In: Proceedings of CAiSE2004
Springer-Verlag LNCS

AutoMed: A BAV Data Integration System for Heterogeneous Data Sources

Michael Boyd, Sasivimol Kittivoravitkul, Charalambos Lazanitis,
Peter McBrien and Nikos Rizopoulos
email: {mboyd,sk297,cl201,pjm,nr600}@doc.ic.ac.uk
http://www.doc.ic.ac.uk/automed/

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

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 in additional provides a method to support a wide range of data modelling languages, and describes transformations between those data modelling languages. This paper documents how BAV has been implemented in the AutoMed repository, and how several practical problems in data integration between heterogeneous data sources have been solved. We illustrate the implementation with examples in the relational, ER, and semi-structured data models.

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 an 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

[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/`.

model 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 transformation template 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, the mapping between schemas can be described as a **pathway** of **primitive transformation** steps applied in sequence. For example, suppose we want to transform the relational schema $\mathsf{S_3}$ in Fig. 2(a) into the the global schema $\mathsf{S_{rg}}$ in Fig. 3(a). Using the approach described in [17], we model each table as a scheme $\langle\langle\mathsf{table\_name}\rangle\rangle$, and each column as a scheme $\langle\langle\mathsf{table\_name,column\_name,cardinality}\rangle\rangle$. When used in queries the cardinality of the column need not be given.

Thus, the level column in $\mathsf{S_3}$ has the scheme $\langle\langle\mathsf{student,level,notnull}\rangle\rangle$, and may be used to divide the table $\langle\langle\mathsf{student}\rangle\rangle$ into those that belong to the undergraduate table $\langle\langle\mathsf{ug}\rangle\rangle$, created by transformations ①–③, and those the belong to the postgraduate table $\langle\langle\mathsf{pg}\rangle\rangle$ by transformations ④–⑥. The IQL [5] query in ① finds in the **generator** $\langle\mathsf{x,y}\rangle \leftarrow \langle\langle\mathsf{student,level}\rangle\rangle$ the tuples $\langle\text{'Mary'},\text{'ug'}\rangle, \langle\text{'John'},\text{'pg'}\rangle, \ldots$ and then the **filter** $\mathsf{y} = \text{'ug'}$ 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.

$\mathsf{S_3} \rightarrow \mathsf{S_{rg}}$

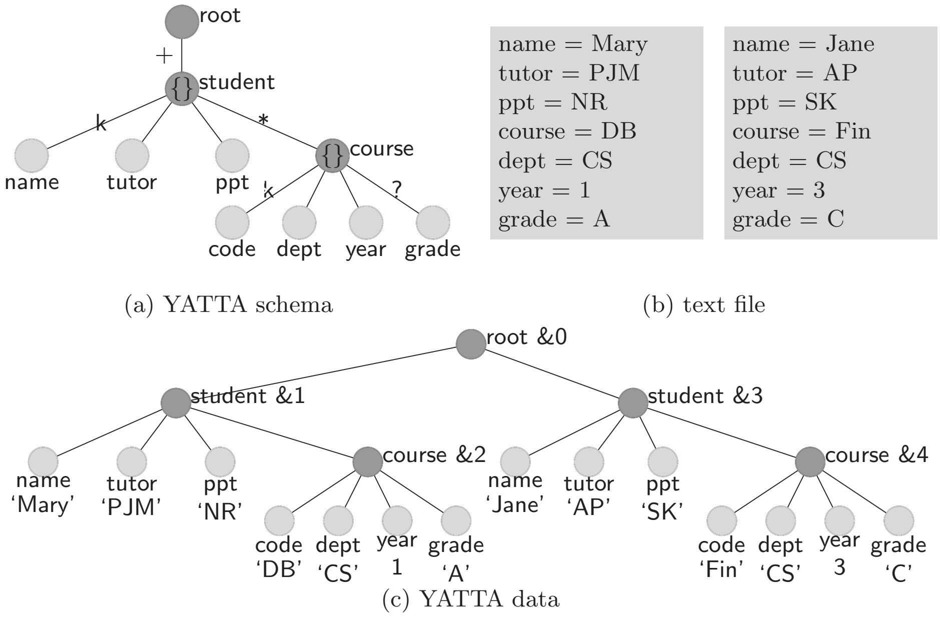
① $\mathsf{addTable}(\langle\langle\mathsf{ug}\rangle\rangle, [\langle\mathsf{x}\rangle \mid \langle\mathsf{x,y}\rangle \leftarrow \langle\langle\mathsf{student,level}\rangle\rangle; \mathsf{y} = \text{'ug'}])$

