# AutoMed

## The AutoMed Repositories and API

Version 2

Michael Boyd amd Nerissa Tong
{*mboyd,nnyt98*}*@doc.ic.ac.uk*

Friday 23$^{\underline{\text{rd}}}$ May 2003

## 1 Introduction

The purpose of this report is to describe the AutoMed repositories API which provides a Java interface to the Model Definitions Repository (MDR) and the Schemas and Transformations Repository (STR) both of which have been developed as part of the AutoMed project. These repositories will be used when developing software for the project such as a graphical schema transformation and intregation tool, schema improvement tool and a JDBC driver. The query processing, optimisation and translation as well as the data migration and schema evolution tools may well interact directly with the reps API as well as indirectly through the JDBC driver.

Because the AutoMed project is concerned with schema integration using a generic Hypergraph Data Model (HDM) with higher-level modelling languages being defined in terms of the lower-level HDM, we need to enable users of the API to describe the higher-level modelling languages in a formal way so that our tools can automatically translate between the two levels. This is the function of the MDR.

The STR stores information about schemas in terms of the modelling language constructs described in the MDR as well as primitive transformations between schemas (thus describing transformation pathways between them). It also stores information about the data source a schama describes such as how to access it, whether its data is extensional or intentional, etc.

## 2 Overview

Figure **??** illustrates how the tool and the various repositories are connected. The DSR (Data Source Repository) is a helper class that provides common functionality to the MDR and STR as well as a test harness for the API. The DSR will not be documented further in this report.

After a breif description of the MDR and STR, we will describe the reps API by giving

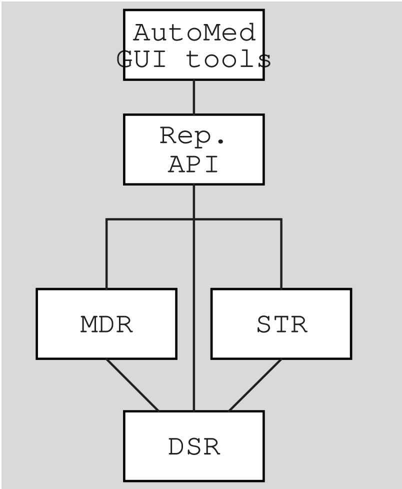


Figure 1: Architecture of the AutoMed Repositories API

example code that first uses the API to construct a basic ER modelling langauge and then uses that language to construct the schemas in the example.

We will give the schemas of the MDR and STR and the data that would be produced by the example code for the reps API as well as some other examples of data for more complex scenarios.

## 2.1 MDR Overview

The MDR (Model Definitions Repository) stores descriptions of modelling languages. In essence, it comprises of a list of constructs for each modelling language along with the arguments needed to create an instance of each construct and the means by which to translate instances of constructs into HDM.

Each argument needed to create an instance can be a simple name, a reference to an existing object of a given construct type, a list of alternatives or a sequence. Each argument has a lower and upper cardinality[1] which specifies how many times the name / construct / alternation / sequence can appear (all occurances make up the one argument for the construct).

Each construct is given an HDM type which can be either nodal, linkage, link-nodal, or constraint (which simply means there is no corresponding HDM), see [**?**] for details. Each HDM type (and therefore each construct) has a template with varaiables that the arguments specified for instantiating the construct can be assigned to.

A very simple example would be an ER entity and attribute. An entity would have a nodal HDM type and would have one construction argument – the entity's name. An attribute would have an HDM type of link-nodal (because it creates a node for the attribute but can only exist in relation to an existing entity) and its two construction arguments would be the name of the attribute and the existing entity to which it should be linked[2].

[1] An upper bound of infinitry is denoted by giving an upper bound which is lower than the lower bound.

[2] A more realistic example might have cardinality arguments as well

## 2.2 STR Overview

The STR (Schemas and Transformations Repository) stores information about schemas and transformation pathways between them. A schema is a list of objects and associated object schemes. Each object is identified as being an instance of a particular construct in the MDR. The object references that make up an object's scheme directly relate to the instantiation arguments in the MDR for the object's construct.

The first instantiation argument of type name (as opposed to a reference to an existing construct, alternation or sequence) is given special treatement. This name is used as the name of the object and the object reference for the argument in which the name appeared references the object it instantiates. If a construct has more than one name in its arguments then extra objects are created to store these values. These extra objects have no construct type. Other than that, the result is the same as it would be if there was a construct representing whatever was named and a reference to an instance of that construct used instead of just its name in the argument list.

Schemas themselves have no inherent modelling language. Instead, each object, through its construct type, is of a particular modelling language. This means that transformation pathways that convert a schema from one language to another can be defined because the intermediatory schemas with some constructs from both languages can be described.

