Advances in Data Management
The AutoMed Heterogeneous Data Integration System
A.Poulovassilis

1 AutoMed

Heterogeneous data integration systems generally support just one Common Data Model (CDM) e.g. relational, Entity/Relationship, Object-Oriented, graph-based.

For each type of data source, there is a *wrapper* for translating its schema into the CDM.

Global schemas are then defined by means of view definitions over the export schemas expressed in this CDM[1].

AutoMed is a schema transformation and integration system developed at Birkbeck and Imperial Colleges — see `http://www.doc.ic.ac.uk/automed`.

AutoMed provides a low-level **hypergraph-based data model (HDM)**.

Higher-level modelling languages (e.g. relational, OO, XML, RDF/S, OWL) can be defined in terms of this lower-level model, using the API of AutoMed's Model Definitions Repository (MDR).

Thus, there is not a single Common Data Model in AutoMed; instead, it is possible to use any of the modelling languages defined in the MDR to specify a global schema. It is also possible to extend this set of modelling languages with variants and new languages.

For any modelling language $\mathcal{M}$ specified in terms of the HDM, AutoMed provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$.

In particular, there is an add and a delete primitive transformation for adding/deleting any construct of $\mathcal{M}$ to/from a schema.

For those constructs of $\mathcal{M}$ which have textual names, there is also a rename primitive transformation.

Schemas are incrementally transformed by applying to them a sequence of such transformations.

Each add or delete transformation is accompanied by a query which defines the extent of the new or deleted schema construct in terms of the rest of the constructs in the schema i.e. this query specifies a *view definition*.

This query is expressed in an **intermediate query language**, **IQL**[2]. IQL is a *functional query language* [3].

We term a sequence of transformations transforming a schema S_1 to a schema S_2 a **transformation pathway**, and we denote it by $S_1 \rightarrow S_2$.

All source, intermediate and integrated schemas, and the pathways between them, are stored in

[1] This is known as **global-as-view (GAV) data integration**, which is what I am assuming for the purposes of this course.

Also possible is **local-as-view (LAV) data integration**, in which local schemas are defined as views over a global schema.

AutoMed is actually a **both-as-view** data integration system, as its schema transformation pathways can be used to generate views for both GAV and LAV query processing.

[2] All primitive transformations have an optional additional argument which specifies a constraint on the current schema extension that must hold if the transformation is to be applied. Constraints are also expressed as IQL queries. For simplicity, none of the examples here need to make use of this feature.

[3] Functional query languages have a computational model which is based on functional programming.

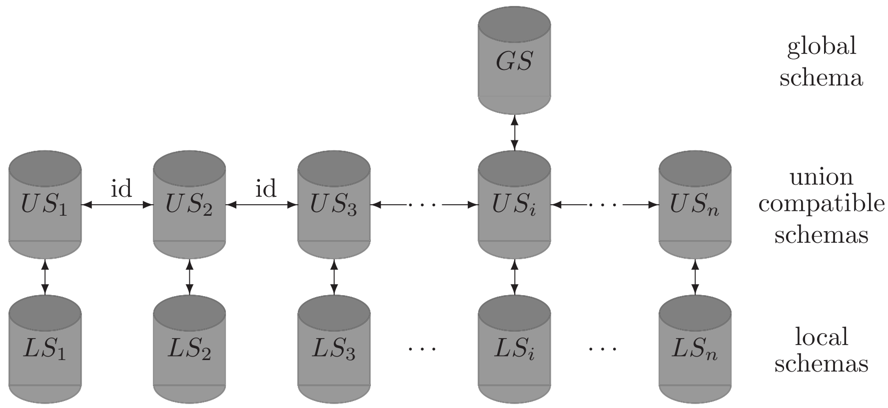

Figure 1: A general AutoMed Integration

AutoMed's Schemas & Transformations Repository (STR).

Schema integration in AutoMed proceeds by forming union-compatible schemas, as illustrated in Figure 1.

In order to integrate n local schemas, $LS_1, \ldots, LS_n$, each LS_i first needs to be transformed into a union-compatible schema US_i. (Henceforth, we use the term 'union schema' synonymously with 'union-compatible schema'.)

These n union schemas are syntactically identical, and this is indicated by creating a sequence of id transformation steps between each pair US_i and US_{i+1}, of the form id $(US_i : c, US_{i+1} : c)$ for each schema construct c in US_i and US_{i+1}[4].

