The Automed HDM Data Store

Dean Williams
dean@dcs.bbk.ac.uk

School of Computer Science and Information Systems,
Birkbeck College, University of London

1 Introduction

AutoMed is a data integration system which supports a hypergraph-based data model, the **HDM**, as its common data model. Constructs of higher-level data models are defined in terms of the HDM, as are sets of primitive schema transformations. In this way collections of schemas, with each schema based on one of a variety of data models, can be transformed into their HDM representation and subsequently integrated into a global schema by means of reversible transformation pathways. The global schema can then be queried using a functional query language, **IQL**, with queries being passed to wrappers to interrogate the actual data sources and return results to the query processor.

Previous technical reports and published work describe the repositories for data models (the **MDR**) and schema definitions (the **STR**), the IQL query language and methods of modelling ER, relational, XML, flat file and RDF data in Automed.

Applications which make use of the Automed infrastructure, such as the ESTEST [5] software, often reason with data in its HDM form and need to store results as HDM. This report describes a data store for HDM instance data.

The store is implemented using Postgres relational tables (as are the Automed Model and Schema Repositories). A Java API allows the store to be populated and queried, alternatively a parser reads a text file description of the data and uses this to populates the store. An Automed wrapper exists to enable the IQL query processor to query this store as it would any other data source.

2 Characteristics of HDM instance data

The HDM data model is described elsewhere [2–4], however the following characteristics of HDM instance data are highlighted:

– Data in the HDM consists of nodes and edges e.g.⟨⟨person⟩⟩ is a node and ⟨⟨worksIn,person,room⟩⟩ is an edge.

- Edges can be of any length e.g. ⟪address,houseNumber,road,town,postCode⟫.

- Edges can be named or unnamed e.g. ⟪_,person,room⟫ or ⟪worksIn,person,room⟫.

- Edges are treated the same whether they are named or unnamed - unnamed edges just happen to have a name _.

- Each span on an edge can be either a nodes or edges e.g. ⟪livesAt,person,⟪address,houseNumber,road,town,postCode⟫⟫

- Edge names only unique for type signature e.g. can have both: ⟪worksIn,person,project⟫ and ⟪worksIn,person,room⟫

- Nodes have an associated data type e.g. integer, string, date etc.

3 An Example

Throughout this report the following sample instance data will be used to illustrate the contents of the HDM store tables and in code fragments:

- Dean, Mat, Hao and Edgar are people.
- Hao, Edgar and Dean work on the Automed project.
- Dean and Mat work on the Tristarp project
- Mat and Dean sit in room B34E.
- Edgar and Hao sit in room BG26.
- Dean lives at 64, Northdown Street, London N1 9BS

This data will be placed in HDM store conforming to the following 'personnel' schema:

Nodes:

⟪person⟫
⟪project⟫
⟪room⟫
⟪houseNumber⟫
⟪road⟫
⟪town⟫
⟪postCode⟫

Edges:

⟪worksIn,person,project⟫
⟪worksIn,person,room⟫
⟪address,houseNumber,road,town,postCode⟫
⟪livesAt,person,⟪address,houseNumber,road,town,postCode⟫⟫

Note that there are two edges named 'worksIn' and that the second span in the edge 'livesAt' is itself an edge. For the purposes of this example it is assumed

that all the data types of the nodes are string except for ⟪houseNumber⟫ which is an integer.
Throughout the Automed project double chevrons are used to indicate a node or edge. In the HDM Store and this report, we enclose instance data in square brackets. These instance data specifications are tuples so it may seem appropriate to use single chevrons, however as the instance data can be nested it can be confusing as to whether single or double chevrons are being used in a particular example.
To illustrate: [dean,[64,Northdown Street,London,N1 9BS]] is an instance of: ⟪livesAt,person,⟪address,houseNumber,road,town,postCode⟫⟫

4 HDM Store Tables

The following Postgres tables are used to store the HDM data:

- hdm_store links an hdm store to the schema in the STR repository to which the data must conform.
- node_datatype gives the data type of each node in the schema.
- node stores the details of an instance of a node.
- edge stores details of each edge instance.
- edge_span stores the value of one span of one edge instance.

