An Overview of The AutoMed Repository

AutoMed Technical Report 26, Draft

Michael Boyd, Charalambos Lazanitis, Sasivimol Kittivoravitkul,
Peter McBrien and Nikos Rizopoulos
Dept. of Computing, Imperial College, London SW7 2AZ
email: {mboyd,cl201,sk297,pjm,nr600}@doc.ic.ac.uk

Friday 13th February 2004

Abstract

This paper describes the AutoMed repository and some associated tools, which provide the first implementation of the **both as view** (**BAV**) approach to data integration. Apart from being a highly expressive data integration approach, BAV has the additional advantages that it provides a method to support a wide range of data modelling languages, and to describe transformations between those data modelling languages. This paper documents how BAV can be implemented, including how the approach can scale up to handle large and evolving systems, by (1) breaking down the system into well defined units called subnets that may be edited independently of each other, (2) providing a template definition system to automate the production of common mappings between data sources, and (3) providing a schema-matching tool that assists in the identification of which transformations should apply. We illustrate the implementation with examples in the relational, ER data models, and semi-structured text files.

1 Introduction

The AutoMed project[1] has developed the first implementation of a data integration technique called **both-as-view** (**BAV**) [17], which subsumes the expressive power of other published data integration techniques **global-as-view** (**GAV**), **local-as-view** (**LAV**), and **global-local-as-view** (**GLAV**) [9]. BAV also distinguishes itself in being the approach which has a clear methodology for handling a wide range of data models in the integration process, as opposed to the other approaches that assume integration is always performed in a single common data model.

In this paper we describe the core **repository** of the AutoMed toolkit, and several packages that make use of this repository. Apart from giving an overview of this freely available software product, we describe the solutions to practical problems of using the BAV approach to integrate large schemas from heterogeneous and evolving data sources.

The paper is structured as follows. Section 2 reviews the BAV approach and demonstrates how it models a relational data source, and introduces a new method that allows BAV to handle semi structured text file data sources. Section 3 then describes how the AutoMed system handles the BAV description of such data modelling languages and their integration. We show how to divide a large integration of data sources into a set of well defined subnetworks. Details of how we approach the transformation between modelling languages are given in Section 4, and the description of how to program higher level transformations as sequences of primitive transformations in a **template**

[1]The AutoMed project was an British EPSRC funded research project, jointly run by Birkbeck and Imperial Colleges, in the University of London. The Imperial College group implemented the data integration toolkit described here, with the the exception of the query processing component based on the IQL language, which was developed at Birkbeck College. Software and documentation are available from the AutoMed website http://www.doc.ic.ac.uk/automed/.

language are given in Section 5. Finally Section 6 addresses the problem of automating the schema matching process in the AutoMed framework.

2 BAV Data Integration

Data integration is the process of combining several data sources such that they may be queried and updated via some common interface. This requires that each **local schema** of each data source be mapped to the **global schema** of the common interface. In the GAV approach [9], this mapping is specified by writing a definition of each global schema construct as a view over local schema constructs. In LAV [9], this mapping is specified by defining each local schema construct as a view over global schema constructs. GLAV [13] is a variant of LAV that allows the head of the view definition to contain any query on the local schema.

In the BAV approach, each construct of a modelling language is mapped to the nodes, edges and constraints of the **hypergraph data model** (**HDM**) [19]. Using this nested hypergraph as the underlying model makes it straightforward to represent any data modelling language. For example, in [15, 17], is was shown that to model a relational model we:

- represent each relation r by the Table construct **scheme** $\langle\langle r\rangle\rangle$, which takes as instances the primary key values of the relation. For example, the relation result in Fig. 3(a) has primary key attributes code and name, and hence instances $[\langle\text{'DB'}, \text{'Mary'}\rangle, \langle\text{'Fin'}, \text{'Jane'}\rangle, \ldots]$ for a scheme $\langle\langle\mathsf{result}\rangle\rangle$.

- represent each attribute a of r by the Column construct scheme $\langle\langle r, a, c\rangle\rangle$, which takes as instances the primary key values of the relation and the associated value of the attribute, where c may be null or notnull. For example, the attribute grade of relation result in Fig. 3(a) has scheme $\langle\langle\mathsf{result}, \mathsf{grade}, \mathsf{notnull}\rangle\rangle$ and instances $[\langle\text{'DB'}, \text{'Mary'}, \mathsf{A}\rangle, \langle\text{'Fin'}, \text{'Jane'}, \mathsf{C}\rangle, \ldots]$.

- represent the primary key integrity constraint named pk on relation r by the PK construct scheme $\langle\langle pk, r, a_1, \ldots, a_n\rangle\rangle$, where $a_1, \ldots, a_n$ are the primary key attributes. For example, the primary key of the result table would be $\langle\langle\mathsf{result\_pk}, \mathsf{result}, \langle\langle\mathsf{result}, \mathsf{code}\rangle\rangle, \langle\langle\mathsf{result}, \mathsf{name}\rangle\rangle\rangle\rangle$ (assuming that result_pk was the name given in the database; and using the abbreviated form of attribute naming omitting the null constraint that shall be explained later). Since this is a constraint, is takes no instances.

Once the constructs of the data modelling language have been defined in this manner, the mapping between schemas can be described as a **pathway** of **primitive transformation** steps applied in sequence, each adding, deleting, or renaming the basic constructs of the schema modelling language being used. For each construct type in the modelling language C, there exists five primitive transformations:

1. add$C$($\langle\langle$s$\rangle\rangle$,q) applied to a schema S produces a new schema S' that differs from S in having a new construct of type C detailed by the **scheme** $\langle\langle$s$\rangle\rangle$. The extent of $\langle\langle$s$\rangle\rangle$ is given by the query q on schema S.

2. extend$C$($\langle\langle$s$\rangle\rangle$,ql,qu) applied to a schema S produces a new schema S' that differs from S in having a new construct of type C detailed by the **scheme** $\langle\langle$s$\rangle\rangle$. The minimum extent of $\langle\langle$s$\rangle\rangle$ is given by query ql, which may take the special value Void if no values of this extent may be derived from S. The maximum extent of $\langle\langle$s$\rangle\rangle$ is given by qu, which may take the special value Any if no limit on this extent may be derived from S.

3. delete$C$($\langle\langle$s$\rangle\rangle$,q) applied to a schema S produces a new schema S' that differs from S in not having a construct of type C detailed by $\langle\langle$s$\rangle\rangle$. The extent of $\langle\langle$s$\rangle\rangle$ may be restored by q on schema S'.

   Note that delete$C$($\langle\langle$s$\rangle\rangle$,q) applied to a schema S producing schema S' is equivalent to add$C$($\langle\langle$s$\rangle\rangle$,q) applied to S' producing S.

4. contract$C$(⟨⟨s⟩⟩,ql,qu) applied to a schema S produces a new schema S' that differs from S in not having a construct of type C detailed by ⟨⟨s⟩⟩. The lower bound query ql and upper bound query qu on schema S' have the same semantics as for extend.

   Note that contract$C$(⟨⟨s⟩⟩,ql,qu) applied to a schema S producing schema S' is equivalent to extend$C$(⟨⟨s⟩⟩,ql,qu) applied to S' producing S.

5. rename$C$(⟨⟨s⟩⟩,⟨⟨s′⟩⟩) applied to a schema S produces a new schema S' that differs from S in not having a construct of type C detailed by ⟨⟨s⟩⟩ and instead having ⟨⟨s′⟩⟩, differing from ⟨⟨s⟩⟩ only in its textual naming.

   Note that contract$C$(⟨⟨s⟩⟩,⟨⟨s′⟩⟩) applied to a schema S producing schema S' is equivalent to rename$C$(⟨⟨s′⟩⟩,⟨⟨s⟩⟩) applied to S' producing S.

