An enhanced transformation language for the HDM Automed Working Document 4, Version 1

A.Poulovassilis

Tuesday 31$^{\underline{\text{st}}}$ July 2001

1 Introduction

Integration of heterogeneous databases requires that semantic differences between schemas are resolved by a process of schema transformation. In previous work, we have developed a general framework to support the schema transformation process, consisting of a hypergraph-based common data model and a set of primitive schema transformations defined for this model [3]. Higher-level common data models and primitive schema transformations for them can be defined in terms of this lower-level model [2].

A key feature of our framework is that both primitive and composite schema transformations are automatically *reversible*, allowing automatic translation of data and queries between schemas [1]. Here we show how that this is still true for an extended transformation language, equipped with iteration and conditional constructs.

The extended language has the advantage that *parametrised* transformations can be written which encapsulate schema equivalences in a generic way. This is contrast to the original transformation language which could only express schema-specific equivalences.

Note: This extended transformation language appears to be very similar to the "template" approach proposed by Nikos Rizopoulos. The difference is that the enhanced transformation language being proposed here need not be compiled into a list of equivalent primitive transformations for each specific application of a transformation, but can be manipulated "as is", for example by the query processor.

2 The Original Schema Transformation Framework

We assume the availability of two disjoint sets, $Value$ (values) and $Name$ (the names of nodes and edges). The set $Scheme$ is defined recursively as follows:

- $Name \subseteq Scheme$
- $\langle n_0, n_1, \ldots, n_m \rangle \in Scheme$ if $m \geq 1$, $n_0 \in Name$, and $n_i \in Scheme$ for all $1 \leq i \leq m$.

The distinguished constant $Null \in Name$. For any set T, $Seq(T)$ denotes the set of finite sequences of members of T.

A **schema**, S, is a triple $\langle Nodes, Edges, Constraints \rangle$ where:

- $Nodes \subseteq Name$

- $Edges \subseteq Name \times Seq(Scheme)$ such that for any $\langle n_0, n_1, \ldots, n_m \rangle \in Edges$, $n_i \in Nodes \cup Edges$ for all $1 \leq i \leq m$.

- $Constraints$ is a set of boolean-valued expressions whose variables are members of $Nodes \cup Edges$.

Thus, $Nodes$ and $Edges$ define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them.

A **query** q **over a schema** $S = (Nodes, Edges, Constraints)$ is an expression whose variables are members of $Nodes \cup Edges$[1].

Given a schema $S = \langle Nodes, Edges, Constraints \rangle$, an **instance of** S is a set $I \subseteq P(Seq(Value))$ such that there exists a function

$$Ext_{S,I} : Nodes \cup Edges \rightarrow P(Seq(Value))$$

where:

(i) each set in $Range(Ext_{S,I})$ is derivable by means of an expression over the sets of I;[2]

(ii) conversely, each set in I is derivable by means of an expression over the sets of $Range(Ext_{S,I})$;

(iii) for each edge $\langle n_0, c_1, \ldots, c_m \rangle \in Edges$, each sequence $s \in Ext_{S,I}(\langle n_0, c_1, \ldots, c_m \rangle)$ contains m subsequences $s_1, \ldots, s_m$ where $s_i \in Ext_{S,I}(c_i)$ for all $1 \leq i \leq m$ (domain integrity);

(iv) for every $c \in Constraints$, the expression $c[c_1/Ext_{S,I}(c_1), \ldots, c_n/Ext_{S,I}(c_n)]$ evaluates to true, where $c_1, \ldots, c_n$ are the elements of $Nodes \cup Edges$ appearing in c.

We call such a function $Ext_{S,I}$ an **extension mapping** from S to I.

A **model** is a triple $\langle S, I, Ext_{S,I} \rangle$ where S is a schema, I is an instance of S and $Ext_{S,I}$ is an extension mapping from S to I. We denote by $Models$ the set of models.

