A Generic Data Level Implementation of ModelGen

Andrew Smith and Peter McBrien

Dept. of Computing, Imperial College London,
Exhibition Road, London SW7 2AZ

Abstract. The model management operator ModelGen translates a schema expressed in one modelling language into an equivalent schema expressed in another modelling language, and in addition produces a mapping between those two schemas. This paper presents an implementation of ModelGen which in addition allows for the translation of data instances from the source to the target schema, and *vice versa*. The translation mechanism is distinctive from others in that it takes a generic approach that can be applied to any modelling language.

1 Introduction

ModelGen is a model management [1] operator that translates a schema from one **data modelling language** (**DML**) into an equivalent schema in another DML and also produces a mapping between the schemas. To date, no implementation of ModelGen completely meets these criteria [2].

In this paper we describe a generic implementation of ModelGen that creates data level translations between schemas by the composition of generic transformations, as well as a bidirectional mapping from the source to the target schema. A distinguishing feature of this work is that the choice of transformations does not rely on knowledge of the source DML. An implementation of ModelGen such as this is useful in a number of circumstances. For example, an e-business may wish to move data between its back end SQL database and its XML based web pages without having to re-engineer the mappings every time the database schema or web pages are changed.

There are two specific prerequisites to translating schemas between DMLs automatically. Firstly we need an accurate and generic UMM capable of describing the schemas and the constructs of both the source and the target DML, so the system can recognise when a given schema matches those constructs. In this paper we make use of the **hypergraph data model** (**HDM**) [3] to accurately describe constructs and schemas. The constructs of a number of DMLs, including XML, UML class diagrams, ER and SQL, have already been defined in terms of the HDM [3, 4]. Secondly we need an information preserving [5] way of *transforming* the resulting HDM schema such that the structure of its constructs match those of the target DML. We use the **Both-As-View (BAV)** data integration technique [6] to transform schemas.

Figure 1 gives an overview of our approach. In step 1 the source schema S_s is translated into an equivalent HDM schema, S_{hdm-s}. Next, a series of transformations are applied to S_{hdm-s} to transform it to S_{hdm-t} that is equivalent to a schema in the target DML. In step 3 the constructs in S_{hdm-t} are translated into their equivalents in the target DML to create S_t.

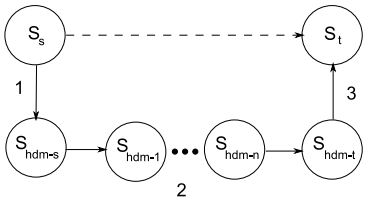

Fig. 1. Overview of the approach taken

Step one of this process depends on existing definitions of high level DML constructs in the HDM. The contribution we make in this paper is to show how steps 2 and 3 can be automated without having to know the DML used to create S_s. Firstly we present an algorithm for identifying schema objects within the HDM schema that match constructs in the target DML and secondly we present an automatic way of choosing the transformation rules at run time that transform a schema expressed in the HDM and its data into an equivalent schema that matches the constructs of the target DML.

The remainder of this paper is structured as follows: Section 2 gives a brief overview of the HDM and the BAV data integration technique and introduces an example schema. We also describe BAV composite transformations and introduce a new one. In Section 3 we present our algorithm for matching HDM schema objects with constructs in a target DML. Section 4 introduces the algorithm we use to select appropriate composite transformations for the translation. Section 5 gives an example translation. In Section 6 we present some experimental results and some analysis. Section 7 describes other proposals for ModelGen as well as some specific model to model translators. Finally Section 8 offers some conclusions.

2 HDM and BAV

The HDM uses a set of three simple constructs: nodes, edges and constraints, to model high level constructs in a given DML. HDM nodes and edges can have associated data values or extents. Each element in the XML instance document is assigned a unique object identifier (OID) shown next to the element. If the node representing the element is not a leaf node and does not have any key nodes associated with it then this OID becomes the extent of the node. For example, Figure 3 shows how HDM represents the XML Schema and accompanying XML instance document in Figure 2. The extent of HDM node $\langle\langle$dept$\rangle\rangle$ is {01,04,07}. If there is a key associated with the element then the extent of the element node is that of the key. For example the extent of $\langle\langle$person$\rangle\rangle$ is {1,2}. The extent of an edge is a tuple made up of values from the nodes or edges it joins. For example the extent of HDM edge $\langle\langle$_,person,name$\rangle\rangle$ is {$\langle$1,'John Smith'$\rangle$, $\langle$2,'Peter Green'$\rangle$}.

