# AutoMed in a Nutshell

http://www.doc.ic.ac.uk/automed/

Software Release 0.4
Document Release 1.0

Peter McBrien
Dept. of Computing
Imperial College London

Thursday 16th March 2006

# Contents

# Chapter 1

# Introduction

The AutoMed project (`http://www.doc.ic.ac.uk/automed/`) has developed a set of tools to support a new approach to data integration called **both as view** (**BAV**) [MP03]. This report gives a brief overview of how to use those tools for the task of data integration, both via the AutoMed GUI and via the AutoMed API. In this introduction, we will give a very brief overview of research into data integration, and review the BAV approach to data integration.

A **data integration** system will provide a unified view of a number data sources, making them appear as a single data source to a user or application program. The process of data integration has two major aspects: **schema integration** [BLN86] in which the structures of the various **local schemas** of data sources are logically related to a single **global schema** by a set of **mappings**, and **query processing** where queries and updates on one schema are mapped (and split) into queries and updates on other schemas.

Our definition of data integration allows for a number of operational interpretations for the implementation of a data integration system. In federated databases [SL90], the global schema (called a **federated schema**) is a **virtual view** of the data sources; each query on the global schema is transformed into a number of logically equivalent queries on the data sources, and the results combined before being returned to a user. A **mediator** [Wie92] approach (where a global schema is called **mediator schema**) differs from a federated database approach in that mediators may source information from each other as well as from the original data sources. In **data warehousing** [JLVV02] the global schema (or **data warehouse schema** is a **materialised view** of the data sources, where data is copied from data sources into the data warehouse by a process known as **extraction transforming and loading** (**ETL**).

A number of approaches to data integration have been proposed, which can be broadly categorised into **global as view** (**GAV**), **local as view** (**LAV**), **global local as view** (**GLAV**), and **both as view** (**BAV**). The approaches differ in how the mappings between schemas are specified, and offer different degrees of precision in that specification. The fact that the approaches differ in the precision of the mappings means that they in turn need conduct the process of schema integration at different levels of precision. Since BAV is the most expressive approach, so it needs the greatest level of precision in schema integration.

## 1.1 BAV approach

In the BAV approach, the integration of schemas is specified as a sequence of bidirectional transformation steps, incrementally adding, deleting or renaming constructs, so as to map one schema to another schema. Optionally associated with each transformation step is a query expression, describing how instances of the construct can be obtained from the other constructs in the schema, which will be used during query processing. Absence of a query indicates that no instances of the construct may derived from the other constructs in the schema.

One of the novel features of the approach used in AutoMed is that it is not tied to using one

particular **common data model** (**CDM**) [SL90] for data integration. Instead, it works of the principle that data modelling languages such as ER, relational, UML, *etc* are graph-based data models, which can be described [MP99, BM04] in terms of constructs in the **hypergraph data model** (**HDM**) [PM98]. The implementation of AutoMed provides only direct support for the HDM, and it is a matter of configuration of AutoMed to provide support for a particular variant of a data modelling language. Once configured to use a data modelling language, schemas and transformations on those schemas can be described in terms of operations on the constructs of that data modelling language.

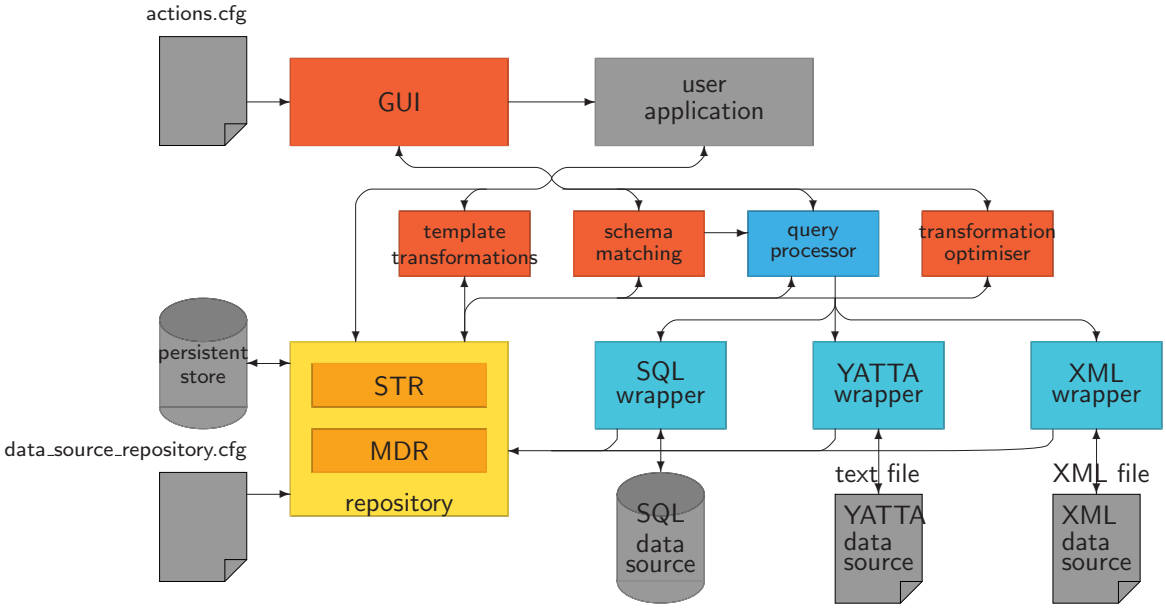



Figure 1.1: AutoMed Software Architecture

The AutoMed software architecture is illustrated in Figure 1.1 [BKL+04]. 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 [JPZ03] 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 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 [FP03].

The remainder of this report provides an overview of some components of the software architecture. Section 2 explains how to install and configure the AutoMed software, and run some simple examples to test that the installation is working correctly. Section 3 explains how the GUI may be used to view the repository contents, and how to use the query processor from the GUI. Section 4 gives a brief overview of the AutoMed API, allowing user applications to be written. Full API documentation is found at `http://www.doc.ic.ac.uk/automed/resources/apidocs/`.

# Chapter 2

# Setting Up The AutoMed Repository

These instructions are based on the assumption that you are working under Linux, with a bash shell command line environment. See Appendix A for notes on how to translate these instructions into the commands used under different shell environments or operating systems.

To use the AutoMed repository software, which is implemented as a Java package, you must have available a Java 1.4 runtime or development environment (the Sun JDK is used for AutoMed project development work, and releases are tested against this version of Java), and a Postgres database account to store the repository data (other databases will be supported in the future).

To determine which version of Java you are running at present, type at the command line:

```
java -version
```

Once you have the appropriate version of Java available, and also login details of a Postgres database you can use, you are ready to start using the AutoMed software. First download the AutoMed API from the AutoMed web site. There are a series of numbered releases (this document is written to describe release 0.4), along with a latest release. Normally you should use the highest numbered release. The latest release might be of use if you have been in contact with AutoMed developers and some bug has been fixed or feature added that you require.

The download is held in a gzipped tar file autoreps.tgz. You should place this in a new directory (say called automed), and set an environment variable AUTOMED to point at this directory. For example, if directory /home/pjm/automed had been used, then you would execute

```
export AUTOMED=/home/pjm/automed
```

You should move into the new directory, and unpack the tar file using the command:

```
tar -zxvf autoreps.tgz
```

This will unpack a number of directories, including at least the following:

- **apidocs**: contains a set of JavaDoc documentation files of the AutoMed API. Point you web browser at apidocs/index.html to view the documentation.
- **bin**: contains utilities to access the database tables used by the AutoMed repository to store its information. These are only intended for use by people working of development of the AutoMed repository.
- **doc**: contains some of the technical reports that are also available on the AutoMed web site.
- **examples**: contains a set of Java application programs that use the AutoMed API, and illustrate its use. These application programs will be referred to in this report.

- **jar**: contains various Java jar files. All the AUTOMED software is contained within automedRepositories.jar, and it is only this jar file that changes between different versions of AUTOMED. However, the AUTOMED software uses the third party jar files: the IQL parser uses java_cup.jar, the XML wrapper jaxp.jar, and parser.jar, and the various relational database wrappers each require the appropriate JDBC driver be available. The distribution comes with a configuration file that allows a Postgres database to be used as persistent storage mechanism, and this requires the use of a Postgres JDBC driver. The pg74.214.jdbc3.jar Postgres JDBC driver supplied has been found to work correctly all recent Postgres versions, but in general, you should use a JDBC2 or JDBC3 driver the corresponds to the version of Postgres you are using.

Before using any AUTOMED applications, you should set your CLASSPATH variable to be:

```
export CLASSPATH=.:$AUTOMED/jar/automedRepositories.jar:$AUTOMED/jar/java_cup.jar:\
$AUTOMED/jar/jaxp.jar:$AUTOMED/jar/parser.jar:$AUTOMED/jar/pg74.214.jdbc3.jar
```

Now change directory to $AUTOMED/examples, and compile and run the DefineRepository example:

```
javac DefineRepository.java
java DefineRepository
```

Given the default settings supplied with AUTOMED, and if you have not used AUTOMED before, this will throw some exceptions, unless you have a database called automed running on a Postgres server on you local machine that needs no password for you to login under you username via JDBC. However it causes a directory and file called $HOME/.automed/data_source_repository.cfg to be created, which you can edit to hold your own details. The file specifies how the AUTOMED Java API should access one or more databases that form a persistent store for the repository information, and is split into a number of DataSource entries covering different aspects of the repository. The two key parts, as illustrated in Figure 1.1, are the MDR, which holds definitions of modelling languages, and the STR which holds definitions of schemas, transformations, and the databases where those schemas are obtained from. All the repositories may be held in one database, which is assumed to be the arrangement for the purposes of these instructions[1].

To configure AUTOMED ready for use you must edit $HOME/.automed/data_source_repository.cfg to contain the details of the Postgres database that you will use to hold the repository data. In particular, for each line

```
JdbcURL jdbc:postgresql://localhost/automed
```

you should change localhost to the domain name of your Postgres database (for example, in DoC at Imperial College London it is db.doc.ic.ac.uk), and you should change AUTOMED to the Postgres database you wish to use for storing AUTOMED data (it is recommended that this is a database dedicated for the purpose; any tables that have the same same as AUTOMED repository table names will be dropped and recreated as part of the AUTOMED repository initialisation process). Also, for each line:

```
Password secret
```

you should change secret to your Postgres database password. If your Postgres username is different from your login username, then you may specify a mapping between the two. For example, if a user logged in under username pjm, but had a Postgres username automed, then the line shown below should be added to each DataSource entry:

```
Username:pjm automed
```

[1] This best arrangement for individuals using AUTOMED in isolation. If a group of users wanted to share the same modelling languages, but have each user have their own individual descriptions of schemas and transformations, then the MDR should be in a common shared database, and the STR (and all other Data Sources) should be in a database belonging to each individual.

Once you have made all the necessary changes to data_source_repository.cfg repeating the execution of DefineRepository should report that a series of repositories are being initialised.

## 2.1 Running the Basic Example Applications

To test your AutoMed installation more fully, and to generate some examples to view with the GUI editor described in Section 3, it is worthwhile to run some of the test applications. These all update the repository with their actions, and report on the command line what action they take.

In the examples directory, compile and run (in the same manner as DefineRepository above) the DefineModels application. This creates in the MDR a simple ER modelling language, a relational modelling language, and an HDM modelling language [PM98].

To create some example schemas, run DefineSchemas. This will creates six schemas in the STR, three in the ER modelling language (called er_s1, er_s2, and er_s3), one relational schema (rel_s1), and two HDM schemas (hdm_s1 and hdm_s2).

Finally, to demonstrate the transformation and integration of schemas, run DefineTransformations, which integrates the three ER schemas into one network of schemas, adding transformations into the STR.

Note that these examples are self contained, in that they do not integrate any real database or data source, but instead specify what the database schema is within the application code. Thus you will be unable to run query processing over these examples, since there is no data source to query. To test query processing you must also run the database integration example detailed in the next section.

## 2.2 Running An Example of Database Integration

The university data integration example demonstrates the integration of several small databases. To run the example, you must first setup the five database schemas university1 to university5 [2]. Then you should compile and execute the UniversityDatabaseWrapping application, giving it as arguments the details of the five databases where you have stored the five schemas (which are assumed to be present on a single DBMS; if this is not the case you will need to modify the program). For example, if you have the databases held on a Postgres database on your own computer, and your Postgres username is pjm with password secret, then you would execute:

```
java UniversityDatabaseWrapping -debug 0 -user pjm -password secret
     -driver org.postgresql.Driver -url jdbc:postgresql://localhost/
```

If you are working with the DoC at Imperial College London you may use a copy of the databases loaded onto the department's SQL Server by the following command:

```
java UniversityDatabaseWrapping -debug 0 -user lab -password lab
     -driver com.microsoft.jdbc.sqlserver.SQLServerDriver
     -url jdbc:microsoft:sqlserver://db-ms.doc.ic.ac.uk\;databaseName=pjm_
```

Once wrapped, the three of the university databases may be integrated by compiling and running the UniversityDatabaseIntegration application.

```
java UniversityDatabaseIntegration
```

Alternatively, the UniversityAutomaticIntegration application may be used, which uses the Merge component of the the **schema match and merge** tool (which will be discussed in detail in Section 5.1).

[2] A tool and scripts to create these databases is available from `http://www.doc.ic.ac.uk/~pjm/databases`. If you are working within DoC at Imperial College London then you will find that these databases are publicly available on the department's Microsoft SQL Server DBMS, as pjm_university1 to pjm_university5, which you may access when you login under username lab and password lab

# Chapter 3

# Using the AutoMed GUI

Once you have configured the AutoMed software as described in Section 2, the graphical user interface Gui application in the AutoMed software can be run by the command:

java uk.ac.ic.doc.automed.editor.Gui

This will open a window titled AutoMed Editor, which initially contains a single sub window titled All Networks (an example of which is shown in Figure 3.1). The all networks window contains an oval for each **network** of schemas you have held in your repository. A network is a set of schemas that are connected to each other by transformations. If you have run the examples detailed in Section 2.1, then you will have networks labelled er_s3-er_s2-er_s1, rel_s1, hdm_s1 and hdm_s2. If you have run the university database integration from Section 2.2 you will have three labelled uni_s3_src-uni_s1_src-uni_s2_src, uni_s4_src and uni_s5_src.