Their are two types of schema wich are stored – schemas of acutal data sources (extensional data) and intermediatory schemas (intentional data) that exist in transformation pathways between the schemas of actual data sources. These intermediatory schemas are usually themselves intentional[3]; that is, the schemas can only be desribed by looking at another schema's extension and a pathway of transformations between the two. The intentional schemas can be temporarily materialised (i.e. made extensional) and indeed this automatically happens when the schema's description is asked for so that these differences are, for the most part, transparent to tools using the repositories.

## 2.3 Simple Example

Throughout this paper the example in figure **??** will be used.

# 3 The AutoMed Repositories API

The repositories API provides methods to create, query, alter and remove models, constructs, construct schemes, schemas, schema objects, object schemes, transformations as well as some ancillary etnitities such as access methods for extensional schemas. An ER diagram of how these entities are related to each other is given in Appendix **??**. The repositories API comprises of Java classes representing each of these entities, all the methods for manipulating the repositories are in these classes.

The AutoMed GUI tools will access the MDR and STR via the AutoMed Repository API. The two major classes that will be created directly by the (AutoMed) tool are the `Model`

[3] Intentional with respect to the schema as opposed to the data described by the schema

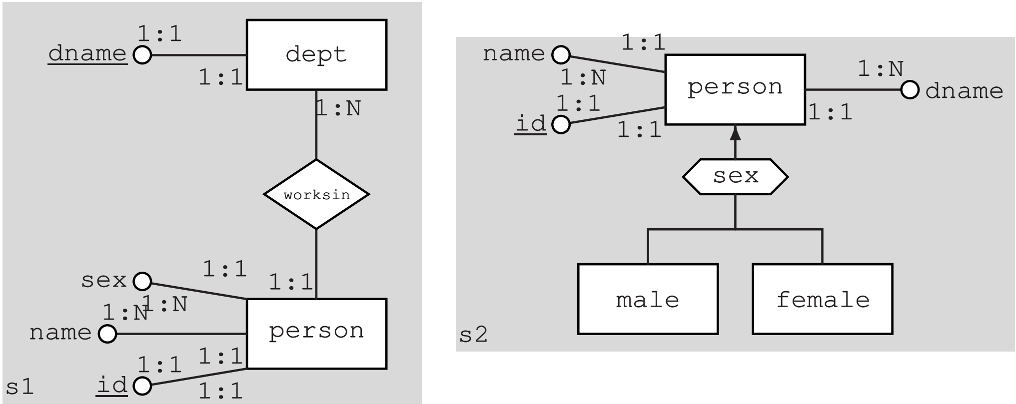


Figure 2: Example ER schemas

| AutoMed Repositories API Classes | | |
|---|---|---|
| Class | Repository | Created By |
| **Model** | MDR | AutoMed tool |
| **Construct** | MDR | Model |
| **SchemePos** | MDR | Construct |
| **Schema** | STR | AutoMed tool |
| **SchemaObject** | STR | Schema |
| **ObjectScheme** | STR | SchemaObject |
| **Transformation** | STR | AutoMed tool |
| **AccessMethod** | STR | AutoMed tool |
| **Protocol** | STR | AutoMed tool |

Table 1: Classes for the AutoMed Repositories API

class (from the MDR) and the `Schema` class (from the STR). The classes we have created (partially) so far for the AutoMed Repositories API are listed in Table **??**.

Examples showing the usage of the classes related to the MDR and STR are included in Section **??** and Section **??** respectively.

### 3.1 Classes related to the MDR

The `Model` class represents data models. The argument required for creating a `Model` object is the `name` of the model. For example, the following line creates a `Model` named `ER`:

```
Model myER = new Model(``ER'');
```

To create `Constructs` for the `Model`, a method is provided in the `Model` class. For example, to create an entity for the `myER` model, we do the following:

```
Construct myEREnt = myER.createConstruct(``Entity'',
                                   Construct.CONSTRUCT_CLASS_NODAL,
```

```
                                         true);
```

The first argument of the `createConstruct` methods takes a `String` value which represents the name of the `Construct`. The second argument is a constant integer indicating the class of the construct, its value must be `Construct.CONSTRUCT_CLASS_$`, where `$` is one of `NODAL`, `LINK`, `LINK_NODAL`, and `CONSTRAINT`. The third argument accepts a `boolean` value. It is set to `true` if the construct is indeed a construct in the model (the **isRoot** field in the *Construct* table of the MDR) and set to `false` otherwise. A model identifier (**mid**) is created upon creation of a `Model` object and a unique construct identifier (**cid**) is created for each `Construct` object. To fill the *construct* table in the MDR, the value for the field **mid** is implied by the calling `Model` object that calls the `createConstruct` method.