When defining the constructs of high level DMLs in the HDM each construct falls into one of four categories [3], the following three of which we use in this paper when describing the XML Schema and SQL data models:

– **Nodal** constructs can exist on their own and are represented by a node. The root node of an XML Schema and an SQL table are examples of a nodal constructs.

– **Link-Nodal** constructs are associated with a parent construct and are represented by a node and an edge linking the node to the parent. XML attributes and elements are link-nodal constructs, as are SQL columns.
– **Constraint** constructs have no extent but rather constrain the values that can occur in the constructs they are associated with. They are represented in HDM by one or more of the HDM constraint operators [3]. Those used in this paper are: inclusion ($\subseteq$), mandatory ($\triangleright$), unique ($\triangleleft$) and reflexive ($\xrightarrow{\text{id}}$). A SQL foreign key is an example of a constraint construct that is represented in the HDM by an inclusion constraint between two HDM nodes representing SQL columns.

The variants of a high level construct can be modelled using different combinations of constraints. For example, the fact that the XML attribute, id, in Figure 2 is a `required` attribute is modelled in HDM by adding a $\triangleright$ operator between $\langle\langle$person$\rangle\rangle$ and $\langle\langle$_,person,id$\rangle\rangle$. This means that every value in $\langle\langle$person$\rangle\rangle$ must also appear in the edge *i.e.* there can be no value of the parent element without an associated attribute value. An attribute that does not have the `required` flag set would not generate this extra constraint.

```
<xsd:complexType name = "person_type">
 <xsd:sequence>
  <xsd:element name = "name" type = "xsd:string" />
 </xsd:sequence>
 <xsd:attribute name = "id" type = "xsd:int" use = "required"/>
</xsd:complexType>
<xsd:element name = "staff">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name = "dept" maxOccurs = "unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name = "person" type = "person_type"
            minOccurs = "0" maxOccurs = "unbounded" />
     </xsd:sequence>
     <xsd:attribute name = "dname" type = "xsd:string"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:key name = "personkey">
  <xsd:selector xpath =  "./dept/person" />
  <xsd:field xpath =  "@id" />
 </xsd:key>
</xsd:element>

<staff> 00
<dept dname = 'Finance'> 01
    <person id = "1"> 02
        <name>John Smith</name> 03
    </person>
</dept>
<dept dname='HR'> 04
    <person id = "2"> 05
        <name>Peter Green</name> 06
    </person>
</dept>
<dept dname='IT'> 07
</dept>
</staff>
```

Fig. 2. S_{xml}

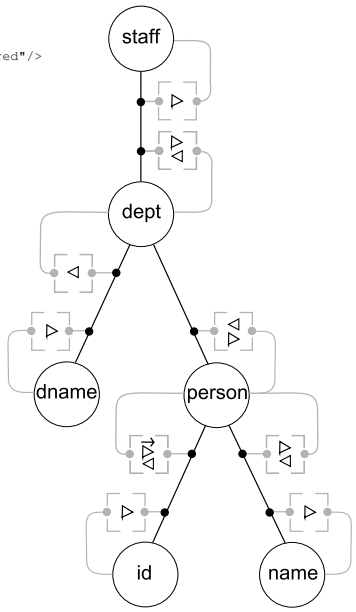

Fig. 3. $S_{hdm-xml}$

We use the BAV data integration technique [6] to transform our schemas. A BAV **transformation pathway** is made up of a sequence of transformations which either add, delete or rename a single schema object thereby generating a new schema. The extent of the new schema object or of the one removed is defined as a query on the extents

of the existing schema objects. In this way the information preserving transformation pathway made up of schemas and transformation operations is created that shows in detail how a source schema is transformed into a target schema. This transformation pathway forms a mapping between the schemas.