These id transformations can be generated automatically by the AutoMed software.

An arbitrary one of the US_i can then be selected for further transformation into a global schema GS.

There may be information within a US_i which is not semantically derivable from the corresponding LS_i. This is indicated by extend transformation steps occurring within the pathway $LS_i \rightarrow US_i$.

Conversely, not all of the information within a local schema LS_i need be transferred into US_i and this is indicated by contract transformation steps occurring within $LS_i \rightarrow US_i$.

These extend and contract transformations behave in the same way as add and delete, respectively, except that they are accompanied by two queries, a lower-bound query and an upper-bound query, which specify a range of values within which the extents of the new/removed constructs lie.

2 Representing Relational and OO Data Models in AutoMed

The following tables show the specification of (a) the relational data model, with four basic modelling constructs table, column, primary_key and foreign_key; and (b) part of a typical object-oriented data model.

[4] id is an additional type of primitive transformation in AutoMed, and $US_i : c$ denotes construct c appearing in schema US_i.

Relational Construct
construct: `table` R scheme: $\ll R \gg$
construct: `column` a of R scheme: $\ll R, a \gg$
construct: `primary_key` on constructs $c_1, \ldots, c_n$ of R scheme: $\ll R\_pk, R, c_1, \ldots, c_n \gg$
construct: `foreign_key` scheme: $\ll R\_fki, R, c, R', c' \gg$

OO Construct
construct: `class` C scheme: $\ll C \gg$
construct: `class_attribute` a of C, of type C' scheme: $\ll C, a, C' \gg$
construct: `scalar_attribute` a of C scheme: $\ll C, a \gg$
construct: `set_atttribute` a of C, of type $set(C')$ scheme: $\ll C, a, C' \gg$
construct: `primary_key` on constructs $c_1, \ldots, c_n$ of C scheme: $\ll C\_pk, C, c_1, \ldots, c_n \gg$

3 Examples

Example 1.

The transformation of the OO schema L_2 into the relational schema C_2 in Example 1 from the earlier Notes can be accomplished by a series of transformation steps that first add the missing relational schema constructs of C_2 and then remove the, no longer needed, OO schema constructs of L_2.

For example, here is a sequence of transformations transforming the *Department* class of L_2 into the *Department* relation of C_2:

```
  //  "growing" phase:
add(table,<<Department_R>>,[{d}|{o,d}<-<<Department,deptName>>])
add(column,<<Department_R,deptName>>,[{d,d}|{o,d}<-<<Department,deptName>>])
add(column,<<Department_R,deptHead>>,[{d,h}|{o1,d}<-<<Department,deptName>>;
                                            {o2,h}<-<<Department,deptHead>>;
                                            o1 = o2)]
add(primary_key,<<Department_R_pk,Department_R,<<Department_R,deptName>>>>)
  //  "shrinking" phase:
contract(primary_key,<<Department_pk,Department,<<Department,deptName>>>>)
contract(scalar_attribute,<<Department,deptHead>>)
contract(scalar_attribute,<<Department,deptName>>)
contract(class,<<Department>>)
  //  "renaming" phase:
rename(<<Department_R_pk,Department_R,<<Department_R,deptName>>,Department_pk)
rename(<<Department_R>>,Department)
```

Example 2.

Consider the export schema ES_1 from the previous Notes:

$Student(\underline{studentId}, name, address, \mathbf{tutorId})$
$Staff(\underline{tutorId}, name, deptName)$
$Lecturer(\underline{lecturerId}, name, deptName)$
$Course(\underline{courseId}, name, programme)$
$Teaches(\underline{\mathbf{lecturerId}}, \underline{\mathbf{studentId}})$

and the export schema ES_2:

$Department(\underline{deptName}, deptHead)$
$Course(\underline{courseId}, courseName, units)$
$Enrollment(\underline{studentId}, \underline{year})$
$Staff(\underline{staffId}, name, \mathbf{deptName})$
$Teaches(\underline{\mathbf{staffId}}, \underline{\mathbf{courseId}})$
$CourseEnrollments(\underline{\mathbf{studentId}}, \underline{\mathbf{year}}, \underline{\mathbf{courseId}})$

and consider their integration discussed in Example 2 in the previous Notes.