Two schemas are **equivalent** if they have the same set of instances. Given a condition f, a schema S **conditionally subsumes** a schema S' w.r.t. f if any instance of S' satisfying f is also an instance of S. Two schemas S and S' are **conditionally equivalent** w.r.t f if they each conditionally subsume each other w.r.t. f.

The primitive transformations for the HDM are listed below. Each transformation is a function that when applied to a model returns a new model i.e. it is of type $ArgType \rightarrow Models \rightarrow Models$ for some $ArgType$.

Each transformation has a proviso associated with it which states when the transformation is **successful**. Unsuccessful transformations return an "undefined" model, denoted by ϕ. Any transformation applied to ϕ returns ϕ.

[1]The language in which constraints and queries are defined is not fixed by our framework, and we are currently experimenting with comprehension languages and graphical query languages within the Automed architecture.

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

1. $renameNode$ $(fromName, toName)$ renames a node.
   Proviso: $toName$ is not already the name of some node.

2. $renameEdge$ $(\langle fromName, c_1, \ldots, c_m \rangle, toName)$ renames an edge.
   Proviso: $toName$ is not already the name of some edge.

3. $addConstraint$ c adds a new constraint c.
   Proviso: c evaluates to true.

4. $delConstraint$ c deletes a constraint.
   Proviso: c exists.

5. $addNode$ $(name, q)$ adds a node named $name$ whose extent is given by the value of the query q.
   The extent of the node is given by the value of the query q.
   Proviso: a node of that name does not already exist.

6. $delNode$ $(name, q)$ 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 (thus, not violating property (ii) of an instance).
   Proviso: the node exists and participates in no edges.

7. $addEdge$ $(\langle name, c_1, \ldots, c_m \rangle, q)$ 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. $delEdge$ $(\langle name, c_1, \ldots, c_m \rangle, q)$ deletes an edge.
   q is a query that states how the extent of the deleted edge could be recovered from the extents of the remaining schema constructs.
   Proviso: the edge exists and participates in no edges.

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

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

Example 1 Transforming between two schemas
Consider two schemas $S_1 = (N_1, E_1, C_1)$ and $S_2 = (N_2, E_2, C_2)$ where
$N_1 = \{person, mathematician, compScientist\}$, $E_1 = \{\}$,
$C_1 = \{mathematician \subseteq person, compScientist \subseteq person\}$,

$N_2 = \{person, dept\}$, $E_2 = \{\langle Null, person, dept\rangle\}$, $C_2 = \{\}$.
The transformation T_{S_1,S_2} below transforms S_1 to S_2 and is s-d:

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

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

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

Thus S_1 and S_2 are c-equivalent. □

2.1 Reversibility of schema transformations

For every primitive transformation t such that $t((S, I, Ext_{S,I})) \neq \phi$ there exists a **reverse primitive transformation** t^{-1} such that $t^{-1}(t((S, I, Ext_{S,I}))) = (S, I, Ext_{S,I})$. We list the reverse transformation of each primitive transformation below. Notice that if t depends on a condition c, since t is successful c must necessarily hold and so need not be verified within t^{-1}:

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

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

2.2 Transforming between non-equivalent schemas

Even when two databases are designed to hold the same information, it is likely that they will differ slightly and will not be precisely equivalent. We define four more low-level transformations to deal with such situations. These are defined in terms of the existing primitive transformations as follows:

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

2.3 Translating Data and Queries

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

- $addNode(n, q)$: the new node n is populated by evaluating q on the extension of S_1.
- $addEdge(e, q)$: the new edge e is similarly populated by evaluating q on the extension of S_1.

For composite transformations, a new extent is generated each time an *addNode* or *addEdge* transformation is applied (see [1] for an example).

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

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

For composite transformations, the above substitutions are successively applied in order to obtain the final migrated query q_2 (see [1] for some examples).

3 Extending the language with iteration and conditional constructs

The next two examples generalise the transformations in Example 1 to work over arbitrary schemas. Note that the function $name : Name \rightarrow Value$ converts a $Name$ to a string $Value$.

