Comparing and Transforming Between Data Models via an Intermediate Hypergraph Data Model

Michael Boyd[1] and Peter McBrien[2]

[1] PSA Parts Ltd, London SW19 3UA, mb@psaparts.co.uk
[2] Dept. of Computing, Imperial College, London SW7 2AZ, pjm@doc.ic.ac.uk

Abstract. Data integration is frequently performed between heterogeneous data sources, requiring that not only a schema, but also the data modelling language in which that schema is represented must be transformed between one data source and another.
This paper describes an extension to the hypergraph data model (HDM), used in the AutoMed data integration approach, that allows constraint constructs found in static data modelling languages to be represented by a small set of primitive constraint operators in the HDM. In addition, a set of five equivalence preserving transformation rules are defined that operate over this extended HDM. These transformation rules are shown to allow a bidirectional mapping to be defined between equivalent relational, ER, UML and ORM schemas.
The approach we propose provides a precise framework in which to compare data modelling languages, and precisely identifies what semantics of a particular domain one data model may express that another data model may not express. The approach also forms the platform for further work in automating the process of transforming between different data modelling languages. The use of the both-as-view approach to data integration means that a bidirectional association is produced between schemas in the data modelling language. Hence a further advantage of the approach is that composition of data mappings may be performed such that mapping two schemas to one common schema will produce a bidirectional mapping between the original two data sources.
keywords: conceptual data modelling, mappings, transformations, multiple representations.

1 Introduction

The AutoMed data integration system [8, 24] distinguishes itself as being an approach which has a clear methodology for handling a wide range of static data modelling languages in the integration process [28], as opposed to the other approaches that assume integration is always performed in a single common data model. This is achieved by allowing a user to relate the modelling constructs of a higher level modelling language such as ER, relational, UML, or ORM, to the constructs in a single lower level common data modelling language called the **hypergraph data model** (**HDM**) [39], *i.e.* using the terminology of model management [5] we perform a ModelGen to convert schemas in the higher level modelling languages into the HDM. Figure 1 illustrates this concept being applied to four schemas which might appear to be equivalent.

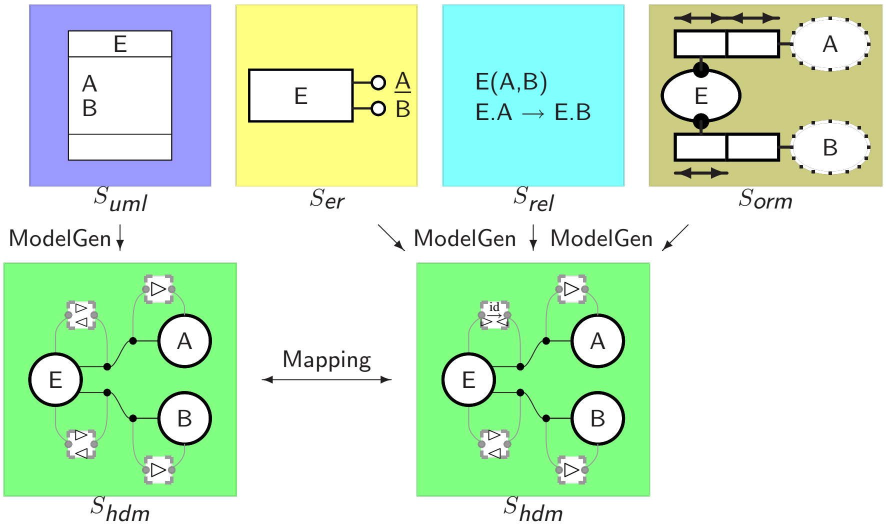


Fig. 1. Conceptual modelling languages represented in the HDM

In [28] a general approach was proposed showing how the data aspects of higher level modelling languages were modelled as nodes and edges in the HDM, with the constraints of the higher level modelling language being represented by writing constraint formulae over the HDM. For example, the ER schema in Figure 1 has an entity labelled E that we represent by a node E in the HDM schema (represented by a black outlined circle), and the attributes A and B of E are also represented as nodes in the HDM, together with edges (the thick black lines) associating them to the node representing E. The fact that each entity instance has only one associated attribute instance in each of A and B is represented by constraint rules in the HDM (the dash grey boxes, introduced in [9] and extended in this paper), as is the fact that A is a key attribute of E. Using the rules for ModelGen presented in Section 2, we will see that the ER, relational and ORM schemas in Figure 1 all produce the same HDM schema, and the UML schema produces an HDM schema with one difference in the constraints. In Section 3 we will describe equivalence preserving rules for the HDM that can be used to map between equivalent HDM schemas: this example is a case where the two HDM schemas shown are not equivalent, and hence the rules do not permit us to 'lose' the extra $\stackrel{\text{id}}{\rightarrow}$ constraint.

The concept of using graphs as an underlying representation for higher level modelling languages has been used in modelling relational schemas [51], and for OO and ER schemas [49], and we argue is an intuitive assumption to make. It also reduces all schemas to an irreducible form [19], and in the context of relational databases has recently been identified as a sixth normal form [13, 12].

This paper extends the approach of [28] to represent the high level modelling language constraints using a set of primitive constraint operators on the HDM. This paper also shows how we may relate the ER, relational, UML and ORM higher level modelling languages — perform **intermodel transformations** — by the application of five types of equivalence rules on the HDM and its primitive constraint operators. This work is a considerable enhancement of our earlier work presented in [9] in that:

1. We give a formal definition of the constraint operators in an extension of the HDM definition found in [39], and have added an additional constraint operator.
2. We give a set of mapping rules to exactly define how the higher level modelling languages are converted into these new constraint operators, and in addition we now review how ORM is modelled using the constraints. We also consider more advanced modelling language features, such as generalisation hierarchies, look-here and look-across cardinality constraints, candidate keys, and n-ary relationships.
3. We define how **both-as-view** (**BAV**) [30] data integration rules can be generated, and use the properties of these BAV rules to demonstrate when we have equivalent higher level schemas, and when there is information loss in the mapping process.

Our approach differs from other work in the area (reviewed in Section 5) in that we use hypergraphs as the common data modelling language combined with producing BAV transformations that exactly define the nature of the relationship (equivalence or non-equivalence) between schemas on a construct by construct basis. Our approach forms a framework, where the constraints we propose in the HDM, and the equivalence preserving transformations, can be extended to handle new modelling languages. Thus we provide a platform for the conversion between any data modelling language, with the limitation that our current work assumes that the data model must have set-based semantics. We also leave for future work the consideration of types in the data models: we make the crude assumption for our current work that the data types match in the data models being compared. This paper demonstrates the approach being applied by converting between the major construct types of ER, relational, UML class and ORM modelling languages, and hence has wide applicability in data modelling.

In addition to providing a mechanism for comparing the expressiveness of modelling languages, the proposed primitive constraints and set of equivalence rules also forms the basis for a method of automating the translation between modelling languages, based on descriptions of their constructs. This would involve further development of an algorithm that would determine which equivalence rules need to be applied to the HDM schema of one higher level schema to form a valid HDM schema of another higher level schema.

The remainder of this paper is structured as follows. Section 2 reviews how to describe higher level data modelling languages by relating them to the HDM schema. The HDM language is extended with a set of constraint operators that form a language used to model the constraints in higher level data modelling languages. Section 3 details how we approach the transformation between schemas in different modelling languages by applying equivalence rules to the HDM schemas, thereby relating basic constructs of the higher level modelling languages with each other. Section 4 considers how some extended operators of these languages are related. Section 5 discusses some related work, and we give a summary and discuss future work in Section 6.

2 Describing a Data Modelling Language

We now review the HDM (first defined in [39]), giving slightly modified definitions that reflect the syntax used in later work on the HDM and the AutoMed implementation. Then we present an example **universe of discourse** (**UoD**), and use it to illustrate in general how the HDM may be used to represent ER, relational, UML and ORM schemas. To keep the initial discussion reasonably concise, we leave some advanced higher level modelling features until Section 4.

We define first the notion of a HDM **schema**, which is the structure in which data may be held.

Definition 1 HDM Schema

Given a set of $Names$ that we may use for modelling the real world, an HDM **schema**, S, is a triple $\langle Nodes, Edges, Cons\rangle$ where:

- $Nodes \subseteq \{\langle\langle n_n\rangle\rangle \mid n_n \in Names\}$
  i.e. $Nodes$ is a set of nodes in the graph, each denoted by its name enclosed in double chevron marks.
- $Schemes = Nodes \cup Edges$
- $Edges \subseteq \{\langle\langle n_e, s_1, \ldots, s_n\rangle\rangle \mid$
  $n_e \in Names \cup \{\_\} \wedge s_1 \in Schemes \wedge \ldots \wedge s_n \in Schemes\}$
  i.e. $Edges$ is a set of edges in the graph where each edge is denoted by its name, together with the list of nodes/edges that the edge connects, enclosed in double chevron marks.
- $Cons \subseteq \{c(s_1, \ldots, s_n) \mid c \in Funcs \wedge s_1 \in Schemes \wedge \ldots \wedge s_n \in Schemes\}$
  i.e. $Cons$ is a set of boolean-valued functions (*i.e.* constraints) whose variables are members of $Schemes$ and where the set of functions $Funcs$ forms the HDM constraint language.

□

The first three items in Definition 1 define a labelled, directed, nested hypergraph (a **hyperedge** is an edge that connects more than two nodes in a graph, and these edges are **nested** in the sense that hyperedges can themselves participate in hyperedges). The last item in Definition 1 will be refined into the constraint language which is one of the contributions of this paper. We list in Example 1 the contents of an example HDM schema that we shall later, in Figure 3, show to be equivalent to an ER schema. Note how the names of edges are sometimes given as the character '_' representing an unnamed edge, and also note that one of the edges is a nested edge, connecting the node ⟨⟨result:grade⟩⟩ to another edge ⟨⟨result,student,course⟩⟩.

Example 1 An HDM schema

$Nodes = \{$⟨⟨ug⟩⟩, ⟨⟨ug:ppt⟩⟩, ⟨⟨student⟩⟩, ⟨⟨student:name⟩⟩, ⟨⟨student:sid⟩⟩, ⟨⟨result:grade⟩⟩, ⟨⟨course⟩⟩, ⟨⟨course:code⟩⟩, ⟨⟨course:dept⟩⟩$\}$

$Edges = \{$⟨⟨_,ug,ug:ppt⟩⟩, ⟨⟨_,student,student:sid⟩⟩, ⟨⟨_,student,student:name⟩⟩, ⟨⟨result,student,course⟩⟩, ⟨⟨_,⟨⟨result,student,course⟩⟩,result:grade⟩⟩, ⟨⟨_,course,course:dept⟩⟩, ⟨⟨_,course,course:code⟩⟩$\}$

□

The fourth component of the HDM schema definition states any extra constraints that all instances of the schema must satisfy. Note that the definition of constraints in [39] makes no restrictions on what the constraint language is. This papers proposes a restricted constraint language, which we show is able to represent constraints in higher level modelling languages, and forms a basis for writing transformations between schemas in those higher level modelling languages. Before defining these constraints we give in Definition 2 a definition of an **instance** of a schema, simplified from the definition in [39].

Definition 2 HDM instance

Given an HDM schema S, an **instance** I of S is a structure for which there exists a function $\mathit{Ext}_{S,I} : Schemes \rightarrow P(Seq(Vals))$ where $Vals$ is the set of values we wish to model as being in the domain of our data model, Seq gives any sequence of those values, and P forms the power set of those sequences. We also have the restriction that:

1. each tuple $\langle a_1, \ldots, a_n \rangle \in \mathit{Ext}_{S,I}(\langle\langle n_e, s_1, \ldots, s_n \rangle\rangle)$, $a_i \in \mathit{Ext}_{S,I}(s_i)$ for all $1 \leq i \leq n$;
2. for every $c \in Cons$, the expression $c(v_1/\mathit{Ext}_{S,I}(v_1), \ldots, v_n/\mathit{Ext}_{S,I}(v_n))$ evaluates to true, where $v_1, \ldots, v_n$ are the variables of c.

We call $Ext_{S,I}(s)$ the **extent** of scheme $s \in Schemes$. □

Note that item (1) of Definition 2 enforces that the extent of an edge is drawn from values present in the extents of nodes and other edges it connects. Note that no restriction is put on the extent of nodes: this would form the basis of **typing** in the HDM, where a type $T \subseteq P(Seq(Vals))$ is associated to each $s \in Schemes$ such that for all I $\mathit{Ext}_{S,I}(s) \subseteq T$. Also note that the semantics of the schema are set based, and hence we at present can only use the HDM to accurately model data modelling languages with set based semantics. Dealing with typing of data, and bag or list based semantics will be the subject of our future work.

The combination of HDM schema and instance, together with an extension mapping function Ext forms what we will term an HDM data source in Definition 3.

Definition 3 HDM data source

A **data source** is a triple $\langle S, I, Ext_{S,I} \rangle$ where S is a schema, I is an instance of S and $Ext_{S,I}$ is an extension mapping from S to I. □

We now introduce to the HDM a set of six primitive constraints that we are proposing as a practical solution to model the constraints of the higher level modelling language in an analogous manner to how the nodes and edges of the HDM models the data aspects of higher level modelling language. The set of six constraint operators might need to be extended in the future to handle a wider range of high level modelling language constructs, but we will demonstrate in this paper that our six constraints can deal with a wide range of modelling concepts used in practice.

In the following descriptions, any variable beginning with s denotes a member of $Schemes$. For each definition we give both a functional form (*e.g.* $\mathbf{inclusion}(s_1, s_2)$, useful for some of our mapping rules that talk in general about a constraint $op(\ldots)$)

and an equivalent infix form (*e.g.* $s_1 \subseteq s_2$, which is used in the diagrams and in most of the descriptions). When using these definitions later, we assume that where a tuple of schemes $\langle s_1, \ldots, s_m \rangle$ is used in a constraint definition, then s_1 may be used as the singleton tuple $\langle s_1 \rangle$. For example, we can write $s_1 \lhd s$ as a shorthand for $\langle s_1 \rangle \lhd s$. Several of the constraint definitions use a **project** function defined in Definition 4 that provides a method of producing a kind of view of an HDM edge restricted to contain a subset of the nodes or edges that the edge connects.