Each of these tables is now described in turn, together with the rows that would be required to store the example data given in section 3 above.

4.1 The 'hdm_store' Table

This table links an HDM store to the schema defining the HDM database, which must already exists in the schema repository. A name for the repository is also defined which can be used to reference the store instead of the id number.

hdm_store	
<u>hid</u>	integer
sid	integer
hdm_store_name	varchar
schema_name	integer

e.g.

hdm_store			
hid	sid	hdm_store_name	schema_name
1	16	bbkdata	personnel

In the example a single hdm store has been defined with id 1. This store is linked to schema 16 in the STR which has the schema name 'personnel'. The hdm store has the name 'bbkdata'.

4.2 The 'node_datatype' Table

The metadata for the schema is stored in the STR with the exception of the data type information which is only required by the HDM store. When the HDM store is created a row is inserted into the node_datatype table for each node in the schema with a default datatype of 'string'.
The default datatype can be overridden at any time providing no instances of the node have previously been stored.

node_datatype	
hid	integer
node_type	string
datatype	string

e.g.

node_datatype		
hid	node_type	datatype
1	person	string
1	project	string
1	room	string
1	road	string
1	town	string
1	postCode	string
1	houseNumber	integer

In this example all the nodes have the default data type of 'string' apart from the houseNumber which is 'integer'.
The datatypes supported are the standard set for Automed wrappers i.e. date, string, integer, float and boolean.

4.3 node

The node table has a row for each instance of a node in the database. The HDM store identifier and the node identifier are the key of the table and the row also contains the node_type, which must exist in the linked schema and the value as a string.
When the row is inserted the value is checked to ensure that it is a valid string representation of whatever the node_datatype of the node_type is set to.

node	
hid	integer
nid	integer
node_type	string
node_value	string

e.g.

node_datatype			
hid	nid	node_type	node_value
1	1	person	dean
1	2	person	hao
1	3	person	mat
1	4	person	edgar
1	5	room	NG26
1	6	room	B34E
1	7	project	automed
1	8	project	tristarp
1	9	houseNumber	64
1	10	road	Northdown Street
1	11	town	London
1	12	postCode	N1 9BS

The rows for the example data can be seen above, the first row shows that 'dean' is an instance of ⟪person⟫.

4.4 The 'edge' Table

There is one row in the edge table for each instance of an edge in the HDM store. The values of each span are held in the edge_span table. The minimum information that needs to be stored at the edge level is the edge_name - the types of the edge spans could be derived from lookups based on the edge_span table. For efficient processing a number of derivable columns are stored at the edge level, the edge type and a representation of the edge value as a string.

edge	
hid	integer
eid	integer
edge_name	string
edge_type	string
edge_value_as_string	string

e.g. for the data in the example:

edge				
hid	eid	edge_name	edge_type	edge_value_as_string
1	1	worksin	⟪worksin,person,project⟫	[dean,tristarp]
1	2	worksin	⟪worksin,person,project⟫	[mat,tristarp]
1	3	worksin	⟪worksin,person,project⟫	[hao,automed]
1	4	worksin	⟪worksin,person,project⟫	[edgar,automed]
1	5	worksin	⟪worksin,person,project⟫	[dean,automed]
1	6	worksin	⟪worksin,person,room⟫	[mat,B34E]
1	7	worksin	⟪worksin,person,room⟫	[dean,B34E]
1	8	worksin	⟪worksin,person,room⟫	[hao,NG26]
1	9	worksin	⟪worksin,person,room⟫	[edgar,NG26]
1	10	address	⟪address,houseNumber,road,town,postCode⟫	[64,Northdown Street,London,N1 9BS]
1	11	livesAt	⟪livesAt,person,⟪address,houseNumber,road,town,postCode⟫⟫	[dean,[64,Northdown Street,London,N1 9BS]]