② $\mathsf{addColumn}(\langle\langle\mathsf{ug,name,notnull}\rangle\rangle,$
$\quad [\langle\mathsf{x,y}\rangle \mid \langle\mathsf{x,y}\rangle \leftarrow \langle\langle\mathsf{student,name}\rangle\rangle; \langle\mathsf{x,z}\rangle \leftarrow \langle\langle\mathsf{student,level}\rangle\rangle; \mathsf{z} = \text{'ug'}])$

③ $\mathsf{addPK}(\langle\langle\mathsf{ug\_pk,ug},\langle\langle\mathsf{ug,name}\rangle\rangle\rangle\rangle)$

④ $\mathsf{addTable}(\langle\!\langle \mathsf{pg} \rangle\!\rangle, [\langle \mathsf{x} \rangle \mid \langle \mathsf{x}, \mathsf{y} \rangle \leftarrow \langle\!\langle \mathsf{student}, \mathsf{level} \rangle\!\rangle; \mathsf{y} = \text{‘}\mathsf{pg}\text{’}])$
⑤ $\mathsf{addColumn}(\langle\!\langle \mathsf{pg}, \mathsf{name}, \mathsf{notnull} \rangle\!\rangle,$
$\quad [\langle \mathsf{x}, \mathsf{y} \rangle \mid \langle \mathsf{x}, \mathsf{y} \rangle \leftarrow \langle\!\langle \mathsf{student}, \mathsf{name} \rangle\!\rangle; \langle \mathsf{x}, \mathsf{z} \rangle \leftarrow \langle\!\langle \mathsf{student}, \mathsf{level} \rangle\!\rangle; \mathsf{z} = \text{‘}\mathsf{pg}\text{’}])$
⑥ $\mathsf{addPK}(\langle\!\langle \mathsf{pg\_pk}, \mathsf{pg}, \langle\!\langle \mathsf{pg}, \mathsf{name} \rangle\!\rangle \rangle\!\rangle)$
⑦ $\mathsf{deleteColumn}(\langle\!\langle \mathsf{student}, \mathsf{level}, \mathsf{notnull} \rangle\!\rangle,$
$\quad [\langle \mathsf{x}, \mathsf{y} \rangle \mid \langle \mathsf{x} \rangle \leftarrow \langle\!\langle \mathsf{ug} \rangle\!\rangle; \mathsf{y} = \text{‘}\mathsf{ug}\text{’}] \mathbin{++} [\langle \mathsf{x}, \mathsf{y} \rangle \mid \langle \mathsf{x} \rangle \leftarrow \langle\!\langle \mathsf{pg} \rangle\!\rangle; \mathsf{y} = \text{‘}\mathsf{pg}\text{’}])$
⑧ $\mathsf{addColumn}(\langle\!\langle \mathsf{ug}, \mathsf{ppt}, \mathsf{notnull} \rangle\!\rangle,$
$\quad [\langle \mathsf{x}, \mathsf{y} \rangle \mid \langle \mathsf{x} \rangle \leftarrow \langle\!\langle \mathsf{ug} \rangle\!\rangle; \langle \mathsf{x} \rangle \leftarrow \langle\!\langle \mathsf{student} \rangle\!\rangle; \langle \mathsf{x}, \mathsf{y} \rangle \leftarrow \langle\!\langle \mathsf{student}, \mathsf{ppt} \rangle\!\rangle]$
⑨ $\mathsf{deleteColumn}(\langle\!\langle \mathsf{student}, \mathsf{ppt}, \mathsf{null} \rangle\!\rangle, [\langle \mathsf{x}, \mathsf{y} \rangle \mid \langle \mathsf{x} \rangle \leftarrow \langle\!\langle \mathsf{student} \rangle\!\rangle; \langle \mathsf{x}, \mathsf{y} \rangle \leftarrow \langle\!\langle \mathsf{ug}, \mathsf{ppt} \rangle\!\rangle])$

