From: 'Advances in Databases',
19th British National Conference on Databases (BNCOD19),
B. Eaglestone, S. North and A. Poulovassilis (Eds),
Springer Verlag LNCS 2405, Pages 42-45, 2002

The AutoMed Schema Integration Repository

Michael Boyd, Peter McBrien and Nerissa Tong
{mboyd,pjm,nnyt98}@doc.ic.ac.uk

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

1 Introduction

The AutoMed EPSRC project, jointly run by Birkbeck and Imperial Colleges, has as part of its aims the implementation of some previous theoretical work, which we know term the **AutoMed approach** to database schema and data integration. In this approach [2] 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 (*i.e.* the data integration) can be obtained from the other constructs in the schema. One feature of AutoMed is that it is not restricted to use a particular modelling language for the description of database models and their 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 [3] in terms of constructs in the **hypergraph data model (HDM)** [5]. 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.

In this paper we describe the first version of the **repository** of the AutoMed toolkit (available from `http://www.doc.ic.ac.uk/automed/`). This is a Java API, that uses a RDBMS to provide a persistent storage for data modelling language descriptions in the HDM, database schemas, and transformations between those schemas [1]. The repository also provides some of the shared functionality that tools accessing the repository may require.

The AutoMed repository has two logical components, assessed via one API. The **model definitions repository (MDR)** allows for the description of how a data modelling language is represented as combinations of nodes, edges and constraints in the HDM. It is used by AutoMed 'experts' to configure AutoMed so that it can handle a particular data modelling language. The **schema transformation repository (STR)** allows for schemas to be defined in terms of the 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 [4].

Before describing how the MDR and STR APIs function, we give in Figure 1 an example of two schemas in a variant of the ER modelling language, together with a sequence of transformations which map between the two schemas.

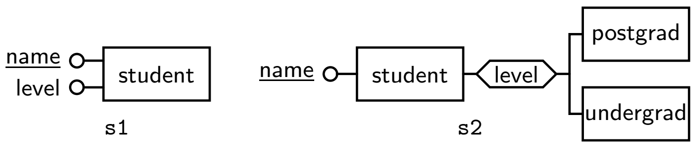

Fig. 1. Example of Schema Integration

The two schemas illustrate the well known attribute-generalisation equivalence. Instances of undergrad are those instances of student which have ug as the value of the level attribute, and postgrad are those instances with pg as the level.

transformation s1→s2
$\textcircled{1}$ *addEntity* ⟨⟨undergrad⟩⟩ $\{x \mid \langle x,$ *'ug'*$\rangle \in$ ⟨⟨student,level⟩⟩$\}$
$\textcircled{2}$ *addEntity* ⟨⟨postgrad⟩⟩ $\{x \mid \langle x,$ *'pg'*$\rangle \in$ ⟨⟨student,level⟩⟩$\}$
$\textcircled{3}$ *addGen* ⟨⟨level,total,student,undergrad,postgrad⟩⟩
$\textcircled{4}$ *delAttribute* ⟨⟨student,level⟩⟩
  $\{x, y \mid x \in$ ⟨⟨undergrad⟩⟩ $\wedge\, y =$ *'ug'* $\vee\, x \in$ ⟨⟨postgrad⟩⟩ $\wedge\, y =$ *'pg'*$\}$
  $(\langle \_, y\rangle \in$ ⟨⟨student,level⟩⟩ $\rightarrow y =$ *'ug'* $\vee\, y =$ *'pg'*$)$

2 Describing a Data Modelling Language in the MDR

In [3] we proposed a general technique for the modelling of any structured data modelling language in the HDM. This has been used as a basis for the design of the MDR. First, to specify a modelling language, we create an instance of the Model class, with an associated identifying name:

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

Constructs in the modelling language are then defined by selecting one of four variants [3] — nodal, link nodal, link, and constraint — and describing details of the **scheme** of the construct. For example, entities in an ER modelling language correspond to nodes in the underlying HDM, which we identify by the general scheme template of ⟨⟨entity_name⟩⟩. Thus in the API, we create a new nodal Construct called "entity" in the "er" Model created earlier, and add to its scheme a new HDM node to hold the entity_name.

