delete(c'', q)</td>
</tr>
</tbody>
</table>

NWF = Not well-formed, [] = removal of transformations

**Table 5. Summary of optimisable transformations**

**Representing Composite Transformations** The results shown in Table 5 are derived by examining the effect of a transformation pair. The effect of a transformation is the construct added/deleted by the transformation. The effect of a composite transformation consisting of two transformations can be found by evaluating the aggregate insertion, aggregate removal, net insertion, and net removal of the pair of transformations. The aggregate insertion made by transformations \( t_m, t_n \) is the union of all the constructs inserted by \( t_m, t_n \), i.e., \( b_m \cup b_n \). The aggregate removal made by transformations \( t_m, t_n \) is the union of all the constructs removed by \( t_m, t_n \), i.e., \( d_m \cup d_n \). The net insertion made by \( t_m, t_n \) is their aggregate insertion minus their aggregate removal, and their net removal is their aggregate removal minus their aggregate insertion.

The resulting simplified transformation \( t_{sim} \) which shows the net effect of \( t_m, t_n \) will have as its positive precondition what \( t_m, t_n \) require to be present before any transformation is executed. However, some positive preconditions of \( t_m \) may be removed by \( t_n \), therefore, their existence and the existence of the constructs they imply (given by \( sc(aggregate\ removal) \)) is not required by \( t_{sim} \). However, the constructs in the net removal of \( t_m, t_n \) must be present before \( t_{sim} \) can be applied. Also, constructs whose existence is implied by the constructs belonging to the net insertion set must also be present in the positive precondition of \( t_{sim} \). Since what is contained in \( b_m \) is the construct to be inserted by \( t_i \), \( t_{sim} \) will have as its negative precondition the net insertion of \( t_m, t_n \). After the execution of \( t_{sim} \), what remains present in the resulting schema would be all the constructs that exist before \( t_{sim} \) is applied, plus the net insertion of \( t_m, t_n \), minus the net removal of \( t_m, t_n \). Finally, the negative postcondition of \( t_{sim} \) will contain the net removal of \( t_m, t_n \). The evaluation of \( t_{sim} \) is summarized in Table 6.

**3.4 An Optimisation Example**

This section shows how optimisation techniques discussed in this paper can be applied to cut down on the number of transformations in a pathway. Example 7 illustrates the optimisation of \( TP_{S_1 \rightarrow S_3} \).
<table>
<thead>
<tr>
<th>Transformation Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate insertion of $t_m, t_n$</td>
<td>$b_n^- \cup b_m^-$</td>
</tr>
<tr>
<td>Aggregate removal of $t_m, t_n$</td>
<td>$d_n^- \cup d_m^-$</td>
</tr>
<tr>
<td>Net insertion of $t_m, t_n$</td>
<td>aggregate insertion - aggregate removal</td>
</tr>
<tr>
<td>Net removal of $t_m, t_n$</td>
<td>aggregate removal - aggregate insertion</td>
</tr>
</tbody>
</table>

Simplified transformation $t_{sim}$ representing the composite transformation $t_m, t_n$:

\[
\begin{align*}
  a_{sim}^+ &= (a_m^+ \cup a_n^+) - sc(aggregate\removal) \\
  &\quad \cup sc(\text{net}\removal) \\
  &\quad \cup (sc(\text{net}\insertion)-\text{net}\insertion) \\
  b_{sim}^- &= \text{net}\insertion \\
  c_{sim}^+ &= a_{sim}^+ \cup \text{net}\insertion - \text{net}\removal \\
  d_{sim}^- &= \text{net}\removal
\end{align*}
\]