Note that the transformation $\mathsf{S_{rg}} \rightarrow \mathsf{S_3}$ is automatically derivable from $\mathsf{S_3} \rightarrow \mathsf{S_{rg}}$ by taking the inverse steps ❾–❶, formed by replacing delete for add, and replacing add for delete, in transformations ①–⑨.



(a) YATTA schema

(b) text file

(c) YATTA data

Fig. 1. $\mathsf{S_1}$: Semi-structured text file of undergraduates

2.1 Handling Semi-Structured Data

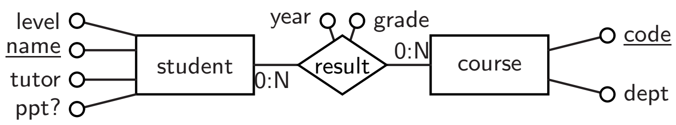
The **YATTA** (YAT for transformation-based approach) is a variation of the YAT model [2] to support the handling of semistructured data in AutoMed. YATTA provides two levels of abstraction: the **schema level** where the structure of data is defined, and the **data level** where actual data is presented. Fig. 1(b) shows

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

course	
code	dept
DB	CS
Fin	CS
Geo	Maths

result			
code	name	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

Fig. 2. S_3: relational database covering all students

student	
name	tutor
Mary	PJM
John	AP
Jane	AP
Fred	PJM

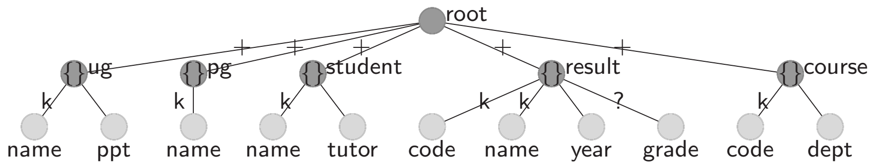
ug	
name	ppt
Mary	NR
Jane	SK

pg
name
Fred
John

course	
code	dept
DB	CS
Fin	CS
Geo	Maths

result			
code	name	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

Fig. 3. Global Schema

a semistructured text file, containing data about the undergraduate students in Fig. 3(a), which together with a similar text file (not shown) of postgraduates, we wish to integrate with the relational schema S_{rg}. Fig. 1(a) gives a schema level YATTA model S_1 of the undergraduate text file (a similar model S_2 exists for postgraduates), and Fig 1(c) gives a data level YATTA model for that file.

In the YATTA model, schemas and data are rooted labelled trees. In a **YATTA schema**, each node is labelled by a tuple $\langle$*name, type*$\rangle$, 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, integer, *etc.*, or compound *i.e.* list (marked '[]'), set (marked '{}'), or bag (marked '$\langle\rangle$'). Each node in a **YATTA data tree** is labelled by a triple $\langle$*name, type, value*$\rangle$, 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 in AutoMed using two construct types. The root of the tree 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 $\mathsf{S_1}$ with $\mathsf{S_{rg}}$, we need to transform $\mathsf{S_1}$ to have the same structure as $\mathsf{S_{rg}}$. Section 4 shows how derive a YATTA schema $\mathsf{S_{yg}}$ shown in Fig. 3(b) that is equivalent to $\mathsf{S_{rg}}$. Now the task is to transform $\mathsf{S_1}$ and $\mathsf{S_2}$ to $\mathsf{S_{yg}}$. To determine the pathway from one YATTA schema Y to another one Y' a three phase methodology is used.