Let us see how AutoMed can express this integration, and as well as automatically generating the necessary schema mappings. For simplicity, I will ignore foreign key constraints.

Let us first express the transformations that were applied to ES_1 and ES_2 in Example 2 as AutoMed transformations:

- add to ES_1 a new relation $Department(\underline{deptName})$ populated by the $deptName$ values in $Staff$ and $Lecturer$:

```
add(table, <<Department>>,
           distinct ([{d}|{i,d}<-<<Staff,deptName>>] ++
                     [{d}|{i,d}<-{<<Lecturer,deptName>>]))
add(column,<<Department,deptName>>,
           distinct ([{d,d}|{i,d}<-<<Staff,deptName>>] ++
                     [{d,d}|{i,d}<-{<<Lecturer,deptName>>]))
add(primary_key,<<Department_pk,Department,<<Department,deptName>>>>)
```

- rename $Staff$ in ES_2 to $Lecturer$:

```
rename(<<Staff_pk,Staff,<<Staff,StaffId>>>>,Lecturer_pk)
rename(<<Staff>>,Lecturer)
```

- rename the attribute $Lecturer.staffId$ in ES_2 to $lecturerId$:

```
rename(<<Lecturer,staffId>>,lecturerId)
```

  and similarly the attribute $Teaches.staffId$:

```
rename(<<Teaches,staffId>>,lecturerId)
```

- rename the attribute $Course.name$ in ES_1 to $courseName$:

```
rename(<<Course,name>>,courseName)
```

- rename $Teaches$ in ES_1 to $TeachesStudents$:

```
rename(<<Teaches_pk,Teaches,<<Teaches,staffId>>,<<Teaches,courseID>>>>,
         TeachesStudents_pk)
rename(<<Teaches>>,TeachesStudents)
```

The above transformations on ES_1, i.e.

```
add(table, <<Department>>,
           distinct ([{d}|{i,d}<-<<Staff,deptName>>] ++
                     [{d}|{i,d}<-{<<Lecturer,deptName>>]))
add(column,<<Department,deptName>>,
           distinct ([{d,d}|{i,d}<-<<Staff,deptName>>] ++
                     [{d,d}|{i,d}<-{<<Lecturer,deptName>>]))
add(primary_key,<<Department_pk,Department,<<Department,deptName>>>>)
rename(<<Course,name>>,courseName)
rename(<<Teaches_pk,Teaches,<<Teaches,staffId>>,<<Teaches,courseID>>>>,TeachesStudents_pk)
rename(<<Teaches>>,TeachesStudents)
```

are applied to ES_1, obtaining a new schema I_1.

Similarly, the above transformations on ES_2, i.e.

```
rename(<<Staff_pk,Staff,<<Staff,StaffId>>>>,Lecturer_pk)
rename(<<Staff>>,Lecturer)
rename(<<Lecturer,staffId>>,lecturerId)
```

are applied to ES_2, obtaining a new schema I_2.

Next, we create a new schema U_1 that extends I_1 with the constructs that it is missing from I_2; and similarly we create a new schema U_2 that extends I_2 with the constructs that it is missing from I_1. This can be done automatically by the AutoMed software, taking I_1 and I_2 as input. The resulting schemas U_1 and U_2 are identical and look as follows:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, \textbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \textbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$TeachesStudents(\underline{\textbf{lecturerId}}, \underline{\textbf{studentId}})$
$Department(\underline{deptName}, deptHead)$
$Enrollment(\underline{studentId}, \underline{year})$
$CourseEnrollments(\underline{\textbf{studentId}}, \underline{\textbf{year}}, \underline{\textbf{courseId}})$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{courseId}})$

Next, we link U_1 and U_2 by a set of id transformation steps — this can be done automatically by the AutoMed software. So if a query is now applied to U_1 or U_2 data will be sourced from *both* L_1 and L_2:

- Schema constructs that appear in both I_1 and I_2 are populated by a bag union of the data from L_1 and L_2.
- Schema constructs that appear only in one of I_1 or I_2 are populated by data from just L_1 or L_2, respectively.