Definition 4 HDM project

The HDM **project** function $\pi(\langle s_x, \ldots, s_y \rangle, s, t)$, which takes a tuple of schemes $\langle s_x, \ldots, s_y \rangle$ that must appear in edge s, together with a tuple t that appears in the extent of s, returns the values in tuple t that corresponds to the schemes $\langle s_x, \ldots, s_y \rangle$:

$$\pi(\langle s_x, \ldots, s_y \rangle, \langle\!\langle n_e, s_1, \ldots, s_x, \ldots, s_y, \ldots, s_n \rangle\!\rangle, \langle a_1, \ldots, a_x, \ldots, a_y, \ldots, a_n \rangle) = \langle a_x, \ldots, a_y \rangle$$

If the tuple t is omitted, a set of values is obtained:

$$Ext_{S,I}(\pi(\langle s_x, \ldots, s_y \rangle, \langle\!\langle n_e, s_1, \ldots, s_x, \ldots, s_y, \ldots, s_n \rangle\!\rangle)) = \{\langle s_x, \ldots, s_y \rangle \mid \langle s_1, \ldots, s_x, \ldots, s_y, \ldots, s_n \rangle \in Ext_{S,I}(\langle\!\langle n_e, s_1, \ldots, s_x, \ldots, s_y, \ldots, s_n \rangle\!\rangle)\} \quad \square$$

Definition 5 HDM Constraints The minimum HDM constraint language should comprise of $Funcs$={inclusion,exclusion,union,mandatory,unique,reflexive}, where the six functions have the following semantics:

1. $\mathbf{inclusion}(s_1, s_2) \equiv s_1 \subseteq s_2$
   States that the extent of s_1 is always a subset of the extent of s_2, *i.e.* for all I, $Ext_{S,I}(s_1) - Ext_{S,I}(s_2) = \emptyset$
2. $\mathbf{exclusion}(s_1, \ldots, s_n) \equiv s_1 \not\lhd \ldots \not\lhd s_n$
   for all $1 \leq x < y \leq n$, and for all I, $Ext_{S,I}(s_x) \cap Ext_{S,I}(s_y) = \emptyset$
3. $\mathbf{union}(s, s_1, \ldots, s_n) \equiv s = s_1 \cup \ldots \cup s_n$
   for all I, $Ext_{S,I}(s) = Ext_{S,I}(s_1) \cup \ldots \cup Ext_{S,I}(s_n)$
4. $\mathbf{mandatory}(\langle s_1, \ldots, s_m \rangle, s) \equiv \langle s_1, \ldots, s_m \rangle \rhd s$
   States that every combination of the values in extents of $s_1, \ldots, s_m$ must appear at least once in the extent of edge s that connects them, *i.e.* for all I
   $$\{\langle a_1, \ldots, a_m \rangle \mid a_1 \in Ext_{S,I}(s_1) \wedge \ldots \wedge a_m \in Ext_{S,I}(s_m)\} - \{\langle \pi(s_1, s, t), \ldots, \pi(s_m, s, t) \rangle \mid t \in Ext_{S,I}(s)\} = \emptyset$$
5. $\mathbf{unique}(\langle s_1, \ldots, s_m \rangle, s) \equiv \langle s_1, \ldots, s_m \rangle \lhd s$
   States that every combination of the values in extents of $s_1, \ldots, s_m$ must appear at most once in the extent of edge s that connects them, *i.e.* for all I
   $$\{t \mid t \in Ext_{S,I}(s) \wedge t' \in Ext_{S,I}(s) \wedge t \neq t' \wedge \pi(s_1, s, t) = \pi(s_1, s, t') \wedge \ldots \wedge \pi(s_m, s, t) = \pi(s_m, s, t')\} = \emptyset$$
6. $\mathbf{reflexive}(s_1, s) \equiv s_1 \xrightarrow{\text{id}} s$
   If an instance of scheme s_1 appears in edge s, then one of the instances of s must be an identity tuple, *i.e.* for all I:
   $$\{\pi(s_1, s, t) \mid t \in Ext_{S,I}(s)\} - \{\pi(s_1, s, t) \mid t \in Ext_{S,I}(s) \wedge t = \langle \pi(s_1, s, t), \pi(s_1, s, t) \rangle\} = \emptyset \quad \square$$

Example 2 may be combined with Example 1 to give a complete HDM as represented diagrammatically in Figure 3(b). The constraint language might appear too fine grain, but we will show in Section 3 that when writing inter model transformations it is useful to be able to test for, to add or to delete constraints expressed at this level of detail. Also, the six constraint operators defined here are not meant to be definitive: handling other modeling language constructs in addition to those presented later in this section might require us to introduce additional constraint primitives.

Example 2 Constraints in an HDM schema

$Cons = \{\langle\langle \mathsf{ug} \rangle\rangle \lhd \langle\langle \mathsf{\_,ug,ug{:}ppt} \rangle\rangle, \langle\langle \mathsf{ug} \rangle\rangle \rhd \langle\langle \mathsf{\_,ug,ug{:}ppt} \rangle\rangle,$
$\langle\langle \mathsf{ug{:}ppt} \rangle\rangle \rhd \langle\langle \mathsf{\_,ug,ug{:}ppt} \rangle\rangle, \langle\langle \mathsf{ug} \rangle\rangle \subseteq \langle\langle \mathsf{student} \rangle\rangle,$
$\langle\langle \mathsf{student} \rangle\rangle \lhd \langle\langle \mathsf{\_,student,student{:}sid} \rangle\rangle, \langle\langle \mathsf{student} \rangle\rangle \rhd \langle\langle \mathsf{\_,student,student{:}sid} \rangle\rangle,$
$\langle\langle \mathsf{student{:}sid} \rangle\rangle \rhd \langle\langle \mathsf{\_,student,student{:}sid} \rangle\rangle, \langle\langle \mathsf{student} \rangle\rangle \lhd \langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle,$
$\langle\langle \mathsf{student} \rangle\rangle \rhd \langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle, \langle\langle \mathsf{student} \rangle\rangle \xrightarrow{\text{id}} \langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle,$
$\langle\langle \mathsf{student{:}name} \rangle\rangle \rhd \langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle,$
$\langle\langle \mathsf{result{:}grade} \rangle\rangle \rhd \langle\langle \mathsf{\_,}\langle\langle \mathsf{result,student,course} \rangle\rangle\mathsf{,result{:}grade} \rangle\rangle,$
$\langle\langle \mathsf{result,student,course} \rangle\rangle \lhd \langle\langle \mathsf{\_,}\langle\langle \mathsf{result,student,course} \rangle\rangle\mathsf{,result{:}grade} \rangle\rangle,$
$\langle\langle \mathsf{course} \rangle\rangle \lhd \langle\langle \mathsf{\_,course,course{:}dept} \rangle\rangle, \langle\langle \mathsf{course} \rangle\rangle \rhd \langle\langle \mathsf{\_,course,course{:}dept} \rangle\rangle,$
$\langle\langle \mathsf{course{:}dept} \rangle\rangle \rhd \langle\langle \mathsf{\_,course,course{:}dept} \rangle\rangle, \langle\langle \mathsf{course} \rangle\rangle \lhd \langle\langle \mathsf{\_,course,course{:}code} \rangle\rangle,$
$\langle\langle \mathsf{course} \rangle\rangle \rhd \langle\langle \mathsf{\_,course,course{:}code} \rangle\rangle, \langle\langle \mathsf{course} \rangle\rangle \xrightarrow{\text{id}} \langle\langle \mathsf{\_,course,course{:}code} \rangle\rangle,$
$\langle\langle \mathsf{course{:}code} \rangle\rangle \rhd \langle\langle \mathsf{\_,course,course{:}code} \rangle\rangle\}$ □

Most of these constraint operators have been used before in the context of describing single modelling languages. In particular, mandatory and unique constraints (though more limited in definition) have been used in a hypergraph model for relational schemas in [51], and inclusion constraints appear in [42]. However the combination of our mandatory, unique and reflexive constraints give a rich framework in which to express various notions of cardinality constraints and keys found in higher level modelling languages.

To illustrate these constraints, consider from the HDM schema in Example 1 the nodes $\langle\langle \mathsf{student} \rangle\rangle$ and $\langle\langle \mathsf{student{:}name} \rangle\rangle$ connected by edge $\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle$. As will be discussed in depth later, this might be used to model an ER entity called student with an attribute name. Suppose we have five data sources with the same schema S but different instances I_1, I_2, I_3, I_4, I_5 for which:

$x \in 1,2,3,4 : Ext_{S,I_x}(\langle\langle \mathsf{student} \rangle\rangle) = \{1,2\}$
$Ext_{S,I_5}(\langle\langle \mathsf{student} \rangle\rangle) = \{\text{‘Peter’}, \text{‘Mike’}\}$
$x \in 1,2,3,4,5 : Ext_{S,I_x}(\langle\langle \mathsf{student{:}name} \rangle\rangle) = \{\text{‘Peter’}, \text{‘Mike’}\}$
$Ext_{S,I_1}(\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle) = \{\langle 1, \text{‘Peter’}\rangle, \langle 1, \text{‘Mike’}\rangle, \langle 2, \text{‘Mike’}\rangle\}$
$Ext_{S,I_2}(\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle) = \{\langle 2, \text{‘Peter’}\rangle, \langle 2, \text{‘Mike’}\rangle\}$
$Ext_{S,I_3}(\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle) = \{\langle 1, \text{‘Mike’}\rangle, \langle 2, \text{‘Mike’}\rangle\}$
$Ext_{S,I_4}(\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle) = \{\langle 1, \text{‘Peter’}\rangle, \langle 2, \text{‘Mike’}\rangle\}$
$Ext_{S,I_5}(\langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle) = \{\langle \text{‘Peter’}, \text{‘Peter’}\rangle, \langle \text{‘Mike’}, \text{‘Mike’}\rangle\}$

To state that every instance of $\langle\langle \mathsf{student} \rangle\rangle$ must appear in the edge (and hence that I_2 is invalid), we use the mandatory constraint:

$\langle\langle \mathsf{student} \rangle\rangle \rhd \langle\langle \mathsf{\_,student,student{:}name} \rangle\rangle$

For the ER model, this constraint would be used when the attribute is mandatory for the entity (*i.e.* to state that each student must have a name). To state that every instance of $\langle\langle\mathsf{student}\rangle\rangle$ must appear no more than once in the edge (and hence that I_1 is invalid), we use the unique constraint:

$\langle\langle\mathsf{student}\rangle\rangle \lhd \langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$

For the ER model, this constraint would be used when the attribute is not multi-valued for the entity (*i.e.* to state that each student has no more than one name). The two above constraints together model that each instance of $\langle\langle\mathsf{student}\rangle\rangle$ must appear exactly once in the edge, and hence for the ER model a mandatory and single valued attribute (*i.e.* to state that each student has exactly one name).

We will illustrate as we consider different higher level modelling languages how these general notions of cardinality may be used to represent optional and mandatory attributes, and also both look-here and look-across [45, 21] cardinality constraints on relationships/associations. The term **look-across** is used to denote the use of cardinality constraints where the cardinality written against entity x_i in an n-ary relationship between entities $x_1, \ldots, x_n$ restricts the participation of the other $n-1$ entities in the relationship. The term **look-here** is used to denote cardinality constraints that restrict the participation of x_i in the relationship.

Here, let us consider the general notion of a key, and how it is represented using our HDM constraints. If we want to model that $\langle\langle\mathsf{student{:}name}\rangle\rangle$ is a **candidate key** for $\langle\langle\mathsf{student}\rangle\rangle$ — that is to say that each value in $\langle\langle\mathsf{student{:}name}\rangle\rangle$ is in a one-to-one correspondence with a value in $\langle\langle\mathsf{student}\rangle\rangle$ (and thus that only I_4 and I_5 are valid out of the five instances above) — we must in addition make $\langle\langle\mathsf{student{:}name}\rangle\rangle$ also be 1:1 in the edge:

$\langle\langle\mathsf{student{:}name}\rangle\rangle \lhd \langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$
$\langle\langle\mathsf{student{:}name}\rangle\rangle \rhd \langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$

Thus in the ER case, name would be unique to each student. If we want to model that $\langle\langle\mathsf{student{:}name}\rangle\rangle$ is the **primary key** for $\langle\langle\mathsf{student}\rangle\rangle$ — that is to say that the values in $\langle\langle\mathsf{student{:}name}\rangle\rangle$ equal those in $\langle\langle\mathsf{student}\rangle\rangle$ (thus only I_5 is valid) — we must make $\langle\langle\mathsf{student}\rangle\rangle$ be reflexive in the edge:

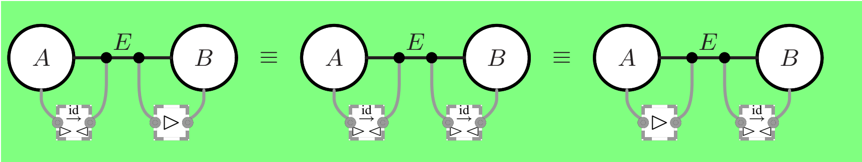
$\langle\langle\mathsf{student}\rangle\rangle \stackrel{\mathrm{id}}{\rightarrow} \langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$

In general, reflexive, mandatory and unique together enforce that the entity has the same values as the attribute, and thus that the extent of the entity is the key values of the entity. This is because mandatory and unique at both ends of an edge enforce a one-to-one mapping, and the reflexive constraint means that this one-to-one mapping is an identity function. By contrast, lack of the reflexive constraint would be used when the entity has as its extent a set of object identifiers.

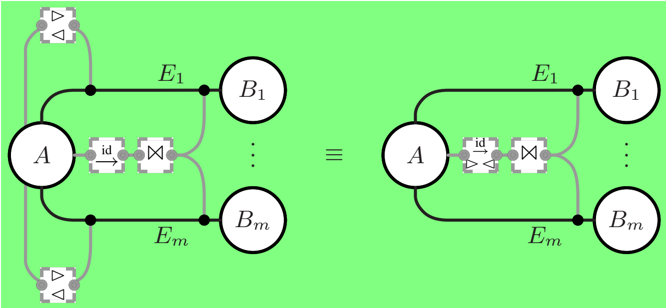
Note that we could have alternatively put the reflexive constraint on the other node as:

$\langle\langle\mathsf{student{:}name}\rangle\rangle \stackrel{\mathrm{id}}{\rightarrow} \langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$

Also note that the unique constraint on the opposite end of the edge to the reflexive constraint is redundant, since it is implied by the other constraints. Hence we draw the equivalences shown in Figure 2(a) that will be used during intermodel transformation, where one higher level modelling language might happen to place constraints in a different but equivalent manner to another higher level modelling language.



(a) Transposing a reflexive constraint across an edge



(b) Mandatory-unique constraints in joins

Fig. 2. Fundamental equivalences on HDM constraints

Compound keys will require that we have a definition of an **edge natural join** given in Definition 6. This will be used to say that the reflexive constraint applies to more that one node. The introduction of the natural join means that we have a variation of the equivalence in Figure 2(a) given in Figure 2(b) that applies across an edge natural join.

Definition 6 Natural join between HDM edges
A view over HDM edges may be formed by joining edges together to form a new virtual edge:
$\langle\langle E, A, B_1, \ldots, B_n\rangle\rangle \bowtie \langle\langle E, A, C_1, \ldots, C_m\rangle\rangle = \langle\langle E, A, B_1, \ldots, B_n, C_1, \ldots, C_m\rangle\rangle$
The extent of the virtual edge is defined by a natural join over the extent of the two joined edges:

$$Ext_{S,I}(\langle\langle E, A, B_1, \ldots, B_n, C_1, \ldots, C_m\rangle\rangle) = \{\langle x, y_1, \ldots, y_n, z_1, \ldots, z_m\rangle \mid \langle x, y_1, \ldots, y_n\rangle \in Ext_{S,I}(\langle\langle E, A, B_1, \ldots, B_n\rangle\rangle) \wedge \langle x, z_1, \ldots, z_m\rangle \in Ext_{S,I}(\langle\langle E, A, C_1, \ldots, C_m\rangle\rangle)\}$$ □

2.1 An Example UoD and Four Schemas

Figures 3(a), 4(a), 5(a) and 6(a) show four data models, in ER, relational, UML and ORM data modelling languages. These are designed to cover the same UoD, and as will be shown later, three of them have the same information capacity [32]. The schemas represent a record of students, the courses that they sit, and the grades they obtain for those courses. Some students are undergraduates, and each ug has an associated personal programming tutor ppt that other students do not have. The use of underlining in the relational and ER schemas indicates what are key attributes, and a question mark follows a nullable attribute in those schemas. In the relational schema, foreign keys are shown by using an implication between the foreign key columns and the referenced table columns. In the ER schema this foreign key may either be represented by a relationship (for example the foreign keys $\mathsf{result.name} \rightarrow \mathsf{student.name}$ and $\mathsf{result.code} \rightarrow \mathsf{course.code}$ are represented in the ER result relationship) or by a subset (for example the foreign key $\mathsf{ug.name} \rightarrow \mathsf{student.name}$ is represented by a subset between the student and ug entities).

To describe how we map between high level modelling languages and the HDM, we use a simple production rule language as described in Definition 7.