Double clicking on any one network will open a network window that shows all schemas within that network. For example, double clicking on that labelled er_s3-er_s2-er_s1 will open up a new window with the title Network er_s3-er_s2-er_s1, which should look similar to that shown in Figure 3.1. Selecting the schema labelled er_s2d (by clicking on the schema in the window) and then clicking on the menu button will bring up a menu of what actions you may perform on the schema. Apart from the options to change the colouring in the diagram, the three options always available when a single schema is selected are listed below:

- **Query schema**: opens a window that allows you to execute an IQL query on the schema. In the example we are using there are no data sources attached to the schemas er_s1, er_s2 and er_s3, and therefore query execution will always fail. We will return to describing how to use this tool in the context of the university integration example in Section 3.1.
- **Schema details**: opens a window that lists the construct type and scheme of each object in the schema. The schema details window for er_s2d is shown in Figure 3.2, and it should be noted that there are two schemes listed for each construct. The key scheme omits all parts of the full scheme that were not declared as being key scheme fields when the modelling language was defined. Either scheme may be used in IQL queries, though for conciseness it is recommended to use key schemes.
- **Apply transformation**: opens a window that allows you to apply a single transformation to the schema. The window will change what you are allowed to enter depending on which type of primitive transformation you intend to create.

Other options on the network menu require that multiple schemas to be selected, which may be done in two different ways. In both cases one schema should be first selected using a normal mouse click. If a second schema is selected whilst holding the control key, then you will have a **group selection**, which contains two schemas (where the second schema may be in a different network). Alternatively, if the second schema is selected whilst holding the shift key then a **pathway selection** is made, where a **pathway** comprises of all the transformations and schemas

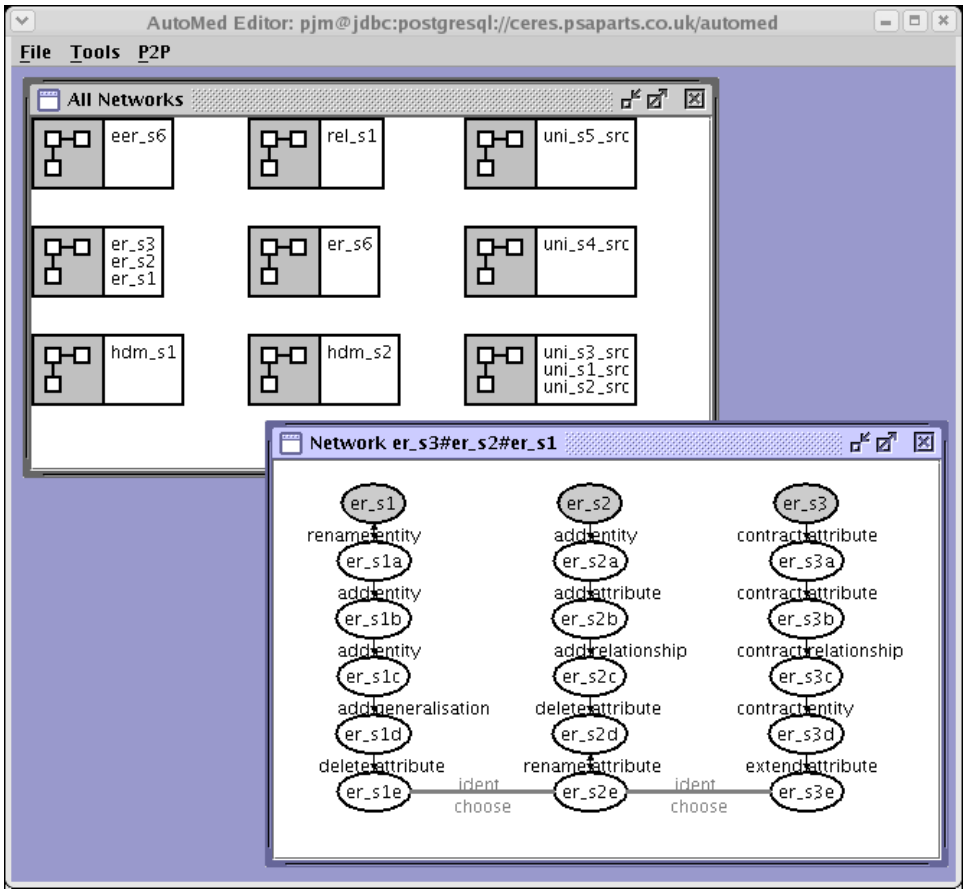



Figure 3.1: AutoMed GUI Tool viewing the basic examples

Schema objects in schema er_s2d

Subnet schema:er_s2

| model | construct | key scheme | full scheme |
|---|---|---|---|
| er | entity | <<person>> | <<person>> |
| er | attribute | <<person,pid>> | <<person,pid,key>> |
| er | attribute | <<person,name>> | <<person,name,notnull>> |
| er | entity | <<male>> | <<male>> |
| er | entity | <<female>> | <<female>> |
| er | generalisation | <<sex,person>> | <<sex,total,person,male,female>> |
| er | entity | <<dept>> | <<dept>> |
| er | attribute | <<dept,dname>> | <<dept,dname,key>> |
| er | relationship | <<worksin,person,dept>> | <<worksin,person,dept,1:1,1:N>> |

Figure 3.2: AutoMed schema object listing for schema er_s2d

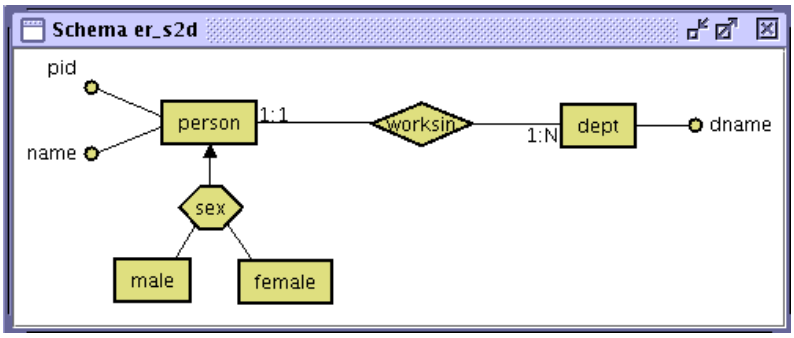



Figure 3.3: AutoMed GUI Tool viewing schema er_s2d

between the two clicked on schemas. You may therefore only make a pathway selection when the two schemas are in the same network.

For example, to view the pathway of transformations that would map from er_s2 to er_s1, in the network er_s3-er_s2-er_s1 window, pathway select schema er_s2 and er_s1, which should result in all schemas between those two schemas being highlighted. You should then use the menu to select **Transformation details**, and you should then obtain a listing of the primitive transformations shown in Figure 3.4.

Transformations in pathway: er_s2->er_s1

| transformation | action | model | construct | full scheme | extent function/new scheme |
|---|---|---|---|---|---|
| er_s2->er_s2a | add | er | entity | <<dept>> | [{x} \| {y,x} <- <<person,dept>>] |
| er_s2a->er_s2b | add | er | attribute | <<dept,dname,key>> | [ {x,x} \| {x} <- <<dept>>] |
| er_s2b->er_s2c | add | er | relationship | <<worksin,person,dept,1:1,1:N>> | <<person,dname>> |
| er_s2c->er_s2d | delete | er | attribute | <<person,dname,notnull>> | <<worksin,person,dept>> |
| er_s2d->er_s2e | rename | er | attribute | <<person,pid,key>> | <<person,id,key>> |
| er_s2e->er_s1e | ident | | | | |
| er_s1e->er_s1d | add | er | attribute | <<person,sex,notnull>> | [ {x,'M'} \| {x} <- <<male>>] ++[ {x,'F'} \| {x} <- <<female>>] |
| er_s1d->er_s1c | delete | er | generalisation | <<sex,total,person,male,female>> | |
| er_s1c->er_s1b | delete | er | entity | <<female>> | [ {x} \| {x,'F'} <- <<person,sex>>] |
| er_s1b->er_s1a | delete | er | entity | <<male>> | [ {x} \| {x,'M'} <- <<person,sex>>] |
| er_s1a->er_s1 | rename | er | entity | <<person>> | <<staff>> |

Figure 3.4: AutoMed GUI view of pathway er_s2 → er_s1

Close this window, and select er_s2e, and select the **Retract** menu option. Not only will the schema disappear, but so also are the ident transformations linking er_s2 to er_s1e and to er_s3e. This means that the single network er_s3-er_s2-er_s1 is now partitioned into three networks er_s3, er_s2, and er_s1.

You may restore the original integration as follows.

1. Comparing the schema details of er_s2d with those of the schema details of er_s1e and er_s3e reveals that er_s2d has an attribute ⟨⟨person, pid, key⟩⟩, which the others call ⟨⟨person, id, key⟩⟩. Selecting er_s2d and then using the menu option **Apply transformation**, you should choose action rename, enter the full scheme as ⟨⟨person, pid, key⟩⟩, and the new scheme ⟨⟨person, id, key⟩⟩. This will generate a new schema named er_s2e.

2. To integrate the new er_s2e with the identical er_s1e, select er_s2e, and then group select (*i.e.* hold select whilst holding the control key) schema er_s1e, and choose menu option **Apply ident transformation**. This will cause the two networks to be merged into one.

3. Repeating the process in step (2) with er_s2e and er_s3e will restore the single network shown in Figure 3.1.

## 3.1 Querying Data Sources using the IQL

The **intermediate query language** (**IQL**) [Pou01, JPZ03, Pou04] is functional programming language based on list comprehensions [Bun94] that is designed to model the query processing

capabilities of a number of database query languages in a single query language. As such, it is not designed to be a user oriented language, but is nevertheless is straightforward for computing professionals to use directly if required.

The IQL query tool is obtained by selecting any schema in a network view, and then selecting the query schema option from the menu. Figure 3.5 shows the IQL tool running on schema uni_ze in the university example (which, provided you have run the applications described in Section 2.2, may be found by double clicking on network uni_s3_src-uni_s1_src-uni_s2_src, and scrolling down to find uni_ze at the bottom of the left hand pathway). The IQL tool may also be executed independently from the GUI. For example, the same schema could be queried by executing the command:

java uk.ac.ic.doc.automed.editor.IQLTool uni_ze

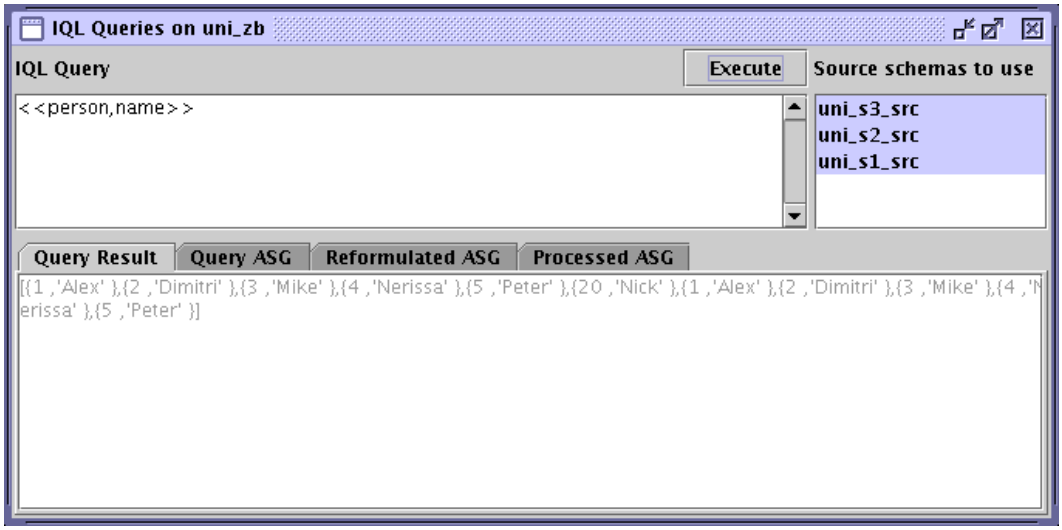



Figure 3.5: AutoMed IQL Tool

The simplest IQL query is to give the scheme of one object in a schema (which in AutoMed is called a **schema object**). Any scheme of a table or column schema object that appears in the schema details window may thus be used as an IQL query, in either the full scheme or key scheme form. Figure 3.5 shows the result of the query ⟨⟨person,name⟩⟩, which returns a list of tuples for the query when executed with all three data sources uni_s1_src, uni_s2_src, and uni_s3_src selected. If you wish to source the results from only a subset of these data sources, you may do so by using the top right **Source schemas to use** selector to choose the data sources to be used when answering a query. If you choose each schema in turn, you will find that uni_s2_src returns one fewer people than uni_s1_src[1]. You will find that uni_s3_src returns no results, since that data source contains no information about the names of people.

To eliminate duplicate answers from a list, the keyword distinct should be prefixed to the list. Hence modifying the query in Figure 3.5 to distinct⟨⟨person, name⟩⟩ would return

[{1, 'Alex'}, {2, 'Dimitri'}, {3, 'Mike'}, {4, 'Nerissa'}, {5, 'Peter'}, {20, 'Nick'}]

Any more complex query processing requires the use of list comprehensions, which produce new lists from a combination of **generators** and **filters**. Anything that produces a list can be used in a generator, and hence any table or column construct from the sql model may be used as a generator. For example, an identical result to that above may be produced by:

distinct[{x, y} | {x, y} <− ⟨⟨person, name⟩⟩]

[1]Given the way the integration has been specified, with the staff table in uni_s1_src and the person table in uni_s2_src being made equivalent by a rename transformation between er_s1 and er_s1a, those two tables should return the same result. The fact that they do not represents a inconsistency between the databases that is quite typical of 'real world' situations: what the AutoMed integration specifies is that from a logical perspective, the data sources *should* contain the same data. It is a matter on going research as to how AutoMed should specify how such inconsistencies should be dealt with.