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

An attribute in an ER model must always be attached to an already existing entity. This is an example of a link nodal construct. Its scheme takes the general form ⟨⟨entity_name,attribute_name,cardinality⟩⟩ where entity_name must already exist as the name of the entity (hence the second line below), attribute_name is the name of a new node holding instances of the attribute (hence the third line), and cardinality will be one of key, notnull or null, and acts as a constraint on instances of the attribute (hence the fourth line).

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

Other ER constructs can be defined in a similar manner, which can be found in the full version of the program available on the AutoMed website.

3 Describing Schemas and Transformations in the STR

A database schema is held in the STR, as a set of SchemaObject instances, each of which must be based on Construct instances that have been created in the MDR. The schema is created by building an instance of Schema, and then populating the schema with instances of SchemaObject.

```
Schema s=Schema.createSchema("s1",false);
SchemaObject student=s.createSchemaObject(ent,new Object[]{"student"});
s.createSchemaObject(att,new Object[]{student,"name","key"});
SchemaObject studentlevel=
    s.createSchemaObject(att,new Object[]{student,"level","notnull"});
```

Two things should be noted about the above example. Firstly, the second argument of createSchemaObject is an Object array, the elements of which must match the types specified in the scheme of the construct in the MDR. For example, a runtime error would result if the text "student" (a string) replaced the student (a SchemaObject instance). Secondly, the Schema could have added to it constructs from different modelling languages. Mixing modelling languages in one schema is useful when translating between different modelling languages [3].

Transformations can be defined by being 'applied' to one schema, generating the next schema in the transformation sequence. Steps ① and ② both add an entity based on a query on the level attribute of student, and this is done by specifying the scheme of the new entity as an argument to the method of creating a transformation below. The first argument is the type of Construct being added, the second argument the scheme of SchemaObject, and the third argument is the query to derive instances of that object:

```
Schema s1a=s1.applyAddTransformation(ent,new Object[]{"undergrad"},
    "{x | ⟨x,ug⟩ in ⟨⟨student,level,notnull⟩⟩}",null);
Schema s1b=s1a.applyAddTransformation(ent,new Object[]{"postgrad"},
    "{x | ⟨x,pg⟩ in ⟨⟨student,level,notnull⟩⟩}",null);
```

The result of these two method calls is a schema held in s1b that contains entities undergrad and postgrad in addition to what is shown in s1 in Figure 1. To create ③, we need to obtain references to these entities so that they can appear in the scheme of the generalisation hierarchy level under student. This is done by finding the 'to' object of the transformations (*i.e.* the new object added by the transformation to the schema), before actually creating the transformation:

```
SchemaObject undergrad=Transformation.getTransformationToObject(s1,s1a);
SchemaObject postgrad=Transformation.getTransformationToObject(s1a,s1b);
Schema s1c=s1b.applyAddTransformation(gen,
    new Object[]{"level","total",student,undergrad,postgrad},null,null);
```

Finally, ④ is specified by creating a delete transformation, which does not need to specify the type of Construct being deleted, since that can be determined from the SchemaObject.

```
Schema s2froms1=s1c.applyDeleteTransformation(studentlevel,
    "{x,y | x in ⟨⟨undergrad⟩⟩,y=ug;x in ⟨⟨postgrad⟩⟩,y=pg}",
    "⟨_,y⟩ in ⟨⟨student,level⟩⟩ −⟩ y=ug;y=pg");
```

The result *is not* the schema s2, but a schema that *appears* to be the same as s2, but derived from the information in s1. To associate these two conceptually identical schemas together, a series of ident transformations are specified to associate pairs of identical objects, as follows:

```
Transformation.createIdentTransformations(s2,s2froms1,null,null);
```

Whilst s2 and s2froms1 appear identical, queries on s2 will be executed on its underlying database, whilst queries on s2froms1 will be executed on the database underlying s1. Hence we are able to control which database is used as the source for instances of a particular.

4 Conclusions

The alpha release of the API presented here has been fully tested, and work on a beta release is almost completed. Current development is focused on developing schema integration tools that work over the repository, and on integrating distributed querying processing software into the repository.

References

1. M. Boyd and N. Tong. The automed repositories and api. Technical report, 2001.
2. P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proceedings of ER99*, volume 1728 of *LNCS*, pages 96–113. Springer-Verlag, 1999.
3. P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer-Verlag, 1999.
4. P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Advanced Information Systems Engineering, 14th International Conference CAiSE2002*, LNCS. Springer-Verlag, 2002.
5. A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.