The **conform phase** uses rename transformations to **conform** the schemas. In $\mathsf{S_1}$, the student node matches in semantics and extent the ug node in $\mathsf{S_{yg}}$, implying:

⑩ renameYattaNode($\langle\langle$root, student, set, $+\rangle\rangle$, $\langle\langle$root, ug, set, $+\rangle\rangle$)

The **growth phase** conducts a search over the nodes n' of Y', and for each n' not found in Y applies transformations to add n' to Y:

1. 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 addYattaNode, with the special function **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.
2. 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 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 we instead use $\mathsf{extendYattaNode}$, with the queries set to q, Any (where Any indicates the source places no upper bound on the extent [18]), and if no query can be determined, then we 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 $\langle \&0, \&1\rangle$ and $\langle \&0, \&3\rangle$ from $\langle \mathsf{r}, \mathsf{u}\rangle \leftarrow \langle\!\langle \mathsf{root}, \mathsf{ug}\rangle\!\rangle$, then $\langle \&1, \text{'Mary'}\rangle$ and $\langle \&3, \text{'Jane'}\rangle$ from $\langle \mathsf{u}, \mathsf{n}\rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{ug}\rangle\!\rangle, \mathsf{name}\rangle\!\rangle$, then $\langle \&1, \&2\rangle$ and $\langle \&3, \&4\rangle$ from $\langle \mathsf{u}, \mathsf{c}\rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{ug}\rangle\!\rangle, \mathsf{course}\rangle\!\rangle$, and finally $\langle \&2, \text{'DB'}\rangle$ and $\langle \&4, \text{'Fin'}\rangle$ 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 lists $[\mathsf{Mary}, \mathsf{DB}]$ and $[\mathsf{Jane}, \mathsf{Fin}]$, and generate $\&5$ and $\&6$ as new identifiers for result. Note that the same identifiers will now be created in ⑫–⑮.

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

Once the growth phase is completed, an analogous **shrinking phase** conducts a search over the nodes n of Y, and for each n not in Y', creates either a $\mathsf{deleteYattaNode}$ or a $\mathsf{contractYattaNode}$ transformation to remove n from Y.

3 The AutoMed Repository for BAV Data Integration

The AutoMed meta data $\mathsf{repository}$ forms a platform for other components of the AutoMed Software Architecture (Fig. 4) to be implemented upon. When a data source is wrapped, a definition of the schema for that data source is added to the repository. The schema matching tool may then be used to identify related objects in various data sources (accessing the query processor [5] to retrieve data from schema objects), and the template transformation tool used to generate transformations between the data sources. A GUI is supplied with AutoMed for

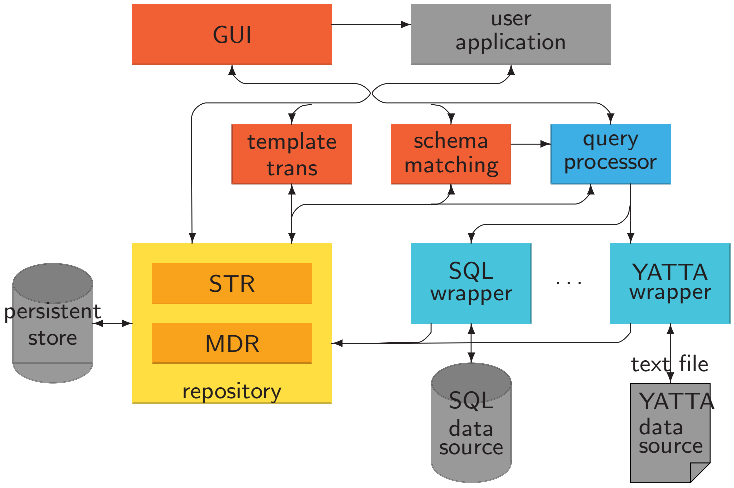


Fig. 4. AutoMed Software Architecture

these components, and it is possible for a user application to be configured to run from this GUI, and use the APIs of the various components. For example, work is in progress on using the repository in data warehousing [4].