4.5 The 'edge_span' Table

The edge_span table has a row for each span on the edge e.g. an edge ⟪worksin,person,project⟫ will have one row on the edge table and two rows on the edge_span table, one to identify which instance of the ⟪person⟫ node is in the edge and one similarly for the ⟪project⟫ node.

edge_span	
hid	integer
eid	integer
span_number	integer
edge_or_node	character
id	string

and so or for the example data:

edge_span				
hid	eid	span_number	edge_or_node	id
1	1	1	n	1
1	1	2	n	8
1	2	1	n	3
1	2	2	n	8
1	3	1	n	2
1	3	2	n	7
1	4	1	n	4
1	4	2	n	7
1	5	1	n	1
1	5	2	n	7
1	6	1	n	3
1	6	2	n	6
1	7	1	n	1
1	7	2	n	6
1	8	1	n	2
1	8	2	n	5
1	9	1	n	4
1	9	2	n	5
1	10	1	n	9
1	10	2	n	10
1	10	3	n	11
1	10	4	n	12
1	11	1	n	1
1	11	2	e	10

The edge_or_node column indicates id the span is an edge or an node. The id column then points to either the node id (nid) in the node table or the edge id (eid) in the edge table.
In the example above all of the id's point to entries of the node table (they are nid's) except for the last row which points to an edge (it is an eid).

5 Using The API

To illustrate the how to create data stores using Java code, an example program is available at http://www.dcs.bbk.ac.uk/ dean/HDMStoreDemo.java
This program is run without arguments and has three methods:

buildModel() this builds the standard repository HDM model and is the same as in other Automed example programs.
buildPersonnelSchema() creates a schema in the STR which complies with the example 'personnel' HDM schema.
populatePersonnelSchema() populates the example schema with the example data.

The following fragment of code from the **populatePersonnelSchema()** method shows how the API is used.

```
HDMStoreDemo.java

HdmStore hdmStore = new HdmStore(debug);

hdmStore.createDBtables();
hdmStore.createHdmStore("personnel","bbkdata");
hdmStore.use("bbkdata");
hdmStore.setDatatype("<<HouseNumber>>","integer");
hdmStore.addNode("<<person>>","[dean]");
hdmStore.addNode("<<person>>","[hao]");

edge = new Edge("<<worksin,person,room>>",new String[] {"[edgar]","[NG26]"});
hdmStore.addEdge(edge);

edge = new Edge("<<livesAt,person,<<address,houseNumber,road,town,postCode>>>>"
  ,new String[] {"[dean]","[64,Northdown Street,London,N1 9BS]"});
hdmStore.addEdge(edge);
```

The **createDBtables()** method will create the Postgres tables used by the HDM store (after dropping them if any already exist).
The the **createHdmStore()** method creates a new HDM store by passing the STR schema name to which the data in this HDM store must conform, together with the name for the HDM store.
As the new HDM store has not yet been populated it is possible to use the **setDatatype()** method to define the data type of ⟨⟨HouseNumber⟩⟩ to be integer.
Then the **use()** method indicates we wish to use the 'bbkdata' store.
Next nodes are added through the **addNode()** method which requires both the node type and the value to be passed, as strings.
The **addEdge()** method takes a Edge class as its argument. The Edge class is created by passing the constructor the string representing the full type description of the edge (e.g. ⟨⟨livesAt,person,⟨⟨address,houseNumber,road,town,postCode⟩⟩⟩⟩)

followed by an array of strings, each string in the array representing the value of a single span in the edge.