Definition 7 HDM Production Rules Higher level modelling language constructs are transformed into the HDM using production rules of the form:

$\langle\mathsf{high\ level\ construct\ name}\rangle\ \langle\mathsf{high\ level\ construct\ scheme}\rangle \leadsto \langle\mathsf{HDM\ scheme}\rangle*$

$\quad\langle\mathsf{condition}\rangle_1 \Rightarrow \langle\mathsf{HDM\ constraint}\rangle_1$

$\quad\vdots$

$\quad\langle\mathsf{condition}\rangle_n \Rightarrow \langle\mathsf{HDM\ constraint}\rangle_n$

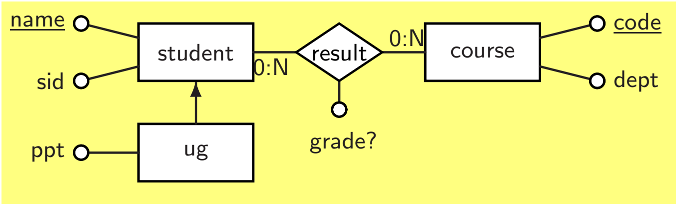
Where

- $\langle\mathsf{high\ level\ construct\ scheme}\rangle$ is the structure used to represent a higher level model construct of type $\langle\mathsf{high\ level\ construct\ name}\rangle$,
- $\langle\mathsf{HDM\ scheme}\rangle$ is the list of zero or more HDM nodes or edges used to represent those aspects $\langle\mathsf{high\ level\ construct\ scheme}\rangle$ that have an extent. Zero such schemes will be denoted using $\perp$, and the last $\langle\mathsf{HDM\ scheme}\rangle$ is used to return the entire extent of the $\langle\mathsf{high\ level\ construct\ scheme}\rangle$
- $\langle\mathsf{condition}\rangle$ is a boolean expression over elements of of $\langle\mathsf{high\ level\ construct\ scheme}\rangle$, which when satisfied, causes $\langle\mathsf{HDM\ constraint}\rangle$ to be added to the $\langle\mathsf{HDM\ scheme}\rangle$s. □

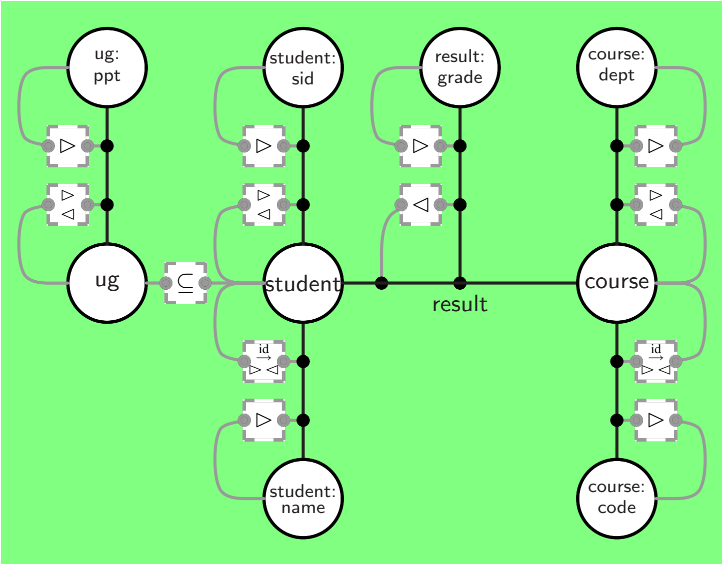
2.2 Describing an ER Modelling Language in the HDM

When the HDM is used to model a higher level modelling language, each construct in that language must be classified as being one of four types, each of which imply a different representation in the HDM. We explain how this methodology (first presented in [28]) is applied to an ER modelling language (which we describe here, see [45, 35] for surveys of variations of ER modelling languages), and illustrate our discussions by showing how the methodology may take the ER schema of Figure 3(a) and produce the HDM schema of Figure 3(b). Note that in the HDM diagrams, HDM nodes are represented by white circles with thick outlines, and HDM edges are represented by thick black lines. The HDM constraint language is represented by grey dashed boxes connected by grey lines to the nodes and edges to which the constraint applies. Edges pass through black circles in a straight line, hence any edge or constraint applying to an edge meets that edge at a angle.

A summary of the AutoMed high level schemes that represent the high level ER schema in textual format are listed in Table 1, along with the class each belongs to. A similar table could be generated for the relational, UML and ORM models considered in the following sections. The production rules presented in this sub-section map these high level model schemes into the HDM schema already given in Examples 1 and 2.



(a) An ER schema of the student-course database



(b) HDM representation of the ER schema

Fig. 3. An ER schema and its equivalent HDM schema

class	construct	scheme
nodal	entity	⟨⟨student⟩⟩
link-nodal	attribute	⟨⟨student,name,notnull⟩⟩
constraint	key	⟨⟨student,name⟩⟩
link-nodal	attribute	⟨⟨student,sid,notnull⟩⟩
nodal	entity	⟨⟨ug⟩⟩
link-nodal	attribute	⟨⟨ug,ppt,notnull⟩⟩
constraint	subset	⟨⟨student,ug⟩⟩

class	construct	scheme
nodal	entity	⟨⟨course⟩⟩
link-nodal	attribute	⟨⟨course,code,key⟩⟩
constraint	key	⟨⟨course,code⟩⟩
link-nodal	attribute	⟨⟨course,dept,notnull⟩⟩
link	relationship	⟨⟨result,student,0:N,course,0:N⟩⟩
link-nodal	attribute	⟨⟨result,grade,null⟩⟩

Table 1. AutoMed high level model schemes for the ER example schema

The methodology in [28] categorises the constructs of a higher level model as being nodal, link-nodal, link, or constraint. We illustrate these categories by considering how the constructs of an ER modelling language will be handled.

Nodal A **nodal** construct is one that may appear in isolation in a schema, such as an ER model **entity**. Using the AutoMed data integration system [8], such constructs are defined by giving a prototype **scheme** that must contain the name of a HDM node used to represent that construct. Hence we represent the ER entity $\mathsf{student}$ by the schema $\langle\langle \mathsf{student} \rangle\rangle$.

The production rule for an ER entity $\langle\langle E \rangle\rangle$ is very simple, since it states that each entity with scheme $\langle\langle E \rangle\rangle$ maps to a single HDM node $\langle\langle E \rangle\rangle$, and has no constraints:

$\mathsf{entity}\ \langle\langle E \rangle\rangle \rightsquigarrow \langle\langle E \rangle\rangle$

Link A **link** construct is one that associates other constructs with each other, and which has an extent which is drawn from those constructs, such as an ER **relationship** construct. In AutoMed, we represent ER relationships by the scheme comprising of the name of the HDM edge used to represent the construct, together with pairs of the entity names and cardinality constraints. For example, we represent the ER relationship result in Figure 3(a) by the scheme $\langle\langle \mathsf{result}, \mathsf{student}, 0{:}\mathsf{N}, \mathsf{course}, 0{:}\mathsf{N} \rangle\rangle$. The production rule uses auxiliary rules to generate the constraints in the HDM necessary to represent the cardinality constraints in the ER schema. Assuming that our ER model uses look-here semantics [45, 21], we need one line for each entity E_x that generates as appropriate $\rhd$ and $\lhd$ using a function $\mathsf{generate\_card}()$ defined in Definition 8.

$$\begin{array}{l}
\mathsf{relationship}\ \langle\langle R, E_1, L_1{:}U_1, \ldots, E_n, L_n{:}U_n \rangle\rangle \rightsquigarrow \langle\langle R, E_1, \ldots, E_n \rangle\rangle \\
\quad \mathsf{true} \Rightarrow \mathsf{generate_card}(E_1, \langle\langle R, E_1, \ldots, E_n \rangle\rangle, L_1, U_1) \\
\qquad\quad \vdots \\
\quad \mathsf{true} \Rightarrow \mathsf{generate_card}(E_n, \langle\langle R, E_1, \ldots, E_n \rangle\rangle, L_n, U_n)
\end{array}$$

Definition 8 Generation of constraints representing cardinality

This function generates cardinality constraints between a set of nodes or edges $\{NE_1, \ldots, NE_n\} \in \mathit{Schemes}$ and $E \in \mathit{Edges}$, where L may be either 0 or 1, and U may be 1 or *.

$$\begin{array}{l}
\mathsf{generate_card}\ (\langle NE_1, \ldots, NE_n \rangle, E, L, U) \rightsquigarrow \bot \\
\quad L = 1 \Rightarrow \langle NE_1, \ldots, NE_n \rangle \rhd E \\
\quad U = 1 \Rightarrow \langle NE_1, \ldots, NE_n \rangle \lhd E
\end{array}$$ □

Note that the rule in Definition 8 takes as its first argument a tuple of nodes and edges $NE_1, \ldots, NE_n$ that must appear in the edge E that is its second argument, and then produces mandatory and unique constraints to determine how many times a particular combination of values from the extent of $NE_1, \ldots, NE_n$ may appear in the extent of E.

Applying our production rule to the relationship $\langle\langle \mathsf{result}, \mathsf{student}, 0{:}\mathsf{N}, \mathsf{course}, 0{:}\mathsf{N} \rangle\rangle$ produces an edge $\langle\langle \_, \mathsf{result}, \mathsf{student} \rangle\rangle$ to represent its extent. The auxiliary constraint rules will produce no constraints, since neither of the guards within the definition of $\mathsf{generate\_card}$ will match $L = 0$ or $U = *$.

Link-Nodal A **link-nodal** construct is one that has associated values, but may only exist when associated with some other construct. They are represented in the HDM by an edge associating a new node with some existing node or edge. For example, ER **attributes** are link-nodal constructs, and the name attribute of the entity $\mathsf{student}$ is represented in AutoMed by the scheme $\langle\langle \mathsf{student}, \mathsf{name}, \mathsf{notnull} \rangle\rangle$. The production rule for ER attributes creates a node and edge.

$\mathsf{attribute}\ \langle\langle E, A, N\rangle\rangle \rightsquigarrow \langle\langle E{:}A\rangle\rangle, \langle\langle \_, E, E{:}A\rangle\rangle$
$\quad \mathsf{true} \quad\quad\quad \Rightarrow \mathsf{generate\_card}(\langle\langle E{:}A\rangle\rangle, \langle\langle \_, E, E{:}A\rangle\rangle, 1, *)$
$\quad N = \mathsf{notnull} \Rightarrow \mathsf{generate\_card}(\langle\langle E\rangle\rangle, \langle\langle \_, E, E{:}A\rangle\rangle, 1, 1)$
$\quad N = \mathsf{null} \quad \Rightarrow \mathsf{generate\_card}(\langle\langle E\rangle\rangle, \langle\langle \_, E, E{:}A\rangle\rangle, 0, 1)$

Thus the production rule when applied to the ER attribute ⟨⟨course,dept,notnull⟩⟩ produces the node ⟨⟨course:dept⟩⟩ and the edge ⟨⟨_,course,course:dept⟩⟩ to represent the extent of the attribute. The first auxiliary constraint rule produces ⟨⟨course:dept⟩⟩ $\rhd$ ⟨⟨_,course,course:dept⟩⟩, the second produces ⟨⟨course⟩⟩ $\rhd$ ⟨⟨_,course,course:dept⟩⟩ and ⟨⟨course⟩⟩ $\lhd$ ⟨⟨_,course,course:dept⟩⟩ (since both guards in the generate_card are met), and then the last rule fails to match in the guard.

Note that since the grade attribute is optional, we obtain just two constraints when the production rule is used on ⟨⟨result,grade,null⟩⟩:
⟨⟨result:grade⟩⟩ $\rhd$ ⟨⟨_,⟨⟨result,student,course⟩⟩,result:grade⟩⟩
⟨⟨result,student,course⟩⟩ $\lhd$ ⟨⟨_,⟨⟨result,student,course⟩⟩,result:grade⟩⟩

Note that in our modelling of the ER model (and relational and UML languages), the fact that attribute names are prefixed by the associated entity name reflects a deliberate choice made when defining the construct. One could alternatively say that attribute names are globally unique, which would change the HDM graph to have just one node ⟨⟨name⟩⟩ to represent both the ⟨⟨student, name, notnull⟩⟩ and ⟨⟨ug, name, notnull⟩⟩ relational columns, but this would not give the correct semantics for a normal ER model. The alternative global naming choice will be used in modelling the value types of ORM models.

In Figure 3(b) it should be noted that the syntax is not ambiguous, but does need careful reading. Each $\rhd$ or $\lhd$ always has a node or edge on its left hand side that appears in the edge on its right hand side. We use this fact to ignore which 'side' we connect $\rhd$ and $\lhd$ constraints to in the diagram. This makes the diagrams more tidy in appearance. (Note that this is different from the approach we followed in our earlier work [9]). Therefore the ⟨⟨course:dept⟩⟩ to ⟨⟨_,course,course:dept⟩⟩ mandatory constraint is drawn using $\rhd$ in the constraint box.

Constraint A **constraint** construct is one that has no associated extent, but instead limits the extent of the constructs it connects to. An example of a constraint construct is the ER model **subset** relationship. For example, the subset between ug and student is represented in AutoMed by the scheme ⟨⟨student, ug⟩⟩.
$\mathsf{subset}\ \langle\langle E, E_s\rangle\rangle \rightsquigarrow \perp$
$\quad \mathsf{true} \Rightarrow \langle\langle E_s\rangle\rangle \subseteq \langle\langle E\rangle\rangle$

ER **generalisations** are another example of constraint constructs. For example, a generalisation that specified the children entities $\langle\langle E_1\rangle\rangle, \ldots, \langle\langle E_n\rangle\rangle$ are disjoint subsets of some parent entity $\langle\langle E\rangle\rangle$ could be defined by the following rule:
$\mathsf{generalisation}\ \langle\langle E, E_1, \ldots, E_n\rangle\rangle \rightsquigarrow \perp$
$\quad \mathsf{true} \Rightarrow \langle\langle E_1\rangle\rangle \subseteq \langle\langle E\rangle\rangle$
$\quad\quad\quad \Rightarrow \vdots$
$\quad \mathsf{true} \Rightarrow \langle\langle E_n\rangle\rangle \subseteq \langle\langle E\rangle\rangle$
$\quad \mathsf{true} \Rightarrow \langle\langle E_1\rangle\rangle \not\bowtie \ldots \not\bowtie \langle\langle E_n\rangle\rangle$

Our example ER schema in Figure 3(a) contains no generalisations, but we will discuss and compare advanced modelling constructs of various data modelling languages in Section 4.

The final constraint in our ER model is the definition of the **key** of an entity, which serves to denote the set of its attributes that may be used to identify instances of the attribute.

$\mathsf{key}\ \langle\!\langle E, A_1, \ldots, A_n \rangle\!\rangle \leadsto \bot$

$\quad \mathsf{true} \Rightarrow \langle\!\langle E \rangle\!\rangle \xrightarrow{\mathrm{id}} \langle\!\langle \_, E, E{:}A_1 \rangle\!\rangle \bowtie \ldots \bowtie \langle\!\langle \_, E, E{:}A_n \rangle\!\rangle$

The constraint limits the instances of the entity to be an identity with the join of its key attributes (we will give an example of how this type of constraint works when looking at the relational result table in the next section).

2.3 Describing relational schemas in the HDM

ug	
name	ppt
Mary	NR
Jane	SK

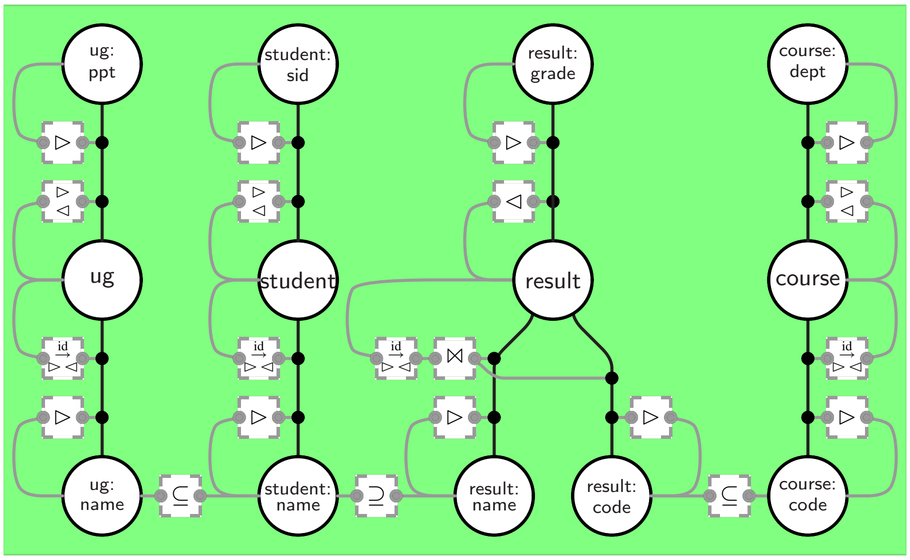
student	
name	sid
Mary	1
John	2
Jane	3
Fred	4

course	
code	dept
DB	CS
Fin	CS
Geo	Maths

result		
code	name	grade?
DB	Mary	A
Fin	Jane	C
Fin	Fred	null
Geo	Fred	A
Geo	John	B

ug.name $\rightarrow$ student.name
result.name $\rightarrow$ student.name
result.code $\rightarrow$ course.code

(a) Relational database schema and data



(b) HDM representation of relational database schema

Fig. 4. A relational schema for the student-course database

Having reviewed the general methodology for representing higher level modelling languages in the HDM in the previous subsection, we will now apply the methodology to the relational schema. Relational model **table**s are nodal constructs, and hence we represent the table $\mathsf{student}$ by the scheme $\langle\langle\mathsf{student}\rangle\rangle$, and the table result by $\langle\langle\mathsf{result}\rangle\rangle$. The production rule for translating such schemes into the HDM is as follows.