2.1 Composite Transformations

BAV transformations are fine grained and allow for accurate translations, but since each step only changes one schema object a large number of transformations are needed for most operations. To avoid the need to programme each transformation step separately, information preserving **composite transformations** (**CT**s) can be defined that are templates, describing common patterns of transformation steps.

Three such CTs are used when we translate between schemas in the XML Schema and SQL modelling languages. Two of those used, namely id_node_expand and inc-_expand, have been previously defined [3] and are shown graphically in Figures 5 and 6, respectively. The third CT, expand_mv, is defined by the pseudo code in Algorithm 1 and illustrated in Figure 4. It is useful when translating from a DML that supports multivalued attributes (such as XML Schema) into to a target DML that does not (such as SQL). In the figure, the $\bowtie$ symbol represents a join operation between $\langle\langle$_,T,B$\rangle\rangle$ and $\langle\langle$_,T,TA$\rangle\rangle$ *i.e.* the constraints linked to the join apply to both edges. The *contains* predicate [3] in the algorithm holds when its first argument appears as a construct in the formula that is in the second argument. We will see in Section 5 how these three CTs can be used to translate the XML schema in Figure 2 to SQL.

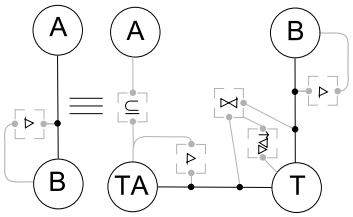

Fig. 4. expand_mv

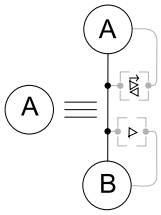

Fig. 5. id_node_expand

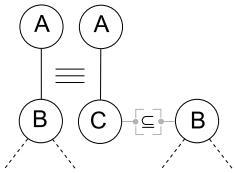

Fig. 6. inc_expand

3 AutoMatch

The detailed constraint language used in the HDM allows us to accurately identify groups of HDM constructs that correspond to an equivalent high level DML construct. Table 3 shows the constraints associated with the various constructs in SQL. $\langle\langle$T$\rangle\rangle$ is an HDM node representing an SQL table, $\langle\langle$C$\rangle\rangle$ represents a column in that table and $\langle\langle$_,T,C$\rangle\rangle$ links the two nodes. In the final line of the table $\langle\langle$C$\rangle\rangle$ represents a foreign key column and $\langle\langle$T$\rangle\rangle$ the table the foreign key links to.

Algorithm 1: expand_mv(⟨⟨B⟩⟩,⟨⟨_,A,B⟩⟩)

if ⟨⟨*A*⟩⟩ ◁ ⟨⟨_,*A*,*B*⟩⟩ **then**
  Exception
addNode(⟨⟨T⟩⟩,⟨⟨_,A,B⟩⟩)
addNode(⟨⟨TA⟩⟩,[{x} | {x, y} ← ⟨⟨_, A, B⟩⟩])
addEdge(⟨⟨_,T,TA⟩⟩,[{{x, y}, x} | {x, y} ← ⟨⟨_, A, B⟩⟩])
addEdge(⟨⟨_,T,B⟩⟩,[{{x, y}, y} | {x, y} ← ⟨⟨_, A, B⟩⟩])
addCons(⟨⟨TA⟩⟩ ⊆ ⟨⟨A⟩⟩)
addCons(⟨⟨T⟩⟩ ◁ ⟨⟨_,T,TA⟩⟩ ⋈ ⟨⟨_,T,TA⟩⟩)
addCons(⟨⟨T⟩⟩ ▷ ⟨⟨_,T,B⟩⟩ ⋈ ⟨⟨_,T,B⟩⟩)
addCons(⟨⟨T⟩⟩ $\xrightarrow{\text{id}}$ ⟨⟨_,T,TA⟩⟩ ⋈ ⟨⟨_,T,B⟩⟩)
addCons(⟨⟨TA⟩⟩ ▷ ⟨⟨_,T,TA⟩⟩)
addCons(⟨⟨B⟩⟩ ▷ ⟨⟨_,T,B⟩⟩)
foreach $c \in Cons$ *forwhich contains(*⟨⟨_, A, B⟩⟩, c*)* **do**
  deleteCons(c)
deleteEdge(⟨⟨_,A,B⟩⟩,⟨⟨T⟩⟩)