Several generators may be placed in a list comprehension, together with a filter to return a selection of the results. For example, to find the names of persons and the departments in which they work requires that we generate tuples from the lists for the schema objects $\langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle$ and $\langle\langle \mathsf{person}, \mathsf{dname} \rangle\rangle$, and then equate the common key of the two:

$\mathsf{distinct}[\{\mathsf{y}, \mathsf{z}\} \mid \{\mathsf{x}, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle; \{\mathsf{x}, \mathsf{z}\} <- \langle\langle \mathsf{person}, \mathsf{dname} \rangle\rangle]$

More details of the IQL may by found in the IQL tutorial [Pou04], but here we provide details of three aspects of the language that are used frequently: arithmetic, aggregation and string functions.

### 3.1.1 IQL Arithmetic and Comparison

IQL supports the four basic arithmetic functions, and six comparison functions found in all programming languages. However, a little care is needed in there use. For example, the following is correct IQL to return only those people with ids greater than 3:

$[\{\mathsf{x}, \mathsf{y}\} \mid \{\mathsf{x}, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle; \mathsf{x} > 3]$

But the following is not correct IQL to return those ids multiplies by 100:

$[\{\mathsf{z}, \mathsf{y}\} \mid \{\mathsf{x}, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle; \mathsf{z} = \mathsf{x} * 100]$

This is because the equals comparison is not an assignment operation, but a boolean valued function. Instead, you should write:

$[\{\mathsf{x} * 100, \mathsf{y}\} \mid \{\mathsf{x}, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle]$

Note that it is correct to test for equality by putting a constant in the head of a generator. For example, to find the name associated with id 3 you could write:

$[\{\mathsf{y}\} \mid \{3, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle]$

### 3.1.2 IQL Aggregation Functions

The IQL provides a number of **aggregation** functions. For example, the IQL query $\langle\langle \mathsf{person} \rangle\rangle$ will return the list containing the ids of persons as found in all the source databases, and include duplicates as the same identifier is found in several data sources. To count the number of distinct identifiers we have in all the databases for different persons, would require the query:

$\mathsf{count}(\mathsf{distinct}\langle\langle \mathsf{person} \rangle\rangle)$

To achieve a query processing similar using aggregate functions with an SQL **GROUP BY** clause, you should use the IQL gc function, with an aggregate function as its first argument. This will first group tuples from a list by the first argument of each tuple in a list, and then apply the aggregate function to the second argument. For example, to count how many (possibly identical) names are associated with each person in the various data sources, you should execute:

$\mathsf{gc\ count}\ [\{\mathsf{x}, \mathsf{y}\} \mid \{\mathsf{x}, \mathsf{y}\} <- \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle]$

The manner in which the integration has been specified means we would expect to find the same set of person ids in all three data sources. However, in practice it is likely that data sources will contain some ids not present in other data sources. Since we know each id should appear three times (once from each data source), we can find those ids not appearing in all data sources by the following IQL query:

$[\mathsf{x} \mid \{\mathsf{x}, \mathsf{y}\} <- \mathsf{gc\ count}\ [\{\mathsf{x}, \mathsf{x}\} \mid \{\mathsf{x}\} <- \langle\langle \mathsf{person} \rangle\rangle]; \mathsf{y} < 3.0]$

The IQL also provides sum, min, max and avg functions.

### 3.1.3 IQL String Functions

The IQL **string** functions provide some of the common string processing capabilities found in most programming languages, and are summerised in Table 3.1. Typically these would appear in

| IQL function | Description |
|---|---|
| Split s p | Return a tuple with string s split into two parts, based on p, which may be an integer (indicating that character number p will start the second string), or a string (used to determine the start of the second string). |
| Concat s1 s2 | Concatenate two strings s1 and s2 together, returning a new string |
| UpperCase s | Return a string which has string s converted to capital letters. |
| LowerCase s | Return a string which has string s converted to small letters. |
| Substring s n1 n2 | Return the substring of s starting at character number n1, and finishing just before n2 |
| Length s | Return an integer representing the number of characters in string s. |

Table 3.1: IQL builtin string functions. Note that IQL string functions count characters from one (not zero).

the head of a comprehension. For example, to return the length of each department name, the Length function is used as follows:

distinct[{x, Length x} | {x, x} <− ⟨⟨dept, dname⟩⟩]

## 3.2 Wrapping a Data Source

Selecting from the tool bar menu Tools option Wrap Data Source will open up a tool that allows you to use any AutoMed wrappers that you have in your CLASSPATH. The AutoMed repository software includes a wrapper for relational databases, that has been tested to work with Postgres, Microsoft SQL Server, and Oracle data sources. You will need however to ensure you have the JDBC driver for the database concerned in your CLASSPATH.

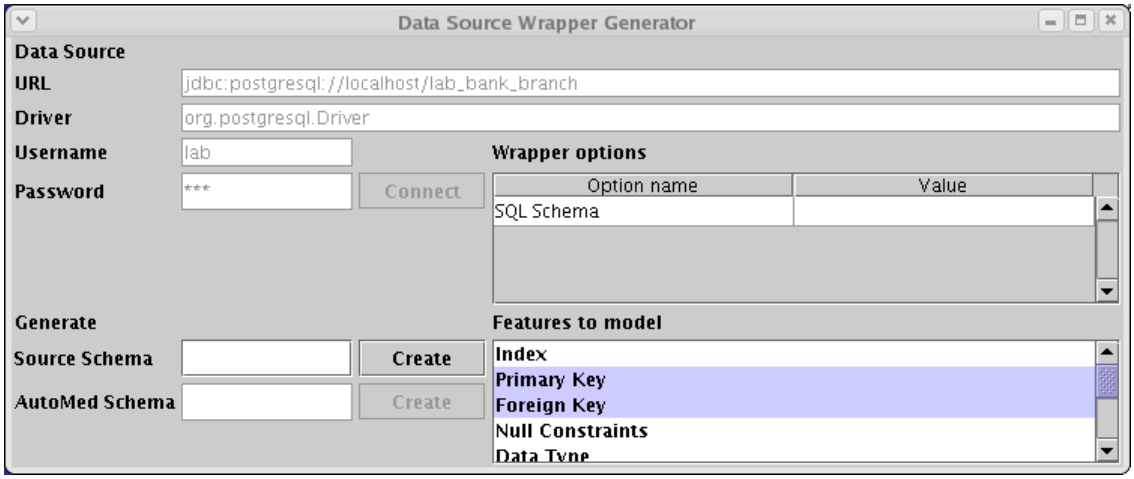



Figure 3.6: Data source wrapper tool, using the relational wrapper

The first step of wrapping a data source requires that you give the connection details of the data source.

- The **URL** specifies where the data source is accessible from. The first part of the URL will determine which AutoMed wrapper should be used: **jdbc** indicates that the SQL database wrapper is used, **yatta** that the YATTA semi-structured file wrapper is used, **dom** that the XML DOM wrapper is used.
- The **driver** specifies which Java driver is used to access the data source, as summerised in Table 3.2. Note that for the AutoMed SQL driver (*i.e.* to use a subclass of SQLWrapper),

the driver should be a Java JDBC driver for the DBMS in question, which you should put in your Java CLASSPATH.

If you leave the driver box blank, then the URL is used to guess which driver is appropriate, and this normally produces correct results, with the exception of the XMLWrapper, which has multiple protocols that it uses to access data sources, and hence is unable to use URLs to determine the driver used.

- The **username** and **password** are those required to access the data source remotely using the driver you have given. For example, on a Postgres database this would be the Postgres username and password of you database account.

| URL Prefix | Wrapper | Driver |
|---|---|---|
| jdbc: | SQLWrapper | JDBC driver appropriate to DBMS |
| yatta: | YattaWrapper | uk.ac.ic.doc.automed.wrapper.YattaWrapper |
| | XMLWrapper | org.w3c.dom |
| p2p: | P2PWrapper | uk.ac.ic.doc.automed.wrapper.P2PWrapper |

Table 3.2: Drivers for use with Wrappers

For example, choosing the URL jdbc:postgresql://db.doc.ic.ac.uk/lab_bank_branch, with driver org.postgresql.Driver, username lab, and password lab, and then pressing connect should allow you to connect to an example Postgres database containing a small banking example[2].

If the tool could successfully connected to the data source, the window will update itself to appear as shown in Figure 3.6, with certain features of the relational model available for you to select, which will then we modelled when later the schema is produced in AutoMed. The features available are:

- **index** models an SQL index as a constraint on the table that is indexed. Note the SQL unique indexes are equivalent to a **candidate key**.
- **primary key** models an SQL primary key as a constraint on the table for which it is a key.
- **foreign key** models an SQL foreign key as a constraint linking table which contains the foreign key to another table that contains the candidate key which the foreign key values appear in.
- **null constraints** causes the SQL NULL and NOT NULL constraints to be modelled as part of the table scheme.
- **data type** causes the SQL column data type, such as INT, STRING, *etc* to be modelled as part of the table scheme.
- **data size** causes maximum size in bytes of the SQL column to be modelled as part of the table scheme.
- **column number** causes the position of a column in the tuples of the table to be modelled as part of the table scheme, counting columns from one.
- **schema aware** causes the name of a schema to be included in the scheme of each table modelled in AutoMed. By default, this feature is not selected, and only the default schema of the database username is modelled in automed. With this feature selected, all schemas in the database are modelled, unless the **SQL Schema** option is used to name a particular schema to model, which when combined with using schema merging, makes the schema appear as a data source. This combination is particularly useful when using the Oracle DBMS.

[2] At present, firewall restrictions prevent this from working outside the Imperial College network.

- **schema merging** being set causes the name of schemas not be form part of the key scheme of the table schema object. This is convenient to use in conjunction with the schema aware option, to cause a particular relational schema appear as the default available to a user of a DBMS (in particular when using an Oracle DBMS data source).

### 3.2.1 Example of Wrapping a Relational Data Source

First, use the Tools menu option Wrap Data Source to open the data wrapping tool, and entire the username, password and URL of the relational database you wish to wrap. For example, with DoC, you will be able to access URL jdbc:postgresql://db.doc.ic.ac.uk/lab_bank_branch with username lab and password lab. Click Connect, and if no problems occur, the Connect button will grey out, and the Create button next to the Source Schema box will be enabled. Enter branch_src as the source schema name, and from the features list, choose at least at least the **primary key** option, and click Create. This will produce a representation of the database where each table is represented as a separate node, with no separate representation of columns, which we call a **source oriented** schema, since it breaks the AutoMed convention of representing each part of the structure that can be changed in a schema by a separate construct type. A schema the obeys this convention is called an **AutoMed oriented** schema. When this convention is applied to the relational model, it results in a schema normalised to **sixth normal form** (**6NF**) [DDL03, Dat04].

To produce an AutoMed oriented schema, enter branch as the AutoMed schema name, and click Create next to it. This maps branch_src to a form where each attribute is also represented as a distinct node. Note that for the YATTA and XML wrappers, the AutoMed and source oriented schemas are identical.

# Chapter 4

# The AutoMed API

In this section we provide a brief overview of using the AutoMed API, describing how to create modelling languages, schemas in those modelling languages, and transformations between those modelling languages. We also explain how applications written over this API can be integrated into the AutoMed GUI, thus extending the functionality of the core AutoMed toolset.

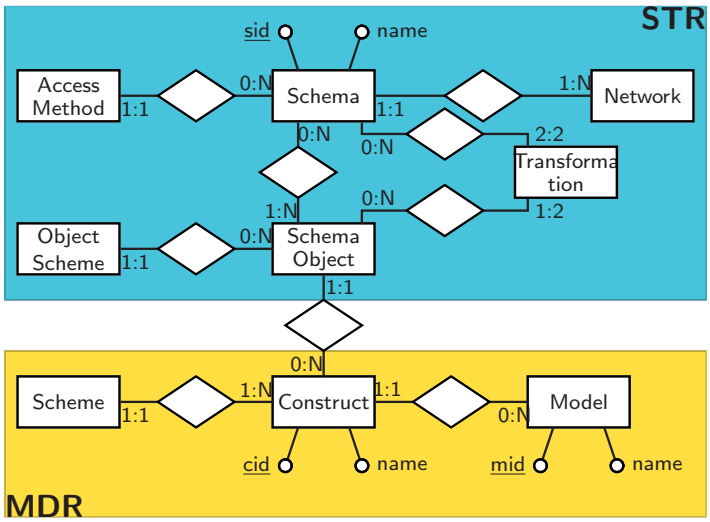



Figure 4.1: Conceptual View of the AutoMed Repository

Figure 4.1 gives an overview of the key object classes available in the AutoMed repository, which are divided into two sections. 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 [PM98, BM05]. 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 [MP02, Fan05].

The following sections give an overview of some of the key features in the AutoMed API that a Java programmer will need to know in order to write applications that use AutoMed. Since the repository forms the foundation for AutoMed applications, the preamble of your Java program will normally import the reps package as follows:

```
import uk.ac.ic.doc.automed.reps.*;
```

Other packages in the AutoMed API should be imported as required by the application.

## 4.1 Wrapping a Data Source

To use the AutoMed wrappers, you need to import classes from the AutoMed wrappers package, by including the following in the Java preamble.

```
import uk.ac.ic.doc.automed.wrappers.*;
```

This package also includes the SQLWrapper and YATTAWrapper wrapper implementations for SQL and semi-structured data sources. For XML data sources, you need to also import the XMLWrapper:

```
import uk.ac.bbk.dcs.automed.xml.XMLWrapper;
```

To create a wrapped data source, you need a implementation of AutoMedWrapperFactory for the type of data source you wish to access. For example, to wrap an SQL data source, you might have the following code:

```
AutoMedWrapperFactory wf=new SQLWrapperFactory();
wf.setFeatures(SQLWrapperFactory.PRIMARY_KEY);
wf.setFeatures(SQLWrapperFactory.FOREIGN_KEY);
wf.setFeatures(SQLWrapperFactory.DATA_TYPE);
wf.setFeatures(SQLWrapperFactory.DATA_SIZE);
wf.setFeatures(SQLWrapperFactory.NULL_CONSTRAINT);
```

The various calls to AutoMedWrapperFactory.setFeatures configure the wrapper factory to model certain features of the data source in the repository. By default, the SQLWrapperFactory only models tables and their columns, with no key or type information. The above sequence of calls to AutoMedWrapperFactory.setFeatures cause primary keys, foreign keys, the SQL data type, the size and the null/notnull distinction to all be modelled.