We now need to fill the *scheme* table of the MDR for the construct we have created. The method `createSchemePos` in the `Construct` class allows us to do so.

```
myEREnt.createSchemePos(SchemePos.SCHEME_TYPE_NAME,
                        null,
                        true,
                        SCHEME_HDM_TYPE_N1,
                        1, 1);
```

The `createSchemePos` method does not return a value. The first argument takes the value `SchemePos.SCHEME_TYPE_$`, where `$` is one of `NAME`, `CONSTRUCT`, `ALTERNATION`, and `SEQUENCE`. The second argument is always `null` if the SCHEME_TYPE is `NAME`. It is of type `Construct` if the SCHEME_TYPE is `CONSTRUCT`. We have not yet arrived at a decision as to what the treatment should be for the other two SCHEME_TYPES. The third argument refers to the **isKey** field of the *scheme* table in the MDR. The fourth argument indicates the corresponding HDM type in the HDM template. Its value is `SchemePos.SCHEME_HDM_TYPE_$`, where `$` is one of `N1`, `N2`, `P1`, `P2`, and `C`. The last two arguments refer to the **lower** and **upper** fields in the *scheme* table, and control the cardinality of the attribute (i.e. how many times the attribute may be repeated). If **upper** $<$ **lower** then the attribute may be repeated an unlimited number of times.

The fragments of code together provide the MDR complete information for the ER entity construct. Table **??** shows the state of the MDR after the execution of the code fragments. An ER attribute can be created in a similar fashion. Table **??** shows the state of the MDR after the attribute construct has also been created.

```
/*........................ code fragment for ER Attribute ...*/
Construct myERAtt = myER.createConstruct(``Attribute'',
                            Construct.CONSTRUCT_CLASS_LINK_NODAL,
                            true);

myERAtt.createSchemePos(SchemePos.SCHEME_TYPE_CONSTRUCT,
                        myEREnt,
                        true,
                        SCHEME_HDM_TYPE_P1,
                        1, 1);
```

| model | |
|---|---|
| mid | name |
| 1 | ER |

(a)

| construct | | | | |
|---|---|---|---|---|
| mid | cid | name | class | isRoot |
| 1 | 1 | Entity | NODAL | true |

(b)

| scheme | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| mid | cid | pos | type | argMid | argCid | isKey | HdmType | lower | upper |
| 1 | 1 | 1 | NAME | - | - | true | N1 | 1 | 1 |

(c)

Table 2: MDR (ER entity created)

| model | |
|---|---|
| mid | name |
| 1 | ER |

(a)

| construct | | | | |
|---|---|---|---|---|
| mid | cid | name | class | isRoot |
| 1 | 1 | Entity | NODAL | true |
| 1 | 2 | Attribute | LINK-NODAL | true |

(b)

| scheme | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| mid | cid | pos | type | argMid | argCid | isKey | HdmType | lower | upper |
| 1 | 1 | 1 | NAME | - | - | true | N1 | 1 | 1 |
| 1 | 2 | 1 | CONSTRUCT | 1 | 1 | true | P1 | 1 | 1 |
| 1 | 2 | 2 | NAME | - | - | true | N2 | 1 | 1 |

(c)

Table 3: MDR (ER entity & attribute created)

```
myERAtt.createSchemePos(SchemePos.SCHEME_TYPE_NAME,
                        null,
                        true,
                        SCHEME_HDM_TYPE_N2,
                        1, 1);
/*....................... end of code fragment ER Attribute ...*/
```

## 3.2 Classes related to the STR

The `Schema` class represents schemas. A `Schema` object is created by calling the `Schema` class constructor with the value for the name of the schema (of type `String`) and three `boolean` values indicating whether or not the schame is an extentional schema, an extentional database, and a data source. The following is an example creating a schema

| schemas | | | | | |
|---|---|---|---|---|---|
| sid | name | es | edb | data_source | is_cached |
| 1 | s1 | 1 | 1 | 1 | 0 |

(a)

| schema_objects | | | | | |
|---|---|---|---|---|---|
| sid | mid | cid | obid | value | is_temp |
| 1 | 1 | 1 | 1 | E1 | 0 |
| 1 | 1 | 2 | 2 | A1 | 0 |

(b)

| object_schemes | | |
|---|---|---|
| obid | scheme_pos | sobid |
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 2 | 2 | 2 |

(c)

Table 4: STR (Schema with ER entity & attribute created)

named `mySchema`.

```
Schema mySchema = new Schema(``s1'', true, true, true);
```

Schema objects are added to the schema by calling the `createSchemaObject` method of the `Schema` class. The following code fragment creates an entity and an attribute for `mySchema`.

```
SchemaObject myEntity1 = mySchema.createSchemaObject(myEREnt,
                                          new Object[] {``E1''});
SchemaObject myAttribute = mySchema.createSchemaObject(myERAtt,
                                    new Object[] {``A1'', myEntity1});
```

The first argument of the `createSchemaObject` method takes an object reference of type `Construct`, which indicates the type of construct to which the schema object being created belongs. The second argument is an `Object` array that accepts values of either of type `String` or `SchemaObject`. Only one `String` value is allowed in the `Object` array. This `String` corresponds to the **value** field in the *schema_objects* table in the STR. An `ObjectScheme` object is automatically created for each `SchemaObject` object by taking the `Object` array argument. In the creation of each `ObjectScheme` object, the type of the elements in the `Object` array are verified against the data in the **scheme** table in the MDR. Data entries are made in the *object_schemes* table in the STR according to the order in which the elements in the `Object` array are declared. The result produced by the fragments of code is shown in Table **??**.