$\mathsf{table}\ \langle\langle T\rangle\rangle \rightsquigarrow \langle\langle T\rangle\rangle$

Relational model **column**s are link-nodal constructs, and hence are modelled by a scheme containing a HDM node that represents the construct it depends on, followed by the name of the HDM node that represents the column, followed by the constraint on whether the attribute may be null. For example, the name column of table $\mathsf{student}$ is represented by the scheme $\langle\langle\mathsf{student},\mathsf{name},\mathsf{notnull}\rangle\rangle$. In the HDM, this becomes a node $\langle\langle\mathsf{student{:}name}\rangle\rangle$ to represent values of the column/attribute, and the nameless edge $\langle\langle\_,\mathsf{student},\mathsf{student{:}name}\rangle\rangle$ to represent the association of these values to table/entity $\langle\langle\mathsf{student}\rangle\rangle$.

$\mathsf{column}\ \langle\langle T,C,N\rangle\rangle \rightsquigarrow \langle\langle T{:}C\rangle\rangle, \langle\langle\_,T,T{:}C\rangle\rangle$

$\quad\mathsf{true} \Rightarrow \mathsf{generate\_card}(\langle\langle T{:}C\rangle\rangle, \langle\langle\_,T,T{:}C\rangle\rangle, 1, *)$

$\quad N = \mathsf{notnull} \Rightarrow \mathsf{generate\_card}(\langle\langle T\rangle\rangle, \langle\langle\_,T,T{:}C\rangle\rangle, 1, 1)$

$\quad N = \mathsf{null} \Rightarrow \mathsf{generate\_card}(\langle\langle T\rangle\rangle, \langle\langle\_,T,T{:}C\rangle\rangle, 0, 1)$

The definition of relational columns and ER attributes are very similar, and as can be seen by comparing Figures 4 and 3, produce similar results in the HDM.

The **primary key** construct of the relational model is a constraint construct. The constraint specifies that the natural join between its key columns gives the extent of the table. Although a slightly unconventional notion of primary key, this definition fits well with the sixth normal form [13, 12], since that normalises tables to have one table for the primary key columns, and then an additional table for each non-key column of the pre-normalised table. The schema of a constraint simply needs to list the table and the columns. For example, the primary key of $\langle\langle\mathsf{result}\rangle\rangle$ would be represented in the HDM by $\langle\langle\mathsf{result}\rangle\rangle \xrightarrow{\text{id}} (\langle\langle\_,\mathsf{result},\mathsf{result{:}code}\rangle\rangle \bowtie \langle\langle\_,\mathsf{result},\mathsf{result{:}name}\rangle\rangle)$. The production rule to produce the HDM constraints is as follows:

$\mathsf{primary\_key}\ \langle\langle T, C_1, \ldots, C_n\rangle\rangle \rightsquigarrow \bot$

$\quad\mathsf{true} \Rightarrow \langle\langle T\rangle\rangle \xrightarrow{\text{id}} \langle\langle\_,T,T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle\_,T,T{:}C_n\rangle\rangle)$

Since any key column must also be a $\mathsf{notnull}$ column in a valid relational schema, this rule need only add the fact that the join of the edges leading to nodes representing key columns is reflexive. Since the table will be connected to these edges using mandatory and unique, it follows that the join is also mandatory and unique, as shown for the result node in Figure 4(b). For the primary key scheme $\langle\langle\mathsf{result},\mathsf{name},\mathsf{code}\rangle\rangle$ for the result table, the production rule generates the reflexive constraint $\langle\langle\mathsf{result}\rangle\rangle \xrightarrow{\text{id}} \langle\langle\_,\mathsf{result},\mathsf{result{:}code}\rangle\rangle \bowtie \langle\langle\_,\mathsf{result},\mathsf{result{:}name}\rangle\rangle$. For example, with the relational data shown in Figure 4(a), this constraint along with the already stated mandatory and unique constraints enforce the following type of instantiation of the $\langle\langle\mathsf{result}\rangle\rangle$ node and key edges:

$\quad\langle\langle\mathsf{result}\rangle\rangle = \{\langle\mathsf{DB},\mathsf{Mary}\rangle, \langle\mathsf{Fin},\mathsf{Jane}\rangle, \langle\mathsf{Fin},\mathsf{Fred}\rangle, \ldots\}$

$\quad\langle\langle\_,\mathsf{result},\mathsf{result{:}name}\rangle\rangle =$

$\qquad\{\langle\langle\mathsf{DB},\mathsf{Mary}\rangle,\mathsf{Mary}\rangle, \langle\langle\mathsf{Fin},\mathsf{Jane}\rangle,\mathsf{Jane}\rangle, \langle\langle\mathsf{Fin},\mathsf{Fred}\rangle,\mathsf{Fred}\rangle, \ldots\}$

$\langle\langle \_,\mathsf{result},\mathsf{result{:}code}\rangle\rangle =$
$\{\langle\langle \mathsf{DB},\mathsf{Mary}\rangle,\mathsf{DB}\rangle,\langle\langle \mathsf{Fin},\mathsf{Jane}\rangle,\mathsf{Fin}\rangle,\langle\langle \mathsf{Fin},\mathsf{Fred}\rangle,\mathsf{Fin}\rangle,\ldots\}$

We represent the **foreign key** constraint by the scheme made up of a name for the constraint, the table and column(s) that are the foreign key, and the table and column(s) of the referenced table.

$\mathsf{foreign\_key}\ \langle\langle FK, T, C_1, \ldots, C_n, T_f, C_{f_1}, \ldots, C_{f_n}\rangle\rangle \rightsquigarrow \bot$
$\mathsf{true} \Rightarrow \pi_{\langle\langle T:C_1\rangle\rangle,\ldots,\langle\langle T:C_n\rangle\rangle}(\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T, T{:}C_n\rangle\rangle) \subseteq$
$\pi_{\langle\langle T_f:C_{f_1}\rangle\rangle,\ldots,\langle\langle T_f:C_{f_n}\rangle\rangle}(\langle\langle \_, T_f, T_f{:}C_{f_1}\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T_f, T_f{:}C_{f_1}\rangle\rangle)$

The somewhat complex constraint simply states that the join of the columns listed in T is a subset of the join of the columns in T_f. For the common case where foreign keys are not compound keys (*i.e.* $n = 1$), the constraint would simplify to $\langle\langle T{:}C_1\rangle\rangle \subseteq \langle\langle T_f{:}C_{f_1}\rangle\rangle$ For example, the foreign key between ug and student is represented by the scheme $\langle\langle$ug_fk, ug, name, student, name$\rangle\rangle$. Using the production rule, this scheme becomes $\langle\langle$ug:name$\rangle\rangle \subseteq \langle\langle$student:name$\rangle\rangle$ in the HDM.

Finally, the relational **candidate key** takes a similar definition to primary key, except that all that is established is a mandatory and a unique association between the table and a join of the candidate key columns.

$\mathsf{candidate\_key}\ \langle\langle T, C_1, \ldots, C_n\rangle\rangle \rightsquigarrow \bot$
$\mathsf{true} \Rightarrow \langle\langle T\rangle\rangle \lhd (\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T, T{:}C_n\rangle\rangle)$
$\mathsf{true} \Rightarrow \langle\langle T\rangle\rangle \rhd (\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T, T{:}C_n\rangle\rangle)$
$\mathsf{true} \Rightarrow (\pi_{\langle\langle T:C_1\rangle\rangle,\ldots,\langle\langle T:C_n\rangle\rangle}(\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T, T{:}C_n\rangle\rangle))\lhd$
$(\langle\langle \_, T, T{:}C_1\rangle\rangle \bowtie \ldots \bowtie \langle\langle \_, T, T{:}C_n\rangle\rangle)$

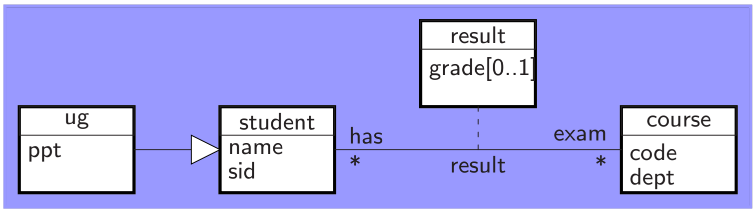
The last line ensures that the combination of columns in the candidate key appears just once in the edge formed by the join of the candidate key column edges. In the common case where the candidate key is not compound (*i.e.* $n = 1$), the last constraint simplifies to $\langle\langle T{:}C_1\rangle\rangle \lhd \langle\langle \_, T, T{:}C_1\rangle\rangle$. Thus if we added the new candidate key $\langle\langle$student,sid$\rangle\rangle$ to the example relational schema, then we would add to our existing relation HDM a $\langle\langle$student:sid$\rangle\rangle \lhd \langle\langle$_,student,student:sid$\rangle\rangle$ constraint.

2.4 Describing UML in the HDM

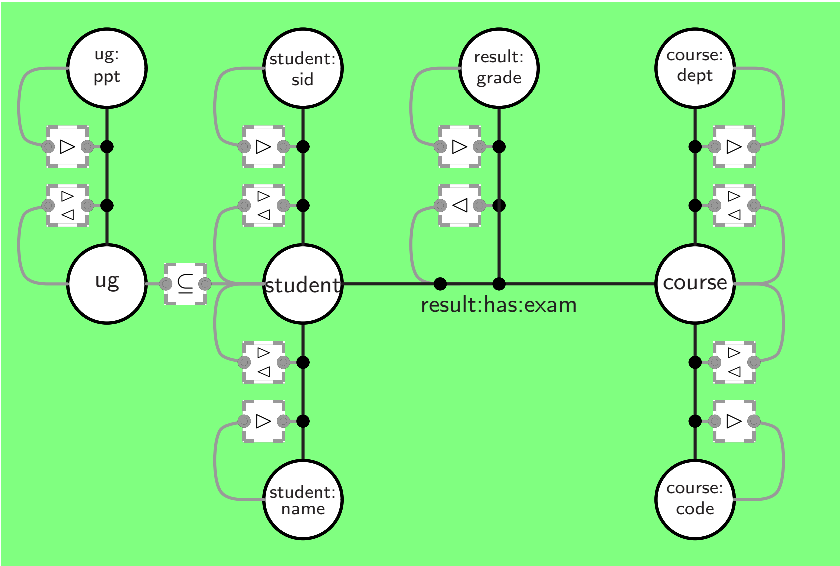
UML **classes** are nodal constructs, and hence each UML class scheme $\langle\langle C\rangle\rangle$ maps to a single node $\langle\langle C\rangle\rangle$. The extent of $\langle\langle C\rangle\rangle$ is the set of unique **object identifiers** (**OID**) of the class.

$\mathsf{class}\ \langle\langle C\rangle\rangle \rightsquigarrow \langle\langle C\rangle\rangle$

The definition of n-ary **associations** in UML states that the multiplicity, $L..U$, of a role, R, defines the number of instances of the class C that are associated with a particular set of values of the other classes in the association A. Thus using ER terminology, it has look-across semantics [45, 21], and hence generate_card() is called for each role class with all the classes except the role class. We also make the assumption that * is simply a shorthand for 0..*, and any single number n is a shorthand for n..n (for example, in UML, one writes 1 as a shorthand for 1..1).



(a) A UML class diagram of the student-course database



(b) HDM representation of the UML Schema

Fig. 5. A UML schema and its equivalent HDM schema

$$\begin{aligned}
&\textsf{association}\langle\!\langle A, R_1, C_1, L_1..U_1, \ldots, R_n, C_n, L_n..U_n\rangle\!\rangle \rightsquigarrow \langle\!\langle A{:}R_1{:}\ldots{:}R_n, C_1, \ldots, C_n\rangle\!\rangle\\
&\quad \textsf{true} \Rightarrow \textsf{generate_card}(\langle C_2, \ldots, C_n\rangle, \langle\!\langle A{:}R_1{:}\ldots{:}R_n, C_1, \ldots, C_n\rangle\!\rangle, L_1, U_1)\\
&\qquad\vdots\\
&\quad \textsf{true} \Rightarrow \textsf{generate_card}(\langle C_1, \ldots, C_{n-1}\rangle, \langle\!\langle A{:}R_1{:}\ldots{:}R_n, C_1, \ldots, C_n\rangle\!\rangle, L_n, U_n)
\end{aligned}$$

For example, the UML association between student and course has the scheme ⟨⟨result,has,student,0..*,exam,course,0..*⟩⟩, and the production rule maps this to the HDM edge ⟨⟨result:has:exam,student,course⟩⟩, with no constraints. Note that the label of the HDM edge is $A{:}R_1{:}\ldots{:}R_n$, which encodes the various labels HDM gives the association in a single HDM identifier. Thus the result association with role names 'has' and 'exam' gets the HDM edge name result:has:exam.

UML **attributes** are link-nodal constructs attached to CA, which is a UML class or a UML association, and hence the production rule takes a similar form to that for ER attributes or relational columns. We make the same assumptions about shorthands for the attribute multiplicity as we did for association multiplicity, as well as noting that the absence of explicit multiplicity means that 1..1 is assumed. Thus the sid attribute of student has the scheme $\langle\langle$student,sid,1..1$\rangle\rangle$.

$\mathsf{attribute}\ \langle\langle CA, A, L..U\rangle\rangle \leadsto \langle\langle CA{:}A\rangle\rangle, \langle\langle \_, CA, CA{:}A\rangle\rangle$
$\quad \mathsf{true}\ \Rightarrow \mathsf{generate\_card}(\langle\langle CA{:}A\rangle\rangle, \langle\langle \_, CA, CA{:}A\rangle\rangle, 1, *)$
$\quad L..U \Rightarrow \mathsf{generate\_card}(\langle\langle CA\rangle\rangle, \langle\langle \_, CA, CA{:}A\rangle\rangle, L, U)$

Note that UML **association classes** are directly supported by these definitions of association and attribute. An association class is simply an association that has one or more attributes placed upon it, each UML attribute becoming an HDM node with a nested edge that connects that node to the HDM edge that represents the association.

UML **generalisations** have a sophisticated constraint system that specifies that the various classes or associations that are children of a parent class or association maybe overlapping or disjoint, and maybe complete or incomplete. The first and last of these keywords are 'noise' in the sense that they add nothing in addition to an unlabelled generalisation. The other two add an exclusion constraint and a union constraint.

$\mathsf{generalisation}\ \langle\langle C, C_1, \ldots, C_n, D\rangle\rangle \leadsto \bot$
$\quad \mathsf{true} \qquad\qquad \Rightarrow \langle\langle C_1\rangle\rangle \subseteq \langle\langle C\rangle\rangle$
$\qquad\qquad\qquad\quad \Rightarrow \vdots$
$\quad \mathsf{true} \qquad\qquad \Rightarrow \langle\langle C_n\rangle\rangle \subseteq \langle\langle C\rangle\rangle$
$\quad \mathsf{disjoint} \in D \quad \Rightarrow \langle\langle C_1\rangle\rangle \not\bowtie \ldots \not\bowtie \langle\langle C_n\rangle\rangle$
$\quad \mathsf{complete} \in D \Rightarrow \langle\langle C\rangle\rangle = \langle\langle C_1\rangle\rangle \cup \ldots \cup \langle\langle C_n\rangle\rangle$

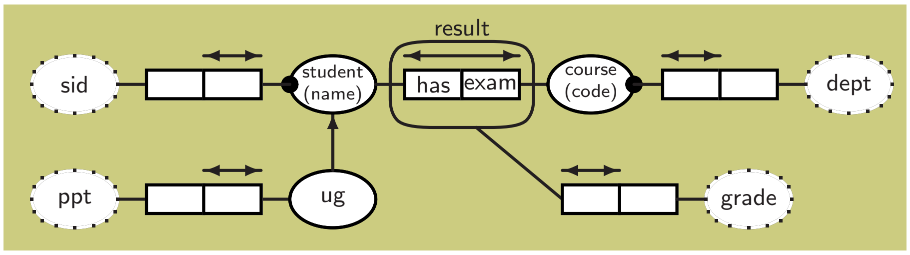
2.5 Describing the ORM in the HDM

From our analysis of the ER, relational and UML modelling languages, it may seem 'obvious' that ORM entity types should be modelled as nodal constructs while value types should be modelled as link-nodal constructs. However, due to ORM's rich semantics, the similarity of value type and entity type roles in fact types, and a value-type's ability to play multiple roles in fact types, it is correct to model both value types and entity types using an HDM nodal construct type.