Example 2 Transforming between two schemas (1)
Consider two schemas $S_1 = (N_1, E_1, C_1)$ and $S_2 = (N_2, E_2, C_2)$ where $N_1 = \{e, e_1, \ldots, e_n\}$, $E_1 = \{\}$, $C_1 = \{\forall e_i \in \{e_1, \ldots, e_n\}.e_i \subseteq e\}$, $N_2 = \{e, a\}$, $E_2 = \{\langle n, e, a\rangle\}$, $C_2 = \{\}$.
The transformation T_{S_1,S_2} below transforms S_1 to S_2 (and is s-d):

$$
\begin{array}{ll}
addNode & (a, \{name(e_1), \ldots, name(e_n)\}); \\
addEdge & (n, e, a, \\
 & \{\langle x, name(e_1)\rangle \mid x \in e_1\} \cup \ldots \cup \{\langle x, name(e_n)\rangle \mid x \in e_n\}); \\
delConstraint & \forall e_i \in \{e_1, \ldots, e_n\}.e_i \subseteq e; \\
delNode & (e_1, \{x \mid \langle x, name(e_1)\rangle \in \langle n, e, a\rangle\}); \ldots \\
delNode & (e_n, \{x \mid \langle x, name(e_n)\rangle \in \langle n, e, a\rangle\});
\end{array}
$$

The reverse transformation automatically derived from this is:

$$
\begin{array}{ll}
addNode & (e_n, \{x \mid \langle x, name(e_n)\rangle \in \langle n, e, a\rangle\}); \\
addNode & (e_1, \{x \mid \langle x, name(e_1)\rangle \in \langle n, e, a\rangle\}); \ldots \\
addConstraint & \forall e_i \in \{e_1, \ldots, e_n\}.e_i \subseteq e; \\
delEdge & (n, e, a, \\
 & \{\langle x, name(e_1)\rangle \mid x \in e_1\} \cup \ldots \cup \{\langle x, name(e_n)\rangle \mid x \in e_n\}); \\
delNode & (a, \{name(e_1), \ldots name(e_n)\});
\end{array}
$$

In the extended transformation language, the parametrised transformation $T1$, say, from S_1 to S_2 which works for any number of nodes e_i is defined as follows:

function $T1(n, a, e : Name, es : set(Name))$ {
  $addNode$ $(a, \{name(i) \mid i \in es\});$
  $addEdge$ $(n, e, a, \{\langle x, name(i)\rangle \mid i \in es; x \in i\});$
  $delConstraint$ $\forall i \in es.i \subseteq e;$
  for i **in** es {
    $delNode$ $(i, \{x \mid \langle x, name(i)\rangle \in \langle n, e, a\rangle\});$
  };
}

The reverse transformation $T1^{-1}$ automatically derived from $T1$ is:

function $T1^{-1}(n, a, e : Name, es : set(Name))$ {
  for i **in** es {
    $addNode$ $(i, \{x \mid \langle x, name(i)\rangle \in \langle n, e, a\rangle\});$
  };
  $addConstraint$ $\forall i \in es.i \subseteq e;$
  $delEdge$ $(n, e, a, \{\langle x, name(i)\rangle \mid i \in es; x \in i\});$
  $delNode$ $(a, \{name(i) \mid i \in es\});$
}

□

Example 3 Transforming between two schemas (2)
The transformation T_{S_2,S_1} below transforms S_2 to S_1 and is i-d since there is a condition $a = \{name(e_1), \ldots, name(e_n)\}$ on the last-but-one step:

$addNode$ $(e_1, \{x \mid \langle x, name(e_n)\rangle \in \langle n,e,a\rangle\});$
$addNode$ $(e_n, \{x \mid \langle x, name(e_1)\rangle \in \langle n,e,a\rangle\}); \ldots$
$addConstraint$ $\forall e_i \in \{e_1, \ldots, e_n\}.e_i \subseteq e;$
$delEdge$ $(n, e, a,$
$\{\langle x, name(e_1)\rangle \mid x \in e_1\} \cup \ldots \cup$
$\{\langle x, name(e_n)\rangle \mid x \in e_n\}) \quad (a = \{name(e_1), \ldots, name(e_n)\});$
$delNode$ $(a, \{name(e_1), \ldots name(e_n)\});$

The reverse transformation automatically derived is T_{S_1,S_2} above.