Since transformations mapping between schemas describe both how to add and delete constructs means that it is possible to extract GAV, LAV, and GLAV rules from a BAV pathway [5]. The BAV approach allows any query language to be used in the transformation rules, but the current AutoMed implementation uses the IQL language [18, 4] together with GAV query processing. The IQL language is a list comprehensions [1] based language able to describe the behaviour of a wide range of database query languages, including SQL and XQuery.

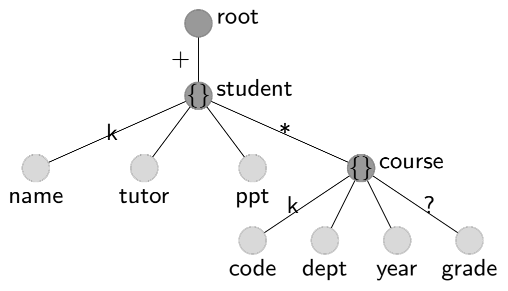
To demonstrate the approach in use, suppose we want to transform the relational $\mathsf{S_3}$ in Fig. 3(a) into the relational version of the global schema $\mathsf{S_{rg}}$ in Fig. 4(a). The level attribute in $\mathsf{S_3}$ may be used to divide students into those that belong to the ug table, created by transformations ① –③ , and those the belong to the pg table by transformations ④ –⑤ . The IQL query in ① finds in the **generator** ⟨x, y⟩ ← ⟨⟨student, level⟩⟩ the tuples ⟨'Mary', 'ug'⟩, ⟨'John', 'pg'⟩, . . . and then the **filter** y = 'ug' restricts the x values returned to be only those that had 'ug' in the second argument. Other IQL queries in square brackets may be read in a similar manner. Once the specialisation tables have been created, transformation ⑦ removes the level attribute from student, since it may be recovered from the ug and pg tables (the IQL ++ operator appends two lists together). Finally ⑧ –⑨ moves the ppt attribute from student to ug, since it only takes non-null values for undergraduate students.

$\mathsf{S_3} \rightarrow \mathsf{S_{rg}}$
① addTable(⟨⟨ug⟩⟩, [⟨x⟩ | ⟨x, y⟩ ← ⟨⟨student, level⟩⟩; y = 'ug'])
② addColumn(⟨⟨ug, name, notnull⟩⟩,
        [⟨x, y⟩ | ⟨x, y⟩ ← ⟨⟨student, name⟩⟩; ⟨x, z⟩ ← ⟨⟨student, level⟩⟩; z = 'ug'])
③ addPK(⟨⟨ug_pk, ug, ⟨⟨ug, name⟩⟩⟩⟩)
④ addTable(⟨⟨pg⟩⟩, [⟨x⟩ | ⟨x, y⟩ ← ⟨⟨student, level⟩⟩; y = 'pg'])
⑤ addColumn(⟨⟨pg, name, notnull⟩⟩,
        [⟨x, y⟩ | ⟨x, y⟩ ← ⟨⟨student, name⟩⟩; ⟨x, z⟩ ← ⟨⟨student, level⟩⟩; z = 'pg'])
⑥ addPK(⟨⟨pg_pk, pg, ⟨⟨pg, name⟩⟩⟩⟩)
⑦ deleteColumn(⟨⟨student, level, notnull⟩⟩,
        [⟨x, y⟩ | ⟨x⟩ ← ⟨⟨ug⟩⟩; y = 'ug'] ++ [⟨x, y⟩ | ⟨x⟩ ← ⟨⟨pg⟩⟩; y = 'pg'])
⑧ addColumn(⟨⟨ug, ppt⟩⟩, [⟨x, y⟩ | ⟨x⟩ ← ⟨⟨ug⟩⟩; ⟨x⟩ ← ⟨⟨student⟩⟩; ⟨x, y⟩ ← ⟨⟨student, ppt⟩⟩]
⑨ deleteColumn(⟨⟨student, ppt⟩⟩, [⟨x, y⟩ | ⟨x⟩ ← ⟨⟨student⟩⟩; ⟨x, y⟩ ← ⟨⟨ug, ppt⟩⟩])

2.1 Handling Semi-Structured Data

YATTA (YAT for Transformation-based Approach) is a variation of the YAT model [2] to support the handling of semistructured data in AutoMed. YATTA extends YAT to distinguish between ordered and unordered data. YATTA simplifies YAT to allow only two levels of abstraction: the **schema level** where the structure of data is defined, and the **data level** where actual data is presented. Fig. 1(b) shows a semistructured text file, containing data about the undergraduate students in Fig. 3(a). Fig. 1(a) gives a schema level YATTA model, and Fig 1(c) gives a data level YATTA model, for that file.

In the YATTA model, schemas and data are both represented by rooted labelled trees. In a **YATTA schema**, each node is labelled by a tuple ⟨*name, type*⟩, where *name* is a string describing what a node represents and *type* is the data type of a node. Type can be either atomic *e.g.* string,

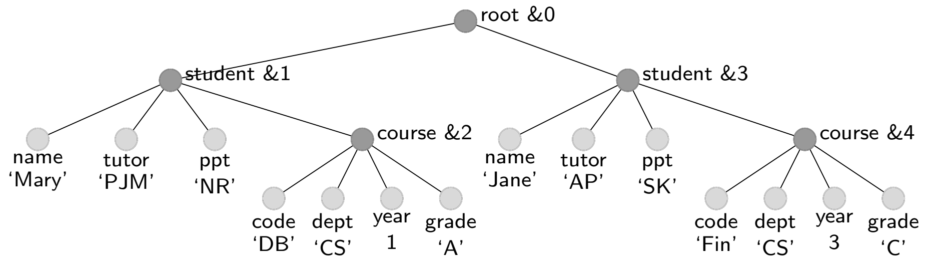


(a) YATTA schema

```
name = Mary
tutor = PJM
ppt = NR
course = DB
dept = CS
year = 1
grade = A
```

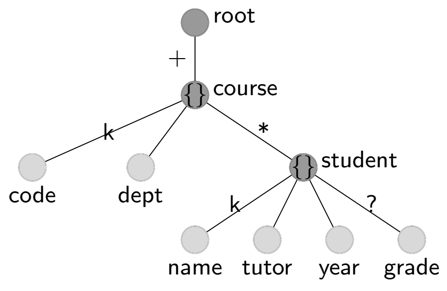
```
name = Jane
tutor = AP
ppt = SK
course = Fin
dept = CS
year = 3
grade = C
```

(b) text file



(c) YATTA data

Figure 1: S_1: Semi-structured text file of undergraduates



```
[Fin,CS]
Fred,PJM,4,A
[Geo,Maths]
Fred,PJM,4,A
John,AP,4,A
```

(a) YATTA schema (b) text file

Figure 2: S_2: Semi-structured text file of postgraduates

integer, *etc.*, or compound *i.e.* list (marked '[]'), set (marked '{}'), or bag (marked '⟨⟩'). Each node in a **YATTA data tree** is labelled by a triple ⟨*name, type, value*⟩, where *value* is the value associated with the node. If the node is of atomic type, the value is a data value of that type. If the node is of compound type, the value is an integer identifier. Outgoing edges of list nodes are ordered from left to right; the edges of set and bag are unordered. The edges of a schema are labelled with cardinality constraints which determine the number of times corresponding edges may occur in a data tree: '$*$' indicates zero or more occurrences, '+' indicates one or more occurrences, '?' indicates zero or one occurrence, and no label indicates exactly one occurrence. A 'k' is used to identify the subset of child nodes, called the key nodes, which uniquely identify the complex node with respect to its parent, all other nodes are non key nodes, which in the schemes we identify by writing 'nk', but in the diagrams simply leave the edge unlabelled.