The repository has two logical components. The **model definitions repository** (**MDR**) defines how a data modelling language is represented as combinations of nodes, edges and constraints in the **hypergraph data model** (**HDM**) [19]. It is used by AutoMed 'experts' to configure AutoMed so that it can handle a particular data modelling language. The **schema transformation repository** (**STR**) defines schemas in terms of the data modelling concepts in the MDR, and transformations to be specified between those schemas. Most tools and users will be concerned with editing this repository, as new databases are added to the AutoMed repository. The MDR and STR may be held in the same or separate persistent storage. The latter approach allows many AutoMed users to share a single MDR repository, which once configured, need not be updated when integrating data sources that conform to a certain set of data modelling languages.

Fig. 5 gives an 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. This Construct describes how the SchemaObject can be constructed in terms of strings and references to other schema objects, and the relationship of the construct to the HDM. Schemas are therefore readily translatable into HDM, and hence we have a common underlying representation of all the data modelling languages handled in AutoMed. Note that each Schema may contain SchemaObjects from more that one data modelling language. This allows AutoMed to describe the mapping between different data modelling languages. Schemas may be related to each other using instances of Transformation.

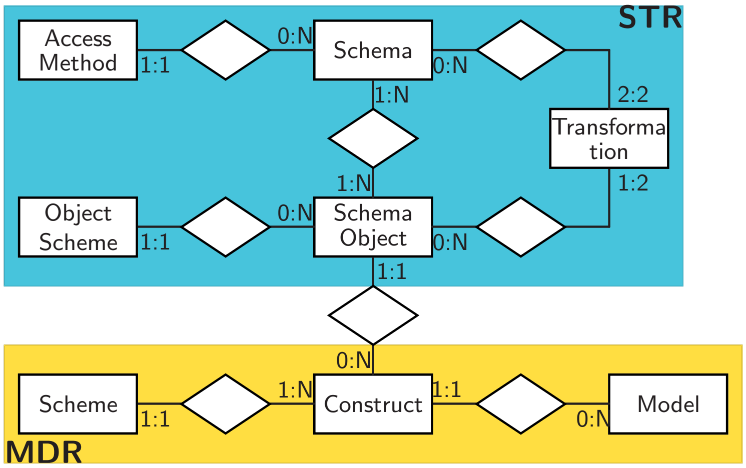


Fig. 5. Repository Schema

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 to manage large and evolving schema integrations.

3.1 Describing a Data Modelling Language in the MDR

In [15] a general technique was proposed 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 description of the Construct is made by defining elements of its scheme in the Scheme class, each instance of which must refer to a HDM node, edge or constraint, or be a textual label to use on the construct. Associating a SchemaObject to a Construct thus gives it a **type** definition. When a SchemaObject is created, and its scheme details are entered into ObjectScheme, they are checked against the corresponding Scheme of its Construct. Each modelling language construct must be classified as being one of four types: nodal, link, link-nodal, and constraint.