SQL Construct	Variant	HDM Constraints
Column	null	$\langle\langle T\rangle\rangle \triangleright \langle\langle \_, T, C\rangle\rangle$, $\langle\langle T\rangle\rangle \triangleleft \langle\langle \_, T, C\rangle\rangle$
Column	not null	$\langle\langle C\rangle\rangle \triangleright \langle\langle \_, T, C\rangle\rangle$, $\langle\langle T\rangle\rangle \triangleleft \langle\langle \_, T, C\rangle\rangle$, $\langle\langle T\rangle\rangle \triangleright \langle\langle \_, T, C\rangle\rangle$
Primary Key		$\langle\langle T\rangle\rangle \xrightarrow{\text{id}} \langle\langle \_, T, C\rangle\rangle$, $\langle\langle C\rangle\rangle \triangleright \langle\langle \_, T, C\rangle\rangle$, $\langle\langle T\rangle\rangle \triangleleft \langle\langle \_, T, C\rangle\rangle$ $\langle\langle T\rangle\rangle \triangleright \langle\langle \_, T, C\rangle\rangle$
Foreign Key		$\langle\langle C\rangle\rangle \subseteq \langle\langle T\rangle\rangle$

Table 1. SQL constructs and the associated constraints

In Figure 3, the constraints associated with ⟨⟨_,person,name⟩⟩ match those of a `not null` SQL column as shown in Table 3, where ⟨⟨person⟩⟩ acts as the table node and ⟨⟨name⟩⟩ the column node. Conversely the constraints on ⟨⟨_,staff,dept⟩⟩ do not match any of the SQL constructs.

AutoMatch as shown in Algorithm 2, loops through all the edges in S, $edges(S)$, comparing the associated constraints with those generated when a construct from the target DML is expressed in the HDM. Each edge in $edges(S)$ has a target model construct label attached to it that is initially set to null. We use this label to identify the target DML construct that the HDM schema object has been matched to. A similar algorithm is used to identify matches between HDM constraint constructs in S, and constructs in the target DML.

get_constraints(S, e) returns the list of constraint operators in S that are attached to e. get_target_constraint_constraints returns the constraint list for ts. For example, if ts was a SQL column, the function would return the first and second lines from Table 3. match(dc, tc) returns true if dc matches any of the variants of ts. label_dependent_schema_objects(e, ts) sets the label of e in $edges(S)$ to ts. If the HDM representation of ts includes constructs other than e these are also labelled with the appropriate target DML construct.

Consider ⟨⟨_,person,name⟩⟩ in Figure 3. If our target model was SQL then the algorithm would identify this edge as part of a SQL column. ⟨⟨person⟩⟩.label would be set to `table` and ⟨⟨_,person,name⟩⟩.label and ⟨⟨name⟩⟩.label would be set to `column`.

Algorithm 2: AutoMatch(S,TM)

Input: S:an HDM schema, TM:the list of target DML constructs
return *true if all edges have been labelled, otherwise false*
$all\_labelled$:= true;
foreach *e in edges(S)* **do**
  dc := get_constraints(S, e);
  foreach *ts in TM* **do**
    if *e.label = null* **then**
      tc := get_target_construct_constraints(ts);
      if match(dc, tc) **then**
        label_dependent_schema_objects(e, ts);
  if *e.label = null* **then**
    $all\_labelled$:= false;
return *all_labelled*;

In contrast ⟨⟨_,dept,person⟩⟩ cannot be matched to any target DML structures and so ⟨⟨_,dept,person⟩⟩.label remains null.

4 AutoTransform

AutoTransform transforms the unidentified HDM constructs of our source schema into equivalent groups of HDM constructs that match those representing a construct in the target DML. It is based on a search of the set of possible schemas that can be created by applying CTs to unidentified schema elements. This set is called the **world space** [7] of the problem. It can be represented as a graph whose nodes are the individual HDM schemas and whose edges are the CTs needed to get from one HDM schema to the next. The world space graph for the example in Section 5 is shown in Figure 7. The algorithm performs a depth first search on the world space starting from the initial state and executing CTs until a solution or a dead end is reached.