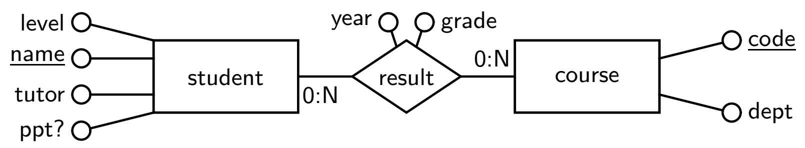
The YATTA model can be represented as BAV schemes in the following way:

student			
<u>name</u>	tutor	ppt	level
Mary	PJM	NR	ug
John	AP	null	pg
Jane	AP	SK	ug
Fred	PJM	null	pg

course	
<u>code</u>	dept
DB	CS
Fin	CS
Geo	Maths

result			
<u>code</u>	<u>name</u>	year	grade
DB	Mary	1	A
Fin	Jane	3	C
Fin	Fred	4	null
Geo	Fred	4	A
Geo	John	4	B

(a) Relational database schema and data



(b) ER model used to design relational database

Figure 3: S_3: relational database covering both undergraduates and MSc students

student	
<u>name</u>	tutor
Mary	PJM
John	AP
Jane	AP
Fred	PJM

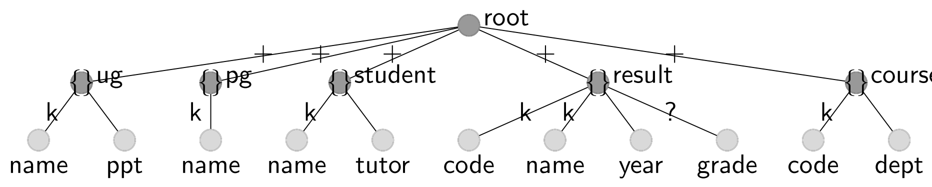
ug	
<u>name</u>	ppt
Mary	NR
Jane	SK

pg
<u>name</u>
Fred
John

course	
<u>code</u>	dept
DB	CS
Fin	CS
Geo	Maths

result			
<u>code</u>	<u>name</u>	year	grade
DB	Mary	1	A
Fin	Jane	3	C
Fin	Fred	4	null
Geo	Fred	4	A
Geo	John	4	B

(a) S_{rg}: Global schema in the relational model



(b) S_{yg}: Global schema in the YATTA model

Figure 4: Global Schema

- a root node r is represented as a RootNode construct with scheme $\langle\langle r, t\rangle\rangle$ where t is one of the YATTA types.
- a non-root node n is represented as a YattaNode construct with scheme $\langle\langle n_p, n, t, c\rangle\rangle$ where n_p is a parent node that may be a RootNode or YattaNode, t is the type of a node and c is a cardinality constraint.

To integrate S_1 with S_{rg}, we need to transform S_1 to have the same structure as S_{rg}. As will be seen in Section 4, it is straightforward to derive the YATTA schema S_{yg} shown in Figure 4(b) that is equivalent to S_{rg}. Now the task is to transform S_1 to S_{yg}. To determine the pathway from one YATTA model Y to another one Y', the following methodology can be used.

1. Use rename transformations to **conform** the schemas, such that if objects have the same name, then they also have the same extent. In $\mathsf{S_1}$, the $\mathsf{student}$ node matches the ug node in $\mathsf{S_{yg}}$, and therefore we should execute:

   $\mathsf{S_1} \rightarrow \mathsf{S_{yg}}$ (conform phase)
   ⑩ $\mathsf{renameYattaNode}(\langle\langle \mathsf{root}, \mathsf{student}, \mathsf{set}, + \rangle\rangle, \langle\langle \mathsf{root}, \mathsf{ug}, \mathsf{set}, + \rangle\rangle)$

2. Conduct a breadth first iteration over the nodes n' of Y', and for each node n' not found in Y create transformations in a **growth** phase to add each n' to Y:

   (a) If n' is of complex type, determine if there is a query q on Y such that there is a one to one mapping between values returned by q and values associated to n'. If there is, then a new node n' is added into Y by applying a rule $\mathsf{addYattaNode}$, with the special function $\mathsf{generateId}$ used on the values returned by q to generate the identifiers of the complex node. This function always returns the *same* identifier for the same input values, and *distinct* identifiers for distinct input values.

   (b) If n' is of simple type, determine if there is a query q on Y such that the values returned by q are equal to the values associated with n'. If there is, then a new node n' is added into Y by applying a rule $\mathsf{addYattaNode}$, with q placed as the query part of the transformation.

   In either case, if the query only returns some of the values of n', then instead use $\mathsf{extendYattaNode}$, with the queries set to q, Any, and if no query can be determined, then use $\mathsf{extendYattaNode}$ with the queries $\mathsf{Void}, \mathsf{Any}$ which states that there is no method to determine anything about the instances of n' in Y' from the information in Y.

   For example, we would find that result node of $\mathsf{S_{yg}}$ does not appear in $\mathsf{S_1}$, and we are able to derive *some* instances in ⑪ -⑮ from $\mathsf{S_1}$, since that contains the results of undergraduates. Step ⑪ generates identifiers for the new result node by finding $\langle \&0, \&1 \rangle, \langle \&0, \&3 \rangle$ from $\langle \mathsf{r}, \mathsf{u} \rangle \leftarrow \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle$, then $\langle \&1, \text{'Mary'} \rangle, \langle \&3, \text{'Jane'} \rangle$ from $\langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle$, then $\langle \&1, \&2 \rangle$, $\langle \&3, \&4 \rangle$ from $\langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle$, and $\langle \&2, \text{'DB'} \rangle, \langle \&4, \text{'Fin'} \rangle$ from $\langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle$. This causes $\mathsf{generateId}$ to receive the pairs $\mathsf{Mary\ DB}$ and $\mathsf{Jane\ Fin}$, and generate &5 and &6 as new identifiers for result. Note that the same identifiers will now be created in ⑫ -⑮ .

   $\mathsf{S_1} \rightarrow \mathsf{S_{yg}}$ (growth phase)
   ⑪ $\mathsf{extendYattaNode}(\langle\langle \mathsf{root}, \mathsf{result}, \mathsf{set}, + \rangle\rangle,$
   $[\langle \mathsf{r}, \mathsf{re} \rangle \mid \langle \mathsf{r}, \mathsf{u} \rangle \leftarrow \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle; \langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle;$
   $\langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle; \langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle;$
   $\mathsf{re} \leftarrow [\mathsf{generateId\ co\ n}]], \mathsf{Any})$
   ⑫ $\mathsf{extendYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{result} \rangle\rangle, \mathsf{code}, \mathsf{string}, \mathsf{k} \rangle\rangle,$
   $[\langle \mathsf{re}, \mathsf{co} \rangle \mid \langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle; \langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle;$
   $\langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle;\ \mathsf{re} \leftarrow [\mathsf{generateId\ co\ n}]], \mathsf{Any})$
   ⑬ $\mathsf{extendYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{result} \rangle\rangle, \mathsf{name}, \mathsf{string}, \mathsf{k} \rangle\rangle,$
   $[\langle \mathsf{re}, \mathsf{n} \rangle \mid \langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle; \langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle;$
   $\langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle;\ \mathsf{re} \leftarrow [\mathsf{generateId\ co\ n}]], \mathsf{Any})$
   ⑭ $\mathsf{extendYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{result} \rangle\rangle, \mathsf{year}, \mathsf{integer}, \mathsf{nk} \rangle\rangle,$
   $[\langle \mathsf{re}, \mathsf{y} \rangle \mid \langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle; \langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle;$
   $\langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle; \langle \mathsf{c}, \mathsf{y} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{year} \rangle\rangle;$
   $\mathsf{re} \leftarrow [\mathsf{generateId\ co\ n}]], \mathsf{Any})$
   ⑮ $\mathsf{extendYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{result} \rangle\rangle, \mathsf{grade}, \mathsf{string}, ? \rangle\rangle,$
   $[\langle \mathsf{re}, \mathsf{g} \rangle \mid \langle \mathsf{u}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{name} \rangle\rangle; \langle \mathsf{u}, \mathsf{c} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle, \mathsf{course} \rangle\rangle;$
   $\langle \mathsf{c}, \mathsf{co} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{code} \rangle\rangle; \langle \mathsf{c}, \mathsf{g} \rangle \leftarrow \langle\langle \langle\langle \mathsf{course}, \langle\langle \mathsf{root}, \mathsf{ug} \rangle\rangle \rangle\rangle, \mathsf{grade} \rangle\rangle;$
   $\mathsf{re} \leftarrow [\mathsf{generateId\ co\ n}]], \mathsf{Any})$