Further investigation is still required for the other STR-related classes and hence they are not included in this report.

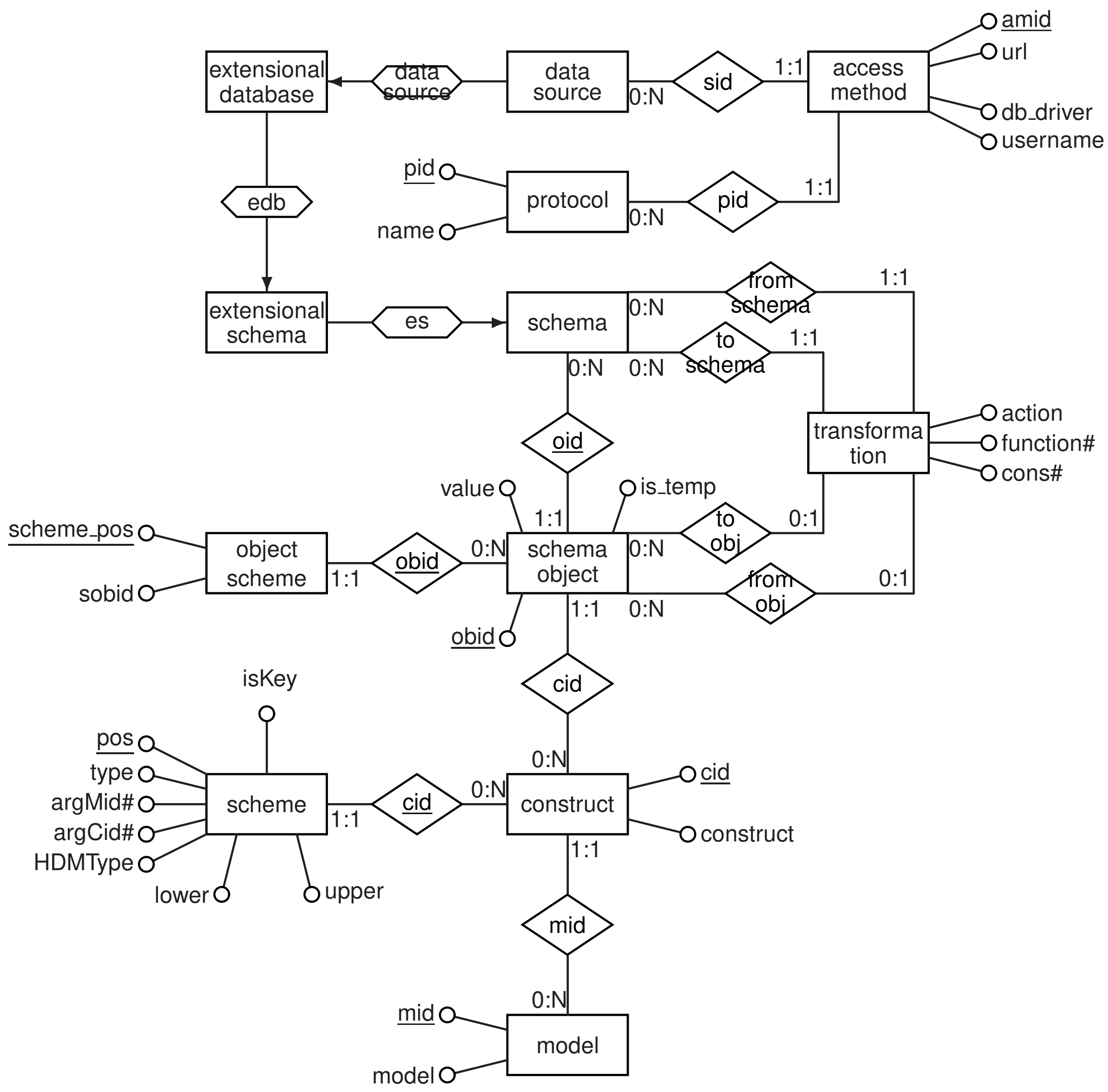


Figure 3: Repository Schema

# A Repositories Schema

Figure ?? illustrates the ER model of the MDR and STR.

# B The STR Repository

The STR is made up of six tables, they are the `schema` table, the `object` table, the `scheme` table, the `transformation` table, the `accessMethod` table, and the `protocol` table.