A **nodal** construct represents a simple list of values. Exactly one element of the construct scheme must be identified as the name of the underlying HDM node, and be of type node_name. 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 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. 2(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 underlying HDM edge's name, and at least two of the instance scheme's elements must refer to other schema objects that have underlying HDM nodes or edges. For example we may express ER n-ary relationships using the following construct scheme:

(link)er:relationship ::= ⟪(edge_name)name, (reference,2:N)er:entity_role⟫
(sequence)er:entity_role ::= ⟪(reference)er:entity, (string,nonkey)cardinality⟫

The scheme relationship has first a edge_name representing the name of the underlying HDM edge, followed by at least two references to an entity_role construct. This 'at least two' cardinality is specified by the 2:N in the brackets; 1:1 is assumed where no explicit cardinality is given. The entity_role is a **sequence** construct, which can only appear in the definition of another construct. Its first element is a reference to the entity construct we have already defined, and the constraint element card is used to denote the use of a constraint expression in the scheme. The scheme instance for the relationship in Fig. 2(b) would be ⟪result, student, 0:N, course, 0:N⟫. Note that the use of nonkey in the definition of the cardinality 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 must contain a reference to an existing construct. For example, an ER attribute can be defined by:

(link)er:attribute ::= ⟪(reference)er:attribute_target,
  (node_name)new_node_name, (constraint,nonkey)card⟫
(alternation)er:attribute_target ::= ⟪(reference)er:entity, (reference)er:relationship⟫

The **alternation** construct allows us to define that the existing construct that this link-nodal refers to may be *either* a entity or a relationship. The last element card corresponds to a constraint expression. With this definition, the name attribute of student in Fig. 2(b) would have the scheme ⟪student,name, notnull⟫, and the grade attribute on result the scheme ⟪⟪result, student, course⟫,grade,null⟫.

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.

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 via its wrapper. 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.

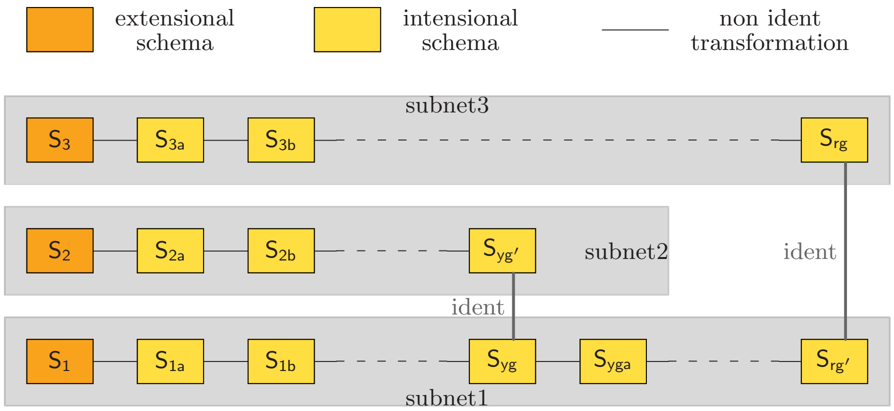


Fig. 6. Overview of Schemas held in AutoMed

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 $S_{yg'}$ (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 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

$\mathsf{S}_{1'} \rightarrow \mathsf{S}_1$. 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

A common task in data integration 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 $\mathsf{S}_{\mathsf{yg}} \rightarrow \mathsf{S}_{\mathsf{rg}'}$ 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. First, a YATTA complex node n of set type with '*' cardinality is created under the root of the YATTA model for each table. Second, 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 $\mathsf{S}_{\mathsf{rg}'}$, these two rules applied to the student table would generate a pathway:

$\mathsf{S}_{\mathsf{rg}'} \rightarrow \mathsf{S}_{\mathsf{yg}'}$

⑯ $\mathsf{addYattaNode}(\langle\!\langle \mathsf{root}, \mathsf{student}, \mathsf{set}, * \rangle\!\rangle,$
$[\langle \mathsf{r}, \mathsf{s} \rangle \mid \langle \mathsf{n} \rangle \leftarrow \langle\!\langle \mathsf{student} \rangle\!\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\!\langle \mathsf{root} \rangle\!\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId}\ [\mathsf{n}, \mathsf{r}]]])$