3. Conduct a breadth first search over the nodes n of Y, and for each node n which do not appear in Y' create transformations in a **shrinking** phase to remove n from Y:

(a) If n is of complex type, determine if there is a query q on the constructs of Y' in Y such that there is a one to one mapping between values returned by q and values associated to n. If there is, then the node n is deleted by applying a rule deleteYattaNode, with the function generateId used on the values returned by q to restore the values of n.

(b) If n is of simple type, determine if there is a query q on the constructs of Y' in Y such that the values returned by q are equal to the values associated with n. If there is, then the node n is deleted from Y by applying a rule deleteYattaNode, with q to restore the values of n.

In a similar manner to step(2), if the query returns partial results, then use contractYattaNode with queries q, Any, and if no query exists use queries Void, Any. In the fragment of the shrinking phase below, we are able to entirely reconstruct the ug node of S_1 from the data in S_{yg}.

$\mathsf{S}_1 \rightarrow \mathsf{S}_{\mathsf{yg}}$ (shrink phase)

⑯ deleteYattaNode(⟪⟪⟪⟪root, ug⟫, course⟫, grade, string, ?⟫,
[⟨c, g⟩ | ⟨re, co⟩ ← ⟪⟪⟪root, result⟫, code⟫; ⟨re, n⟩ ← ⟪⟪⟪root, result⟫, name⟫;
⟨re, g⟩ ← ⟪⟪⟪root, result⟫, grade⟫; c ← [generateId co n]])

⑰ deleteYattaNode(⟪⟪⟪⟪root, ug⟫, course⟫, year, integer, nk⟫,
[⟨c, y⟩ | ⟨re, co⟩ ← ⟪⟪⟪root, result⟫, code⟫; ⟨re, n⟩ ← ⟪⟪⟪root, result⟫, name⟫;
⟨re, y⟩ ← ⟪⟪⟪root, result⟫, year⟫; c ← [generateId co n]])

⑱ deleteYattaNode(⟪⟪⟪⟪root, ug⟫, course⟫, dept, string, nk⟫,
[⟨c, d⟩ | ⟨re, co⟩ ← ⟪⟪⟪root, result⟫, code⟫; ⟨re, n⟩ ← ⟪⟪⟪root, result⟫, name⟫;
⟨x, co⟩ ← ⟪⟪⟪root, course⟫, code⟫; ⟨x, d⟩ ← ⟪⟪⟪root, course⟫, dept⟫;
c ← [generateId co n]])

⑲ deleteYattaNode(⟪⟪⟪⟪root, ug⟫, course⟫, code, string, k⟫,
[⟨c, co⟩ | ⟨re, co⟩ ← ⟪⟪⟪root, result⟫, code⟫; ⟨re, n⟩ ← ⟪⟪⟪root, result⟫, name⟫;
c ← [generateId co n]])

⑳ deleteYattaNode(⟪⟪⟪root, ug⟫, course, set, +⟫,
[⟨u, c⟩ | ⟨r, re⟩ ← ⟪⟪root, result⟫; ⟨re, co⟩ ← ⟪⟪⟪root, result⟫, code⟫;
⟨re, n⟩ ← ⟪⟪⟪root, result⟫, name⟫; ⟨ug, n⟩ ← ⟪⟪⟪root, ug⟫, name⟫;
c ← [generateId co n]])

A complete set of transformations $\mathsf{S}_1 \rightarrow \mathsf{S}_{\mathsf{yg}}$ is available from the AutoMed web site, as is the similar pathway to translate $\mathsf{S}_2 \rightarrow \mathsf{S}_{\mathsf{yg}}$.

3 The AutoMed Repository for BAV Data Integration

The AutoMed repository has at its core the **reps** Java package which forms a platform for other components of the AutoMed to be implemented upon (such as the template language and schema matching system that we describe later). The current implementation of the reps API uses a RDBMS to provide persistent storage for data modelling language descriptions in the HDM, database schemas, and transformations between those schemas. The repository also provides some of the shared functionality that tools accessing the repository may require.

The AutoMed repository has two logical components, assessed via the reps API. The **model definitions repository** (**MDR**) allows for the description of how a data modelling language is represented as combinations of nodes, edges and constraints in the HDM. It is used by AutoMed 'experts' to configure AutoMed so that it can handle a particular data modelling language. The **schema transformation repository** (**STR**) allows for schemas to be defined in terms of the data modelling concepts in the MDR. It also allows for transformations to be specified between those schemas. Most AutoMed tools and users will be concerned with editing this repository, as new databases are added to the AutoMed repository, or those databases evolve [16]. The MDR and STR may be held in the same persistent storage, or separate persistent storage. The latter approach allows many AutoMed users to share a single MDR repository, which once correctly

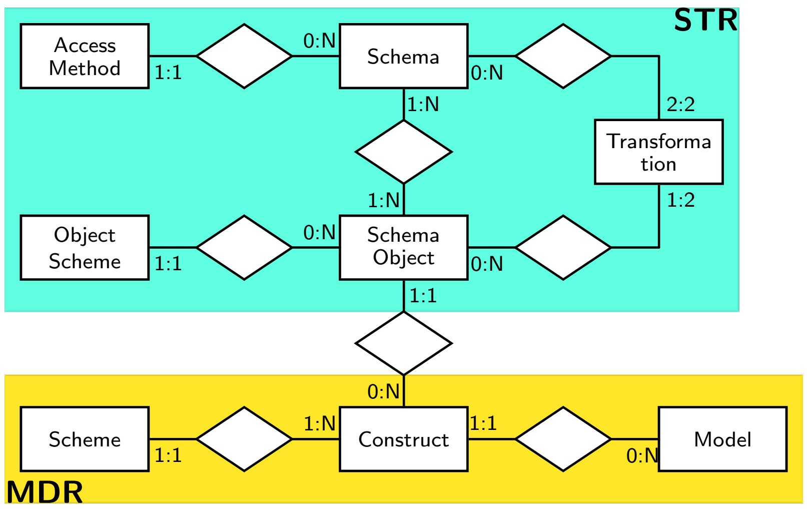


Figure 5: Repository Schema

configured, should not need to be update when integrating data sources that conform to a certain set of data modelling languages.

Figure 5 gives a overview of the key objects in the repository. The STR contains a set of descriptions of Schemas, each of which contains a set of SchemaObject instances, each of which must be based on a Construct instance that exists in the MDR, which in effect is the **type** of the construct. This Construct describes how the SchemaObject can be constructed in terms of strings and references to other schema objects, as well as the relationship of each of these strings and references to the HDM. Schemas are therefore readily translatable into HDM.