In the extended transformation language, the parametrised transformation $T2$, say, from S_2 to S_1 which works for any number of nodes e_i is:

function $T2(n, a, e : Name, es : set(Name))$ {
  for i **in** es {
    $addNode$ $(i, \{x \mid \langle x, name(i)\rangle \in \langle n,e,a\rangle\});$
  };
  $addConstraint$ $\forall i \in es.i \subseteq e;$
  $delEdge$ $(n, e, a, \{\langle x, name(i)\rangle \mid i \in es; x \in i\}) \quad (a = \{name(i) \mid i \in es\});$
  $delNode$ $(a, \{name(i) \mid i \in es\});$
}

The reverse transformation $T2^{-1}$ automatically derived from this is the same as $T1$ above (the constraint in the last-but-one step need not to be preserved). □

Example 4 Transforming between two schemas (3)

In the previous two examples, the transformations between S_1 and S_2 assume that the values of the attribute a in S_2 are identical to the names of the sub-types $e_1, \ldots, e_n$ in S_1. We can write slightly more general transformations that take also a 1-1 function $corresp : Name \rightarrow Value$ which associates each e_i with a value of a:

function $T1'(n, a, e : Name, es : set(Name), corresp : Name \rightarrow Value)$ {
  $addNode$ $(a, \{corresp(i) \mid i \in es\});$
  $addEdge$ $(n, e, a, \{\langle x, corresp(i)\rangle \mid i \in es; x \in i\});$
  $delConstraint$ $\forall i \in es.i \subseteq e;$
  for i **in** es {
    $delNode$ $(i, \{x \mid \langle x, corresp(i)\rangle \in \langle n,e,a\rangle\});$
  };
}

function $T2'(n, a, e : Name, es : set(Name), corresp : Name \rightarrow Value)$ {
  for i **in** es {
    $addNode$ $(i, \{x \mid \langle x, corresp(i)\rangle \in \langle n,e,a\rangle\});$
  };
  $addConstraint$ $\forall i \in es.i \subseteq e;$
  $delEdge$ $(n, e, a, \{\langle x, corresp(i)\rangle \mid i \in es; x \in i\}) \quad (a = \{corresp(i) \mid i \in es\});$
  $delNode$ $(a, \{corresp(i) \mid i \in es\});$
} □

4 The extended transformation language

The syntax of a transformation, $transf$, in the extended transformation language is as follows, where $primTransf$ is one of the primitive transformations listed in Section 2 above:

$$\begin{array}{lcl} transf & = & primTransf \mid \\ & & \text{“\{”}\ transf_1\ \text{“;”}\ \ldots\ \text{“;”}\ transf_n\ \text{“\}”} \mid \\ & & \text{“}for\text{”}\ var\ \text{“}in\text{”}\ query\ transf \mid \\ & & \text{“}if\text{”}\ query\ \text{“}then\text{”}\ transf \mid \\ & & \text{“}if\text{”}\ query\ \text{“}then\text{”}\ transf_1\ \text{“}else\text{”}\ transf_2 \mid \\ & & functionName\ \text{“(”}\ actualParameterList\ \text{“)”} \end{array}$$

The syntax of a transformation function definition is:

$$transfFunDef \;=\; \text{“}function\text{”}\ functionName\ \text{“(”}\ formalParameterList\ \text{“)”}\ transf$$

The syntax of a formal parameter list and an actual parameter list is straightforward.

In the for construct, the $query$ is evaluated once, with respect to the model input to the transformation.

There are a number of additional semantic restrictions on transformations:

1. well-typedness: the query in the for construct is set-valued and the query in the if constructs is boolean valued/
2. well-definedness: in order for the for construct to be well-defined, $transf$ should be independent of the order in which it is applied to the set of elements resulting from evaluating $query$;
3. the $query$ in the for construct is independent of the $transf$ — this guarantees reversibility;
4. similarly, the query in the if constructs is independent of the transformations in the $then$ and $else$ branches.