### B.1 The `schema` table

| schema | |
|---|---|
| SID | integer |
| name | varchar |
| extSchema | integer |
| extDB | integer |
| dataSource | integer |
| isCached | integer |

Each schema entered into the repository is assigned a schema ID (`SID`) which uniquely identifies itself in the table. Schemas are assumed to have unique names. Information about the nature of the schemas is also recorded – whether a schema is an extensional schema (`extSchema`), whether it is an extensional database (`extDB`), and whether it is a data source (`dataSource`). An extentional schema is one which physically exists, whereas an intentional schema is one which is computed or derived from some other extentional or intentional schema(s). An extentional database is one with instances of the schema objects stored. An intentional database has its schema object instances derived from an extiontional database when they are required to be materialized. A data source refers to an actual external data repository. Data sources whose schemas and objects are materialized and stored locally are not considered to be data sources. The `isCached` field is a special field indicating whether or not a schema's objects have been materialized and temporarily stored in the STR. This field is designed to be used only by the interface above this layer.

Figure **??** shows two example ER schemas `s1` and `s2`. The following table shows how these two schemas are stored in the `schema` table.

| schema | | | | | |
|---|---|---|---|---|---|
| SID | name | extSchema | extDB | dataSource | isCached |
| 1 | s1 | 1 | 1 | 1 | 0 |
| 2 | s2 | 1 | 1 | 1 | 0 |

### B.2 The `object` table

| object | |
|---|---|
| SID | integer |
| MID | integer |
| CID | integer |
| OBID | integer |
| value | varchar |
| isTemp | integer |

The `object` table stores information about each object (or construct) in a schema. Each object is given a unique object identifier (`OBID`). The `MID` (model ID) and `CID` (construct ID) fields indicate the modelling language used and the type of construct the object belongs to in this modelling language. The `SID` field records to which schema a particualr object belongs and the `value` field stores the name (or simply value for constraints) of the object. The `isTemp` field is a special field that will be used by the inteface at a higher level. It is set to `true` (1) when an object of an intentional schema is materialized, i.e., when the object is copied from the schema from which the intentional schema is derived.

Assume we have the following two tables that define the model identifier and construct identifier for each model:

| model | |
|---|---|
| MID | model |
| 1 | relational |
| 2 | ER |
| 3 | ER+ |
| 4 | UML |

| construct | | |
|---|---|---|
| MID | CID | construct |
| 1 | 1 | relation |
| 1 | 2 | attribute |
| 1 | 3 | func. dep. |
| 2 | 1 | entity |
| 2 | 2 | attribute |
| 2 | 3 | relationship |
| 2 | 4 | generalisation |
| 2 | 5 | constraint |
| | | ⋮ |

The following table shows how the objects of the two schemas `s1` and `s2` are stored in the `object` table (omitting cardinality constraints for attributes).

| object | | | | | |
|---|---|---|---|---|---|
| SID | MID | CID | OBID | value | isTemp |
| 1 | 2 | 1 | 1 | person | 0 |
| 1 | 2 | 1 | 2 | dept | 0 |
| 1 | 2 | 2 | 3 | id | 0 |
| 1 | 2 | 2 | 4 | name | 0 |
| 1 | 2 | 2 | 5 | sex | 0 |
| 1 | 2 | 2 | 6 | dname | 0 |
| 1 | 2 | 3 | 7 | worksin | 0 |
| 1 | 2 | 5 | 8 | 1:1 | 0 |
| 1 | 2 | 5 | 9 | 1:N | 0 |
| 2 | 2 | 1 | 10 | person | 0 |
| 2 | 2 | 1 | 11 | male | 0 |
| 2 | 2 | 1 | 12 | female | 0 |
| 2 | 2 | 2 | 13 | id | 0 |
| 2 | 2 | 2 | 14 | name | 0 |
| 2 | 2 | 2 | 15 | dname | 0 |
| 2 | 2 | 4 | 16 | sex | 0 |
| 2 | 2 | 5 | 17 | total | 0 |

### B.3 The scheme table

| scheme | |
|---|---|
| OBID | integer |
| schemePos | integer |
| SOBID | integer |

The `object` table records details about each objects individually, however, it does not provide information about the linkage between different objects in the schema. The `scheme` table is designed to hold this information. Every construct in each of the models has a **scheme**. The definition of what the scheme should be for each construct is stored in the MDR. The following shows four example schemes for the ER model.

| Schemes for ER constructs | |
|---|---|
| construct | scheme |
| entity | $\langle\langle E \rangle\rangle$ |
| attribute | $\langle\langle E, A \rangle\rangle$ |
| relationship | $\langle\langle R, E_1, E_2, C_1, C_2 \rangle\rangle$ |
| generalisation | $\langle\langle G, T, E, S_1, ..., S_n \rangle\rangle$ |

The scheme of an entity contains only the entity itself. An attribute has the entity with which it is associated in its scheme, followed by itself. The scheme of a (binary) relationship contains the relationship as the first argument, followed by the (two) entities connected by the relationship, and their associated cardinality constraints for the relationship. The size (number of arguments) of the scheme for a generalisation is not fixed. It is dependent on the number of sub-entities in the generalisation. The first argument in the generalisation scheme is the generalisation itself, followed by the type of the generalisation (either partial or total), the super-entity and a number of sub-entities.