It should be noted that there is no direct restriction between Model and Schema, allowing each Schema to contain SchemaObjects from more that one data modelling language. This allows us to describe within AutoMed the mapping between different data modelling languages. Schemas may be related to each other using instances of Transformation. Each BAV transformation may only alter at most one SchemaObject in each Schema it is associated to.

We now describe in detail how the MDR may be programmed to describe a modelling language, and then describe some features of the STR that allow it manage large and evolving schema integrations.

3.1 Describing a Data Modelling Language in the MDR

In [15] we proposed a general technique for the modelling of any structured data modelling language in the HDM, which was used as the basis for the design of the MDR. The premise of this approach is that in any data modelling language, the various constructs of the language can be viewed as describing sets, bags and lists of values (which henceforth we refer to just as **lists** since lists can be used to model sets and bags), and constraints between these lists. In HDM we can represent a construct that takes a list of values by with a graph **node**, and when the construct models a data source, the values in the data source are called the **extent** of the node. As constraints describing relationships between extents of constructs are so common (for example instance of an ER relationship would imply instances of the ER entities it connects) we use **edge**s to represent constructs where the extent contains a list of tuples of values, and where every member of the tuple must appear in the extent of the nodes or edges which the edge connects.

The description of the construct that a schema object is a instance of is in effect a **type** definition of the schema object, and thus constrains which schemes are valid for a construct. In particular, when a SchemaObject is created, and its scheme details are entered into ObjectScheme, then they are checked against the type definition of the corresponding Construct held in Scheme.

Each construct must be classified as being one of six types: nodal, link, link-nodal, constraint, alternation and sequence. Each type restricts what may appear in Scheme, as described below.

A **nodal** construct represents a simple list of values. Exactly one element of the construct scheme must be identified as the name of the resulting HDM node and be of type node_name (i.e. a scheme would have to be supplied with the name of the node). Often a nodal construct has just this one scheme element, for example in an ER model, the construct for an entity would be defined by:
(nodal)er:entity ::= ⟪(node_name)hdm_node_name⟫
The brackets contain an indication of the type being used, so we read the above as stating that the entity construct in the er modelling language has a scheme that contains a single string, which is the name of the HDM node. Hence the schema objects representing entities student and course in Fig. 3(b) would have the schemes ⟪student⟫ and ⟪course⟫.

A **link** construct is one that can only be instantiated (*i.e.* a schema object of its type be constructed) by referring to other schema objects. One scheme element may be identified as the resulting HDM edge's name and at least two of the instance scheme's elements must refer to other schema objects that have resulting HDM nodes or edges. For example we may express ER binary relationships with mandatory names using the following construct scheme:
(link)er:relationship ::=
⟪(edge_name)name, (reference,2:2)er:entity, (constraint,2:2,nonkey)card⟫
The scheme has first a edge_name representing the name of the underlying HDM edge, followed by exactly two references to a schema object of construct type entity. The 'exactly two' is implied by the 2:2 after reference, and gives a cardinality constraint on the occurrences of the argument. Note that where no explicit cardinality constraint is given for any scheme position then 1:1 is implied. The constraint element card is used to denote the use of a constraint expression in the scheme, that may be used to lookup a macro that expresses the constraint in terms of the underlying HDM node and edges associated with this construct. The scheme instance for the relationship in Fig. 3(b) would be ⟪result, student, course, 0:N, 0:N⟫. The use of nonkey in the definition of the card element means that this element only has to appear in the definition of this schema object (as it appears in the first argument of a transformation) and need not appear in queries. Hence in a query one may also use the abbreviation ⟪result, student, course⟫ for the result relationship.

A **link-nodal** construct is a combination of a link and a node. It models a node type which cannot exist in isolation but requires another construct with which to be associated. The construct scheme must contain one string element for the name of the new HDM node, an optional name for the HDM edge name, and a mandatory reference to an existing construct. For example, an ER attribute can be defined by:
(link)er:attribute ::=
⟪(reference)er:entity, (node_name)new_node_name, (constraint,nonkey)card⟫
where the last element card corresponds to macros constraining the attribute instances. The attributes of student in Fig. 3(b) would then be ⟪student, name, notnull⟫, ⟪student, tutor, notnull⟫, ⟪student, ppt, null⟫, and ⟪student, level, notnull⟫.

A **constraint** construct has no extent, and must be associated with at least one other construct on which it places a constraint on its extent. For example, a subset relationship in a ER model places a restriction on two entities such that the extent of one is a subnet of the extent of another. This would be defined by:
(constraint)er:subset ::= ⟪(reference)er:entity, (reference)er:entity⟫
which would allow the scheme ⟪student, ug⟫ to exist. Again there is an associated macro for the constraint.

An **alternation** does not correspond to a modelling language construct (and hence no schema object can be created of this type), but is used to create a construct to be referenced by other constructs, where those other constructs may be related to alternative constructs. In Fig. 3(b) we have two non-key attributes on the relationship result. If we allow attributes on relationships, and do not want to have two different constructs for attributes on relationships and attributes on entities, then we could define an alternation:
(alternation)er:attribute_target ::= ⟪(reference)er:entity, (reference)er:relationship⟫

and redefine attribute as:
(link-nodal)er:attribute ::= ⟨⟨(reference)er:attribute_target,
(node_name)new_node_name, (constraint,nonkey)card)⟩⟩
Now the first position in an attribute scheme may reference either an entity or relationship. Attributes on student have the same schemes as before, and the attributes on result have the schemes ⟨⟨⟨⟨result, student, course⟩⟩, year, 1 : N⟩⟩ and ⟨⟨⟨⟨result, student, course⟩⟩, grade, 1 : N⟩⟩.

Like an alternation, a **sequence** does not have any corresponding modelling language construct, but is used to create a construct to be referenced by other constructs that need a repeatable sequence of construct types. For example, to provide n-ary ER relationships (rather than just binary relationships above), we could *not* write:
(link)er:relationship ::=
⟨⟨(string)name, (reference,2:N)er:entity, (constraint,2:N,nonkey)card⟩⟩
since this would allow a relationship to have a different number of entity associations from the number of card constraints on those associations. We could however construct an entity-constraint sequence and give that the required cardinality:
(sequence)er:entity_role ::= ⟨⟨(reference)er:entity, (string,nonkey)cardinality⟩⟩
(link)er:relationship ::= ⟨⟨(string)name, (reference,2:N)er:entity_role⟩⟩
Now we can use this definition to create our results relationship as the scheme ⟨⟨result, ⟨⟨student, 1 : N⟩⟩, ⟨⟨course, 1 : N⟩⟩⟩⟩, and can add additional entity and cardinality pairs as required.

The definitions for the other models we have used in this paper are much simpler than the ER model. For example, the YATTA model is defined by:
(alternation)yatta:yattanode_target ::=
⟨⟨(reference)yatta:yattanode, (reference)yatta:rootnode⟩⟩
(link-nodal)yatta:yattanode ::= ⟨⟨(reference)yatta:yattanode_target,
(node_name)new_node_name,(constraint,nonkey)type,(constraint,nonkey)card)⟩⟩
(nodal)yatta:rootnode ::= ⟨⟨(node_name)new_node_name, (constraint,nonkey)type)⟩⟩

The relational model is defined by the following rules (note how the primary key definition uses a label type to denote a string that is used just as a label in the modelling language, and does not correspond to any HDM construct):
(nodal)rel:table ::= ⟨⟨(node_name)new_node_name⟩⟩
(link-nodal)rel:column ::=
⟨⟨(reference)rel:table, (node_name)new_node_name,(constraint,nonkey)card)⟩⟩
(constraint)rel:pk ::=
⟨⟨(label)pk_name, (reference,1:N)rel:table, (reference,1:N)rel:column⟩⟩

3.2 Describing Schemas and Transformations in the STR

In a large data integration, there will be many schemas produced as intermediate steps in the process of mapping one data source to another. At first this would appear to make the BAV approach unworkable, since there are so many versions of schemas being kept. The AutoMed approach addresses this issue by distinguishing between **extensional** and **intensional** representations of schemas. Each data source will be represented in the AutoMed repository by describing its schema as a set of schema objects, which is the **extensional** representation of schemas. Extensional Schemas may be associated with an AccessMethod to describe the driver, username, password and URL of how a data source may be accessed. Transformations applied to the extensional schema produce new intensional schemas, for which the schema objects are not stored, but which can be derived when required by applying transformation rules in sequence to an extensional schema.