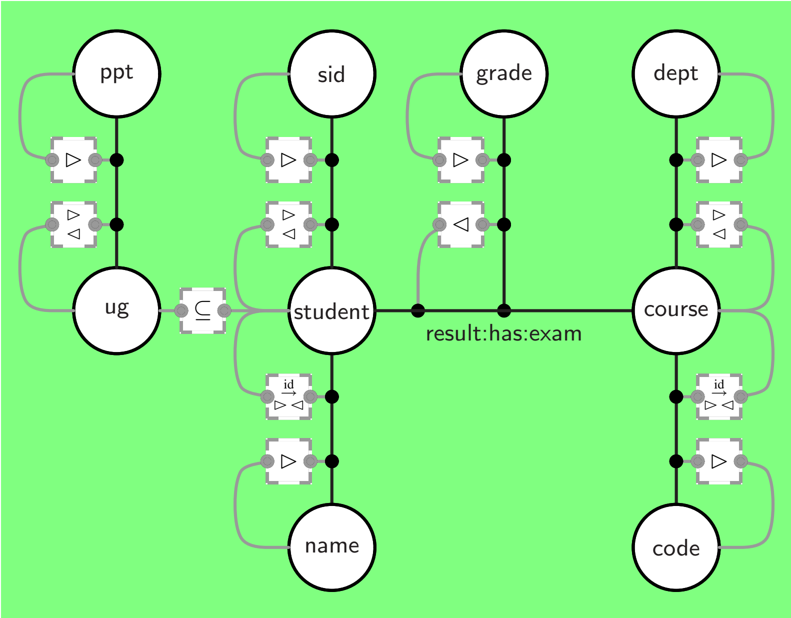
Each **entity type** $\langle\langle E\rangle\rangle$ maps to a single node $\langle\langle E\rangle\rangle$. The extent of entity type $\langle\langle E\rangle\rangle$ is the extent of its primary reference mode while the extent of a **value type** $\langle\langle V\rangle\rangle$ is just the ORM value type's set of values. Hence we have the simple definitions:

$\mathsf{entity\_type}\ \langle\langle E\rangle\rangle \leadsto \langle\langle E\rangle\rangle$
$\mathsf{value\_type}\ \langle\langle V\rangle\rangle \leadsto \langle\langle V\rangle\rangle$

An ORM n-ary **fact type** is an association between n objects where each object is an entity type, value type, or objectified fact type. A fact type's extent is drawn from the objects it associates and is hence modelled in the HDM as a link construct. The scheme for the ORM fact type should describe the name FT of the fact type (if any) along with the role name N (if any), role object R, and the mandatory nature of the object type M in the role. Hence the fact type between $\langle\langle$student$\rangle\rangle$ and $\langle\langle$course$\rangle\rangle$ has the scheme $\langle\langle$result,has,student,_,exam,course,_$\rangle\rangle$ and the fact type between $\langle\langle$student$\rangle\rangle$ and $\langle\langle$sid$\rangle\rangle$ has scheme $\langle\langle$_,student,_,$\bullet$,sid,_,_$\rangle\rangle$ (where $M = \bullet$ corresponds to the black circle used in ORM on objects to denote mandatory roles). These then map into the HDM using the following production rule:



(a) An ORM schema of the student-course database



(b) HDM representation of the ORM schema

Fig. 6. An ORM schema of the student-course database

$$\text{fact\_type}\ \langle\langle FT, N_1, R_1, M_1, \ldots, N_n, R_n, M_n \rangle\rangle \rightsquigarrow \langle\langle FT{:}N_1{:}\ldots{:}N_n, R_1, \ldots, R_n \rangle\rangle$$
$$M_1 = \bullet \Rightarrow \text{generate\_card}(R_1, \langle\langle FT{:}N_1{:}\ldots{:}N_n, R_1, \ldots, R_n \rangle\rangle, 1, *)$$
$$\vdots$$
$$M_n = \bullet \Rightarrow \text{generate\_card}(R_n, \langle\langle FT{:}N_1{:}\ldots{:}N_n, R_1, \ldots, R_n \rangle\rangle, 1, *)$$

The names of fact types and roles are encoded into a single HDM edge label in a similar manner to that used to encode UML association and role names into a single HDM edge label. Note that all fact types have an implied uniqueness constraint across all n roles, and HDM has a similar edge constraint because the extent of an edge is a set of tuples. In ORM one can specify uniqueness constraints across $n-1$ roles of an n role fact type. Hence we define the scheme of **uniqueness** to take both a fact type and the role R_x that is uniquely identified by the other roles:

$\mathsf{uniqueness}\ \langle\langle\langle\langle FT, N_1, R_1, M_1, \ldots, N_n, R_n, M_n\rangle\rangle, R_x\rangle\rangle \leadsto \bot$
$\quad\mathsf{true} \Rightarrow \mathsf{generate\_card}(\langle R_1, \ldots, R_{x-1}, R_{x+1}, \ldots, R_n\rangle,$
$\qquad\qquad\qquad\qquad \langle\langle FT{:}N_1{:}\ldots{:}N_n, R_1, \ldots, R_n\rangle\rangle, 0, 1\quad)$

The production rule implements the cardinality constraint using $\mathsf{generate\_card}()$ being called with all roles of the fact type accept the R_x being uniquely identified.

We note in passing that ORM also has a general **frequency constraint** type across any number of roles. We have not needed this in our examples, but could have modelled the mandatory and unique constraints using the more general frequency constraint; but as there are implicit unique and mandatory constraints in an ORM schema and these constraints are used heavily in the rules regarding a schema's well formedness, it is useful to model mandatory and unique as we have here.

ORM can express **subtype** relationships between fact roles as well as entity types. We only use subtyping between entity types in our examples, hence we shall restrict ourselves to just defining that below, together with the notion of disjointness and totality of such subtypes which ORM also supports:

$\mathsf{subset}\ \langle\langle EV, EV_s\rangle\rangle \leadsto \bot$
$\quad\mathsf{true} \Rightarrow \langle\langle EV_s\rangle\rangle \subseteq \langle\langle EV\rangle\rangle$
$\mathsf{disjoint}\ \langle\langle EV_1\rangle\rangle, \langle\langle EV_2\rangle\rangle \leadsto \bot$
$\quad\mathsf{true} \Rightarrow \langle\langle EV_1\rangle\rangle \not\cap \langle\langle EV_2\rangle\rangle$
$\mathsf{total}\ \langle\langle EV, EV_1\rangle\rangle, \langle\langle EV, EV_2\rangle\rangle \leadsto \bot$
$\quad\mathsf{true} \Rightarrow \langle\langle EV\rangle\rangle = \langle\langle EV_1\rangle\rangle \cup \langle\langle EV_2\rangle\rangle$

Our simple UoD does not use the above ORM constructs, but applying the above definitions to the ORM diagram in Figure 6(a) produces the HDM in Figure 6(b), almost the same HDM we arrived at using the ER schema bar some trivial renaming of nodes and edges. Note that we have assumed that the value classes implied by the primary reference modes for each entity class have been made explicit before applying the definitions (ORM allows these to be implicit, and not stated in the ORM schema).

3 Inter Model Transformations

We now introduce five general purpose equivalence mappings that may be used on our HDM constraint operators, and which allow us to transform between different modelling languages. In particular, the relational HDM schema in Figure 4(b) may be transformed into the ER HDM schema in Figure 3(b) by applying a sequence of transformations using four of the equivalence relationships.

Section 3.6 describes the fifth general purpose equivalence preserving rule, and shows how it is used as part of the transformation of the UML HDM schema in Figure 5(b) to the ER HDM schema. However, the UML to ER transformation as a whole will be demonstrated to be non-equivalence preserving.

In order to give our mappings a rigorous basis, we define them using the BAV transformation language [39, 28, 30, 31] which allows the specification of bidirectional mappings between equivalent data sources, and also the specification of the situation where one data source has greater information capacity than another. Hence, we first review BAV primitive transformations for the HDM in the next subsection, before giving the five mappings in Sections 3.2–3.6 defined using those transformation primitives. Finally we discuss in Section 3.7 how non-equivalent data sources are identified and handled in our approach.

3.1 HDM BAV Transformations

primitive transformation $S \rightarrow S'$	reverse transformation $S' \rightarrow S$	conditions on S, S'	information capacity
addNode(n,q)	deleteNode(n,q)	$n \notin Nodes, n \in Nodes'$	$S \equiv S'$
addEdge(e,q)	deleteEdge(e,q)	$e \notin Edges, e \in Edges'$	$S \equiv S'$
addCons(c)	deleteCons(c)	$c \notin Cons, c \in Cons'$	$S \equiv S'$
renameNode(n,n')	renameNode(n',n)	$n \in Nodes, n \notin Nodes'$, $n' \notin Nodes, n' \in Nodes'$	$S \equiv S'$
renameEdge(e,e')	renameEdge(e',e)	$e \in Edges, e \notin Edges'$, $e' \notin Edges, e' \in Edges'$	$S \equiv S'$
extendNode(n,q_l,q_u)	contractNode(n,q_l,q_u)	$n \notin Nodes, n \in Nodes'$	$S \subset S'$
extendEdge(e,q_l,q_u)	contractEdge(e,q_l,q_u)	$e \notin Edges, e \in Edges'$	$S \subset S'$
extendCons(c)	contractCons(c)	$c \notin Cons, c \in Cons'$	$S \supset S'$

Table 2. BAV primitive transformations applied to HDM schema $S = \langle Nodes, Edges, Cons \rangle$, to generate a new schema $S' = \langle Nodes', Edges', Cons' \rangle$

In the BAV approach, schemas are incrementally transformed by applying to them a **pathway** of primitive schema transformations $t_1, \ldots, t_r$. Each primitive transformation t_i makes a 'delta' change to the schema by adding, deleting or renaming just one HDM node, edge or constraint. The model management [5] concept of a Mapping between S_1, S_2 is implemented by having a well-formed pathway [47] between the two schemas. Note that the model management concept of Compose is directly supported by the BAV approach, since it allows a pathway between S_2, S_3 to be appended to the pathway between S_1, S_2 to give a pathway between S_1, S_3. Details of how pathways can be analyzed for their impact on information capacity can be found in [27], and work in using BAV to perform the model management Match is found in [40] and Merge is found in [41].

Table 2 lists those primitive transformations of the BAV language that we use in this paper, and we now briefly review their semantics.

The primitive transformation that adds a node n to a schema S in order to generate new schema S' is addNode(n, q), where q is a query over S specifying the extent of n in terms of the existing constructs of S. The logical semantics of this kind of transformation are

$$\forall I.Ext_{S,I}(n) = q \tag{1}$$

and for this reason we categorise addNode as an **exact transformation** [25], and the two schemas S, S' have equivalent information capacity, summarised in Table 2 by putting $S \equiv S'$ in the information capacity column.

When it is not possible to specify the exact extent of the new node n in terms of the existing schema constructs, we must instead of addNode use extendNode(n, q_l, q_u), where q_l gives the lower bound on the extent of n, and q_u gives the upper bound. The logical semantics of this kind of transformation are

$$\forall I.q_u \supseteq Ext_{S,I}(n) \supseteq q_l \tag{2}$$

and so we term extend a **sound transformation** [25] when considering q_l and a **complete transformation** [25] when considering q_u. The query q_l may just be the constant Void, indicating no values in the extent can be derived from other constructs in the schema. The query q_u may just be the constant Any, indicating that no limit of the values in the extent can be derived from other constructs in the schema. If $q_l = \mathsf{Void}$ and $q_u = \mathsf{Any}$ then the two queries may be omitted (and only this form of the transformation is used in the paper). Note that S has an information capacity which is a subset of that of S', which in Table 2 by putting $S \subset S'$ in the information capacity column.

The exact transformation deleteNode(c, q) when applied to schema S' generates a new schema S with node n removed. The extent of n may be recovered using the query q on S, and Equation 1 above holds. Note that this implies that from a primitive transformation deleteNode(n,q) used to transform $S' \rightarrow S$ we can automatically derive that addNode(n,q) transforms $S \rightarrow S'$, and *vice versa*.

When it is not possible to specify the exact extent of node n being deleted from S' in terms of the remaining schema constructs, contractNode(n,q_l,q_u) must be used instead of deleteNode, where Equation 2 above holds. Again, it is possible that sound query q_l may just be Void, and the complete query q_u be Any, indicating that the extent of n cannot be specified even partially, in which case the queries can be omitted from the transformation. Note that from a primitive transformation contractNode(n,q_l,q_u) used to transform $S' \rightarrow S$ we can automatically derive that extendNode(n,q_l,q_u) transforms $S \rightarrow S'$, and *vice versa*.

The last type of transformation dealing with nodes is renameNode(n, n') causes a node n in a schema S to be renamed to n' in a new schema S', where in logical terms

$$\forall I.Ext_{S,I}(n) = Ext_{S,I}(n') \tag{3}$$

Note that this definition implies that from renameNode(n, n') used to transform $S \rightarrow S'$ we can automatically derive that renameNode(n', n) transforms $S' \rightarrow S$, and *vice versa*.

Entirely analogous arguments will give the definitions of the primitive transformations in Table 2 that handle manipulation of edges. For constraints, there are three differences. Firstly, since constraints have no extent, there is no query in the primitive transformations handling constraints. Secondly, constraints have no name, and hence there is no renameCons transformation. Thirdly, the extendCons transformation causes S' to be more restrictive than S, and hence the information capacity of S' is less than that of S.

Note that the ORM HDM schema in Figure 6(b) is the same as the ER HDM schema in Figure 3(b), except for trivial renaming of constructs. Hence we can apply equivalence preserving rename transformations to make the ORM HDM schema match those in the ER HDM schema:

renameEdge(⟪result:has:exam,student,course⟫, ⟪result,student,course⟫)

Hence the pathway of transformations describing the transformation from ER to relational schemas in the following sub-sections also defines the mapping from relational to ORM schemas, with this extra transformation step appended to the pathway.

We now move on to describe in the subsequent subsections the five mappings that we propose as a solution to transforming between the four models detailed in the previous section.

3.2 Inclusion Merge

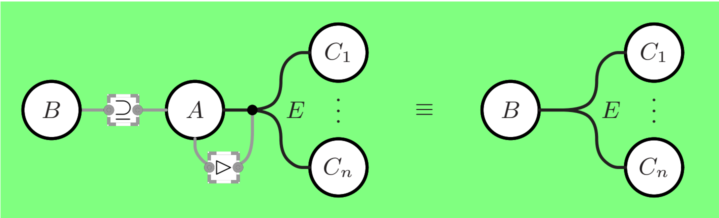


Fig. 7. Equivalence Relationships: Inclusion Merge

The **Inclusion Merge** equivalence in Figure 7 allows us to merge two nodes $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$ together where $\langle\langle A\rangle\rangle$ is a subset of $\langle\langle B\rangle\rangle$ and there is a mandatory constraint from $\langle\langle A\rangle\rangle$ to an edge $e = \langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$. The mandatory constraint is dropped as we merge $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$. Any edges or constraints that applied to $\langle\langle B\rangle\rangle$ remain, and any other (unillustrated) edges on $\langle\langle A\rangle\rangle$ are also redirected to $\langle\langle B\rangle\rangle$. Definition 9 gives a pseudo code definition of this equivalence, that generates primitive transformations on the HDM. The pseudo code first deletes the constraint between $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$, and then checks that the subset node $\langle\langle A\rangle\rangle$ is not associated in any other subset, union or exclusion with any other nodes, and raises an exception if that is the case. The pseudo code then iterates over all edges that connect to node $\langle\langle A\rangle\rangle$ and removes any mandatory constraints involving $\langle\langle A\rangle\rangle$. Then move_dependents function (defined in Definition 10) is used to move all edges on $\langle\langle A\rangle\rangle$ to connect to $\langle\langle B\rangle\rangle$. Note that this will include e and cause a new edge $e' = \langle\langle E, B, C_1, \ldots, C_n\rangle\rangle$ to now exist. The final line of Definition 9 then deletes $\langle\langle A\rangle\rangle$, giving a query that can restore the values of $\langle\langle A\rangle\rangle$ from the new non-mandatory edge e'.