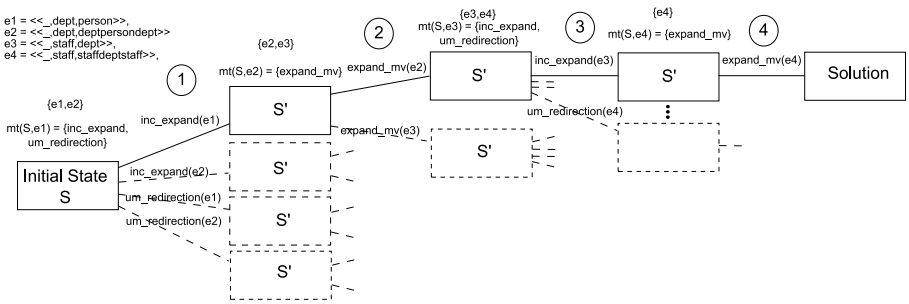

Fig. 7. The world space graph for the example

To limit the number of possible actions that may be performed at each node of the world space graph, each action must satisfy certain preconditions before it can be exe-

cuted. In our algorithm the preconditions rely on the structure of the graph surrounding the schema object the CT is to be applied to. Some of the CTs and their preconditions are shown in Table 2. In addition to those mentioned in Section 2.1 we include um_redirection [3]. The DNC in the table means we Do Not Care (DNC) whether the precondition is met or not. If we assume so is the current schema object the preconditions are:

edge is so an edge
leaf is so a leaf node or connected to a leaf node
reflexive is there a reflexive constraint attached to so
join does so take part in a join

Transformation	edge	leaf	reflexive	join
inc_expand	Y	N	N	N
um_redirection	Y	N	DNC	N
expand_mv	Y	Y	N	N
id_node_expand	N	Y	N	N

Table 2. The preconditions of the CTs used in the example

As an example consider the inc_expand transformation. It can only be applied to an edge, the edge must not be attached to leaf node, there must not be a reflexive constraint on the edge and the edge must not take part in a join. As we saw in the previous section, AutoMatch was unable to match $\langle\langle$_,dept,person$\rangle\rangle$ in Figure 3 to any target DML construct. We see, however, that this edge matches all the preconditions for inc_expand. These preconditions provide a heuristic method of selecting the CTs to execute. Those CTs that match the preconditions for a given node in the world space graph are put into a list and those have the fewest DNCs, *i.e.* that match the preconditions most closely, are put at the top of the list.

AutoTransform works as follows, first AutoMatch is run to label $edges(S)$. If AutoMatch is able to label all the edges in $edges(S)$ the transformation has been a success, the current schema is added to the result pathway and the algorithm returns the pathway. Otherwise, the algorithm loops through all the edges in $edges(S)$ looking for those with null labels. When one is found the matching_cts function is called to create an ordered list of CTs whose preconditions match the structure of the graph surrounding the edge. The hashmap, CT_tried, is checked to make sure the CT at the top of the list has not been tried on the current edge in the current schema. If it has the next CT is tried. If not the CT is applied to the edge to create schema S'. The current schema, S, is then added to the result pathway and CT_tried is updated with the current edge and schema. The algorithm is then called again with the transformed schema and the tail of the pathway.

If no suitable transformation can be found for any of the unidentified schema elements then the head function is used to remove the most recent schema from the result pathway to allow backtracking. For example, in Figure 7 if we came to a dead end after step 1 we could backtrack to schema S and try the inc_expand transformation on $e2$. If

Algorithm 3: AutoTransform(S, TM, CT, pathway)

```
Input: S: an HDM schema, TM: the list of target DML constructs,
CT: the set of possible CTs, pathway: the transformation pathway, initially []
return a transformation pathway describing how to transform the source schema into one
that matches the constructs of the target DML
CT_tried = new HashMap;
if AutoMatch(S, TM) then
    pathway := Concatenate(S, pathway);
    return pathway;
else
    S' := null;
    foreach e in edges(S) do
        if e.label = null then
            mt[] := matching_cts(S, e);
            foreach t in mt[] do
                if !CT_tried.(t) contains (S,e) then
                    S' := the result of applying t to e;
                    pathway := Concatenate(S, pathway);
                    CT_tried.put((S, e), t);
                    AutoTransform(S', TM, CT, pathway);
    if S' = null then
        S' := head(pathway);
        if S' = null then
            Exception;
        else
            AutoTransform(S', TM, CT, tail(pathway));
```

the result path is empty then we have failed to transform the schema. If it does not fail the algorithm is run again on S' with the updated result pathway.