Furthermore, a special transformation called **ident** is introduced, which states that two schemas have the same logical set of schema objects, but that they are derived from distinct extensional schemas. For example, in Fig. 6, the schema S_{yg} is derived from S_1, and schema $\mathsf{S}_{\mathsf{yg}'}$ (identical to S_{yg}) is derived from S_2. The identity of these two schemas may then be stated by adding an ident transformation between them, which query processing can use to retrieve data from alternative

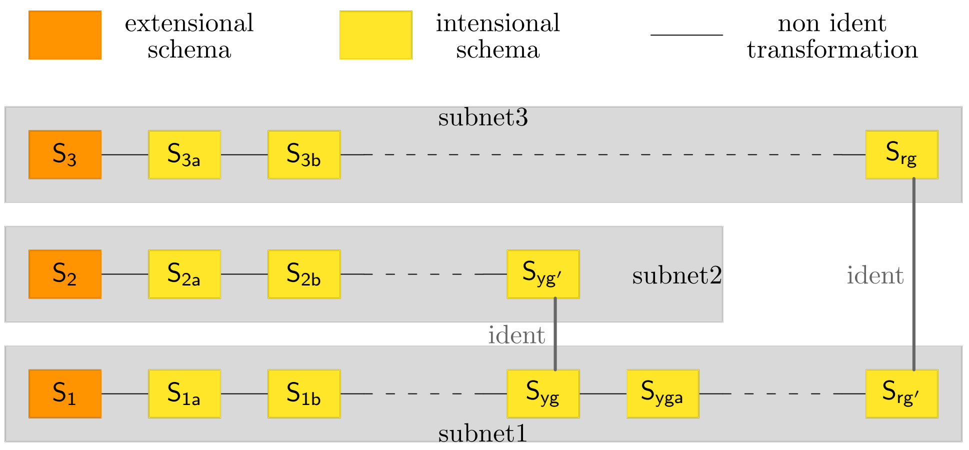


Figure 6: Overview of Schemas held in AutoMed

data sources. The YATTA model S_{yg} is then translated to its relational equivalent $S_{rg'}$ which can then be idented with the corresponding S_{rg} derived from S_3

The set of schemas connected together by transformations other than ident is called a **subnet**. By the nature of this arrangement, each subnet can exist independently of other subnets. This means that a subnet can be created, edited, connected to other subnets via ident transformations, and deleted all without changing anything in another subnet. It also allows for the schema evolution techniques in [16] to be supported. If say S_1 has been found to evolve to $S_{1'}$, then a new subnet 4 can be created for $S_{1'}$, with transformations to describe $S_{1'} \rightarrow S_1$. Then S_1 may have its AccessMethod removed, and query processing will be directed to the new version of the data source.

4 Inter Model Transformations

With there being a wide range of data modelling languages in use, a common task in data integration methodologies is to implement a **wrapper** to translate all the component schemas into a common data model. In AutoMed, this wrapping step can be formalised within the data integration methodology if the data modelling languages used for the component schemas are described in the MDR. In addition, this translation process may occur in the middle of the data integration (as illustrated by pathway $S_{yg} \rightarrow S_{rg'}$ in Fig. 6), allowing a mixture of data modelling languages to be used in a large integration.

To illustrate this process, consider the following rules that map relational constructs to YATTA constructs:

1. a YATTA complex node n of set type with '*' cardinality is created under the root of the YATTA model for each table, and

2. a YATTA atomic node is created under this complex node for each column of the table, where the cardinality of the column is '?' if it is a nullable column in the relational model, and 'k' if it is a primary key column in the relational model.

For S_{rg}, this would generate a pathway that starts:
$S_{rg} \rightarrow S_{yg'}$

㉑ $\mathsf{addYattaNode}(\langle\langle \mathsf{root}, \mathsf{student}, \mathsf{set}, * \rangle\rangle,$
$[\langle \mathsf{r}, \mathsf{s} \rangle \mid \langle \mathsf{n} \rangle \leftarrow \langle\langle \mathsf{student} \rangle\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\langle \mathsf{root} \rangle\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId\ r\ n}]])$
㉒ $\mathsf{addYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{name}, \mathsf{string}, \mathsf{k} \rangle\rangle,$
$[\langle \mathsf{s}, \mathsf{n} \rangle \mid \langle \mathsf{n} \rangle \leftarrow \langle\langle \mathsf{student} \rangle\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\langle \mathsf{root} \rangle\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId\ r\ n}]])$
㉓ $\mathsf{addYattaNode}(\langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{tutor}, \mathsf{string}, \mathsf{nk} \rangle\rangle,$
$[\langle \mathsf{s}, \mathsf{t} \rangle \mid \langle \mathsf{n}, \mathsf{t} \rangle \leftarrow \langle\langle \mathsf{student}, \mathsf{tutor} \rangle\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\langle \mathsf{root} \rangle\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId\ r\ n}]])$
㉔ $\mathsf{deleteColumn}(\langle\langle \mathsf{student}, \mathsf{tutor} \rangle\rangle, [\langle \mathsf{n}, \mathsf{t} \rangle \mid \langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{name} \rangle\rangle;$
$\langle \mathsf{s}, \mathsf{t} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{tutor} \rangle\rangle])$
㉕ $\mathsf{deleteColumn}(\langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle, [\langle \mathsf{n}, \mathsf{n} \rangle \mid \langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{name} \rangle\rangle])$
㉖ $\mathsf{deleteTable}(\langle\langle \mathsf{student} \rangle\rangle, [\langle \mathsf{n} \rangle \mid \langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\langle \langle\langle \mathsf{root}, \mathsf{student} \rangle\rangle, \mathsf{name} \rangle\rangle])$

Note that this pathway may be used in reverse to generate $\mathsf{S_{yg}} \rightarrow \mathsf{S_{rg'}}$ as shown in Figure 6.

In general, translating a schema from a source to a target modelling langauge involves using the MDR definitions to convert constructs in the source and target langauge to HDM, analysing the constraint information, and building an association between the two. A common aspect of this analysis is that constraint information will involve the cardinality constraints on edges in the HDM, which can be represented by just two constraint templates:

1. $N \rhd E$ states that there must be at least one tuple in the edge E for each value in the extent of node N to which is is associated.

2. $N \lhd E$ states that there must not be more than one tuple in the edge E for each value in the extent of node N.

Combining these rules gives the following cardinality constriaints on the node N's role in the edge E:

None	$\rightarrow$	N has $\mathsf{0{:}N}$ occurances in E
$N \rhd E$	$\rightarrow$	N has $\mathsf{1{:}N}$ occurances in E
$N \lhd E$	$\rightarrow$	N has $\mathsf{0{:}1}$ occurances in E
$N \rhd E \wedge N \lhd E$	$\rightarrow$	N has $\mathsf{1{:}1}$ occurances in E

Now we are in a position to more formally analyse the relational to YATTA mapping. In the relational model, a column L of table T is represented by the scheme $\langle\langle T, L, C \rangle\rangle$, which in Section 3.1 we modelled as a $\mathsf{link\text{-}nodal}$ construct that references an existing nodal construct which has HDM node E_r, and has a new edge E and node N to represent the column's association with the table. Now the macros for C can be expressed over those HDM constructs: the macro $\mathsf{notnull}{=}N \rhd E \wedge E_r \rhd E \wedge E_r \lhd E$, and the macro $\mathsf{null}{=}N \rhd E \wedge E_r \lhd E$.