6 Using The Parser

As alternative a parser HdmStoreCmd exists which takes in a text file with database commands and runs each command in turn. This provided is to remove the necessity to write new code each time an HDM store is required or amended. The usage is:

```
Usage: hdmStoreCmd [-options] <db commands filename>
       -drop = drop & recreate tables first
       -usage or -help = this message
```

To illustrate the format of the text file the commands for the personnel example would be:

```
createdb;
newstore personnel birkbeck;
use birkbeck;

settype houseNumber integer;

add <<person>>      [dean];
add <<person>>      [hao];
add <<person>>      [mat];
add <<person>>      [edgar];
add <<room>>        [NG26];
add <<room>>        [B34E];
add <<project>>     [automed];
add <<project>>     [tristarp];
add <<houseNumber>> [64];
add <<road>>        [Northdown Street];
add <<town>>        [London];
add <<postCode>>    [N1 9BS];

add <<worksIn,person,project>> [dean,tristarp];
add <<worksIn,person,project>> [mat,tristarp];
add <<worksIn,person,project>> [hao,automed];
add <<worksIn,person,project>> [edgar,automed];
add <<worksIn,person,project>> [dean,automed];

add <<worksIn,person,room>> [mat,B34E];
add <<worksIn,person,room>> [dean,B34E];
add <<worksIn,person,room>> [hao,NG26];
add <<worksIn,person,room>> [edgar,NG26];
add <<address,houseNumber,road,town,postCode>>
    [64,Northdown Street,London,N1 9BS];

add <<livesAt,person,<<address,houseNumber,road,town,postCode>>>>
    [dean,[64,Northdown Street,London,N1 9BS]];
```

These commands map on to the API methods described in Section 5 above.

6.1 Shortcuts

If creating the text file of commands by hand then the syntax as described above will lead to lengthy files even for reasonably small databases. To make this more manageable a number of syntax shortcuts are provided which the parser converts into the full commands before execution. Each of these shortcuts is described with an example.

Multiple Instances Of The Same Node. Typically it will be required to add many instances of the same type e.g.

```
add <<person>>      [dean];
add <<person>>      [hao];
add <<person>>      [mat];
add <<person>>      [edgar];

add <<worksIn,person,project>> [dean,tristarp];
add <<worksIn,person,project>> [mat,tristarp];
add <<worksIn,person,project>> [hao,automed];
add <<worksIn,person,project>> [edgar,automed];
add <<worksIn,person,project>> [dean,automed];
```

this can be specified in the text file by giving the node once and following this by any number of instances e.g.

```
add <<person>> [dean] [mat] [hao] [edgar]
add <<worksIn,person,project>> [dean,tristarp] [mat,tristarp] [hao,automed]
    [edgar,automed] [dean,automed];
```

Missing Nodes On An Edge. An optional additional parser command **addmissingnodes** instructs the parser to insert into the database any nodes mentioned in edge definitions which do not themselves currently exist in the database. Instead of:

```
add <<person>> [dean] [mat] [hao] [edgar]
add <<worksIn,person,project>> [dean,tristarp] [mat,tristarp] [hao,automed]
    [edgar,automed] [dean,automed];
```

it is possible to specify:

```
addmissingnodes;
add <<worksIn,person,project>> [dean,tristarp] [mat,tristarp] [hao,automed]
    [edgar,automed] [dean,automed];
```

Edge Names. As mentioned in section 2 above, edge names do not have to be unique in a schema. However in practice they probably will be, the parser allows for the edge name to be used and not the full type description in cases where there is not more than one edge with the same name.
so it is possible to state:

```
add <<livesAt>> [dean,[64,Northdown Street,London,N1 9BS]];
```

instead of:

```
add <<livesAt,person,<<address,houseNumber,road,town,postCode>>>>
    [dean,[64,Northdown Street,London,N1 9BS]];
```

but not:

```
add <<worksIn>> [mat,B34E];
```

as there are two edges defined in the schema with the name worksIn.

Typing Macros. If something needs to be retyped it can be flagged and the label used for future use e.g.

```
add <<address>> [64,Northdown Street,London,N1 9BS] &1;
add <<livesAt>> [dean,&1];
```

is equivalent to:

```
add <<address>> [64,Northdown Street,London,N1 9BS] &1;
add <<livesAt>> [dean,[64,Northdown Street,London,N1 9BS]];
```