⑰ $\mathsf{addYattaNode}(\langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{name}, \mathsf{string}, \mathsf{k} \rangle\!\rangle,$
$[\langle \mathsf{s}, \mathsf{n} \rangle \mid \langle \mathsf{n} \rangle \leftarrow \langle\!\langle \mathsf{student} \rangle\!\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\!\langle \mathsf{root} \rangle\!\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId}\ [\mathsf{n}, \mathsf{r}]]])$

⑱ $\mathsf{addYattaNode}(\langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{tutor}, \mathsf{string}, \mathsf{nk} \rangle\!\rangle,$
$[\langle \mathsf{s}, \mathsf{t} \rangle \mid \langle \mathsf{n}, \mathsf{t} \rangle \leftarrow \langle\!\langle \mathsf{student}, \mathsf{tutor} \rangle\!\rangle; \langle \mathsf{r} \rangle \leftarrow \langle\!\langle \mathsf{root} \rangle\!\rangle;\ \mathsf{s} \leftarrow [\mathsf{generateId}\ [\mathsf{n}, \mathsf{r}]]])$

⑲ $\mathsf{deletePK}(\langle\!\langle \mathsf{student\_pk}, \mathsf{student}, \langle\!\langle \mathsf{student}, \mathsf{name} \rangle\!\rangle \rangle\!\rangle)$

⑳ $\mathsf{deleteColumn}(\langle\!\langle \mathsf{student}, \mathsf{tutor}, \mathsf{notnull} \rangle\!\rangle, [\langle \mathsf{n}, \mathsf{t} \rangle \mid$
$\langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{name} \rangle\!\rangle; \langle \mathsf{s}, \mathsf{t} \rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{tutor} \rangle\!\rangle])$

㉑ $\mathsf{deleteColumn}(\langle\!\langle \mathsf{student}, \mathsf{name}, \mathsf{notnull} \rangle\!\rangle,$
$[\langle \mathsf{n}, \mathsf{n} \rangle \mid \langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{name} \rangle\!\rangle])$

㉒ $\mathsf{deleteTable}(\langle\!\langle \mathsf{student} \rangle\!\rangle, [\langle \mathsf{n} \rangle \mid \langle \mathsf{s}, \mathsf{n} \rangle \leftarrow \langle\!\langle \langle\!\langle \mathsf{root}, \mathsf{student} \rangle\!\rangle, \mathsf{name} \rangle\!\rangle])$

Note that these rules could equally well be applied in reverse as 22–16 to generate $\mathsf{S}_{\mathsf{yg}} \rightarrow \mathsf{S}_{\mathsf{rg}'}$, which would be the method used to generate the integration shown in Fig. 6. In general, translating a schema from a source to a target modelling language involves using the MDR definitions to convert constructs in the source and target language to HDM, analysing the constraint information, and building an association between the two. One common aspect of this analysis [1] 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 for each value V in the extent of node N there must be at least one tuple in the extent of edge E that contains V.
2. $N \lhd E$ states that for each value V in the extent of node N there must be no more than one tuple in the extent of edge E that contains V.

Note that if neither of these constraints applies then N has 0:N occurrences in E; if $N \rhd E$ then N has 1:N occurrences in E; if $N \lhd E$ then N has 0:1 occurrences in E; and of $N \rhd E \wedge N \lhd E$ then N has 1:1 occurrences 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 $\langle\langle T, L, C\rangle\rangle$. Using the methodology in Section 3.1, we model this column as a link-nodal construct, that references an existing nodal construct $\langle\langle T\rangle\rangle$ that represents table T. In the HDM, the table has node N_t, and the column has an edge E that connects N_t to node N containing the attribute value. Now C can be expressed over those HDM constructs as notnull=$N \rhd E \wedge N_t \rhd E \wedge N_t \lhd E$, and null=$N \rhd E \wedge N_t \lhd E$.

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. If in addition T indicates this is a simple type node, then we have a match between the YATTA node and the concept of a column in the relational model. In particular, the YATTA constraint C is expressed as nk=$N \rhd E \wedge N_t \rhd E \wedge N_t \lhd E$ (matching the relational constraint notnull), and ?=$N \rhd E \wedge N_t \lhd E$ (matching null).