It may be possible to relax the last two of these conditions, if necessary. However:

A preliminary examination of the schema transformation examples given in our previous papers indicates that the language (with the above restrictions) is sufficient to encode all of them.

5 Query translation in the extended transformation language

The following translation function, $Trans$, takes a query on a schema, and a transformation on that schema, and returns the equivalent query on the transformed schema[3]:

[3]The $[\![\ldots]\!]$ notation indicates that we are operating in the semantic domain.

$$
\begin{array}{lcl}
Trans\,[\![\, q, addConstraint\ c\,]\!] & = & q \\
Trans\,[\![\, q, delConstraint\ c\,]\!] & = & q \\
Trans\,[\![\, q, addNode\ (n, q')\,]\!] & = & q \\
Trans\,[\![\, q, delNode\ (n, q')\,]\!] & = & q[n/q'] \\
Trans\,[\![\, q, addEdge\ (\langle n, s\rangle, q')\,]\!] & = & q \\
Trans\,[\![\, q, delEdge\ (\langle n, s\rangle, q')\,]\!] & = & q[\langle n, s\rangle/q'] \\
Trans\,[\![\, q, renameNode\ (n, n')\,]\!] & = & q[n/n'] \\
Trans\,[\![\, q, renameEdge\ (\langle n, s\rangle, n')\,]\!] & = & q[\langle n, s\rangle/\langle n', s\rangle] \\
Trans\,[\![\, q, \{\ t_1\ ;\ \ldots\ ;\ t_n\ \}\,]\!] & = & Trans[\![\, \ldots Trans[\![\, Trans[\![\, q, t_1\,]\!], t_2\,]\!] \ldots, t_n\,]\!] \\
Trans\,[\![\, q, for\ v\ in\ q'\ t\,]\!] & = & let\ \{s_1, \ldots, s_n\} = Eval[\![\, q'\,]\!]\ in \\
 & & Trans[\![\, q, \{t[v/s_1]\ \ \ldots\ ; t[v/s_n]\}\,]\!] \\
Trans\,[\![\, q, if\ q'\ then\ t\,]\!] & = & if\ q'\ then\ Trans[\![\, q, t\,]\!]\ else\ q \\
Trans\,[\![\, q, if\ q'\ then\ t_1\ else\ t_2\,]\!] & = & if\ q'\ then\ Trans[\![\, q, t_1\,]\!]\ else\ Trans[\![\, q, t_2\,]\!]
\end{array}
$$

6 Data translation in the extended transformation language

Handled similarly, except that in this case new extents are generated for each *addNode* and *addEdge* transformation, by evaluating the supplied query with respect to the current schema.

7 Reversibility of the extended transformation language

If t is a primitive transformation, then

$$Rev[\![\, t\,]\!] \ = \ t^{-1}$$

as defined in Section 2.

For the remaining cases, *Rev* is defined recursively as follows, where $Rev : transf \rightarrow transf$:

$$
\begin{array}{lcl}
Rev[\![\, \{\ t_1\ ;\ \ldots\ ;\ t_n\ \}\,]\!] & = & \{\ Rev[\![\, t_n\,]\!]\ ;\ \ldots\ ;\ Rev[\![\, t_1\ ]\!]\} \\
Rev[\![\, for\ v\ in\ q\ t\,]\!] & = & for\ v\ in\ q\ Rev[\![\, t\,]\!] \\
Rev[\![\, if\ q\ then\ t\,]\!] & = & if\ q\ then\ Rev[\![\, t\,]\!] \\
Rev[\![\, if\ q\ then\ t_1\ else\ t_2\,]\!] & = & if\ q\ then\ Rev[\![\, t_1\,]\!]\ else\ Rev[\![\, t_2\,]\!]
\end{array}
$$

References

[1] 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.

[2] 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.

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