The `scheme` table records the content of each position of the scheme of a particular schema object. The `OBID` field refers to the schema object whose scheme content is stored in the row. The `schemePos` field indicates which position in the scheme `SOBID`, which is another schema object, is located. The following table shows how the `scheme` table of the two schemas `s1` and `s2` are filled.

| scheme | | |
|---|---|---|
| OBID | schemePos | SOBID |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 1 |
| 3 | 2 | 3 |
| 4 | 1 | 1 |
| 4 | 2 | 4 |
| 5 | 1 | 1 |
| 5 | 2 | 5 |
| 6 | 1 | 2 |
| 6 | 2 | 6 |
| 7 | 1 | 7 |
| 7 | 2 | 1 |
| 7 | 3 | 2 |
| 7 | 4 | 8 |
| 7 | 5 | 9 |

| scheme (cont'd) | | |
|---|---|---|
| OBID | schemePos | SOBID |
| 10 | 1 | 10 |
| 11 | 1 | 11 |
| 12 | 1 | 12 |
| 13 | 1 | 10 |
| 13 | 2 | 13 |
| 14 | 1 | 10 |
| 14 | 2 | 14 |
| 15 | 1 | 10 |
| 15 | 2 | 15 |
| 16 | 1 | 16 |
| 16 | 2 | 17 |
| 16 | 3 | 10 |
| 16 | 4 | 11 |
| 16 | 5 | 12 |

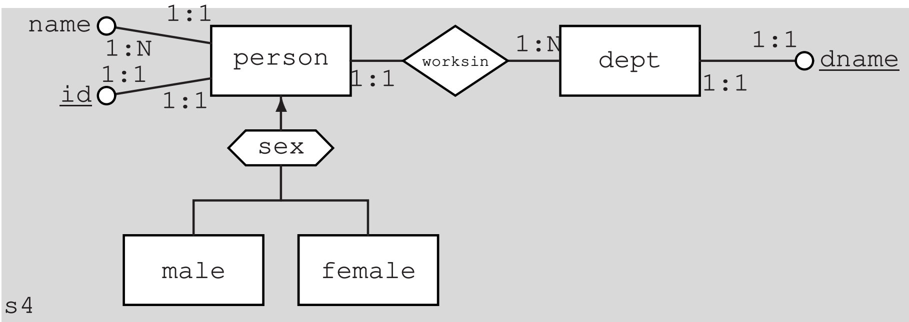


Figure 4: Example ER schema `s4`

### B.4 The `transformation` table

| **transformation** | |
|---|---|
| action | varchar |
| fromObj | integer |
| toObj | integer |
| <u>fromSchema</u> | integer |
| <u>toSchema</u> | integer |
| function | varchar |
| cons | varchar |

An intentional schema is formed by transforming an extentional schema, or another intermediate intentional schema. Transformations between schemas are specified in the `transformation` table. When a particular intentional schema is to be materialized, the corresponding transformations are looked up from this table and operated on by the interface at a level above the MDR and the STR.

The specification of a transformation includes an `action`, which is one of "`add`", "`del`", "`ren`", "`ext`", and "`con`". The input of the action is a schema object, whose ID (`OBID`) is recorded in the `fromObj` field of the table. The output of the action is another schema object – the `toObject` in the table. For all the transformation operations except "`ren`", only one entry of either `fromObj` or `toObj` is made, a negative `OBID` (which is an invalid `OBID`) is entered into the field that is not applicable. In the case of an "`ren`" (rename) transformation, the `fromObj` field contains the ID of the schema object with its original name, and the `toObj` field contains the ID of the schema object with the new name. The `fromSchema` and `toSchema` fields store the ID of the source schema and result schema of the transformation, respectively. The query whose value represents the extent of the transformed construct, if any, is stored in the `function` field. Any constraints associated with a transformation are entered into the `cons` field.