5 Example transformation from XML to SQL

In this section, we show how AutoTransform is used to transform the schema shown in Figure 3 into one that matches the structure of an SQL schema represented in the HDM. The world space for the example is shown in Figure 7 and the list of CTs selected by the algorithm is shown below.

1. inc_expand(⟨⟨person⟩⟩,⟨⟨_,dept,person⟩⟩)
2. expand_mv(⟨⟨deptpersondept⟩⟩,⟨⟨_,dept,deptpersondept⟩⟩)
3. inc_expand(⟨⟨dept⟩⟩,⟨⟨_,staff,dept⟩⟩)
4. expand_mv(⟨⟨staffdeptstaff⟩⟩,⟨⟨_,staff,staffdeptstaff⟩⟩)

In the first iteration AutoMatch returns ⟨⟨_,dept,person⟩⟩ and ⟨⟨_,staff,dept⟩⟩ with null labels. If we consider ⟨⟨_,dept,person⟩⟩ first, and compare the structure of the surrounding schema with the preconditions in Table 2, we see that two CTs match. inc_expand

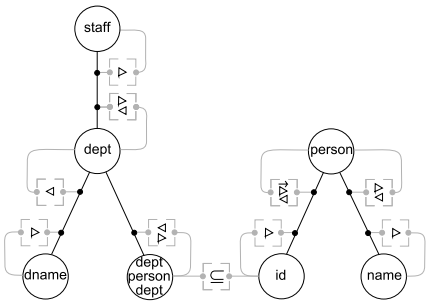

Fig. 8. After applying CT 1

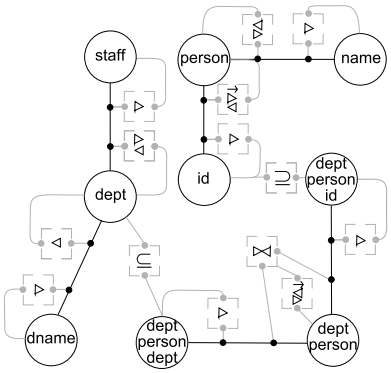

Fig. 9. After applying CT 2

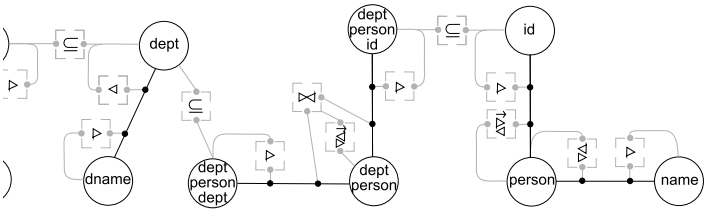

Fig. 10. Final HDM Schema

matches with one DNC, whereas um_redirection has two DNCs, so inc_expand is executed. The resulting schema is shown in Figure 8. In the second iteration ⟨⟨_,staff,dept⟩⟩ and the newly created edge ⟨⟨_,dept,deptpersondept⟩⟩ will be returned with null labels by AutoMatch. The only CT whose preconditions are met by ⟨⟨_,dept,deptpersondept⟩⟩ is expand_mv. The resulting schema is shown in Figure 9. Two similar iterations that execute CTs 3 and 4, transform ⟨⟨_,staff,dept⟩⟩ to create the schema shown in Figure 10, where all the HDM constructs match those of the SQL model. The ⟨⟨person⟩⟩, ⟨⟨deptperson⟩⟩, ⟨⟨dept⟩⟩, and ⟨⟨staffdept⟩⟩ nodes become tables, the nodes linked to them become columns in those tables. The remaining ⊆ constraints become foreign keys.