Once a AutoMedWrapperFactory has been created and configured, any number of data sources of that type may be wrapped by calls to AutoMedWrapper.selectNewAutoMedWrapper. For example, to create the uni_s1_src schema used in the example SQL Server data source of Section 2.2 held in a SQL Server database could be achieved by using the TransactSQLWrapper class:

```
TransactSQLWrapper sw=(TransactSQLWrapper)
    TransactSQLWrapper.newAutoMedWrapper("lab","lab",
        "com.microsoft.jdbc.sqlserver.SQLServerDriver",
        "jdbc:microsoft:sqlserver://db-ms.doc.ic.ac.uk\\\;databaseName=pjm_university1",
        null,wf);
```

However, If the URL and/or driver is distinctive enough to identify the wrapper being used, the AutoMedWrapper.selectNewAutoMedWrapper method can we used as follows:

```
AutoMedWrapper sw=
    AutoMedWrapper.selectNewAutoMedWrapper("lab","lab",null,null,
        "jdbc:microsoft:sqlserver://db-ms.doc.ic.ac.uk\\\;databaseName=pjm_university1",
        null,wf);
```

The AutoMed wrapper sw will now be connected to the data source (in the example, this would mean that it has logged in to the database), with an associated AccessMethod that holds the username, password, url, driver and wrapper features used when the data source was wrapped. To obtain a representation in AutoMed of the data source schema, you should call AutoMedWrapper.getSchema, which returns a Schema instance that models that data source. Note that this that the schema could have been created during the method call above if the second to last parameter was changed from null to "uni_s1_src". Also note that if the AutoMedWrapperFactory was changed to null, one would be created with default feature settings.

If the wrapper provides distinctive source oriented and AutoMed oriented schemas[1] you can also obtain the AutoMed oriented schema by calling AutoMedWrapper.newAutoMedSchema.

```
Schema s=sw.getSchema("uni_s1_src");
Schema as=sw.newAutoMedSchema("uni_s1");
```

Note that unless the SQLWrapper.SCHEMA_AWARE feature is selected, only the default schema is read into AutoMed. If the tables we were interested in were in a SQL schema called staffdept, then the following executed before the getSchema method would cause that SQL schema to be read in and have the same key schemes as if it were the default database (though the schema name will be stored in the full table scheme):

```
wf.setFeatures(SQLWrapperFactory.SCHEMA_AWARE);
wf.setFeatures(SQLWrapperFactory.SCHEMA_MERGING);
sw.setOption(SQLWrapper.OPTION_SCHEMA_NAME,"staffdept");
```

Note that this arrangement of features and options is typical of how an Oracle data source should be wrapped in AutoMed. Also note that having SQLWrapperFactory.SCHEMA_AWARE selected, but not the SQLWrapperFactory.SCHEMA_MERGING nor having the SQLWrapper.OPTION_SCHEMA_NAME would cause all SQL schemas in the database to modelled in one AutoMed schema, which the SQL schema name forming part of the key scheme for each table modelled.

Since AccessMethod and Schema are repository objects, they are persistent and hence will be available to other Java applications even after the current application has terminated. When starting an application that uses a data source schema, the wrapper for the data source is retrieved by finding the AccessMethod associated with the data source schema. The AccessMethod contains all the details associated with the wrapper, and allows connections to data sources to be reestablished as required. In principle more that one AccessMethod may be associated with a data source, each with a different Protocol. For example, a relational database might in principle be accessed via either the JDBC or ODBC protocols. In the current implementation, only one Protocol is supported per data source type, and thus a reliable method to retrieve the wrapper for uni_s1_src would be:

```
Schema s=Schema.getSchema("uni_s1_src");
AccessMethod am[]=s.getAccessMethods();
AutoMedWrapper sw=AutoMedWrapper.selectNewAutoMedWrapper(am[0]);
```

## 4.2 Transforming Schemas

In the schema created in Section 4.1, we might wish to rename the table ⟨⟨staff⟩⟩ to ⟨⟨person⟩⟩. To do this, you find the SchemaObject that represents the staff table using Schema.getSchemaObject. Then you can use Schema.applyRenameTransformation to create a new schema that is the same as the original schema, except ⟨⟨staff⟩⟩ being named ⟨⟨person⟩⟩:

```
SchemaObject staff=uni.getSchemaObject("⟨⟨staff⟩⟩");
Schema unia=uni.applyRenameTransformation(staff,new Object[]{"person"});
```

To obtain the new SchemaObject that represents ⟨⟨person⟩⟩, we need to inspect the result of the transformation. At a conceptual level, each transformation either deletes, adds or changes one SchemaObject. Thus each end of the transformation has at most one SchemaObject associated with it, which may be found by asking for the object that is associated with the Schema at that end of the transformation. In practice, there is a complexity that a rename of one SchemaObject was cause others to be renamed. In the example, renaming ⟨⟨staff⟩⟩ to ⟨⟨person⟩⟩ means that column ⟨⟨staff,id⟩⟩ is renamed to ⟨⟨person,id⟩⟩, primary key ⟨⟨staff_pk,staff,⟨⟨staff,id⟩⟩⟩⟩ is renamed to ⟨⟨staff_pk,person,⟨⟨person,id⟩⟩⟩⟩, *etc.* In AutoMed this is handled by the SchemaObject being represented as an array, where the first element of the array is the object being renamed, and the remaining elements of the array are objects renamed as consequence of that renaming.

[1] At present, SQLWrapper (and all its subclass wrappers) provide distinctive source and AutoMed schemas, whilst YattaWrapper and XMLWrapper do not.

Hence, after the applying the transformation above, we may find the renamed SchemaObject ⟨⟨person⟩⟩ as follows:

```
SchemaObject map=Transformation.getTransformation(uni,unia).getSchemaObject(ns);
SchemaObject person=(SchemaObject)map.getSchemeDefinition()[0];
```

Other transformations are applied in a similar manner, but some important details differ. When applying an **add** or **extend** transformation, you need to give as a parameter the Construct instance you are going to create a SchemaObject for when performing the transformation. For example, to add ⟨⟨male⟩⟩ and ⟨⟨female⟩⟩ ER entities based on which ⟨⟨person⟩⟩ instances are associated to a particular value in ⟨⟨person,sex⟩⟩ (and assuming that the DefineModels example application has been run), the following code should be used:

```
Model er=Model.getModel("er");
Construct ent=er.getConstruct("entity");
Schema unib=unia.applyAddTransformation(ent,new Object[]{"male"},
        "[ {x} | {x,'M'} <- ⟨⟨person,sex⟩⟩]",null);
Schema unic=unib.applyAddTransformation(ent,new Object[]{"female"},
        "[ {x} | {x,'F'} <- ⟨⟨person,sex⟩⟩]",null);
```

If we wish the to create a generalisation between the ⟨⟨male⟩⟩, ⟨⟨female⟩⟩, and ⟨⟨person⟩⟩ entities, we must obtain the SchemaObject instances that represent this elements of the schema (and not use strings) as illustrated below:

```
SchemaObject person=uni.getSchemaObject("⟨⟨person⟩⟩");
SchemaObject male=Transformation.getTransformation(s1a,s1b).getSchemaObject(s1b);
SchemaObject female=Transformation.getTransformation(s1b,s1c).getSchemaObject(s1c);
Construct gen=er.getConstruct("generalisation");
Schema unid=unic.applyAddTransformation
        (gen,new Object[]{"sex","total",person,male,female},null,null);
```

Note that any transformation creating a constraint should never supply a query (if you do, then a runtime exception will be generated).

To apply delete or contract transformations, you need only give the SchemaObject instance to be deleted, along with any query that will restore the extent of the object being deleted if its Construct class is not of type constraint. For example, to delete the ⟨⟨person,sex⟩⟩ attribute, the following code would be used:

```
SchemaObject personsex==unid.getSchemaObject("⟨⟨person,sex⟩⟩");
Schema unie=applyDeleteTransformation(personsex,
        "[ {x,'M'} | {x} <−⟨⟨male⟩⟩] ++ [ {x,'F'} | {x} <−⟨⟨female⟩⟩]",
        "distinct [{x} | {y,x} <−⟨⟨person,sex⟩⟩] = ['F','M']");
```

Note that using the various apply methods in Transformation can only ever generate a new Schema.VIRTUAL_TYPE type Schema instance, means that it is never possible to trasform one Schema.DATA_SOURCE_TYPE, Schema.STORED_TYPE, or Schema.MATERIALISED_TYPE type Schema to another just using the apply methods. To achieve this integration of two data sources, you must at some point create an **ident** transformtion. Suppose that we have translated another schema to get a SchemaObject instance held in uniebis. Provided that these two Schema instances hold SchemaObjects with the same name and Contruct type, you will be able to ident the schemas together using the static method Transformation.createIdentTransformation:

```
Transformation.createIdentTransformation(unie,uniebis,null,null)
```

## 4.3 Query Processing

The IQL query processing is split into three stages that allow application programmers to have significant control over the query processing, and to add their own extensions onto the IQL. To begin with, we have a query a schema (which we shall refer to as the **query schema**), which

shall be represented in a graph structure called the **AbstractSyntaxGraph** (ASG). Then each of the three stages will require its own class that rewrites the ASG to another form. All query processing classes are located in a single package that should be included in applications using the query processor:

```
import uk.ac.bbk.dcs.automed.qproc.*;
```

The first stage in query processing is to map the queries on the query schema into queries on extensional schemas connected to the query schema by pathways. This process is conducted by the QueryReformulator, one of which needs to be created for each query schema that is to be queried with respect to a set of source schemas. For example, to query uni_ze in the University example against *all* attached data sources, you would create QueryReformulator as follows:

```
Schema qs=Schema.getSchema("uni_ze");
Schema extSchemas[]=qs.findAttachedExtSchemas();
QueryReformulator qr=new QueryReformulator(qs,extSchemas);
```

The output of the QueryReformulator with be a version of the query with constructs from the query schema replaced by constructs from the source schemas. This is when broken up into fragments by the FragmentProcessor, which is created without any parameters:

```
FragmentProcessor fp=new FragmentProcessor();
```

Finally, each query fragment with need to be evaluated against the data source, and replaced by the values that form the result of the query. This process is conducted by the Evaluator, that is created to evaluate a specific version of the IQL. If no user functions are to be added, the StandardFunctionTable should be used as follows:

```
FunctionTable ft=new StandardFunctionTable();
Evaluator e=new Evaluator(ft);
```

Once these classes have been created, any number of queries can be made on the query schema using the same class instances. For example, one query evaluation could be conducted by:

```
ASG q=new ASG("distinct ⟨⟨person, name⟩⟩");
qr.reformulate(q);
fp.process(q);
e.evaluate(q);
```

## 4.4 Adding Tools to the GUI

The AutoMed **GUI** provides a graphical tool to view the contents of the STR, and perform certain API calls from graphical tools, in particular the creation of transformations, and the wrapping of data sources. Apart from running the GUI from the command line as described in Section 3, an application may open the tool using the following code:

```
Gui gui=Gui.openMainWindow();
```

The Gui.openMainWindow method will not create a new Gui instance if it has been called before, and hence it is safe to place this is application code without checking if the code is being called from the GUI tool. The Gui class can display a windown describing any Postionable instance in the repository (*i.e.* any instance of Schema, SchemaObject, and Network) using the GuicreateEditorFrame method. It should be noted that at present the view of SchemaObject produces an empty window, but in a later release the window will contain information about the schema object. For example, to open as view of a Schema called er_s2:

```
Schema s=Schema.getSchema("er_s2");
gui.createEditorFrame(s);
```

The AutoMed GUI tool allows users to add their own applications to the menu structure of the GUI, and to pass to the application any instances that have been selected by the user. Any updates that the application makes to the repository are automatically reflected in the GUI. The GUI editor and classes are included by the following statement:

```
include uk.ac.ic.doc.automed.editor.*
```

Any application that is to be called by the GUI must provide a static method that takes an array of Object instances as an argument, and returns a single UserActionResult. It is valid (and often the case) for null to be returned. For example, the IQLTool described in Section 3.1 contains the following method:

```
public static UserActionResult getTool(Object o[]) {
    new IQLTool((Schema)o[0]);
    return null;
}
```

Note that the method implicitly assumes that the object array passed to it contains a single instance of the Schema class. The configuration files associated with the GUI ensure that this will always be the case, by controlling which objects in the GUI may be selected before the method is called, by placing in configuration file a series of ⟨method statement⟩ entries, the syntax for which is given in Table 4.1.