Finally, we can now select either of U_1 or U_2, let's say U_1, for further improvement into the final global schema:

- remove the redundant relation $TeachesStudents$ since this information can be derived by joining $Teaches$ and $CourseEnrollments$:

```
delete(primary_key,<<TeachesStudents_pk,TeachesStudents,<<TeachesStudents,lecturerId>>,
         <<TeachesStudents,studentId>>>>)
delete(column,<<TeachesStudents,studentId>>,
         [{l,s,s} | {l,c1}<-<<Teaches>>; {s,y,c2}<-<<CourseEnrollments>>; c1 = c2])
delete(column,<<TeachesStudents,lecturerId>>,
         [{l,s,l} | {l,c1}<-<<Teaches>>; {s,y,c2}<-<<CourseEnrollments>>; c1 = c2])
delete(table, <<TeachesStudents>>,
         [{l,s} | {l,c1}<-<<Teaches>>; {s,y,c2}<-<<CourseEnrollments>>; c1 = c2])
```

- remove the redundant relation *Enrollment* since this information can be derived by projecting on the $studentId, year$ attributes of $CourseEnrollments$

```
delete(primary_key,<<Enrollment_pk,Enrollment,<<Enrollment,studentId>>,
                                            <<Enrollment,year>>)
contract(column,<<Enrollment,year>>,[{s,y,y} | {s,y,c}<-<<CourseEnrollments>>])
contract(column,<<Enrollment,studentId>>,[{s,y,s} | {s,y,c}<-<<CourseEnrollments>>])
contract(table,<<Enrollment>>, [{s,y} | {s,y,c}<-<<CourseEnrollments>>])
```

The global schema, GS, resulting from all the above transformations is:

$Student(\underline{studentId}, name, address, \textbf{tutorId})$
$Staff(\underline{tutorId}, name, \textbf{deptName})$
$Lecturer(\underline{lecturerId}, name, \textbf{deptName})$
$Course(\underline{courseId}, courseName, units, programme)$
$Department(\underline{deptName}, deptHead)$
$CourseEnrollments(\underline{\textbf{studentId}}, \underline{\textbf{year}}, \underline{\textbf{courseId}})$
$Teaches(\underline{\textbf{lecturerId}}, \underline{\textbf{courseId}})$

4 Generating the Schema Mappings

In Automed, each primitive transformation has an automatically derivable *reverse transformation*:

- each add or extend transformation is reversed by a delete or contract transformation with the same arguments;
- each delete or contract transformation is reversed by an add or extend transformation with the same arguments;
- each rename transformation is reversed by a rename that restores the original name;
- each id transformation is reversed by an id transformation with the two arguments swapped.

View definitions which define the constructs of a global schema in terms of the constructs of a set of source schemas are generated automatically by the AutoMed software. The view generation algorithm makes use of the transformation pathways between the source schemas and the global schema.

For example, AutoMed's view generation algorithm will create views which define the constructs of the global schema GS in the example earlier in terms of the constructs of the two source schemas L_1 and L_2.

These views are generated from the transformation pathways that we defined from L_1 and L_2 to GS.

This view generation is done by traversing the *reverse* pathways from GS down to each L_i. The only transformations that are significant for the purposes of the view generation are those that delete, contract or rename a construct:

- delete: This has an associated query which shows how to reconstruct the extent of the construct being deleted. Any occurrence of the deleted construct within the current view definitions is replaced by this query.
- contract: This is treated similarly to delete, except that its associated pair of lower and upper bound queries is used.
- rename: All references to the old construct in the current view definitions are replaced by references to the new construct.

For example, here are the successive rewriting steps that define the global construct `GS:<<Department>>` as a view over the relational source schema L_1 and the OO source schema L_2:

```
GS:<<Department>>
=>
I1:<<Department>> ++ I2:<<Department>>
=>
(distinct ([{d}|{i,d}<-ES1:<<Staff,deptName>>] ++
           [{d}|{i,d}<-ES1:<<Lecturer,deptName>>]))
++ ES2:<<Department>>
=>
(distinct ([{d}|{i,d}<-LS1:<<Staff,deptName>>] ++
           [{d}|{i,d}<-LS1:<<Lecturer,deptName>>]))
++ [{d} | {o,d}<-LS2:<<Department,deptName>>]
```

Such view definitions are then used by AutoMed for global query processing by substituting the view definitions into queries that are expressed over the global schema, so as to reformulate such queries into queries that are expressed on the source schemas.