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 table with a mandatory column and a table with specializations for each instance of the mandatory column is found in transformations 1–7 in Section 2, where the $\langle\langle$student, level$\rangle\rangle$ column is used to generate new tables $\langle\langle$ug$\rangle\rangle$ and $\langle\langle$pg$\rangle\rangle$.

To describe such equivalences between schemas, we have created a package around the AutoMed repository that enables the definition of parameterised **template** transformations [8] which are schema and data independent. For example, the parameters of the template transformation that decomposes a table to its specializations are: (a) the schema that the template transformation is going to be performed upon, (b) the existing table, (c) its mandatory column, (d) the specialization tables that are going to be added to the schema and (e) the instances of the mandatory column that correspond to the specializations. These are specified as follows:

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

Note that each parameter has a description for use in a dialogue box, and optionally a Construct, which in this example ensures that the first parameter is a SchemaObject of type table and the second two parameters are of type column. After the parameters, a number of statements must be defined. In our example, since in general there are n different specialization tables 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 specialization table. For example, the transformations 1 and 4 that create the specialization 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 a column from a table and moves it down to its specialization tables.

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. In practice, a key issue is the identification of the semantic relationships between the schema objects [6], which then imply which primitive or template transformations should 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 [21], 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. These are: **equivalence**, **subsumption**, **intersection**, **disjointness** and **incompatibility**. Rules of the transformations that should be performed on the schemas based on the discovered semantic relationships are defined in [20].

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

In our approach to the automatic discovery of these semantic relationships, we perform a bidirectional comparison of schema objects, 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 intersection 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 subsumes X; and finally $0 < d(X,Y) < 1$ and $0 < d(Y,X) < 1$ when X, Y are intersecting. The problem with this approach is 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 intersection 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

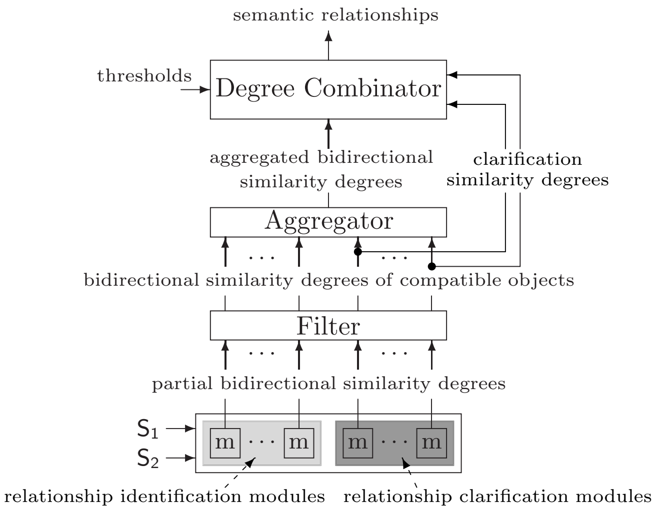


Fig. 7. Architecture of Schema Matching Tool

their semantic relationships. The output of the Aggregator becomes the input of the Degree Combinator, which based on the relationship clarification modules and 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.

7 Conclusions

This paper details the implementation of repository for BAV transformations 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 be 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.

References

1. M. Boyd and P.J. McBrien. Towards a semi-automated approach to intermodel transformations. Technical Report No. 29, AutoMed, 2004.
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. H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. DOLAP03*, pages 86–93, New Orleans, 2003.
5. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report No. 20, 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 25, AutoMed, 2004.
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, 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, 2002.
17. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*. IEEE, 2003.
18. P.J. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P, at VLDB'03*, Berlin, Germany, 2003.
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. BAV transformations on relational schemas based on semantic relationships between attributes. Technical Report 22, AutoMed, 2003.
21. N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. of 6th ICEIS*, 2004.