For example, if we have an intentional schema called `s4` with schema ID `4` (shown in Figure **??**), the transformations between `s1` and `s4` can be represented in the `transformation` as follows (An intermediate schema is created after each transformation. The intermediate schemas between `s1` and `s4` are named `s1a`, `s1b`, and `s1c`, with schema ID `5`, `6`, and `7` respectively):

| transformation | | | | | | |
|---|---|---|---|---|---|---|
| action | fromObj | toObj | fromSchema | toSchema | function | cons |
| add | null | 18 | 1 | 5 | query | cons |
| add | null | 19 | 5 | 6 | query | cons |
| add | null | 20 | 6 | 7 | query | cons |
| del | 22 | null | 7 | 4 | query | cons |

The following tables show the portions of data in the `schema` and `object` tables that are related to the transformations between `s1` and `s4`. Arbitrary object identifiers (`OBID`) are assigned to the schema objects belonging to `s4` and the intermediate schemas between `s1` and `s4`. Note that all the objects of `s4` and the intermediate schemas have the value of `1` for the `isTemp` field because `s4` and the intermediate schemas are intentional schemas, hence their objects are "copied" from their parent schema(s) and are not to be permanently stored in the STR.

| schema | | | | | |
|---|---|---|---|---|---|
| SID | name | extSchema | extDB | dataSource | isCached |
| ⋮ | | | | | |
| 4 | s4 | 0 | 0 | 0 | 0 |
| 5 | s1a | 0 | 0 | 0 | 0 |
| 6 | s1b | 0 | 0 | 0 | 0 |
| 7 | s1c | 0 | 0 | 0 | 0 |

| object | | | | | |
|---|---|---|---|---|---|
| SID | MID | CID | OBID | value | isTemp |
| 1 | 2 | 2 | 5 | sex | 0 |
| ⋮ | | | | | |
| 5 | 2 | 1 | 18 | male | 1 |
| 6 | 2 | 1 | 19 | female | 1 |
| 7 | 2 | 4 | 20 | sex | 1 |
| 7 | 2 | 5 | 21 | total | 1 |
| 7 | 2 | 2 | 22 | sex | 1 |

## B.5 The `accessMethod` table

| accessMethod | |
|---|---|
| AMID | integer |
| SID | integer |
| URL | varchar |
| PID | integer |
| dbDriver | varchar |
| username | varchar |

The `accessMethod` table stores information about how to make connection to databases. Each access method has a unique identifier (`AMID`), the schema identifier (`SID`) that describes the schema of the database, the URL (`URL`) of the database, the protocol identifier (`pid` – see section **??**) that indicates the protocol used for the connection, the database driver (`dbDriver`), and the userame (`username`) that can be used to log in to the database.

The following table shows some sample access methods stored in the `accessMethod` table.

| accessMethod | | | | | |
|---|---|---|---|---|---|
| AMID | SID | URL | PID | dbDriver | username |
| 1 | 1 | jdbc:mysql://x.y.z:1234/db | 1 | org.mysql.Driver | dbadmin |
| 2 | 1 | jdbc:postgresql:emp | 1 | org.postgresql.Driver | foo |
| 3 | 2 | jdbc:odbc:People | 1 | sun.jdbc.odbc.JdbcOdbcDriver | bar |

### B.6 The `protocol` table

| protocol | |
|---|---|
| PID | integer |
| name | varchar |

The `protocol` table stores the types of database connection protocols. Each protocol is given an identifier (`PID`) and described in the `name` field.

| protocol | |
|---|---|
| PID | name |
| 1 | JDBC |
| 2 | ODBC |
| 3 | XML |

## C MDR Schema

Table **??** lists the tables which make up the **model definitions repository schema (MDR Schema)**. Each type of modelling language (for example ER, Relational, UML, *etc*) that may be held in the MDR will have one entry in the **model** table, where the fields have the following meaning:

- **mid** is a numeric identifier for the modelling language, used inside the MDR.
- **name** is the textual name the modelling langauge may be referred to as in API calls made to the MDR.

The **construct** relation has one tuple per modelling language construct. Examples of modelling language constructs are entities, attributes, relationships etc. The class field determines the correspondence between the construct and its representative HDM. This determines, via the HDM template below, what values the HdmType field may take in this construct's different scheme positions. Note that all HDM types may have constraints associated with them; the constraint HDM type serves to model a constaint alone.

Each construct has a number of scheme tuples associated with it. Each scheme tuple corresponds to one argument in the constructor for its construct. The HdmType of an

| model | |
|---|---|
| mid | integer |
| name | varchar |

(a)

| construct | | |
|---|---|---|
| **mid** | integer | references model |
| **cid** | integer | |
| **name** | varchar | |
| **class** | (Nodal \| Link \| Link-Nodal \| Constraint) | |
| **isRoot** | boolean | |

(b)

| scheme | | |
|---|---|---|
| mid | integer | references model |
| cid | integer | |
| pos | integer | |
| type | (Name \| Construct \| Alternation \| Sequence) | |
| argMid | integer | |
| argCid | integer | |
| isKey | boolean | |
| HdmType | $(N_1 \mid P_1 \mid N_2 \mid P_2 \mid C)$ | |
| lower | integer | |
| upper | integer | |
| (mid, cid) references construct | | |
| (argMid, argCid) references scheme | | |

(c)