For example, a menu option called Query schema ... to call IQLTool.getTool is configured by the following entry, where the schema(single,inside) entry indicates that the menu option is active wherever a single schema is selected, or the menu is brought up inside a schema window with nothing selected.

```
schema(single,inside) {
    name "Query schema ...",
    method uk.ac.ic.doc.automed.editor.IQLTool.getTool
}
```

Sometimes, the basic choice of enabling menu items based on there being just the ⟨selection type⟩s listed in Table 4.1 is not sufficient, in which case an enabler method may be provided, which takes an object array, and returns a boolean to indicate if the menu option should be enabled. For example, applying a ident transformation requires that exactly two schemas are selected. This may be configured by using a ⟨selection type⟩ of multi, with an enabler method that returns true if passed an Object array of two elements:

```
schema(multi) {
    name "Apply ident transformation",
    method uk.ac.ic.doc.automed.editor.Gui.applyIdent,
    enabler uk.ac.ic.doc.automed.editor.Gui.exactlyTwo
}
```

The GUI on startup reads a master configuration file actions.cfg held in the $HOME/.automed directory. If you are using the default one written by the AutoMed API, this should look roughly like the file shown in Figure 4.2 in structure.

```
//
// AutoMed GUI editor master configuration
//
include "::standard_editor_actions.cfg"
include "my_actions.cfg"
// include "::developer_actions.cfg"
// include "::test_actions.cfg"
```

Figure 4.2: Example actions.cfg file

The include statements cause other configuration files to be included into the main configuration file. All the include statements that have :: in front of the filename cause the file to be sourced from the editor package directory (and hence cannot be altered by a user). The three such supplied configuration files illustrated above have the following purpose:

- The standard set of tools used by the GUI (and which were described in Section 3) are listed in standard_editor_actions.cfg.
- New experimental tools distributed with AutoMed are described in developer_actions.cfg.
- A complete set of all possible configurations are described in test_actions.cfg. You might wish to uncomment this entry, and comment out all the other entries, and then run the GUI tool to experiment with which configuration patterns are obtained when certain windows and objects are selected.

| ⟨selection object⟩ | description |
|---|---|
| root | Refers to the master All Networks window, where only the inside select option has any effect. |
| network | Refers to anything that represents the Network class, *i.e.* any node in the All Networks window, or being inside any Network ... window. |
| schema | Refers to anything that represents the Schema class, *i.e.* any node in a Network ... window, or being inside any Schema ... window. |
| schemaobject | Refers to anything that represents the SchemaObject class, *i.e.* any construct in a Schema ... window, or being inside any SchemaObject ... window. |

| ⟨selection type⟩ | description |
|---|---|
| inside | Menu has been called inside a window of selection object, with nothing selected in that window. |
| single | Menu has been called with a single selection object selected. |
| multi | Menu has been called with two or more selection objects selected. Multiple objects are selected either by dragging box over a set of objects, or by control-clicking on several individual objects. |
| path | Menu has been called with a pathway of the selection objects selected. Paths are obtained by selecting one object, and then shift-clicking on another object linked to the first by some set of arcs in the diagram. |

```
⟨gui configuration file⟩ ::= ⟨method statement⟩*

⟨selection⟩ ::= ⟨selection object⟩( ⟨selection type⟩ [ , ⟨selection type⟩ ]* )

⟨method statement⟩ ::= ⟨selection⟩ [, ⟨selection⟩]* {
      name "⟨menu text⟩" ,
      method "⟨full method name⟩" [ ,
      enable "⟨enabler method name⟩" ]
}
```

Table 4.1: BNF definition of the GUI configuration file format.

The my_actions,cfg configuration file is sourced from the same directory as actions.cfg, and is where a user should put there own GUI tool ⟨method statement⟩ declarations. This ensures that if the repository updates the actions.cfg file you do not loose your personal configuration options.

## 4.5 Describing a Schema

Schemas of Schema.DATA_SOURCE_TYPE are normally created using a wrapper over a data source, and then schemas of Schema.VIRTUAL_TYPE type are created by applying transformations to schemas. However, if you want to write your own wrapper, or write a tool that

is designed to create new schemas independently from a data source (which should by of type Schema.STORED_TYPE or Schema.MATERIALISED_TYPE), then you need to understand how the API may be used to describe a schema. The first step is to give the schema a name and create it using the Schema.createSchema method as follows:

```
if (force)
    Schema.retract("mdr");
Schema s=Schema.createSchema("mdr");
```

Note, that Schema.createSchema will throw an IntegrityException if a schema with the name mdr already exists. To prevent this occurring, we preceded the attempt to create the schema with a call to Schema.retract if the boolean flag force is set. This will remove any schema with that name (and all dependent schemas). A schema is **dependent** on another schema if former schema is generated by the application of Transformation.applyAddTransformation, Transformation.applyDeleteTransaction *etc*methods (in the Transformation class to the later schema.

The you need to obtain references to the Construct instances that model the type of the SchemaObjects which represent the objects in the schema. Suppose we want to create the ER model of the MDR shown in Figure 4.1 using the er Model created by the example application DefineModels (q.v. Section 2.1). Then we would need to get the Model by referencing by its name using Model.getModel, and then get associated Constructs by using Construct.getConstruct.

```
Model er=Model.getModel("er");
Construct ent=er.getConstruct("entity");
Construct att=er.getConstruct("attribute");
Construct rel=er.getConstruct("relationship");
```

The SchemaObject class models various objects within a schema, and are created by calls to Schema.createSchemaObject, which takes as its first argument an instant of Construct that defines the type of the SchemaObject, and its second argument an Object array containing the **full scheme** of the object. For example, to create the model and construct entities, and its attributes mid and name would require the following calls:

```
SchemaObject model=s.createSchemaObject(ent,new Object[]{"model"});
SchemaObject construct=s.createSchemaObject(ent,new Object[]{"construct"});
```

Once the entities have been created, attributes on entities can be created, which in their scheme will reference the entity SchemaObject instances created above. The definition of the er model will be described in Section 4.6, and will show that the first field of the scheme of attributes is defined to always reference SchemaObject that was created with an entity Construct.

```
s.createSchemaObject(att,new Object[]{model,"mid","key"});
s.createSchemaObject(att,new Object[]{model,"name","notnull"});
s.createSchemaObject(att,new Object[]{construct,"cid","key"});
s.createSchemaObject(att,new Object[]{construct,"name","notnull"});
```

A relationship between entities is created in a similar manner to attributes, in the sense that the scheme must name two entity SchemaObject instances, followed by the two cardinality constraints:

```
s.createSchemaObject(rel,new Object[]{"",construct,model,"1:1","0:N"});
```

## 4.6 Describing a Modelling Language

For each modelling language, an instance of the Model class should be created. For example, the following creates a modelling language called er:

```
Model=Model.createModel("er");
```

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:

```
Construct entity=er.createConstruct("entity",Construct.CLASS_NODAL,true);
entity.addNodeNameScheme();
```

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:

```
Construct attribute=er.createConstruct("attribute",Construct.CLASS_LINK_NODAL,true);
attribute.addReferenceScheme(entity);
attribute.addNodeNameScheme();
attribute.addConstraintScheme(false);
```

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 define a binary relationships construct scheme as follows:

```
Construct relationship=er.createConstruct("relationship",Construct.CLASS_LINK,true);
relationship.addEdgeNameScheme();
relationship.addReferenceScheme(entity,2,2);
relationship.addConstraintScheme(2,2,false);
```

Note that the numbers in the last two method calls specify a lower and upper bound on the number of occurrences of the item being added to the scheme. Hence the above states that there is one name of an edge, followed by two references to entity constructs, followed by two constraint expressions.

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 generalisation in a ER model places a restriction on entities such that the extent of one is a subnet of the extent of another. This would be defined by:

```
Construct generalisation=er.createConstruct("generalisation",Construct.CLASS_CONSTRAINT,true);
generalisation.addLabelScheme();
generalisation.addLabelScheme(false);
generalisation.addReferenceScheme(entity);
generalisation.addReferenceScheme(entity,2,0,false);
```

Note that in the last Construct.addReferenceScheme, the use of an upper bound value of 0 being less than the lower bound value of 2 means that the repository will not enforce any upper bound on the number of subclass entities that can appear in the generalisation.

### 4.6.1 Alternatives and Sequences

Apart from the four basic types of construct outlined above, the AutoMed API supports two more constructs that allow for the accurate representation of more complex modelling languages. These are virtual constructs, which may be used in the addReferenceScheme method to specify patterns that may occur in the scheme of a construct, but may not be used to create a SchemaObject instant.

In the ER modelling so far defined, it is assumed that attributes are only placed on entities. If we wanted to allow attributes to be also placed on relationships, we have the problem that the Construct.addReferenceScheme needs to reference more than one type of Construct. This may be

achieved by defining an **alternation** Construct, which can not be instantiated in any schema, but can be used in references to constructs to represent a choice of different constructs, as the following code fragment illustrates.

```
Construct entityOrRelationship=eer.createConstruct("entity_or_relationship",
    Construct.CLASS_ALTERNATION,false);
Construct attribute=eer.createConstruct("attribute",Construct.CLASS_LINK_NODAL,true);
attribute.addReferenceScheme(entityOrRelationship);
attribute.addNodeNameScheme();
attribute.addConstraintScheme(false);
```

The entityOrRelationship alternation construct is used the place of the attribute definition where we want a reference to either an entity or relationship appear in the scheme of the attribute.

The **sequence** construct allows a list of construct references, labels and constraints to be handled as a single unit. This is useful when one wishes to ensure that the list occurs either in its entirety or not at all. For example, if we wanted to change our binary ER relationship construct to instead be an $n$-ary relationship, it might be thought that we could change the last two scheme definitions to be:

```
relationship.addReferenceScheme(entity,2,0);
relationship.addConstraintScheme(2,0,false);
```

indicating that there is no upper bound on the number of entities referenced, and cardinalities. However, this would make the scheme ⟨⟨worksin,person,dept,1:1,0:N,1:1⟩⟩ a valid scheme, even though the number of cardinality constraints does not match the number of entities referenced. Instead we should create a sequence that contains both an entity reference and cardinality constraint. This could be defined as follows:

```
Construct relationshipTarget=eer.createConstruct("relationship_target",
    Construct.CLASS_SEQUENCE,false);
relationshipTarget.addReferenceScheme(entityOrRelationship,1,1);
relationshipTarget.addConstraintScheme(1,1,false);
```

This may then be used in the definition of relationships by modifying the scheme definitions to be as follows:

```
relationship.addEdgeNameScheme();
relationship.addReferenceScheme(relationshipTarget,2,0);
```

This alternative definition allows for a scheme of the form ⟨⟨worksin,person,1:1,dept,0:N⟩⟩ but would not allow ⟨⟨worksin,person,1:1,dept,0:N,1:1⟩⟩ since the last constraint does not appear after a reference to an entity.

## 4.7 Customising the Appearance of a Model

The AutoMed GUI has a default representation of SchemaObject instances which uses the HDM associated with Construct type to determine the appearance of the SchemaObject as follows:

- Construct.CLASS_NODAL types are represented by a rectangular box, with the node name drawn within.
- Construct.CLASS_LINK_NODAL types are represented by a small circle, with a line connecting the circle to any types referenced by the link nodal scheme, and the name of the node drawn on the opposite side of the circle to the first of these lines.
- Construct.CLASS_LINK types are represented by a diamond shape, with a line connecting the diamond to any types referenced by the link scheme.
- Construct.CLASS_CONSTRAINT types are represented by a dashed rounded rectangle, with

the first label of the constraint placed inside the rectangle, and with a line connecting the rounded rectangle to any types referenced by the constraint scheme.

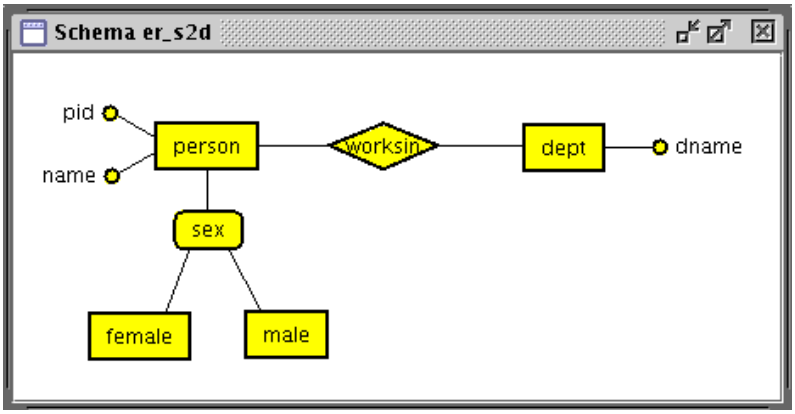



Figure 4.3: AutoMed GUI Tool viewing schema er_s2d using the default Drawable implementations

Figure 4.3 illustrates the appearance of all four of these default appearances. Whilst this gives a 'pseudo-ER' appearance to the diagrams, and is adequate for basic viewing of models, it will invariably be the case that not all information in the modelling language is presented. Even the 'pseudo-ER' diagram omits the cardinality constraints on the relationships in the example ER modelling language, and the cardinality of attributes, and draws generalisations in an ambiguous manner. The AutoMed API allows a developer to implement custom classes for drawing the objects in a modelling language, and store the names of those classes in the repository, such that the GUI tool then uses them for drawing schemas. Figure 3.3 illustrates the classes shipped with the AutoMed GUI to enhance the appearance of ER diagrams.

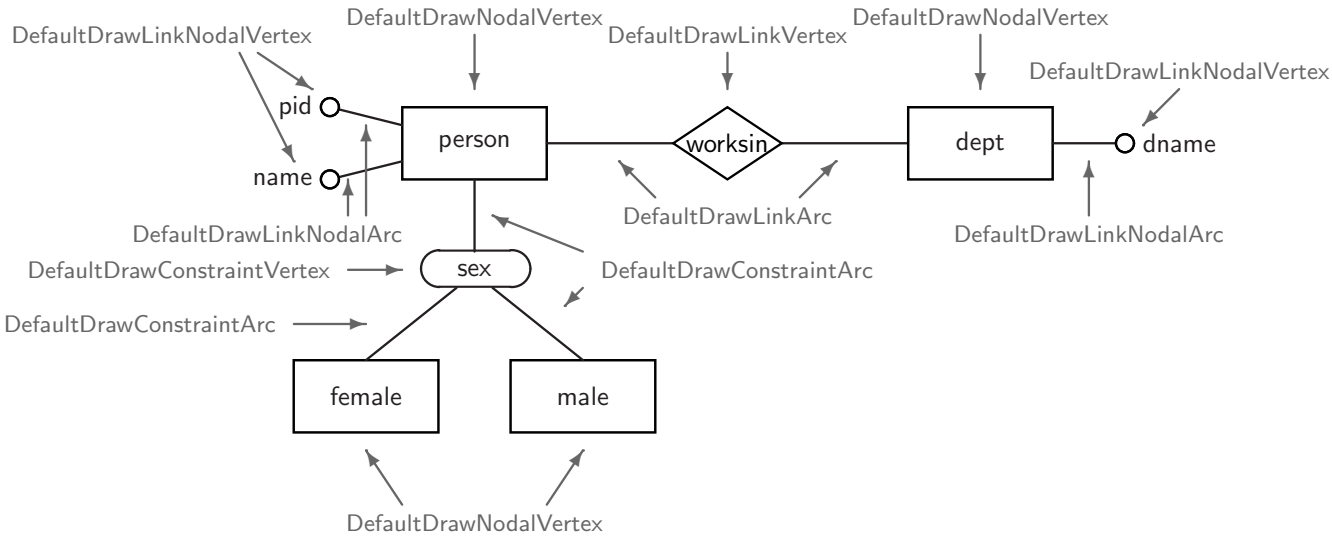



Figure 4.4: Default Drawable implementations used to paint AutoMed SchemaObjects

The Gui in the uk.ac.doc.ic.automed.editor package conceptualises each object it needs to draw as being an implementation of the Positionable interface, that the repository classes Network, Schema and SchemaObject each implement. Each appearance of a Positionable in a EditorPanel requires that the Positionable be wrapped inside a Drawable implementation. Each SchemaObject based on a Construct that is not of nodal type will have two graphical components: a vertex and several arcs connecting that vertex to other constructs. Constructs that are of nodal type have just the vertex. These vertex and arcs are drawn by implementations of the DrawableVertex and