6.2 Shorthand Version Of The Example

Using the quick syntax the example commands can be replaced with:

```
createdb;
newstore personnel birkbeck2;
use birkbeck2;
addmissingnodes;
settype houseNumber integer;
add <<worksIn,person,project>>
    [dean,tristarp] [mat,tristarp] [hao,automed] [edgar,automed] [dean,automed];
add <<worksIn,person,room>>
    [mat,B34E] [dean,B34E] [hao,NG26] [edgar,NG26];
add <<address>> [64,Northdown Street,London,N1 9BS] &1;
add <<address>> [12,Malet Street,London,WC1E 7HX] &2;
add <<livesAt>> [dean,&1];
add <<livesAt>> [edgar,&2];
```

7 The Automed Wrapper

Wrappers for the various physical data sources in AutoMed implement the abstract **AutoMedWrapper** class and are built by a class implementing keywordAutoMed-WrapperFactory. Documentation for these classes can be found in the API repository documentation (`http://www.doc.ic.ac.uk/automed/resources/apidocs/index.html`), however the key methods are described below.
The data structure used to pass and return queries is the Abstract Syntax Graph and these are described, together with an explanation of IQL query processing in Automed in [1].
The HdmWrapperFactory class has a method:

```
HdmWrapper build(String schemaName)
```

Which returns a HdmWrapper when passed the name of a schema.The key methods of the HdmWrapper are those for passing select, update and delete queries

```
ASG executeIQL(ASG q)
boolean insertIQL(ASG q)
boolean deleteIQL(ASG q)
```

The HdmWrapper makes use of a keywordLowLevelWrapperHdm class which uses data structures more closely tied to the HDM data model, in particular the Edge and EdgeElement classes. The EdgeElement class includes a static method for converting from these structures back to the ASG structure used to return query results. An example of a query using the low level wrapper is:

```
LinkedList resultList;
LowLevelWrapperHdm lowLevelWrapperHdm;
lowLevelWrapperHdm = new LowLevelWrapperHdm();

// Example Query 3
resultList = lowLevelWrapperHdm.query("personnel",
     "<<livesAt,person,<<address,houseNumber,road,town,postCode>>>>");

EdgeElement.display(resultList);
```

which returns the results:

```
Edge: <<livesAt,person,<<address,houseNumber,road,town,postCode>>>>
        Node: <<person>>             dean               string
        Edge: <<address,houseNumber,road,town,postCode>>
                Node: <<houseNumber>>        64                 integer
                Node: <<road>>               Northdown Street   string
                Node: <<town>>               London             string
                Node: <<postCode>>           N1 9BS             string
```

8 The Demonstration Programs

A number of demonstration programs are contained in the hdmstore package and each of these is now described:

- DemoHdmStoreApi This program:
  - Builds the HDM Automed Model
  - Build the personnel department schema used in the demo programs. This includes building an access method for the schema which will need to be amended for the local postgres system. In particular the url for a hdm store is of the form PROTOCOL:DBMS:DATABASE:HDMSTORE
    e.g. "jdbc:postgresql:dwAutomed:bbkdata" i.e. its the same as for a postgres database but with the name of the hdm store added at the end.
  - Populates the hdm store with example data using the api.

  Running this program will also create the hdm store tables if required.
- HdmStoreCmd is the command line processor for the HDM store and two example text files are included which make use of the personnel department schema.
  - DemoHdmStoreCmdLong.txt is the longhand version of the file from Section 6.
  - DemoHdmStoreCmdShort.txt is the shorter version, which makes use of shortcuts, from Section 6.2.
- DemoHdmStoreWrapper shows the use of the IQL wrapper with the HDM store including:
  - Queries against existing data
  - Insert and Delete queries for nodes
  - Insert and Delete queries for edges

References

1. E. Jasper and A.Poulovassilis. Processing iql queries and migrating data in the automed toolkit. Technical report, Automed Project, 2003.
2. P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99, LNCS 1626*, pages 333–348, 1999.
3. P.J. McBrien and A. Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Proc. CAiSE'01, LNCS 2068*, pages 330–345, 2001.
4. P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02, LNCS 2348*, pages 484–499, 2002.
5. D. Williams. Combining database and information extraction techniques to discover structure from partially structered data. Technical report, Automed Project, 2003.