The final HDM schema is, however, not equivalent to a well designed SQL schema. The algorithm has identified ⟨⟨dept⟩⟩ and ⟨⟨staff⟩⟩ as tables but there is no key column for either table. As part of Step 3 from Figure 1 a number of target DML specific rules to overcome cases such as this are defined. Here id_node_expand(⟨⟨dept⟩⟩) can be applied to ⟨⟨dept⟩⟩ and ⟨⟨staff⟩⟩ to create ⟨⟨dept_pk⟩⟩ and an edge linking it to ⟨⟨dept⟩⟩, along with ⟨⟨staff_pk⟩⟩ and an edge linking it to ⟨⟨staff⟩⟩ that represent key columns for the tables.

6 Analysis and Experimental Results

In analysing AutoMatch, we count the number of checks for equality between the source graph structures and those of the target DML. If we let the number of objects in the graph be num_o and the number of constructs, including all variants, in the target

dept

staff_pk
00
00
00

staffdept

staffdeptstaff	staffdeptdept
00	01
00	06
00	09

dept

dept_pk	dname
01	Finance
06	HR
09	

person

id	name
1	John Smith
2	Peter Green

deptperson

deptpersondept	deptpersonid
01	1
06	2

deptperson.deptpersondept → dept.dept_pk, deptperson.deptpersonid → person.id
staffdept.staffdeptstaff → staff.staff_pk, staffdept.staffdeptdept → dept.dept_pk

Fig. 11. Translated SQL Schema

DML be num_s, then the total number of checks is $num_o \times num_s$. This is O(num_o) in the number of objects in the schema, since num_s is a constant. In the example num_s is four since there are four different constructs in the SQL model that we represent in the HDM.

We can analyse AutoTransform by counting the number of times we need to run AutoMatch. In the worst case, no structures in the source graph are identified as matching target structures by AutoMatch, and we will need to iterate num_o times. If we further assume we have num_t different composite transformations to choose from, and a world space graph of depth x, in a worst case scenario we will need to visit $(num_o \times num_t)^x$ nodes in the world space graph. Within each node of the world space graph we will need to perform the checks in AutoMatch.

It is clearly vital to limit both the size of $num_e \times num_t$ and x. There is a trade off here though. The more CTs we use the more likely we are to reach our goal in fewer steps, but each extra one will increase the size of the world space graph exponentially. To get around this, the CTs we use have stringent preconditions so that in practice the number that can be chosen at each iteration of the algorithm is limited. We also want to limit the chances of costly backtracking in the algorithm. The preconditions also help here in that they ensure as far as possible that CTs are only chosen in the correct circumstances. In our experiments so far we have found that very little backtracking is necessary and in most cases the most useful transformation is chosen first. We have successfully translated a number of different ER, SQL and XML schemas using the six existing CTs [3] and the new CT defined in Algorithm 1.

Figure 12 shows the number of match operations verses the number of schema objects required to translate various subsets of an XML Schema representation of DBLP into SQL. The gradient is steepest when schema objects from the source DML that have not direct equivalent in the target DML are added to the source schema, in this case nested XML Schema complex types. Where the graph is flatter constructs that could be matched directly with the target model, like XML Schema attributes, were added. Figure 13 shows matches vs schema objects for the translation of a SQL database to ER. Again the graph is steeper when tables with foreign keys are added.

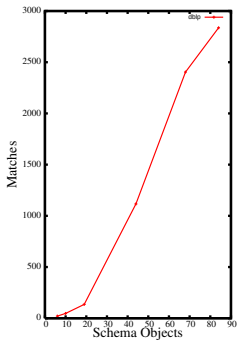

Fig. 12. DBLP XML Schema to SQL

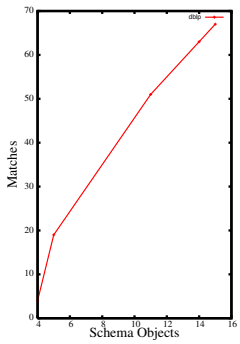

Fig. 13. SQL database to ER

The experimental results described here were produced using the AUTOMED [8] inter model data integration tool. AUTOMED implements the BAV schema transformation approach and uses the HDM as its UMM.