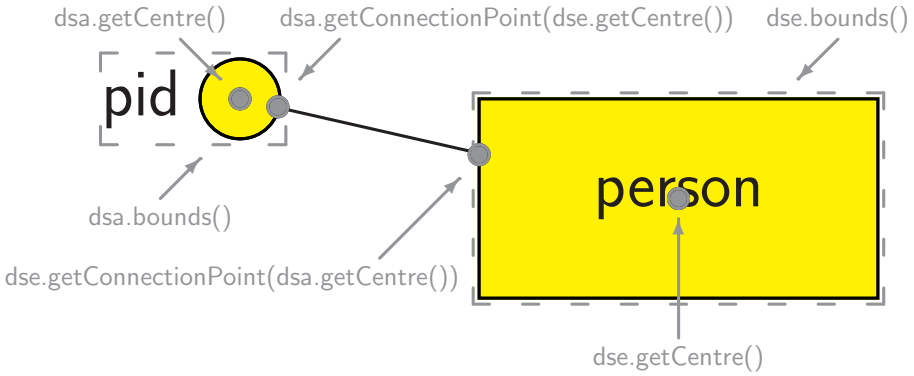
| Construct type | DrawableVertex implementation | DrawableArc implementation |
|---|---|---|
| CLASS_NODAL | DefaultDrawNodalVertex | N/A |
| CLASS_LINK_NODAL | DefaultDrawLinkNodalVertex | DefaultDrawLinkNodalArc |
| CLASS_LINK | DefaultDrawLinkVertex | DefaultDrawLinkArc |
| CLASS_CONSTRAINT | DefaultDrawConstraintVertex | DefaultDrawConstraintArc |

Table 4.2: Default Classes used for drawing Constructs

DrawableArc interfaces. The default implementations used for various construct types are listed in Table 4.2, and used to annotate the schema objects shown in Figure 4.4. Whilst it is possible to write your own replace implementations of these two interfaces, it is recommended that instead you extend two classes DrawVertex and DrawArc, overriding methods as necessary. All the classes in Table 4.2 extend these two classes, and if you wish your construct to appear similar to one of these, you might consider overriding one of these default implementations.

The methods of DrawVertex that you might wish to override are:

- DrawVertex.getBounds() returns the Rectangle within which you want your object to appear. You should use the Positionable.getPosition() method on the Positionable p to calculate this position.
- DrawVertex.getCentre() returns the Point which is the absolute location of Positionable.getPosition() relative to the top-left corner of the DrawVertex.getBounds() rectangle.
- DrawVertex.getCentreOffset() returns the Point which is the location of Positionable.getPosition() relative to the top-left corner of the DrawVertex.getBounds() rectangle.
- DrawVertex.setText sets what text should be used to label the vertex. The default implementation uses the Positionable.getLabel().
- DrawVertex.paint draws the object in a Graphics2D canvas.
- DrawVertex.getConnectionPoint takes a Point as an argument, and calculates when a line representing an arc from that point to the graphical centre of the drawable should stop.




```
Schema s=Schema.getSchema("er_s2d");
SchemaObject se=s.getSchemaObject(⟪person⟫);
SchemaObject sa=s.getSchemaObject(⟪person,pid⟫); Gui gui=Gui.openMainWindow();
gui.createEditorFrame(s);
DrawableVertex dse=(DrawableVertex)gui.getDrawable(se);
DrawableVertex dsa=(DrawableVertex)gui.getDrawable(sa);
```

Figure 4.5: Points and shapes computed by DrawVertex methods

Figure 4.5 illustrates the Points and Rectangle returned on an example schema. Note that the result of getConnectionPoint() is used by the default implementation of DrawArc to decide where it should draw the arc connecting nodes in the diagram.

The following illustrates a complete example of a Drawable implementation that enhances the representation of attributes in our ER modelling languages, by putting a question mark after any nullable attribute, and a asterisk after a multi-valued attribute.

```
public class DrawERAttributeVertex extends DefaultDrawLinkNodalVertex {
    public DrawERAttributeVertex(EditorPanel ep,Positionable schemaObject,
            Positionable d[]) {
        super(ep,schemaObject,d);
        Object s[]=((SchemaObject)schemaObject).getSchemeDefinition();
        String card=s[2].toString();

        if ("null".equals(card))
            text+="?";

        if ("multi".equals(card))
            text+="*";

        setText(text);
    }
}
```

To make the GUI use this method for drawing a SchemaObject called attribute, a call to the Construct.setVertexDrawable method should be made as follows:

```
attribute.setVertexDrawable(DrawERAttributeVertex.class.getName());
```

There is a analogous Construct.setArcDrawable method to supply a new DrawableArc implementation. Note that the Drawable implementation must be in your CLASSPATH both when you compile your application, and when you run the GUI.

## 4.8 Writing an AutoMed Wrapper

Implementing an AutoMed Wrapper for a particular type of date source requires that:

1. You design an instance of Model and associated instances of Construct that can be used as a basis to represent the schema of the data source. If the data source modelling language is very different from the standard AutoMed representation of languages, you might create two models, one source oriented, and one AutoMed oriented.
2. You write an extension of AutoMedWrapperFactory, that includes methods to create the instances you designed in (1), and which can also use either data source metadata and/or input from an expert user to create instances of SchemaObject to represent the schema of a particular data source.
3. You write an extension of AutoMedWrapper that can connect to the data source, and which can return the extent of any nodal, link-nodal or linking SchemaObject in the Schema associated with the data source. All persistent information about a wrapper must be held in AccessMethod instance. Thus an AutoMedWrapper should be created by giving either the connection details for a data source not used before, or by giving the AccessMethod for a data source that has previously been wrapped.

In general, one wrapper may map a data source into AutoMed is a variety of ways, and may or may not model certain features of the data source. This is handled in AutoMed wrappers by the concept of **features**, which is a bitmap what indicates which features are on or off in a particular wrapper instance. The features should be

1. Described by static constants in your extension of AutoMedWrapperFactory, numbered 1,2,4,8,...

2. Recorded in the AccessMethod created by the wrapping process
3. Used by your AutoMedWrapper in deciding how it interacts with the data source when running queries and updates.

The purpose of separating functionality into AutoMedWrapper and AutoMedWrapperFactory is that each instance of AutoMedWrapperFactory may read several schemas of several data sources in a consistent manner, and create several AutoMedWrapper and associated Schema instances which represent several data sources using the same data modelling language. Thus features should only be settable on an AutoMedWrapperFactory before it has been used to map its first schema. After this, any attempt to set the features will throw an IntegrityException, provided the implementation of AutoMedWrapperFactory sets the protected variable factoryInUse after creating its first schema.

### 4.8.1 Extending AutoMedWrapper

To allow users to call AutoMedWrapper.newAutoMedWrapper with appropriate parameters, and have your wrapper class selected, your wrapper should register itself with AutoMedWrapper using the AutoMedWrapper.registerWrapper, by including a static initialiser. For example, the Transact-SQLWrapper registers itself by:

```
static {
    AutoMedWrapper.registerWrapper(new TransactSQLWrapper()).getClass(),
        "com.microsoft.jdbc.sqlserver.SQLServerDriver",null);
}
```

To know that your wrapper class is loaded when the GUI is running (and therefore it will be offered as an option to users of the GUI), you should ensure that it is loaded in your data_source_repository.cfg or one of its included files. For example, the reps_schema.cfg included by default in data_source_repository.cfg contains the following line to load PostgresWrapper:

```
load "uk.ac.ic.doc.automed.wrappers.PostgresWrapper"
```

To create a wrapper, it is expected that each wrapper implementation provides a factory method AutoMedWrappernewAutoMedWrapper that given details of a date source, attempts to connect to it, and if given the schema name, calls the AutoMedWrapper.getSchema method to create the Schema that represents the data source. If the schema name is not given, then

A wrapper for a data source stores is configuration in the repository class AccessMethod. Every subclass of AutoMedWrapper is expected to implement a constructor method that takes an AccessMethod and String password. For example, the Postgres wrapper has the constructor:

```
public PostgresWrapper(AccessMethod am, String password)
throws DataSourceException {
    this.am=am;

    if (password==null)
        this.password=am.getPassword();
    else
        this.password=password;

    url=am.getURL();
    s=am.getSchema();
    username=am.getUsername();
    driver=am.getDBDriver();
    awf=new PostgresWrapperFactory();
    try {
        awf.setFeatures(am.getAutoMedWrapperFactoryFeatures());
    }
    catch (Exception e) {
```

```
        throw InconsistentException.newInconsistentException("Unable to set "+
            "features of brand new WrapperFactory! ",e);
        }
        connect();
        cacheWrapper();
}
```

The test on the password allows the user to override the password held inside the AccessMethod. Then the details of the AccessMethod are unloaded into variables provided in AutoMedWrapper for the purpose. The call to the AutoMedWrapper.connect method (which must be supplied within the wrapper implementation) then opens the data source, and finally AutoMedWrapper.cacheWrapper is called to ensure that this wrapper instance is always returned by methods that attempt to locate wrappers by reference to AccessMethod. Apart from constructors, AutoMedWrapper.connect is one of four methods that the wrapper implementation must provide:

- AutoMedWrapper.connect should ensure that the driver (if any) used to assess the data source can be loaded, and that the data source can be opened for reading using any username and password supplied. The implementation of connect should only use the password, username, url, and driver fields on AutoMedWrapper, and not the AutoMedWrapperFactory which might to be present at the time connect is called. This allows features of the AutoMedWrapperFactory to be changed before the schema is populated.
- AutoMedWrapper.getProtocolName must return the name of the protocol associated with this wrapper. This is used only for identification purposes, and does not (at present) impact on the operation of the system. For example, the PostgresWrapper and TransactSQLWrapper wrappers both return jdbc to indicate their reliance on the JDBC protocol.
- AutoMedWrapper.executeIQL must take an ASG containing a single scheme, and return an ASG representing the list associated that represents the content of the data source corresponding to that scheme.
- AutoMedWrapper.getDefaultWrapperFactory return an instance of the AutoMedWrapperFactory that is designed to populate Schema for the wrapper, and also create Model and associated Constructs those schemas will be described in.

### 4.8.2 Extending AutoMedWrapperFactory

The AutoMedWrapperFactory.getModel method should create a new instance of the Model your wrapper needs only if that Model is not already present in the repository. Otherwise it should return the existing Model instance. Hence in outline, a getModel will follow the following structure:

```
try {
        sql=Model.getModel(modelName);
}
catch (NotFoundException e) {
        sql=Model.createModel(modelName);
        table=sql.createConstruct(...);
        ⋮
}
```

The AutoMedWrapperFactory.getAutoMedModel method should similarly create or return the AutoMed oriented schema if your wrapper has one, or if your wrapper does not, just return the return the result of getModel.

The AutoMedWrapperFactory.populateSchema should be passed a AutoMedWrapper which has already has its AutoMedWrapper.connect method called, and had an empty Schema created. The AutoMedWrapperFactory.populateSchema method then reads in the schema of a data source, and creates SchemaObject instances to represent the data source schema.

The AutoMedWrapperFactory.getFeatureNames method should return an array of strings, containing the names of features that may be used in this wrapper. The elements of the array correspond to a bit map that is used as an integer to AutoMedWrapperFactory.setFeatures: the first array element should be a feature that has a static constant 1 associated with it, the second a static constant 2, the third a static constant 4, *etc.*

# Chapter 5

# AutoMed Tools

The standard AutoMed distribution comes with a number of tools to assist in performing data integration using the AutoMed repository. In this chapter, we detail two tools, the first providing a mechanism to automate the integration of schemas, and the second to allow for P2P exchange schemas, pathways and queries between several AutoMed repositories.

## 5.1 Schema Matching and Merging

In **model management** [Ber03], the Match operator determines what are the semantic relationships between SchemaObject instances in different schemas, and the Merge operator uses these semantics relationships to combine two Schemas into a single new Schema.

Since AutoMed is much more precise in its specification of the relationship between schemas, it follows that the AutoMed Merge operator requires a more precise description of semantic relationships from the Match operator than is found in other approaches. The semantic relationships supported by AutoMed Match [Riz04] compare two SchemaObjects $\langle\langle A \rangle\rangle$ and $\langle\langle B \rangle\rangle$ from two Schemas $S_A$ and $S_B$ and determine what is the semantic relationship between the two schema objects. We say that two schema objects are **compatible** $\langle\langle A \rangle\rangle \overset{S}{\sim} \langle\langle B \rangle\rangle$ if it is always the case that (1) when the same data value appears in the extent of both schema objects then it represents the same real world concept, and (2) if the data values differ, then they represent different real world concepts. From this definition it follows that if we had a SchemaObject $\langle\langle person \rangle\rangle = [1, 2, 3]$ in one schema representing the identifiers of students, and a SchemaObject $\langle\langle dept \rangle\rangle = [1, 2, 3]$ in another schema representing identifiers of departments, then we would that $\langle\langle person \rangle\rangle$ and $\langle\langle dept \rangle\rangle$ are **incompatible**. By contrast if we had the same $\langle\langle person \rangle\rangle$ SchemaObject in one schema, and in another schema a SchemaObject $\langle\langle staff \rangle\rangle = [1, 3, 5]$, where the value 1 is both cases represented the same person 'Alex', value 2 represents person 'Dimitri' who only appears in the first schema, value 3 represented 'Mike' in both, *etc*, then we say the schema objects are **compatible**.