Table 6. Representing composite transformation $t_{sim}$

**Example 7** Optimising $TP_{S_1\rightarrow S_2}$:

\[
\begin{align*}
  TP_{S_1\rightarrow S_2}: \\
  t_1 &\text{ addEnt}([m], \{X \mid (X', m') \in \langle p.s \rangle\}) \\
  t_2 &\text{ addEnt}([f], \{X \mid (X', f') \in \langle p.s \rangle\}) \\
  t_3 &\text{ addGen}([\langle p, m, f \rangle]) \\
  t_4 &\text{ deleteAtt}([\langle p.s \rangle], \{X, Y \mid X \in \langle m \rangle \wedge Y \equiv m' \lor X \in \langle f \rangle \wedge Y \equiv f'\}) \\
  t_5 &\text{ extendAtt}([\langle d.s \rangle]) \\
  t_{10} &\text{ addAtt}([\langle p.s \rangle], \{X, Y \mid X \in \langle m \rangle \wedge Y \equiv m' \lor X \in \langle f \rangle \wedge Y \equiv f'\}) \\
  t_{15} &\text{ deleteGen}([\langle p, m, f \rangle]) \\
  t_{14} &\text{ deleteEnt}([f], \{X \mid (X', f') \in \langle p.s \rangle\}) \\
  t_{13} &\text{ deleteEnt}([m], \{X \mid (X', m') \in \langle p.s \rangle\}) \\
  t_{12} &\text{ renameAtt}([\langle d.s \rangle], [\langle d.l \rangle]) \\
  t_{11} &\text{ renameEnt}([\langle p \rangle], [\langle e \rangle])
\end{align*}
\]

The above pathway is formed by joining $TP_{S_1\rightarrow S_2}$ and $TP_{S_2\rightarrow S_3}$. First $t_4$ and $t_5$ are reordered. Since $t_4$ and $t_{10}$ are redundant, they are removed from the pathway. We apply the same optimisation to transformation pairs $t_3$ and $t_{15}$, $t_2$ and $t_{14}$, and $t_1$ and $t_{13}$. By now, the number of transformations in the pathway has dramatically decreased as shown in $TP'_{S_1\rightarrow S_2}$ in Table 7. We further optimise $t_5$ and $t_{12}$ to form $t_{17}$ as $\text{extendAtt}([\langle \text{dept}, \text{location}, \text{null} \rangle])$ as shown in Example 6. The final optimised pathway $TP''_{S_1\rightarrow S_3}$ is shown in Table 7.

<table>
<thead>
<tr>
<th>$TP'_{S_1\rightarrow S_2}$</th>
<th>$TP''_{S_1\rightarrow S_3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_5$ \text{extendAtt}([\langle d.s \rangle])</td>
<td>$t_{17}$ \text{extendAtt}([\langle d.l \rangle])</td>
</tr>
<tr>
<td>$t_{12}$ \text{renameAtt}([\langle d.s \rangle], [\langle d.l \rangle])</td>
<td>$t_{11}$ \text{renameEnt}([\langle p \rangle], [\langle e \rangle])</td>
</tr>
</tbody>
</table>

Table 7. Optimising $TP_{S_1\rightarrow S_3}$
4 Conclusion

We have discussed in this paper the AutoMed integration system which adopts the BAV approach and techniques for optimising transformations in this system. We have looked at how transformations can be expressed in the TML, and shown how TML rules can be applied for pathway optimisation. A transformation pathway optimisation tool using the TML has been implemented in the AutoMed project. This tool, which is currently fully functional, is being optimised for more speedy performance. An evaluation of performance gain by using the TML techniques is also scheduled to be carried out.

The use of the TML can also be extended to automatically detect any possible needs for repairing the global schema [11] in the face of evolving source schemas. An initial idea of how this could be achieved is to periodically scan all the pathways connected to the global schema. If a removal of a particular construct is found in each and every of the pathways, which means this construct has now become obsolete, then this construct should be removed from the global schema to give a more updated reflection of the changes in its connected sources. The techniques on using the TML to resolve some of the issues raised by schema evolution will be investigated in the near future.

While the study of using techniques on database schema optimisation as a way to increase the efficiency in schema integration and query processing receives considerable attention [7, 18], the study of optimisation focused solely on transformations is a rather new topic. It is our intention to develop the TML as a general transformation manipulation language that can be used by other schema transformation formalisms. Generally speaking, the TML is applicable with other schema transformation languages, so long as these languages clearly indicate the pre- and postconditions of the transformations and the associations between new and existing constructs. The possibility of using the TML with other transformation languages described in [4, 7, 17] will be investigated.

References