Table 5: Model Definitions Repository Schema

argument must be either constraint or one of the types in the HDM template for the class of the associated construct. If the type is name then argMid and argCid have no meaning and the only HDM type allowed (other than constraint) is $N_i$. Otherwise, they refer to another construct and the only allowable HDM type is $P_i$. Fields lower and upper determine how many times the scheme argument can be repeated, repetitions are comma seperated and together form the HDM type $N_i$ or $P_i$. When (argCid, argMid) refer to another construct, all schemes for the construct with isKey true must be named to uniquely identify it. If upper is less than lower (typically lower = 1, upper = 0) then there is no upper bound.

| class | HDM Template |
|---|---|
| Nodal | $\langle\langle N_1 \rangle\rangle$ |
| Link | $\langle\langle N_1, P_1, P_2 \rangle\rangle$ |
| Link-Nodal | $\langle\langle N_1, P_1, N_2 \rangle\rangle$ |
| Contstraint | $X$ |

Table 6: HDM Types supported by the MDR

| model | |
|---|---|
| mid | name |
| 1 | ER |

| construct | | | | |
|---|---|---|---|---|
| mid | cid | name | class | isRoot |
| 1 | 1 | entity | Nodal | true |
| 1 | 2 | attribute | Link-Nodal | true |
| 1 | 3 | relationship | Link | true |

| scheme | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| mid | cid | pos | type | argMid | argCid | isKey | HdmType | lower | upper |
| 1 | 1 | 1 | name | - | - | true | $N_1$ | 1 | 1 |
| 1 | 2 | 1 | construct | 1 | 1 | true | $P_1$ | 1 | 1 |
| 1 | 2 | 2 | name | - | - | true | $N_2$ | 1 | 1 |
| 1 | 3 | 1 | name | - | - | true | $N_1$ | 1 | 1 |
| 1 | 3 | 2 | construct | 1 | 1 | true | $P_1$ | 1 | 1 |
| 1 | 3 | 3 | construct | 1 | 1 | true | $P_2$ | 1 | 1 |

Figure 5: ER Model Represented in the MDR

The type construct simply means that this argument must refer to an existing construct of the type referred to by argCid and argMid. The type alternation means that any one of the schemes of the construct referred to can appear as the argument. It could be used, for instance to state that an attribute is always associated with an existing entity or relationship. In this case the construct that is referenced would have two scheme arguments of type construct and it would probably have isRoot set to false as it is unlikely to be a construct of the modelling language itself. The type sequence is similar to alternation, the difference being that all the scheme arguments of the referenced construct, in order, make up the scheme argument.

# D ER Example

Figure **??** is an example of how to represent simple **entity relationship (ER)** [**?**], which contains entities, simple attributes on entities, and binary relationships without cardinality constraints.

In this example there is a direct one-to-one correspondence between the modelling language constructs and the HDM types. Note that although attribute has HDM type Link-Nodal, it has no $N_1$ in the constructor. This means that the HDM name of the edge between the attribute node and entity node has no name. E.g.

| Construct | Resulting HDM |
|---|---|
| ER.entity(person) | ⟨⟨person⟩⟩ |
| ER.attribute(person,age) | ⟨⟨null,person,age⟩⟩ |
| ER.entity(department) | ⟨⟨department⟩⟩ |
| ER.relationship(works_in,person,department) | ⟨⟨works_in,person,deparment⟩⟩ |

If we allow relationships between more than two entities then we simply change the upper limit on scheme(1,3,2) or scheme(1,3,3) to 0. If we also wanted to allow attributes on relationships then we need to define an alternation for the attribute to refer to.

| construct | | | | |
|---|---|---|---|---|
| mid | cid | name | class | isRoot |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 1 | 4 | entity_relationship | constraint | false |

This defines the construct and states that it is not a user construct of the modelling language. The class constraint simply means that an instance of this construct does not directly relate to any HDM.

We now need to define the alternatives for the construct.

| scheme | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| mid | cid | pos | type | argMid | argCid | isKey | HdmType | lower | upper |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 1 | 4 | 1 | consruct | 1 | 1 | true | $C$ | 1 | 1 |
| 1 | 4 | 2 | name | - | - | true | $C$ | 1 | 1 |

Note that the construct itself is not defined as an alternation, a reference to it specifies what its arguments mean. Also note that the HdmType is $C$, this simply means that there is no correspinding HDM for this argument (as must be the case as the construct itself is of type constraint).

We can now change the definition of attribiute so that the referenced construct $P_1$ can be either an entity or relationship.

| scheme | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| mid | cid | pos | type | argMid | argCid | isKey | HdmType | lower | upper |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 1 | 2 | 1 | consruct | 1 | 4 | true | $P_1$ | 1 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

We can now add an attribute to the works_in relationship.

| Construct | Resulting HDM |
|---|---|
| ER.attribute((works_in, person, department), salary) | $\ll null, \ll works\_in, person, deparment \gg, salary \gg$ |

Note that we have to name all the key scheme positions of the relationship works_in.