In the YATTA model, a node N is represented by the scheme $\langle\langle P, N, T, C \rangle\rangle$, which is again a link-nodal construct, that references a parent node P that may be either another YATTA node, or a root node. In the latter case P will be nodal, and in the former case, if P is attached to the root, then that association can be ignored since it is made to the constant value &0 stored in the root node. If in addition N does not appear as the parent of any other YATTA node, we have a match between the YATTA node and the concept of an column in the relational model. In particular, we find that the YATTA nk constraint C corresponds to $N \rhd E \wedge E_r \rhd E \wedge E_r \lhd E$, and the $\mathsf{?}$ constraint corresponds to $N \rhd E \wedge E_r \lhd E$.

5 Template Transformations

Schema integration in the AutoMed framework frequently relies on the reuse of specific sequences of primitive transformations. These sequences are called **composite** transformations and resemble well-known equivalences between schemas [7, 14]. For example, the equivalence between a relation with a mandatory attribute and a relation with specializations for each instance of the mandatory attribute is found in transformations ① –⑦ in Section 2, where the $\langle\langle \mathsf{student}, \mathsf{level} \rangle\rangle$ attribute is used to generate new relations $\langle\langle \mathsf{ug} \rangle\rangle$ and $\langle\langle \mathsf{pg} \rangle\rangle$.

To describe such equivalences between schemas, we have created a package around the AutoMed reps package that enables the definition of **template** transformations [20, 8] which are schema and data independent and can therefore be reused in different situations. Based on this framework, a template transformation is a parameterised definition of a composite transformation. These parameters are instantiated in each template transformation execution based on the schema that the template is performed upon, the underlying data and the desired outcome. Thus, a template transformation can be applied on any schema by appropriately setting its parameters. In our example, the parameters of the template transformation that decomposes a relation to its specializations are: (a) the schema that the template transformation is going to be performed upon, (b) the existing relation, (c) its mandatory attribute, (d) the specialisation relations that are going to be added to the schema and (e) the instances of the mandatory attribute that correspond to the specializations. These are specified as follows:

```
INPUTS();
  OBJECT parentTable=askForObject("Existing parent relation",table);
  OBJECT mandatoryColumn=askForObject("Mandatory attribute",column);
  OBJECT parentPrimaryKey=askForObject("Primary key of parent relation",column);
  NAMELIST specializationTableNames=askForNameList("Names of specializations");
  NAMELIST descriptiveInstances=askForNameList("Values of the attribute ...",
      SIZEOF(specializationTableNames));
```

Note that each parameter has a description and a *type* associated with it. The example template transformation has been defined in the relational data model, where relations are of type Table and attributes of type Column. Also, note that the *initial schema* parameter is always implicitly defined in every template transformation, so it does not appear in the INPUTS.

Except from the parameters, the number of statements that the template consists of must be defined. In our example, since in general there are n different specialisation relations to create, we require to put statements in a loop which iterate over the LIST we have specified in the INPUTS:

```
  FOREACH();
    NAME specializationTableName=IN(specializationTableNames)
    NAME descriptiveInstance=IN(descriptiveInstances);
    OBJECT parentcolumn = VARIES_WITH(mandatoryColumn);
    OBJECT parent = VARIES_WITH(parentTable);
    OBJECT primaryKey = VARIES_WITH(parentPrimaryKey);
    DO();
```

This loop may contain the instructions to create each specialisation relation. For example, the transformations ① and ④ that create the specialisation Table constructs are produced by the following template definition:

```
  FUNCTION tableExtent=DEFINE_FUNCTION("[ {x} |
    {x,y} <- @parentcolumn?scheme; y='@descriptiveInstance']");
  OBJECT newSpecialization=ADD(CONSTRUCT.IS(table),
    SCHEME.IS(new Object[]{my(specializationTableName)}),
    FUNCTION.IS(tableExtent));
```

Similar definitions can create the Column transformations ② and ⑤ and the PK transformations ③ and ⑥ .

Note that the rest of the transformations ⑧ -⑨ in $S_3 \rightarrow S_{rg}$ can be produced by another template transformation that removes an attribute from a relation and moves it down to its specialisation relations.

Our template transformation framework is not only schema and data independent, it is additionally model-independent. Template transformations can be written for any modelling language by specifying the correct types for the template parameters. For example, a template transformation can be written for YATTA schemas, *e.g.* flattening YATTA subtrees by moving complex nodes one level up in the YATTA hierarchy, as illustrated in the pathway $S_1 \rightarrow S_{yg}$ in Section 2.1.

6 Schema Matching

In all the examples seen so far, an expert user specifies the primitive transformations to be applied on the available schemas and integrate the underlying data sources. The key issue, as in every manual data integration framework, is the identification of the existing semantic relationships between the schema objects [6]. Based on these relationships the appropriate primitive or template transformations can be performed.

The process of discovering semantic relationships between schema objects is called **schema matching**. Most of the existing methodologies are focused on discovering equivalence relationships between schema objects [3, 10, 12], or **direct matches**, but in many cases more expressive relationships exist between schema objects, which yield **indirect matches** [11].

In our framework, we define five types of semantic relationships between schema objects based on the comparison of their *intentional domains*, i.e. the sets of real-world entities represented by the schema objects. Our relationships are similar to the ones described in [7] but our definitions differ:

1. **equivalence**: Two schema objects A and B are equivalent, $\mathsf{A} = \mathsf{B}$, iff

   $Dom_{int}(\mathsf{A}) = Dom_{int}(\mathsf{B})$

2. **subsumption**: Schema object A subsumes schema object B, $\mathsf{B} \subset \mathsf{A}$, iff

   $Dom_{int}(\mathsf{B}) \subset Dom_{int}(\mathsf{A})$

3. **overlapness**: Two schema objects A and B are overlapping, $\mathsf{A} \simeq \mathsf{B}$, iff

   $Dom_{int}(\mathsf{A}) \cap Dom_{int}(\mathsf{B}) \neq \emptyset$,
   $\exists \mathsf{C} : Dom_{int}(\mathsf{A}) \cap Dom_{int}(\mathsf{B}) = Dom_{int}(\mathsf{C})$

4. **disjointness**: Two schema objects A and B are disjoint, $\mathsf{A} \not\simeq \mathsf{B}$, iff

   $Dom_{int}(\mathsf{A}) \cap Dom_{int}(\mathsf{B}) = \emptyset$,
   $\exists \mathsf{C} : Dom_{int}(\mathsf{A}) \cup Dom_{int}(\mathsf{B}) \subseteq Dom_{int}(\mathsf{C})$

5. **incompatibility**: Two schema objects A and B are incompatible, $\mathsf{A} \neq \mathsf{B}$, iff

   $Dom_{int}(\mathsf{A}) \cap Dom_{int}(\mathsf{B}) = \emptyset$,
   $\neg\exists \mathsf{C} : Dom_{int}(\mathsf{A}) \cup Dom_{int}(\mathsf{B}) \subseteq Dom_{int}(\mathsf{C})$

It is important to notice that schema object C in the definition of overlapness and disjointness may or may not exist in the existing schemas. The notation $\exists \mathsf{C}$: *condition* means that there is a real-world concept that can be represented by an existing or non-existing schema object C that satisfies the *condition*. The notation $\neg\exists \mathsf{C}$: *condition* in the definition of incompatibility means that there is no real-world concept that would be represented by a schema object C to satisfy the specified *condition*.