Definition 9 Inclusion Merge

inclusion_merge($\langle\langle B\rangle\rangle$,$\langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$)
    deleteCons($\langle\langle A\rangle\rangle \subseteq \langle\langle B\rangle\rangle$);
    if $op(\langle\langle A\rangle\rangle, d) \in Cons \wedge op \in \{\subseteq, \not\cap, \cup\}$ then
        exception
    endif;
    foreach $e \in Edges$ forwhich $e = \langle\langle E_a, A, \ldots\rangle\rangle$
        deleteCons($\langle\langle A\rangle\rangle \rhd e$)
    endforeach;

move_dependents($\langle\langle A\rangle\rangle$,$\langle\langle B\rangle\rangle$,id $\langle\langle A\rangle\rangle$);
deleteNode($\langle\langle A\rangle\rangle$,$\{\langle x\rangle \mid \langle x,y_1,\ldots,y_n\rangle \in \langle\langle E,B,C_1,\ldots,C_n\rangle\rangle\}$); □

The last line of inclusion_merge projects out the single arity tuples $\langle x\rangle$ that form the extension of $\langle\langle A\rangle\rangle$ from edge $\langle\langle E,B,C_1,\ldots,C_n\rangle\rangle$.

The definition of move_dependents takes three arguments, the first two (a,b) of which must be a node or edge, and the third a mapping set that maps instances of a to instances of b. For use in inclusion merge, the third argument should be an identity function id, defined as $id(\langle\langle A\rangle\rangle) = \{\langle a,a\rangle \mid a \in \langle\langle A\rangle\rangle\}$.

Definition 10 Move Dependents

move_dependents(a,b,map)
    foreach $op(a,d) \in Cons$ forwhich $b \neq d$
        addCons($op(b,d)$);
        deleteCons($op(a,d)$)
    endforeach;
    foreach $op(d,a) \in Cons$ forwhich $b \neq d$
        addCons($op(d,b)$);
        deleteCons($op(d,a)$)
    endforeach;
    foreach $e \in Edges$ forwhich $e = \langle\langle E,a,C_1,\ldots,C_n\rangle\rangle$
        let e'=$\langle\langle E,b,C_1,\ldots,C_n\rangle\rangle$;
        addEdge(e',$\{\langle y,z_1,\ldots,z_n\rangle \mid \langle x,z_1,\ldots,z_n\rangle \in e \wedge \langle x,y\rangle \in map\}$);
        move_dependents(e,e',
            $\{\langle\langle x,z_1,\ldots,z_n\rangle,\langle y,z_1,\ldots,z_n\rangle\rangle \mid \langle x,y\rangle \in map \wedge \langle x,z_1,\ldots,z_n\rangle \in e\}$);
        deleteEdge(e,$\{\langle x,z_1,\ldots,z_n\rangle \mid \langle y,z_1,\ldots,z_n\rangle \in e' \wedge \langle x,y\rangle \in map\}$)
    endforeach; □

In Example 3, the series of transformations that will convert the HDM schema in Figure 4(b) into that in Figure 3(b) are listed. The first two steps in the series are applications of inclusion merge, which after 2 result in the intermediate HDM schema shown in Figure 8.

Example 3 Transforming between relational and ER HDM schemas

1 inclusion_merge($\langle\langle$student:name$\rangle\rangle$, $\langle\langle$_,result:name,result$\rangle\rangle$)
2 inclusion_merge($\langle\langle$course:code$\rangle\rangle$, $\langle\langle$_,result:code,result$\rangle\rangle$)
3 identity_node_merge($\langle\langle$_,ug:name,ug$\rangle\rangle$)
4 unique_mandatory_redirection($\langle\langle$_,student:name,result$\rangle\rangle$,
    $\langle\langle$_,student:name,student$\rangle\rangle$)
5 unique_mandatory_redirection($\langle\langle$_,course:code,result$\rangle\rangle$, $\langle\langle$_,course:code,course$\rangle\rangle$)
6 identity_edge_merge($\langle\langle$_,result,student$\rangle\rangle$, $\langle\langle$_,result,course$\rangle\rangle$)
7 move_dependents($\langle\langle$student:name$\rangle\rangle$, $\langle\langle$student$\rangle\rangle$, $\langle\langle$_,student:name,student$\rangle\rangle$) □

Taking transformation step 1 and applying Definition 9, we may expand the steps into a series of primitive transformation steps shown in Example 4. Step 1.1 is a result of the first foreach loop in Definition 9, Steps 1.2 and 1.3 result from the call to move_dependents, and 1.4 and 1.5 result from the last two lines of Definition 9.

Example 4 Primitive steps associated with transformation ①

(1.1) $\mathsf{deleteCons}(\langle\langle \mathsf{result{:}name}\rangle\rangle \rhd \langle\langle \_,\mathsf{result{:}name},\mathsf{result}\rangle\rangle)$

(1.2) $\mathsf{addEdge}(\langle\langle \_,\mathsf{student{:}name},\mathsf{result}\rangle\rangle,$
$\{\langle \mathsf{b},\mathsf{c}\rangle \mid \langle \mathsf{a},\mathsf{c}\rangle \in \langle\langle \_,\mathsf{result{:}name},\mathsf{result}\rangle\rangle \wedge \langle \mathsf{a},\mathsf{b}\rangle \in \mathsf{id}\langle\langle \mathsf{result{:}name}\rangle\rangle\}$

(1.3) $\mathsf{deleteEdge}(\langle\langle \_,\mathsf{result{:}name},\mathsf{result}\rangle\rangle,$
$\{\langle \mathsf{a},\mathsf{c}\rangle \mid \langle \mathsf{b},\mathsf{c}\rangle \in \langle\langle \_,\mathsf{student{:}name},\mathsf{result}\rangle\rangle \wedge \langle \mathsf{a},\mathsf{b}\rangle \in \mathsf{id}\langle\langle \mathsf{result{:}name}\rangle\rangle\}$

(1.4) $\mathsf{deleteCons}(\langle\langle \mathsf{result{:}name}\rangle\rangle \subseteq \langle\langle \mathsf{student{:}name}\rangle\rangle)$

(1.5) $\mathsf{deleteNode}(\langle\langle \mathsf{result{:}name}\rangle\rangle,$
$\{\langle \mathsf{b}\rangle \mid \langle \mathsf{b},\mathsf{c}\rangle \in \langle\langle \_,\mathsf{student{:}name},\mathsf{result}\rangle\rangle\}$ □

The expansion of transformations ① illustrates that inclusion merge is a data preserving transformation, since it only uses add and delete BAV transformations. In particular the edge $\langle\langle \_,\mathsf{result},\mathsf{student{:}name}\rangle\rangle$ may be recovered by the query in (1.3) (which in turn uses the query of (1.5) to find the extent of $\langle\langle \mathsf{result{:}name}\rangle\rangle$), and the new edge $\langle\langle \_,\mathsf{result},\mathsf{student{:}name}\rangle\rangle$ may be derived from existing data in (1.2). Note that a very similar expansion into primitive steps may be performed for ②, with a similar argument about data preservation.

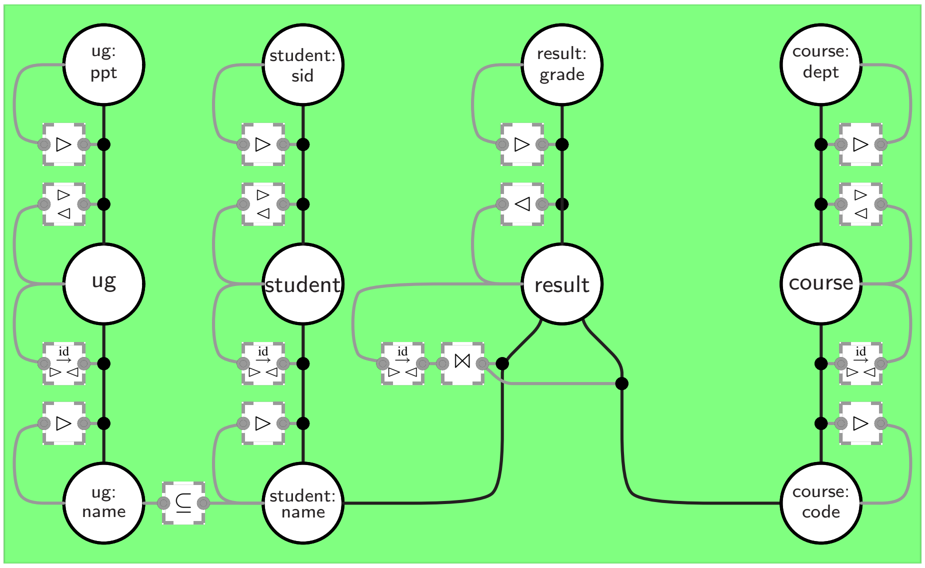


Fig. 8. Intermediate HDM schema in relational to ER conversion, after steps ① and ②

3.3 Identity Node Merge

The **Identity Node Merge** in Figure 9 allows us to merge the two nodes $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$ together because they are identical. The constraints $\langle\langle A\rangle\rangle \xrightarrow{\text{id}} \langle\langle E,A,B\rangle\rangle$, $\langle\langle A\rangle\rangle \rhd$

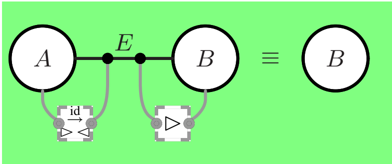


Fig. 9. Equivalence Relationships: Identity Node Merge

$\langle\langle E, A, B\rangle\rangle$, and $\langle\langle A\rangle\rangle \lhd \langle\langle E, A, B\rangle\rangle$ taken together mean that every instance of the edge $\langle\langle E, A, B\rangle\rangle$ is an identity mapping for $\langle\langle A\rangle\rangle$, and there is exactly one such mapping in $\langle\langle E, A, B\rangle\rangle$ for every element of $\langle\langle A\rangle\rangle$. As each element in $\langle\langle E, A, B\rangle\rangle$ is an identity mapping, each element in $\langle\langle A\rangle\rangle$ must be in $\langle\langle B\rangle\rangle$. Conversely because we also have $\langle\langle B\rangle\rangle \rhd \langle\langle E, A, B\rangle\rangle$, each element in $\langle\langle B\rangle\rangle$ must be in $\langle\langle A\rangle\rangle$, and so $\langle\langle A\rangle\rangle = \langle\langle B\rangle\rangle$. This implies both $\langle\langle B\rangle\rangle \xrightarrow{\text{id}} \langle\langle E, A, B\rangle\rangle$ and $\langle\langle B\rangle\rangle \lhd \langle\langle E, A, B\rangle\rangle$, and thus the equivalences illustrated in Figure 2(a) hold. Because we have identified them as equal, nodes $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$ can be merged together and the edge $\langle\langle E, A, B\rangle\rangle$ dropped, using Definition 11. Note any node can have this transformation applied in reverse, copying instances into a new node and linking the old node to the new node via an edge containing the identity instances.

Definition 11 Identity Node Merge

```
identity_node_merge(⟨⟨E, A, B⟩⟩)
    let e=⟨⟨E, A, B⟩⟩;
    move_dependents(⟨⟨A⟩⟩,⟨⟨B⟩⟩,e);
    foreach c ∈ Cons forwhich contains(e,c)
        deleteCons(c);
    endforeach;
    deleteEdge(e,{⟨x, x⟩ | ⟨x⟩ ∈ ⟨⟨B⟩⟩});
    deleteNode(⟨⟨A⟩⟩,⟨⟨B⟩⟩);
```
□

This identity mapping comes about by the way some modelling languages specify a certain attribute as being an entity's identifying attribute (such as the primary key constraint in the relational schema).

In Figure 8 we can use identity node merge to merge nodes $\langle\langle\mathsf{ug{:}name}\rangle\rangle$ and $\langle\langle\mathsf{ug}\rangle\rangle$ by step 3 in Example 3. Note that the constraint $\langle\langle\mathsf{ug{:}name}\rangle\rangle \subseteq \langle\langle\mathsf{student{:}name}\rangle\rangle$ is not lost, but becomes $\langle\langle\mathsf{ug}\rangle\rangle \subseteq \langle\langle\mathsf{student{:}name}\rangle\rangle$. Figure 11 is partially derived by applying this merge.

3.4 Unique-Mandatory Redirection

The **Unique-Mandatory Redirection** equivalence in Figure 10 allows us to move an edge $\langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$ from node $\langle\langle A\rangle\rangle$ to node $\langle\langle B\rangle\rangle$ because both $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$ have a unique and mandatory constraint on the common edge $\langle\langle E_{AB}, A, B\rangle\rangle$. These constraints together are equivalent to stating that there is exactly a one to one correspondence between the elements of $\langle\langle A\rangle\rangle$ and $\langle\langle B\rangle\rangle$ so whatever is related to an element of $\langle\langle A\rangle\rangle$

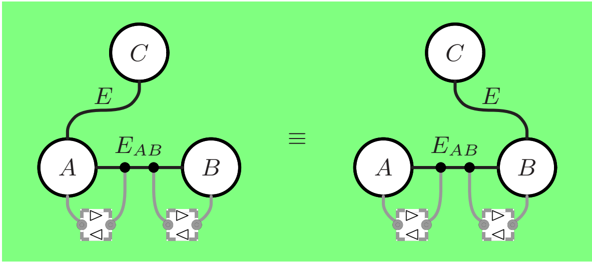


Fig. 10. Equivalence Relationships: Unique-Mandatory Redirection

through $\langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$ is equally related to the corresponding element in $\langle\langle B\rangle\rangle$. Moving the edge requires us to rewrite the elements of the edge, replacing in each the value that came from $\langle\langle A\rangle\rangle$ with the corresponding value from $\langle\langle B\rangle\rangle$ (via $\langle\langle E_{AB}, A, B\rangle\rangle$).

Definition 12 Unique-Mandatory Redirection

unique_mandatory_redirection($\langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$,$\langle\langle E_{AB}, A, B\rangle\rangle$)
  let $e = \langle\langle E, A, C_1, \ldots, C_n\rangle\rangle$;
  let $map = \langle\langle E_{AB}, A, B\rangle\rangle$;
  if $(A \xrightarrow{\text{id}} e) \in Cons$ then exception endif;
  let $e' = \langle\langle E, B, C_1, \ldots, C_n\rangle\rangle$;
  addEdge(e',$\{\langle y, z_1, \ldots, z_n\rangle \mid \langle x, z_1, \ldots, z_n\rangle \in e \wedge \langle x, y\rangle \in map\}$);
  move_dependents(e,e',map)
  deleteEdge(e,$\{\langle x, z_1, \ldots, z_n\rangle \mid \langle y, z_1, \ldots, z_n\rangle \in e' \wedge \langle x, y\rangle \in map\}$); □

For the HDM schema in Figure 8, we can apply 4 in Example 3 to move the edge ⟨⟨_,result,student:name⟩⟩ from node ⟨⟨student:name⟩⟩ to node ⟨⟨student⟩⟩, becoming edge ⟨⟨_,result,student⟩⟩. This transformation does not lose information, because of the constraints on the edge ⟨⟨_,student:name,student⟩⟩, in particular ⟨⟨student:name⟩⟩ $\lhd$ ⟨⟨_,student,student:name⟩⟩ is implied by the other constraints present on the edge, as illustrated in Figure 2(a). Similarly we can apply 5 to move edge ⟨⟨_,result,course:text⟩⟩ to become ⟨⟨_,result,course⟩⟩. Applying these two edge redirections in addition to the previous identity node merge results in Figure 11.