For compatible schema objects we identify four subclasses of compatibility, giving the five semantic relationships listed below:

1. **equivalence** $\langle\langle A \rangle\rangle \overset{S}{=} \langle\langle B \rangle\rangle$ holds iff $\langle\langle A \rangle\rangle \overset{S}{\sim} \langle\langle B \rangle\rangle$ holds, and it is semantically correct to say that at all times $\langle\langle A \rangle\rangle = \langle\langle B \rangle\rangle$

2. **subsumption** $\langle\langle B \rangle\rangle \overset{S}{\subset} \langle\langle A \rangle\rangle$ holds iff $\langle\langle A \rangle\rangle \overset{S}{\sim} \langle\langle B \rangle\rangle$ holds, $\langle\langle A \rangle\rangle \overset{S}{=} \langle\langle B \rangle\rangle$ does not hold, and it is semantically correct to say that at all times $\langle\langle B \rangle\rangle$ — $\langle\langle A \rangle\rangle = []$

3. **intersection**: $\langle\langle A \rangle\rangle \overset{S}{\cap} \langle\langle B \rangle\rangle$, iff $\langle\langle A \rangle\rangle \overset{S}{\sim} \langle\langle B \rangle\rangle$ holds, $\langle\langle A \rangle\rangle \overset{S}{=} \langle\langle B \rangle\rangle$, $\langle\langle B \rangle\rangle \overset{S}{\subset} \langle\langle A \rangle\rangle$, $\langle\langle A \rangle\rangle \overset{S}{\subset} \langle\langle B \rangle\rangle$ do not hold, and it is semantically correct to say that at some time count $[x \mid x <- \langle\langle A \rangle\rangle; x <- \langle\langle B \rangle\rangle] > 0$ and that at all at times there exists a real world concept which we can represent by a schema object $\langle\langle C \rangle\rangle = [x \mid x <- \langle\langle A \rangle\rangle; x <- \langle\langle B \rangle\rangle]$

4. **disjointness** $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\not\cap} \langle\langle \mathsf{B} \rangle\rangle$ iff $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\sim} \langle\langle \mathsf{B} \rangle\rangle$ holds, and $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{=} \langle\langle \mathsf{B} \rangle\rangle$, $\langle\langle \mathsf{B} \rangle\rangle \overset{\mathrm{S}}{\subset} \langle\langle \mathsf{A} \rangle\rangle$, $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\subset} \langle\langle \mathsf{B} \rangle\rangle$, and $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\cap} \langle\langle \mathsf{B} \rangle\rangle$ do not hold

5. **incompatibility** $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\not\sim} \langle\langle \mathsf{B} \rangle\rangle$ iff $\langle\langle \mathsf{A} \rangle\rangle \overset{\mathrm{S}}{\sim} \langle\langle \mathsf{B} \rangle\rangle$ does not hold.

Several properties of the semantic relationships should be noted. Firstly, it is clear from the definitions that any pair of schema objects must fall into exactly one of the five types of relationships. Secondly, the phrase 'semantically correct to say' allows any particular extent of the schemas to violate the IQL constraint. For example, is if two schemas contain the schema objects $\langle\langle \mathsf{person} \rangle\rangle$ and $\langle\langle \mathsf{staff} \rangle\rangle$ that are supposed to represent *all* staff members of a university, then we would say that $\langle\langle \mathsf{person} \rangle\rangle \overset{\mathrm{S}}{=} \langle\langle \mathsf{staff} \rangle\rangle$, even if at any given time there might be some staff members appearing in one schema and not the other, due to asynchronous updates or errors in updates to the schemas.

In general, determining the semantic relationships between schema objects can only be fully determined by a domain expert. However, certain properties of the schema and of the data associated with the schema may be used to get an approximation of the semantic relationship, which is what the AutoMed schema matching tool attempts to achieve. As distributed, the tool has a number of modules examining different properties as follows:

- Data-Type: the data type of schema objects are compared, and if both have the same type then the schema objects are regarded as equivalent..
- Statistics: for String data types, the frequency occurrence of different characters are compared, and if similar, the schema objects are regarded as equivalent.
- Name: the naming of the schema objects are compared. If the strings are similar, then the schema objects are regarded as equivalent, if one has a name that is a substring of a second schema object, then the first is regarded as being a subset of the other.
- Number of Instances: the number of instances in the data source are compared, and the smaller number indicates which is a subset of the other.
- Existence: A sample of instances is taken from each data source, and tested for membership of the other, giving indications of equivalence, subset or overlapping.
- Naive-Bayes: A Naive-Bayes algorithm is used on the data sources.
- Precision: the minimum and maximum length of strings or value of numbers in the data source are compared. If the intervals are the same, then the schema objects are equivalent, if one interval is a subset of another then one schema object is a subset of the other, and if the intervals overlap, then the schema objects overlap.

There is an API for matching modules, which allows for new modules to be written and added into the tool. The tool aggregates the rusults of all the modules, and suggests which semantic relationship is most likely to exist between pairs of schema objects from different schemas. The user may then correct the tools choices, before using the tool to also perform a Merge of the schemas, using techniques described in [RM05].

### 5.1.1 Example of using the Match Tool

The schema matching tool requires that you first select two schemas that you wish to match, using the same method as you would if you wish to add an ident transformation between then (*i.e.* by clicking on one schema in a network window, and then control-clicking on a second schema). You should then bring up the menu and select Discover semantic relationships. For example, in the university example, the application described in Section 3.2 wraps two data sources in networks uni_s4_src and uni_s5_src but leaves them unintegrated. Opening up these two networks, selecting uni_s4, control-selecting uni_s5, and then selecting Discover semantic relationships results in the window illustrated in Figure 5.1, listing all the compatible semantic relationships that the tool

has found. (The tool will take sometime to run depending on how many modules it has been configured to use).

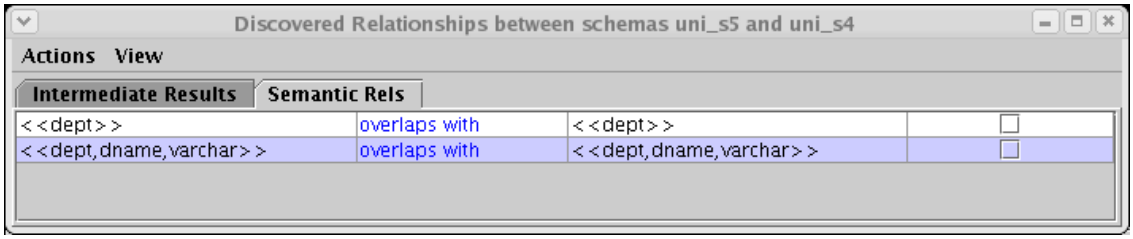



Figure 5.1: Schema matching and merging tool

Note that in this example, the modules have given a resonable solution, identifying that the dept tables of the two schemas overlap. If you disagree which the tools choice of semantic relationship, then you may change it by clicking the menu over the current setting. Choosing anything apart from the tools choice of semantic relationship will result in the semantic relationship being displayed in black rather than blue. If the tool has entirely missed a compatible pairing, then you should select the Actions tool bar menu option Add Mapping option to add it it. You may use the View tool bar menu to activate the viewing of the results of different modules, and hence discover which module(s) contribute to the tool reaching a certain decision.

If you look at the instances of the ⟨⟨dept⟩⟩ tables (using the IQL query tool), you will find that the instances are disjoint, but we can assume that the two tables may share values in the future. Tick both of the semantic relationships as being correct, and then use the Action tool bar menu option to select Merge Schemas. This will create the necessary transformations in the repository to map between the two schemas, and the two networks are replaced by a single new one.

## 5.2 Peer-to-Peer Data Integration

The term **peer-to-peer** (**P2P**) is used to describe computer systems where computers interact with each other as equals, and there is no notion of a central server to control the interaction. The AutoMed P2P data integration implementation [Laz05] supports this notion by allowing separate AutoMed systems to publish the fact that they have a pathway from a set of data sources to one or more **public schemas** [MP03]. As illustrated in Figure 5.2, the logical view of a AutoMed P2P data integration is that a **peer** $P_n$ will integrate a number of data sources $DS_a, DS_b, \ldots$ into some global schema $GS_n$, and then transform that global schema into one or more public schemas $PS_s, PS_t, \ldots$. When two peers $P_x, P_y$ share a public schema $PS_s$, then they will have a pathway mapping between the data sources at each peer. For example, in Figure 5.2, the fact the $P_1$ and $P_2$ share $PS_2$ means that taking the combined knowledge of the two peers together, there is a pathway from $DS_3$ to both $DS_1$ and $DS_2$. These logical associations may be chained together. In the figure, the fact that $PS_3$ is shared by $P_2$ and $P_3$ means that combining the knowledge of $P_1, P_2$ and $P_3$ gives a pathway from $DS_1$ to $DS_4$, $DS_1$ to $DS_5$, *etc.*

Note that a data source may be accessible via very different peers. For example, $DS_4$ is accessible from peers $P_2$ and $P_3$. Also note that if just one schema is being mapped to public schemas, then no global schema is necessary. For example, $DS_6$ on $P_4$ is mapped directly to two public schemas $PS_1$ and $PS_4$. Finally, also note that a peer may store information about several data sources without itself integrating those data sources. For example, $P_4$ has pathways from $DS_6$ and $DS_7$ to public schemas, but does not integrate them together. However, the fact that $DS_7$ has a pathway to $PS_3$ and $DS_6$ has a pathway to $PS_4$ means that the knowledge of $P_3$ combined with $P_4$ does give an integration of $DS_6$ and $DS_7$.

This flexible method of describing the logical associated of peers to data sources and public schemas may be used in two operational ways of interaction between peers:

- $P_x$ may send a query on $PS_s$ to peer $P_y$, requesting that $P_y$ evaluate the result. $P_y$ may

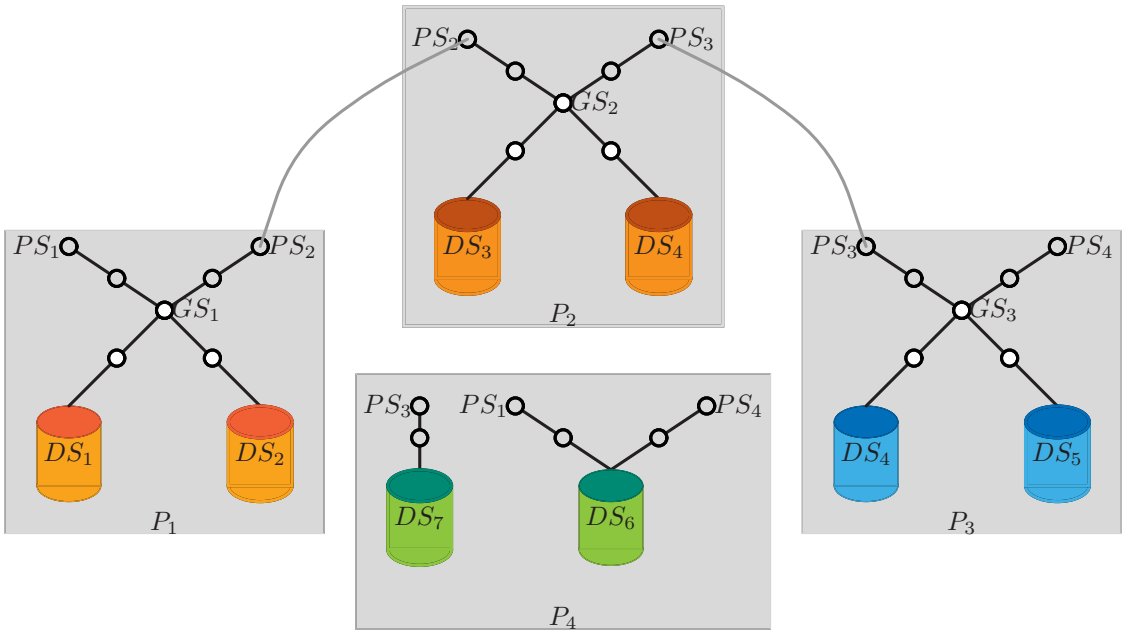



Figure 5.2: Public Schemas in a P2P Data Integration Environment

then transform the query into one on its local data sources, and pass back the results to $P_x$. For example $P_1$ may execute queries, evaluating the results against $DS_1$ and $DS_2$. It may then ask for a remote execution of the query on $P_2$, and obtain an answer from $DS_3$.

In addition, $P_y$ may act as a **broker**, passing on the query to any other peers it knows support $PS_s$, or any other public schemas $PS_t$ that $P_y$ has a pathway between $PS_s$ and $PS_t$. For example, the query from $P_1$ having been passed to $P_2$ may be transformed into a query on $PS_3$, and then the logical association with $P_3$ used to then pose the query on $PS_3$ on $P_3$, and obtain answers from $DS_4$ and $DS_5$.

- $P_x$ may request that $P_y$ send the pathways from $PS_s$ to the local data sources of $P_y$ over to $P_x$. Peer $P_x$ may store these pathways in its local repository, and evaluate queries directly of the remote data sources.

  For example, $P_1$ may ask for the pathway on $P_2$ from $PS_2$ to $DS_3$, and add that information into its local repository. This means it may then perform the translation of queries from $PS_2$ to $P_2$, placing no load on $P_2$

To implement these two methods of interaction, the architecture in Figure 5.3 has been developed and implemented. The components have the following purpose:

1. In order that peers may discover other peers that share common public schemas, the P2PDirectory provides a central record of which peers exist on the network, and which public schemas those peers implement. It also records which host a named peer last appeared at.

2. In order that a number of peers can run on a single host, yet all peers on the network need only know from the directory which host a peers runs on, the P2PRegistry provides access to all the AutoMed peers on the same host as it runs. It executes on what should be a standard port (and hence 'well known', defaulting to 8282) throughout the AutoMed P2P network. All requests from other peers are sent to this standard port number.

   Note that the combination of P2PDirectory and P2PRegistry allow for a peer to migrate from one host to another. The peer would simply logoff one P2PRegistry and login to the P2PRegistry of another host, which in turn would inform P2PDirectory of the peer's new location.

3. To provide a interface that allows two AutoMed repositories to interact, the AutoMedPeer provides a set of messages that can be exchanged with other AutoMed peers, containing

information about schemas, pathways, and requests to execute queries. There should be one AutoMedPeer instance for each AutoMed repository in the P2P network, and each AutoMedPeer should have a name that is unique to that peer.