In our example schemas in Figs. 1 and 2, an indirect match exists between the **disjoint** student nodes in $\mathsf{S_1}$ and $\mathsf{S_2}$. These nodes should be renamed in order to be distinguished, therefore transformation ⑩ renames student in $\mathsf{S_1}$ to ug and an equivalent transformation renames student in $\mathsf{S_2}$ to pg. These are **equivalent** to the ug and pg nodes in $\mathsf{S_{yg}}$ respectively, and can therefore be unified using ident transformations. Other indirect matches exist between the **overlapping** course nodes and between ug,pg and the **subsumming** student node.

In our automatic schema matching approach to automatically discover these semantic relationships, a bidirectional comparison of schema objects is performed, which has been motivated by the fact that a bidirectional comparison of the schema objects' intentional domains can be used to identify equivalence, subsumption and overlapness relationships. This is depicted by the following formula:

$d(X,Y) = \frac{|Dom_{int}(X) \cap Dom_{int}(Y)|}{|Dom_{int}(X)|}$,

where X, Y are schema objects and $|Z|$ defines the number of entities in set Z. This formula gives $d(X,Y) = d(Y,X) = 1$ when X, Y are equivalent, $d(X,Y) = 1$ and $0 < d(Y,X) < 1$ when Y

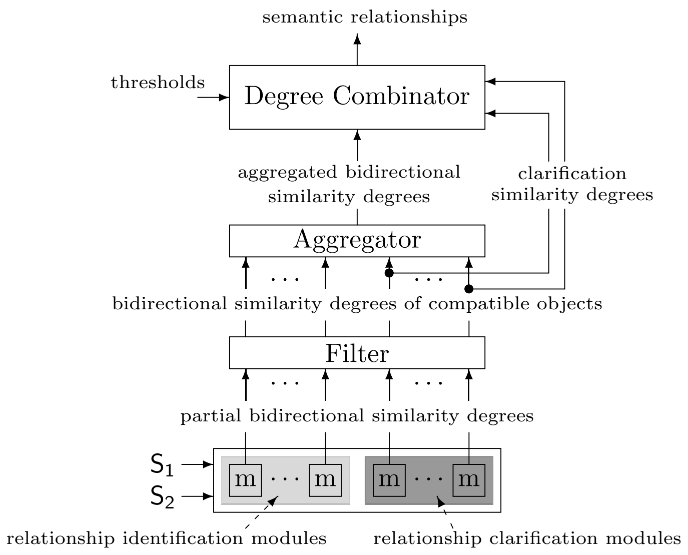

semantic relationships
thresholds
Degree Combinator
aggregated bidirectional similarity degrees
clarification similarity degrees
Aggregator
bidirectional similarity degrees of compatible objects
Filter
partial bidirectional similarity degrees
S_1
S_2
m
relationship identification modules
relationship clarification modules

Figure 7: Architecture

subsumes X, and $0 < d(X,Y) < 1, 0 < d(Y,X) < 1$ when X, Y are overlapping. The problems with this approach are that the disjointness and incompatibility relationships cannot be distinguished, and that the *bidirectional similarity degrees* $d(X,Y), d(Y,X)$ cannot be automatically computed since a comparison of the schema objects' real-world entities is required.

We attempt to simulate the behaviour of the above formula by examining the schema objects instances and their metadata. The equivalence, subsumption and overlapness relationships can still be discovered as explained previously. Now, however, the similarity degrees are fuzzier, e.g. $d(X,Y)$ and $d(Y,X)$ are unlikely to have values equal to 1 when X and Y are equivalent, but they will be above an equivalence threshold. Disjointness can also be discovered since disjoint schema objects will exhibit similarity in their instances and metadata, arising from their relationship with the same *super* schema object. Thus, disjoint pairs of schema objects will have higher similarity degrees than incompatible pairs.

This relationship discovery process is implemented by the architecture in Fig. 7, which consists of several modules that exploit different types of information to compute bidirectional similarity degrees of schema objects. Our currently implemented modules compare schema object names, instances, statistical data over the instances, data types, value ranges and lengths. There are two types of modules: **relationship identification** modules attempt to discover compatible pairs of schema objects, and **relationship clarification** modules attempt to specify the type of the semantic relationship in each compatible pair.

Initially in the schema matching process, the bidirectional similarity degrees produced by the modules are combined by the Filter, using the average aggregation strategy, to separate the compatible from the incompatible pairs of schema objects. Then, the Aggregator component combines the similarity degrees of the compatible schema objects using the product aggregation strategy and indicates their semantic relationships. The output of the Aggregator becomes the input of the Degree Combinator, which based on (a) the relationship clarification modules, (b) the fuzzy thresholds and (c) the previous discussion on the values of the similarity degrees, it outputs the discovered semantic relationships. The user is then able to validate or reject these relationships and proceed to the data integration process.

More details about the implemented tool and an evaluation of it can be found in [21].

7 Conclusions

This paper details the implementation of the BAV produced by the AutoMed project, and illustrates how the AutoMed system may be used to model a number of data modelling languages, and in particular introduces the YATTA model as a method to handle semistructured text files in the BAV approach. The paper also deals with practical issues concerned with data integration, by providing a template system for defining common patterns of transformations, and a schema matching system to help automate the generation of transformations. It also introduces the notion of subnetworks into the BAV approach, which allows complex and large integrations to divided into clearly identifiable independent units.

The AutoMed approach has the unique property that it does not insist that an entire data integration system be conducted in a single data modelling language. This gives the flexibility of integrating different domains in a modelling language suited to each domain, and then using inter-model transformations to connect between the domains.

A complete version of the example presented in this paper, together with technical reports, API document and the latest release of the AutoMed software may be downloaded from http://www.doc.ic.ac.uk/automed.

References

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

[2] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! *SIGMOD Record*, 27(2):177–188, 1998.

[3] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map ontologies on the Semantic Web. In *Proceedings of the World-Wide Web Conference (WWW-02)*, pages 662–673, 2002.

[4] E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL Queries and Migrating Data in the AutoMed toolkit. Technical Report No. 20, AutoMed, 2003.

[5] E. Jasper, N. Tong, P.J. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. Technical Report No. 16, Version 3, AutoMed, 2003.

[6] V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: a context-based approach. *VLDB Journal*, 5(4):276–304, 1996.

[7] J.A. Larson, S.B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, April 1989.

[8] C. Lazanitis. Template transformations in automed. Technical report, AutoMed Project, 2003.

[9] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.

[10] W.-S. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33:49–84, 2000.

[11] L.Xu and D.W. Embley. Discovering direct and indirect matches for schema elements. In *8th International Conference on Database Systems for Advanced Applications (DASFAA '03), Kyoto, Japan, March 26–28, 2003*, pages 39–46, 2003.

[12] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proc. 27th VLDB Conference*, pages 49–58, 2001.

[13] J. Madhavan and A.Y. Halevy. Composing mappings among data sources. In *Proc. VLDB'03*, pages 572–583, 2003.

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

[15] P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer-Verlag, 1999.

[16] P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Advanced Information Systems Engineering, 14th International Conference CAiSE2002*, volume 2348 of *LNCS*, pages 484–499. Springer-Verlag, 2002.

[17] P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*. IEEE, 2003.

[18] A. Poulovassilis. The automed intermediate query language. Technical Report No. 2, AutoMed, 2001.

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

[20] N. Rizopoulos. A database integration tool. Technical report, Imperial College, 2001.

[21] N. Rizopoulos. Discovery of semantic relationships between schema elements. Technical Report No.23, AutoMed, 2003.