3.5 Identity Edge Merge

The **Identity Edge Merge** in Figure 12 allows us to replace the node $\langle\langle A\rangle\rangle$ and edges $\langle\langle E_1, A, B_1\rangle\rangle \ldots \langle\langle E_m, A, B_m\rangle\rangle$ with the single edge $\langle\langle A, B_1 \ldots B_m\rangle\rangle$. The constraints $\xrightarrow{\text{id}}$, $\rhd$, and $\lhd$ between $\langle\langle A\rangle\rangle$ and the natural join of $\langle\langle E_1, A, B_1\rangle\rangle \ldots \langle\langle E_m, A, B_m\rangle\rangle$ mean that for each instance of node $\langle\langle A\rangle\rangle$ there is exactly one instance of the join of edges $\langle\langle E_1, A, B_1\rangle\rangle \ldots \langle\langle E_m, A, B_m\rangle\rangle$. The extent of the hyper edge $\langle\langle A, B_1 \ldots B_m\rangle\rangle$ is obtained from the corresponding values in $\langle\langle B_1\rangle\rangle \ldots \langle\langle B_m\rangle\rangle$ for each instance of the node $\langle\langle A\rangle\rangle$. Because of the identity mapping, these are the same values was in $\langle\langle A\rangle\rangle$, and hence there is no information in the node $\langle\langle A\rangle\rangle$ that is not in this new edge.

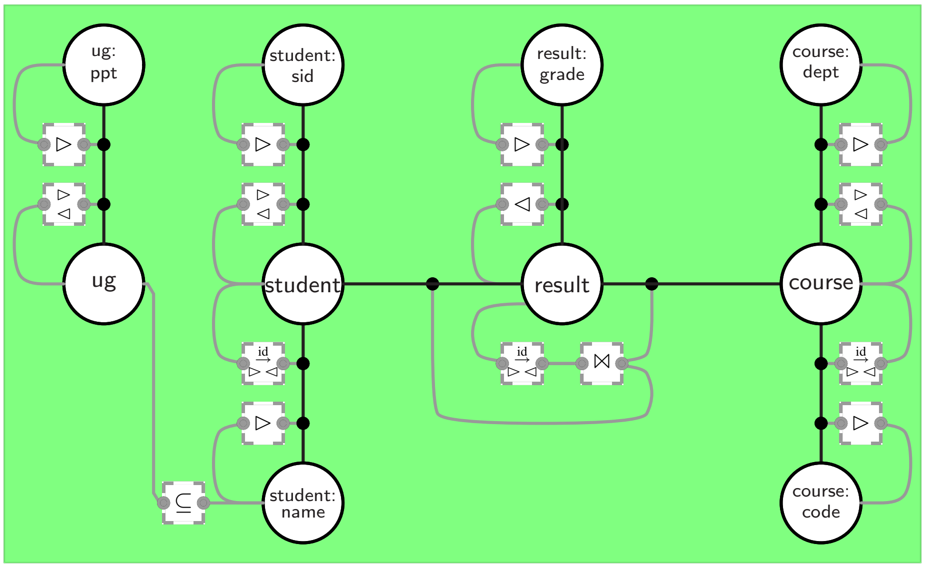


Fig. 11. Intermediate HDM schema in relational to ER conversion, after steps ①–⑤

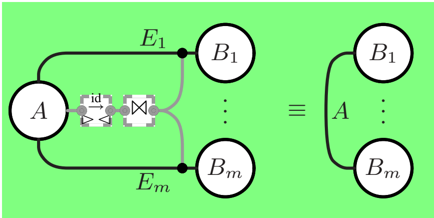


Fig. 12. Equivalence Relationships: Identity Edge Merge

Definition 13 Identity Edge Merge

identity_edge_merge($\langle\langle E_1, A, B_1\rangle\rangle, \ldots, \langle\langle E_m, A, B_m\rangle\rangle$)
  let $a = \langle\langle A, B_1, \ldots, B_m\rangle\rangle$;
  addEdge($a, \{\langle b_1, \ldots, b_m\rangle \mid$
      $\langle a, b_1\rangle \in \langle\langle E_1, A, B_1\rangle\rangle \wedge \ldots \wedge \langle a, b_m\rangle \in \langle\langle E_m, A, B_m\rangle\rangle\}$);
  foreach $(A\ op\ e) \in Cons$ forwhich $e \in \{\langle\langle E_1, A, B_1\rangle\rangle, \ldots, \langle\langle E_m, A, B_m\rangle\rangle\}$
    deleteCons($A\ op\ e$)
  endforeach;
  move_dependents($\langle\langle A\rangle\rangle$,$a$,id $\langle\langle$A$\rangle\rangle$);
  deleteNode($\langle\langle A\rangle\rangle$,$a$) □

In Figure 11 we can use identity node merge to replace the node ⟨⟨result⟩⟩ with the edge ⟨⟨result,student,course⟩⟩, in step ⑥ of Example 3. In this case the new edge is binary because the natural join was between two edges. Note that as part of this process, the edge ⟨⟨_,result,result:grade⟩⟩ from ⟨⟨result⟩⟩ to ⟨⟨result:grade⟩⟩ becomes ⟨⟨_,⟨⟨result,student,course⟩⟩,result:grade⟩⟩.

All that is left for us to do in order to obtain the HDM ER schema is to move the constraint ⟨⟨ug⟩⟩ $\subseteq$ ⟨⟨student:name⟩⟩ to ⟨⟨ug⟩⟩ $\subseteq$ ⟨⟨student⟩⟩. This is correct to do for similar reasons to the unique-mandatory redirection being correct for edges, but here we are moving a constraint between two nodes for which, in addition, we know the extent to be identical. This redirection is achieved by using the move_dependents subroutine in (7), and the result is Figure 3(b).

3.6 Node Reidentify

Object orientation introduces the concept of there being a unique **object identifier** (**OID**) that is associated to instances of a class, and that OID is not represented as an attribute. Thus when we look at the HDM representation of the UML shown in Figure 5, although similar to those for the relational, ER and ORM schemas, there is no use of the $\xrightarrow{\text{id}}$ constraint made between nodes representing the UML class, such as ⟨⟨student⟩⟩, and edges to nodes representing UML attributes, such as ⟨⟨_,student,student:name⟩⟩. This is because ⟨⟨student⟩⟩ has as its extent the object identifiers of the student UML class, whilst ⟨⟨student:name⟩⟩ has as its extent the names of students.

Definition 14 Node Reidentify

node_reidentify($\langle\langle A\rangle\rangle$,$map$)
    addNode($\langle\langle A'\rangle\rangle$,$\{\langle b\rangle \mid \langle a\rangle \in \langle\langle A\rangle\rangle \wedge \langle a,b\rangle \in map\}$);
    foreach $(\langle\langle A_s\rangle\rangle \subseteq \langle\langle A\rangle\rangle) \in Cons$
        node_reidentify($\langle\langle A_s\rangle\rangle$,$map$)
    endforeach
    move_dependents($\langle\langle A\rangle\rangle$,$\langle\langle A'\rangle\rangle$,$map$);
    deleteNode($\langle\langle A\rangle\rangle$,$\{\langle a\rangle \mid \langle b\rangle \in \langle\langle A'\rangle\rangle \wedge \langle a,b\rangle \in map\}$)
    renameNode($\langle\langle A'\rangle\rangle$,$\langle\langle A\rangle\rangle$) □

Example 5 Transforming between UML and ER HDM schemas

(8) extendCons(⟨⟨student:name⟩⟩ $\lhd$ ⟨⟨_,student,student:name⟩⟩)
(9) inverse_identity_node_merge(⟨⟨student⟩⟩, ⟨⟨student:oid⟩⟩)
(10) deleteCons(⟨⟨student⟩⟩ $\xrightarrow{\text{id}}$ ⟨⟨_,student,student:oid⟩⟩)
(11) node_reidentify(⟨⟨student⟩⟩, {⟨x, y⟩ |
    ⟨o, x⟩ $\in$ ⟨⟨_,student,student:oid⟩⟩ $\wedge$ ⟨o, y⟩ $\in$ ⟨⟨_,student,student:name⟩⟩})
(12) addCons(⟨⟨student⟩⟩ $\xrightarrow{\text{id}}$ ⟨⟨_,student,student:name⟩⟩)
(13) extendCons(⟨⟨course:code⟩⟩ $\lhd$ ⟨⟨_,course,course:code⟩⟩)
(14) inverse_identity_node_merge(⟨⟨course⟩⟩, ⟨⟨course:oid⟩⟩)
(15) deleteCons(⟨⟨course⟩⟩ $\xrightarrow{\text{id}}$ ⟨⟨_,course,course:oid⟩⟩)
(16) node_reidentify(⟨⟨course⟩⟩, {⟨x, y⟩ |
    ⟨o, x⟩ $\in$ ⟨⟨_,course,course:oid⟩⟩ $\wedge$ ⟨o, y⟩ $\in$ ⟨⟨_,course,course:code⟩⟩})

⑰ addCons($\langle\langle$course$\rangle\rangle \xrightarrow{\text{id}} \langle\langle$_,course,course:code$\rangle\rangle$)
⑱ renameEdge($\langle\langle$:has:exam,student,course$\rangle\rangle$, $\langle\langle$result,student,course$\rangle\rangle$)
⑲ deleteCons($\langle\langle$course$\rangle\rangle \lhd \langle\langle$_,course,course:oid$\rangle\rangle$)
⑳ deleteCons($\langle\langle$course$\rangle\rangle \rhd \langle\langle$_,course,course:oid$\rangle\rangle$)
㉑ deleteCons($\langle\langle$course:oid$\rangle\rangle \lhd \langle\langle$_,course,course:oid$\rangle\rangle$)
㉒ deleteCons($\langle\langle$course:oid$\rangle\rangle \rhd \langle\langle$_,course,course:oid$\rangle\rangle$)
㉓ contractEdge($\langle\langle$_,course,course:oid$\rangle\rangle$)
㉔ contractNode($\langle\langle$course:oid$\rangle\rangle$)
㉕ deleteCons($\langle\langle$student$\rangle\rangle \lhd \langle\langle$_,student,student:oid$\rangle\rangle$)
㉖ deleteCons($\langle\langle$student$\rangle\rangle \rhd \langle\langle$_,student,student:oid$\rangle\rangle$)
㉗ deleteCons($\langle\langle$student:oid$\rangle\rangle \lhd \langle\langle$_,student,student:oid$\rangle\rangle$)
㉘ deleteCons($\langle\langle$student:oid$\rangle\rangle \rhd \langle\langle$_,student,student:oid$\rangle\rangle$)
㉙ contractEdge($\langle\langle$_,student,student:oid$\rangle\rangle$)
㉚ contractNode($\langle\langle$student:oid$\rangle\rangle$)

□

When transforming between between an OO model such as UML, and key based models such as ORM, ER or relational, we must overcome the fundamental difference in data modelling based on OIDs and natural keys. This will require us finding attributes or associations of the UML class that can be used to identify instances of the UML class.

Comparing the UML schema in Figure 5 with the ER schema in Figure 3, the HDM schemas of the two appear similar. One difference is trivial: the edge between $\langle\langle$student$\rangle\rangle$ and $\langle\langle$course$\rangle\rangle$ has a different name in the two schemas. The other difference is between the use of OIDs and natural keys. The ER HDM schema, using natural keys, has $\langle\langle$student$\rangle\rangle \xrightarrow{\text{id}} \langle\langle$_,student,student:name$\rangle\rangle$ and $\langle\langle$course$\rangle\rangle \xrightarrow{\text{id}} \langle\langle$_,course,course:code$\rangle\rangle$, whereas the UML HDM, using OIDs, does not have these constraints. Example 5 lists a sequence of transformations that converts the UML schema into an 'ER compatible' HDM schema that has explicit attributes for the OIDs, and uses a natural key to identify the ER entity instances. The following steps explain the example:

1. Missing from the UML schema is any definition of natural keys for the UML classes. Hence step ⑧ introduces a new constraint that indicates that name is a candidate key for student.
2. The inverse of identity node merge in step ⑨ generates a new node $\langle\langle$student:oid$\rangle\rangle$, connected to $\langle\langle$student$\rangle\rangle$ by a new edge $\langle\langle$_,student,student:oid$\rangle\rangle$. If for example the node $\langle\langle$student$\rangle\rangle$ had the extent $\{\langle\&1\rangle, \langle\&2\rangle, \langle\&3\rangle, \langle\&4\rangle\}$ before this step, then the new edge will have as its extent
$\{\langle\&1,\&1\rangle, \langle\&2,\&2\rangle, \langle\&3,\&3\rangle, \langle\&4,\&4\rangle\}$.
3. Transformations ⑩–⑫ have the net effect of repopulating the $\langle\langle$student$\rangle\rangle$ node with values of the $\langle\langle$student:name$\rangle\rangle$ attribute, and changing its key from oid to name. For example, if in the schema that results from ⑨ $\langle\langle$_,student,student:name$\rangle\rangle$ had extent
$\{\langle\&1, \text{'Mary'}\rangle, \langle\&2, \text{'John'}\rangle, \langle\&3, \text{'Jane'}\rangle, \langle\&4, \text{'Fred'}\rangle\}$
then the *map* generated would be the same list, and in the schema after ⑪, the node $\langle\langle$student$\rangle\rangle$ would have the extent $\{\langle\text{'Mary'}\rangle, \langle\text{'John'}\rangle, \langle\text{'Jane'}\rangle, \langle\text{'Fred'}\rangle\}$, and the

edge ⟨⟨_,student,student:oid⟩⟩ would have the extent

{⟨'Mary', &1⟩, ⟨'John', &2⟩, ⟨'Jane', &3⟩, ⟨'Fred', &4⟩}

The result of after 12 is shown in Figure 13.

4. Transformations 13–17 perform a similar conversion of the ⟨⟨course⟩⟩ node into a natural key based construct, using code as the key.
5. Transformation 18 deals with the trivial problem of renaming the edge between ⟨⟨student⟩⟩ and ⟨⟨course⟩⟩ to match the name in the ER schema.
6. Transformations 19–30 delete the ⟨⟨student:oid⟩⟩ and ⟨⟨course:oid⟩⟩ nodes (with their associated constraints and edges). Note that the use of contract transformations represents the fact that you are unable to derive the oid values from the ER schema.

If the transformations in Example 5 are compared with the BAV transformation summary Table 2, we see that the UML schema has higher information capacity than the ER schema, due to its use of OIDs and lack of key constraints. Note that alternatively, step (6) could be omitted, and the ER schema could be enhanced with oid attributes, if it was intended to use the ER schema to more fully represent the UML schema. However, the UML would still lack the key information present in the ER schema.

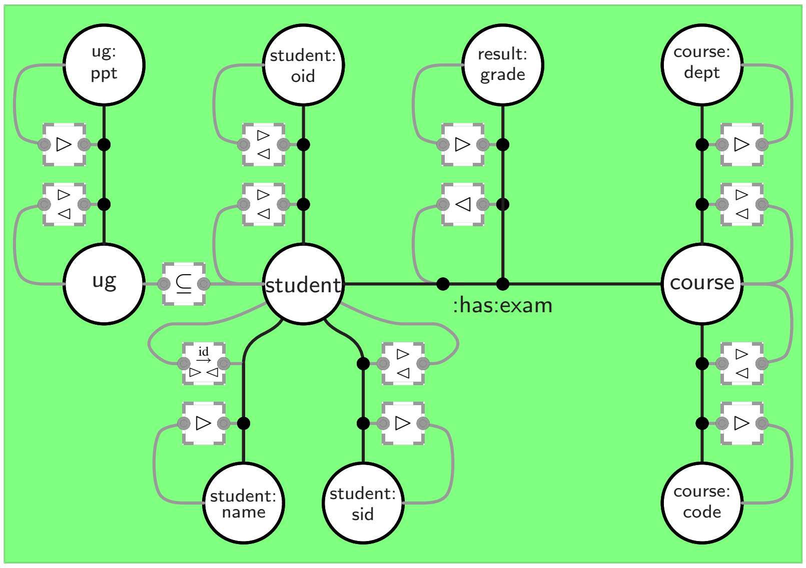


Fig. 13. UML to ER mapping after 12

3.7 Non-Equivalent Schemas

The examples in Figures 3–6 were deliberately chosen to illustrate how we could draw an equivalence between schemas with the same information capacity (with the exception of the UML object identifiers and the other models use of keys). In practice, modelling languages have different expressive powers, and hence there may be no equivalent schema.