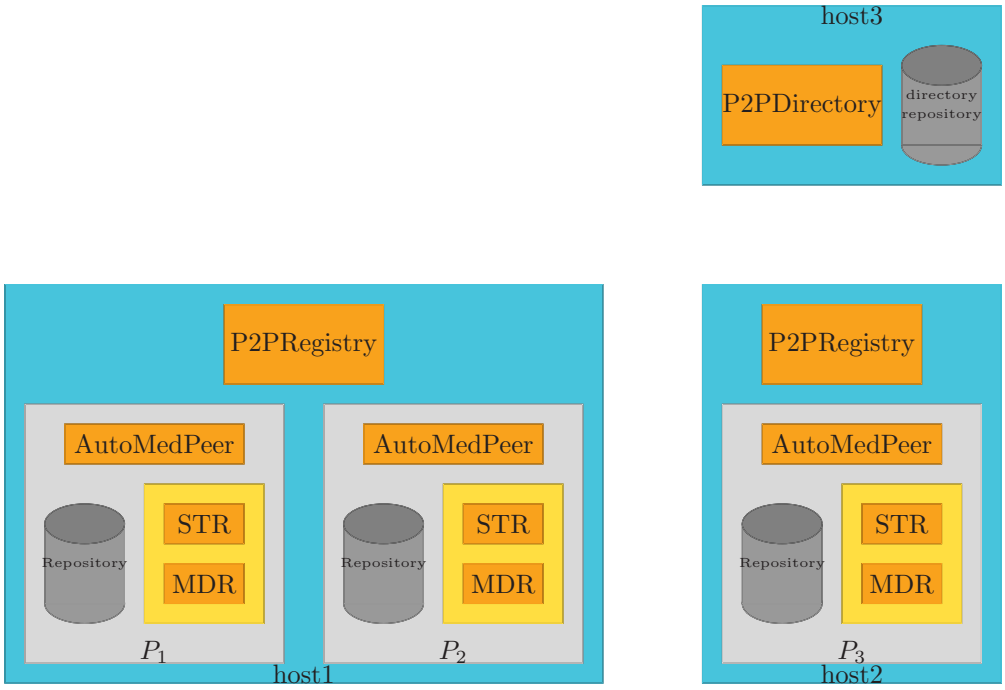



Figure 5.3: The AutoMed P2P Architecture

### 5.2.1 Configuring and Running the Directory Service

The AutoMed P2P directory requires the following configuration files for setup and operation:

- p2p_directory_repository.cfg file to setup the database tables used by the directory
- p2p_directory_server.cfg file to run as a background process. This file includes settings from another file called p2p_common.cfg.

Note that these configuration files are automatically created in your .automed directory with default settings the first time their associated programs are run. Normall, you do not need to do anything with these files unless you want to customise them for your environment.

```
//
// AutoMed P2P Directory Repository
//
JdbcURL jdbc:postgresql://host3.example.net/automedp2p
Username automed
Password secret
JdbcDriver org.postgresql.Driver
```

Figure 5.4: Example p2p_directory_repository.cfg file, where the directory repository is to be held on the local host, in a Postgres database called automedp2p

The AutoMed P2P directory needs a JDBC accessible database in order to hold the persistent information for the directory. An example repository configuration file is illustrated in Figure 5.4, which should be present in your .automed directory (q.v. Page 2). This configuration file is created the first time the directory initialiser program is run by the command:

java uk.ac.ic.doc.automed.p2p.directory.P2PDirectoryRepositry

```
//
// AutoMed P2P Directory and Registries
//
//UDP settings for server
//the thread's wait time (in seconds)
udpWaitTime 0
// the gap time (in seconds) between two successive runs of a thread
udpThreadInterval 2
numberOfUDPThreads 5

//other parameters
//the time (in milliseconds) during which a thread must wait for a partner thread to finish
threadWaitTime 50

// common settings
include p2p_common.cfg
```

Figure 5.5: Example p2p_directory_server.cfg file

Note if you had an existing AutoMed P2P directory installed, this would cause its repository tables to be cleared. Once the directory repository has been initialised, the P2PDirectory should be run as a background task. This will require a directory configuration such as that illustrated in Figure 5.5 to be also present in your .automed directory. The settings in this and the included p2p_common.cfg file (Figure 5.6) should be agreed to by all other AutoMed peers so that the port numbers for registry and directory services are consistent across the P2P network. These files are created the first time the directory program is executed by the command:

nohup java uk.ac.ic.doc.automed.p2p.directory.P2PDirectory

### 5.2.2 Running an AutoMed Peer

The AutoMed peer requires the following configuration files to run:

- p2p_peer.cfg file which includes settings from p2p_common.cfg file (Figure 5.6) for the peer's operations.

These files are automatically created with the default settings the first time their associated programs are run. These programs will be discussed below.

Before using the AutoMed P2P system on a particular host, you must ensure that there is a local P2PRegistry running on that host. To run a registry you execute the command:

nohup java uk.ac.ic.doc.automed.p2p.P2PRegistry

Once the registry is running, you may start AutoMed peer application by the command:

java uk.ac.ic.doc.automed.editor.Gui

This program initialises one AutoMedPeer class instance for the current **Java Virtual Machine** (**JVM**) to handle all communication with other AutoMed peers. To create it use the P2P tool bar menu, and select Login. Alternatively, you may directly run the standalone application:

java uk.ac.ic.doc.automed.p2p.gui.PeerApplication

You should select a name that you would like to identify your peer by, and use it to join the P2P network, and login. The default suggested for you by the tool is a combination of the database username and URL of the database used for your STR repository, and should ensure that your peer is globally unique across any AutoMed P2P work.

```
//
// AutoMed P2P Peer and Directory common configuration
//
// TCP settings
serverHost flagstaff.doc.ic.ac.uk
serverPort 56056
registryPort 55055

// UDP settings
udpServerHost flagstaff.doc.ic.ac.uk
udpServerPort 56156

// packet buffer size esp. for TCP connections
//TCP settings
numberOfTCPThreads 5
timeOut 0
packetBufferSize 1000000

//timer values
// – helloTime = keep alive;
// – updateTime = time period during which data update occurs b/w peer and directory server
// – refreshTime = periodic update interval b/w peer and directory
// – holdTime = time to keep entries even after refresh period; flushTime = time
// – to wait before deleting entries after hold time period
helloTime 10
updateTime 15
refreshTime 30
holdTime 35
flushTime 40

//Peer status codes
psActive ac
psInactive in
psFlush fl
```

Figure 5.6: Example p2p_common.cfg file

```
//
// AutoMed P2P Peer configuration
//
//UDP settings for client
udpWaitTime 5
numberOfUDPThreads 1

//TCP settings
numberOfTCPThreads 1
timeOut 20000
packetBufferSize 100000

// cache age time (in seconds)
queryCacheAgeTime 30

// common settings
include p2p_common.cfg
```

Figure 5.7: Example p2p_peer.cfg file

### 5.2.3 Publishing Schemas and Obtaining Listings of Published Schemas

Once you have logged in to the P2P network, you can perform the following operations from the Directory option of the GUI P2P toolbar menu.

- Schema Info to list which peers have a copy of a public schema in their repositories, and send queries to those peers.
- Get Schema to copy a public schema into the local AutoMed repository
- Advertise Path to inform the directory that the local peer has a pathway from the public schema to one or more data sources.

### 5.2.4 Example of using the P2P System

Let us suppose that we have a situation where the three databases from the University database integration (q.v. Section 2.2) have been held separately by three different AutoMed repositories, and that the owners of these repositories wished to exchange schema, pathway and data with each other. To set this situation up, you need to remove the schemas from you repository if you them already loaded, which you may do by selecting schemas uni_s1_src, uni_s2_src and uni_s3_src in turn, and using the **retract** menu option on each one.

#### Setting up Peer 1 holding database university2

To configure the first peer, use the Wrap Data Source (q.v. 3.2) tool to wrap the second university database (with DoC pjm_university2, as discussed in Section 2.2). You should use the tool to both create a source oriented schema called uni_s2_src, and then to create an AutoMed oriented schema called uni_s2

Then use the GUI tool bar File menu option Import to load the university_staff.atm file held in the data directory. This will create a network and schema called university_staff, which is a public schema for the university database that has already been designed for you. (Of course, you could also design your own).

Select university_staff, and then control select uni_s2, and use the Discover semantic relationships menu option. This will bring up a list of semantic relationships that the schema match tool has determined. You should just tick off those pairs where the schemes are identical on left and right, before choosing Actions menu option Merge Schemas.

Now choose the tool bar GUI P2P menu option Directory, click on the Publish Schema button, and enter the schema name university_staff and give it a description (such as 'details of university staff members and their departments'). This will publish the schema onto the P2P network. Clicking the Refresh button will get this information back from the directory.

Now select university_staff entry in the list, and click the Advertise Path button. This informs the directory that the peer has a path of a data source for that public schema.

#### Setting up Peer 2 holding database university3

To run the second peer, you need a second AutoMed repository. If you run under a different username (with a different URL for the repository) then you will have no conflicts with Peer 1 that we setup above. However, if you run under the same username, then the same data_source_repository.cfg will be used. To avoid this, then use the -c switch on the Gui application as in the following example:

```
java uk.ac.ic.doc.automed.editor.Gui -c  /.automed/anon.cfg
```

which will load anon.cfg instead of the standard data_source_repository.cfg. (The -h switch may be used to find all Gui options). If you are using the same database server for both repositories, then within anon.cfg you might also use the database username aliasing. For example:

Username:pjm anon

would cause the Linux user pjm to contact the repository database under username anon.

Once you have the second repository running, you should wrap university3 in a similar way to how Peer 1 wrapped university2, and login to the P2P network.

When you run the P2P directory viewer from this peer, you should see that university_staff is available as a public schema. Selecting it, and then clicking the Schema Info button will bring up a window listing the details of Peer 1. Selecting Peer 1 from the list, and then clicking the Query button will send a query over to Peer 1 for it to process.

Clicking on the Get Schema button will insert a copy of university_staff in Peer 2's repository. You may then match your local uni_s3 with that public schema in a similar way to how this was performed on Peer 1. The use the Directory view tool to Advertise Path.

Now Peer 1 will be able to query uni_s3.

# Appendix A

# Using the Software Under Other Environments

## A.1 Linux csh environments

To use the configuration instructions under a csh command line environment, you need only change the setting of environment variables to the csh equivalent. For example, under bash the variable setting

```
export MYVAR=MyValue
```

becomes under a csh:

```
setenv MYVAR MyValue
```

## A.2 Microsoft Windows environments

To use the configuration instructions under a Windows cmd command line environment, you need only change the setting of environment variables to the Windows equivalent. Also, any directory separators are changed from colons to semi-colons. For example, under bash the variable setting

```
export CLASSPATH=$AUTOMED/jar/automedRepositories.jar:$AUTOMED/jar/java_cup.jar
```

becomes under Windows:

```
set CLASSPATH=%AUTOMED%/jar/automedRepositories.jar;%AUTOMED%/jar/java_cup.jar
```

The AutoMed configuration directory .automed is placed in the users home directory.

# Appendix B

# Known Problems

- **AbstractMethodError Exceptions being throw**

  The Postgresql driver jdbc7.0-1.1.jar gives the error:

  ```
  Exception in thread "main" java.lang.AbstractMethodError
        at uk.ac.ic.doc.automed.Util.resultSetToArray(Util.java:338)
  ```

  If is recommended that you use the pgjdbc.jar driver supplied with the AutoMed distribution in the $AUTOMED/jar directory.

- **Empty networks in the GUI All Networks window**

  After performing a retract of a transformation that causes the network to be partitioned into two or more networks, it is sometimes the case the a spurious empty network also appears in the **All Networks** window. You should not select this network, and it will not appear the next time that the GUI tool is run, and otherwise causes no harm.

- **Spurious tables appearing the schema of a wrapped Postgres data source**.

  After wrapping a Postgres data source, tables which do not appear in the schema as listed by the **psql** command are appearing in the AutoMed repository view of the schema. This is probably due to the wrong JDBC driver being used for the version of Postgres you are wrapping. For example, Postgres 7.4 should be wrapped with the jdbc3 JDBC driver.

# Bibliography


[Ber03] P.A. Bernstein. Applying model management to classical meta data problems. In *Proc. CIDR'03*, 2003.

[BKL+04] M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE'04*, volume 3084 of *LNCS*, pages 82–97. Springer, 2004.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BM04] M. Boyd and P.J. McBrien. Towards a semi-automated approach to intermodel transformations. In *Proc. EMMSAD 04, CAiSE Workshop Proceedings Volume 1*, pages 175–188, 2004.

[BM05] M. Boyd and P.J. McBrien. Comparing and transforming between data models via an intermediate hypergraph data model. *Journal on Data Semantics*, IV:69–109, 2005.

[Bun94] P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[Dat04] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 8th edition edition, 2004.

[DDL03] C.J. Date, H. Darwen, and N.A. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2003.

[Fan05] H. Fan. Using schema transformation pathways for incremental view maintenance. In *Proc. DaWak'05*. 2005.

[FP03] H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. DOLAP03*, pages 86–93, New Orleans, 2003.

[JLVV02] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 2nd edition edition, 2002.

[JPZ03] E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report No. 20, AutoMed, 2003.

[Laz05] C. Lazanitis. Schema based peer-to-peer data integration. Master's thesis, Imperial College London, 2005.

[MP99] 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.

[MP02] P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02*, volume 2348 of *LNCS*, pages 484–499. Springer, 2002.

[MP03] P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, pages 227–238. IEEE, 2003.

[PM98] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

[Pou01] A. Poulovassilis. The automed intermediate query language. Technical Report No. 2, AutoMed, 2001.

[Pou04] A. Poulovassilis. A tutorial on the IQL query language. Technical Report No. 28, AutoMed, 2004.

[Riz04] N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. of 6th ICEIS*, 2004.

[RM05] N. Rizopoulos and P.J. McBrien. A general approach to the generation of conceptual model transformations. In O. Pastor and J.F. e Cunha, editors, *Proc. CAiSE'05*, volume 3520 of *LNCS*, pages 326–341. Springer, 2005.

[SL90] A. Sheth and J. Larson. Federated database systems. *ACM Computing Surveys*, 22(3):183–236, 1990.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

# Index