7 Related Work

The work most closely related to ours is that done by Atzeni et al in their MIDST system [9, 10]. They also generate data-level translations by composition of elementary transformations to translate schemas and data between a number of different DMLs. Each DML, however, is defined with a number of variants. This is not necessary with our approach. In contrast to our schema level approach the rules they use are applied across an entire pair of data models and must be predefined for each pair of models in the system. Our rules are chosen at run time independently of the source DML.

The design of our UMM also differs from that used in MIDST. They create a complex, high-level model that includes abstractions of all the constructs of the models the UMM is to represent. We, on the other hand, use a set of simple UMM constructs and use combinations of these to create any complex structures needed. This is a more flexible approach and is the one most commonly adopted, Batini et al., in their survey of a data integration methods [11], suggest that a simpler UMM has advantages over more complex models.

Schema only implementations of ModelGen include Rondo [12] and AutoGen [13]. Numerous examples of systems for translating between specific models exist in the literature. XML and relational schemas [14] as well as ER and relational [15] and ER and XML schemas [16]. More recent work on object relational to SQL translation has been done by Mork and Bernstein [17].

8 Conclusion

This paper has presented a generic data level implementation of the ModelGen model management operator that returns the translated schema along with its data instances as well as a mapping from the source to the target schema. We have shown how a schema

and its associated data instances can be translated from one DML to another by the application of information preserving CTs. We have described an algorithm for choosing the most suitable CT at each stage of the translation process and a mechanism for determining when a given schema matches the constructs of the target DML. Finally we presented some experimental results. Our on going work in this area includes investigating the translation of OWL schemas into the other DMLs we currently support.

References

1. Bernstein, P.A., Halevy, A.Y., Pottinger, R.: A vision of management of complex models. SIGMOD Record **29**(4) (2000) 55–63
2. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: SIGMOD Conference. (2007) 1–12
3. Boyd, M., McBrien, P.: Comparing and transforming between data models via an intermediate hypergraph data model. J. Data Semantics IV (2005) 69–109
4. McBrien, P., Poulovassilis, A.: A semantic approach to integrating XML and structured data sources. In: Advanced Information Systems Engineering. Volume 2068 of LNCS., Springer Verlag (2001) 330–345
5. Hull, R.: Relative information capacity of simple relational database schemata. SIAM J. Comput. **15**(3) (1986) 856–886
6. McBrien, P., Poulovassilis, A.: Data integration by bi-directional schema transformation rules. In: ICDE. (2003) 227–238
7. Weld, D.S.: An introduction to least commitment planning. AI Magazine **15**(4) (1994) 27–61
8. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien and N. Rizopoulos: AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In: CAiSE04. Volume 3084 of LNCS., Springer Verlag (2004) 82–97
9. Atzeni, P., Cappellari, P., Bernstein, P.A.: Modelgen: Model independent schema translation. In: ICDE. (2005) 1111–1112
10. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model-independent schema and data translation. In: EDBT. (2006) 368–385
11. Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of methodologies for database schema integration. ACM Comput. Surv. **18**(4) (1986) 323–364
12. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A programming platform for generic model management. In: SIGMOD Conference. (2003) 193–204
13. Song, G.L., Kong, J., Zhang, K.: Autogen: Easing model management through two levels of abstraction. J. Vis. Lang. Comput. **17**(6) (2006) 508–527
14. Jayavel Shanmugasundaram et al.: Efficiently publishing relational data as XML documents. VLDB Journal: Very Large Data Bases **10**(2–3) (2001) 133–154
15. Premerlani, W.J., Blaha, M.R.: An approach for reverse engineering of relational databases. Commun. ACM **37**(5) (1994) 42–49, 134
16. Arijit Sengupta, S.M., Doshi, R.: XER - Extensible Entity Relationship Modeling. In et al., J.H., ed.: Proceedings of the XML 2003 Conference, Philadelphia, PA, USA (2003)
17. Mork, P., Bernstein, P.A., Melnik, S.: Teaching a schema translator to produce o/r views. In: ER. (2007) 102–119