For example, changing the cardinality constraint in Figure 3(a) of student being associated with result from 0:N to 1:N would result the addition in Figure 3(b) of a HDM constraint ⟨⟨student⟩⟩ ▷ ⟨⟨result,student,course⟩⟩. If we review the arguments outlined in Sections 3.2–3.5 with this extra constraint in place then we would run into a problem. The reversed edge redirection from ⟨⟨_,result,student⟩⟩ in Figure 11 to ⟨⟨_,result,student:name⟩⟩ in Figure 8 carries the mandatory constraint introduced by 1:N, giving an HDM constraint ⟨⟨student:name⟩⟩ ▷ ⟨⟨result,student,course⟩⟩. When we come to reverse the inclusion merge that merged ⟨⟨result:name⟩⟩ into ⟨⟨student:name⟩⟩ to enable the relationship between ⟨⟨result⟩⟩ and ⟨⟨student:name⟩⟩ to be represented as a foreign key, we are unable to carry this mandatory constraint down to ⟨⟨result:name⟩⟩. This is because the relational schema in Figure 4(a) cannot be altered to express the fact that every student.name must be referenced by at least one result.name. This lost constraint is, therefore, not a weakness in the approach, but an example of the approach formally identifying what information from the ER schema cannot be represented in the relational schema. In this particular case, it might appear that we could repair the relational schema by adding the foreign key constraint student.name → result.name, but this would not be legal since result.name is not a candidate key of result.

4 Handling Additional Modelling Concepts

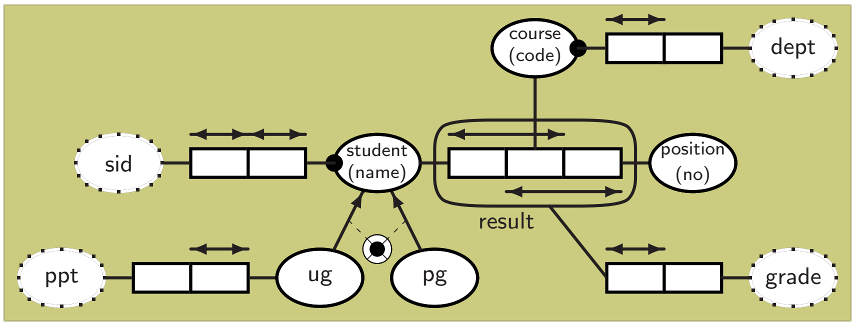
Figure 14(a) illustrates an ORM schema of an extended version of the student-course database where, as we will show, the ORM schema is able to represent some aspect of the UoD that one or more of our other three data models is unable to represent. The additions made to the ORM schema of Figure 6 are described in the following paragraphs under headings which indicate the category of modelling concept they fall under, and we discuss the extent to which the relational, ER and UML models may handle these types of concepts.

Candidate Keys The ORM model of Figure 14(a) has reference/predicate between value ⟨⟨sid⟩⟩ and entity ⟨⟨student⟩⟩ where both roles are key. This implies that we can identify ⟨⟨student⟩⟩ by either ⟨⟨name⟩⟩ or by ⟨⟨sid⟩⟩. This concept can be represented in a relational schema with the ⟨⟨student,sid⟩⟩ attribute being a candidate key. Neither the ER nor UML models have a method of representing this concept however.

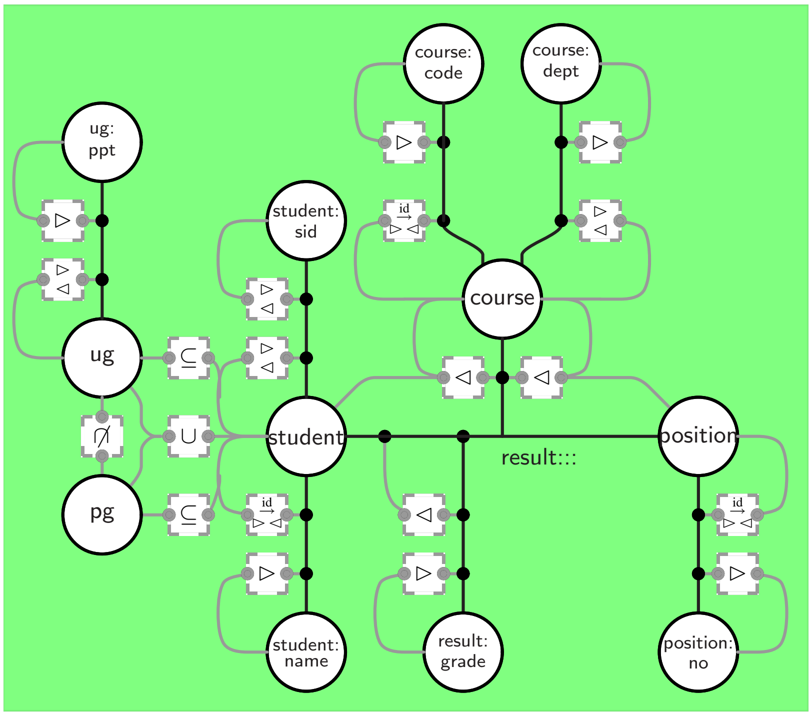
Typically, ER models do not make explicit the relationships between an entity and its attributes, but instead use some sort of syntax with an attribute's name to indicate that it is (or is part of) the primary identifier: no other uniqueness constraints can be expressed. With the more elaborate ER syntax where attribute/entity cardinality constraints are explicit, a one-to-one relationship is synonymous with the primary identifier and therefore can only be expressed once per entity.

Because of UML's reliance on object identifiers, it does not require classes to have value-based reference schemes and indeed requires nonstandard extensions to its notation to express an attribute's uniqueness in its association with its class.

Note that in our HDM production rules for UML there is no rule that generates a uniqueness constraint from an attribute to the edge associating it to its class. In our HDM production rules for ER the only way to generate this constraint is in conjunction with a reflexive constraint.



(a) ORM schema of an extend student-course database



(b) HDM representation of the ORM schema

Fig. 14. An extended student-course database

If we were to convert our extended ORM schema into an HDM schema that could have been produced by an equivalent ER schema, we would have to drop the uniqueness constraint from either ⟨⟨sid⟩⟩ or ⟨⟨student⟩⟩, or extend the ER language to handle candidate keys. For UML, both uniqueness constraints must be dropped, since it has no support for any keys (except by using its constraint language).

Disjointness between entities The ORM model of Figure 14(a) has an additional subclass entity ⟨⟨pg⟩⟩ that is disjoint from ⟨⟨ug⟩⟩, and in addition, ⟨⟨pg⟩⟩ and ⟨⟨ug⟩⟩ are total w.r.t. entity ⟨⟨student⟩⟩.

The disjointness and totality can not be represented in the relational model, since the relational model has no constructs that make use of either the HDM disjoint or union constraints. If we wanted to model our extended ORM example using a relational schema we would have to drop the exclusion constraint between ⟨⟨ug⟩⟩ and ⟨⟨pg⟩⟩ as well as the union constraint between these subsets and ⟨⟨student⟩⟩.

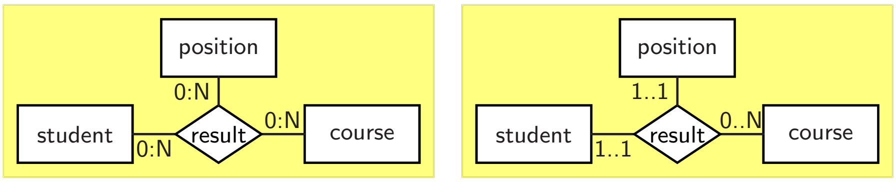
There are several well known ways we may attempt to model the total partition.

- We could represent each subset with a table containing the columns common to just that subset, and a primary key that is also a foreign key to the superclass table ⟨⟨student⟩⟩. The problem is that there is no way to enforce that at most one instance of some referring foreign key exists for each instance of a primary key, *i.e.* that the subclass tables are disjoint.
- With some more elaborate transformations we could change the subclass identification into a column of the superclass telling us which subclass table we should join each instance to. This would enforce the exclusion constraint, and if we made the column not nullable it would also enforce the union constraint. The problem is that the relational model has no way of specifying that a value in a column in the superclass table means that the superclass table joins with a particular subclass table.
- With yet more transformations we could use the subclass-identifying column as in (4) and make each column of each subtype a nullable column of the supertype table where it is set to null when not applicable, and have no subtype tables. Again, the relational model has no way of specifying that certain columns must, or must not, be null depending on the value held in another column.

Therefore, despite there being several ways to model subclasses in the relational model, we can not fully represent the exclusion and union constraints involving ⟨⟨ug⟩⟩ and ⟨⟨pg⟩⟩. The UML and ER modelling languages are able to represent these constraints.

Cardinality constraints of n-ary relationships The ORM model of Figure 14(a) has a ternary ORM fact type between entities ⟨⟨student⟩⟩, ⟨⟨course⟩⟩, and ⟨⟨position⟩⟩. This fact type has two overlapping keys, the first states that any pair of ⟨⟨student⟩⟩ and ⟨⟨course⟩⟩ instances may appear at most once in the fact, and the second that any pair of ⟨⟨position⟩⟩ and ⟨⟨course⟩⟩ instances may appear at most once in the fact.

Whilst it is well known that look-across and look-here cardinality constraints have equivalent expressive power for binary relationships, it is rather less well known that this equivalence does not extend to n-ary relationships [21]. In particular, the use of keys on the ORM ternary fact type is not representable in the ER modelling language as we defined it in Section 2.2, since we choose look-here cardinality constraints. Look-here constraints are restricted to express the cardinality of just a single entity in the relationship. The use of HDM unique constraints in Figure 14(b) has no representation as ER model look-here constraints. Hence the nearest representation of the ternary relationship we can achieve is shown in Figure 15(a), where all cardinality constraints are removed (*i.e.* 0:N is used, meaning that the relationship is unrestricted).



(a) Look here semantics (b) Look across semantics

Fig. 15. Alternative ER ternary-relationship cardinality constraints

The UML model is capable of describing this concept, since it uses look-across cardinality constraints. Many ER modelling languages also use look-across semantics, and hence we could represent the ORM in such languages as shown in Figure 15(b), where the use of the symbol .. between lower and upper bounds for cardinality constraints indicates the use of look-across semantics[1].

To generalise, for **look-here** semantics, we have the restriction on the use of HDM mandatory and unique constraints from Definition 5 to $m = 1$ (*i.e.* only one node or edge is restricted as mandatory or unique), and for **look-across** semantics the restriction is that $m = n - 1$ for an n-ary relationship.

5 Related Work

Graphs and graph transformations are a subject of fundamental interest to computer science, and therefore have been very widely studied [48], and in particular have been studied with application to schema transformation between different modelling languages. What we will now do is set our work in the context of other work (and thus demonstrate it to be distinctive) by discussing the answer to three general questions one might ask about work in the area of 'intermodel transformation'. Note that we exclude from the discussion issues associated with modelling languages that are not set oriented, and also the typing of data, since these matters are outside the scope of the work presented in this paper.

What is being modelled, and how is it modelled? The first distinction is between graphs which model the **dynamic behaviour** of a computer system (a recent survey may be found in [3], and general tools to model software have been constructed such as the

[1] Whilst this example might indicate that look-across semantics have advantages, there will be other modelling situations that would be better modelled with look-here constraints. There are also operational issues to consider, in that look-across constraints on n-ary relationships are difficult to implement: for example, if mandatory were used in n-ary relationships, then inserting or deleting an instance from one entity will require more than one relationship instance to be inserted or deleted. However, since we are in this paper only studying the logical aspects of data models, we will not study this issue further.

PROGRES system [33]), and graphs which model the **static** (*i.e.* data) aspect of the system, in which area our work falls into. There is a degree of overlap between the two, in that programs must manipulate data to perform useful tasks (for example in the PROGRES system [34]).

Considering work that is focused on static data modelling via graphs, the next distinction we can make is between systems that are data **model specific**, or those which handle **multiple models**. This is not a binary distinction, but denotes a spectrum. At one extreme we find systems that map between schemas in a single modelling language: normally the relational model such as in the self-documenting data models of [26], or the initial version of Clio [50]. In the middle of the spectrum, much work has concentrated on converting between just a few specific modelling languages: for example between relational and ER [1, 36], ORM to UML or relational [20], relational and generic object oriented models [11, 23], XML and relational data [37], *etc*. Our work, along with [18, 2, 10, 7, 44], attempts to provide a more flexible framework, and demonstrate how the framework is adaptable to a wide range of data modelling languages.

Within approaches that deal with multiple models, there is then the distinction to make as to how the approaches handle the requirements of multiple data modelling languages. Invariably, some **common data model** (**CDM**) [43] is used as a intermediate language, into which schemas of all other data modelling languages are translated. There is then the choice of how expressive with CDM language should be [17]. Some approaches take a union of all modelling constructs to form a **high level** CDM which has all the features of the models being represented (for example, by having constructs for key, aggregation, function dependency, and so on). **DB-MAIN** [18, 22, 17] and **MDM** [2] are examples of tools that take this approach. Our work, and [10, 7, 44], instead use a **low level** CDM, where the CDM has a few simple modelling constructs, plus the ability to express some constraints over those modelling constructs. Our work differs from other approaches in having developed a small set of constraint primitives that may be used instead of general logical expressions. The constraint primitives are relatively fine grained, which has the advantage that our transformations can deal with just those aspects of the constraints that are relevant to the particular transformation.

What type of Graph Language is used? The notion of a 'graph' is a very general one, but we will restrict ourselves to saying that we are considering languages that have the concept of there being nodes, along with some method of defining relationships between those nodes. From that basis, there have been a wide range of variations of what semantics are attached to the nodes and relationships. A major distinction is between graph models that are **schema models** where data is just the values associated with the extent of the nodes and relationships. The HDM, and [26, 2, 44, 18, 10, 16, 46] fall into this category. Alternatively, the graph models may be **two-level models**, modelling as nodes and associations between nodes both the schema and the data instances. Examples of this approach include [7, 23].

Another important feature to distinguish is what relationships are provided in the graph model. Using the terminology of the HDM, we can distinguish between relationships that model (1) edges (*i.e.* data that is restricted in extent to values that appear in the nodes that the edge connects), (2) constraints (*i.e.* just restrict the values that may appear in the nodes that the relationship connects) or (3) edge-constraints (*i.e.* are edges, with some implied constraints).

Perhaps surprisingly, some graph models have just nodes and constraints. For example, **MDM** has three types of node — abstract, aggregation and lexicals — and six types of constraints (which they call edges) that represent functional dependency, multivalued functional dependency, components of aggregation, keys of aggregation, and keys of abstract. Also, the **ULD** [7] representation language has nodes representing sets of tuples called 'constructs' which may have first order logic constraints placed on their extent, and **WOL** [16] models nodes as classes, with the notion of keys used to identify class instances, and general purpose constraint language to use over the class instances.

A common type of graph language is to have nodes and binary edges, such as in Clio, possibly with additional constraints, such as [26, 10]. In [44] the **reserved graph grammar** (**RGG**) is used to represent data models, where an RGG is comprised of two types of node, and binary edges between these nodes, and some constraint relationships.

A hypergraph model was used in the specific case of relational schema transformations in [51], and nested hypergraphs have been studied as a general framework for modelling complex objects [38]. The **higher-order ER model** (**HERM**) [46] is similar to the HDM, in that it is also a nested hypergraph model: it allows for n-ary relationships which may connect to entities or other relationships. However, the HERM still distinguishes between attributes and entities, and it uses relatively course grained constraints. In particular, cardinality constraints are not decomposed into two primitives as they are in this paper, nor is there the equivalent of our reflexive constraint. Although there is some discussion in [46] about mapping HERM to relational and network models, we are not aware of other work that uses nested hypergraphs as a general basis for mapping between schemas in different modelling languages. An advantage of the HDM over other approaches is that it clearly separates those constructs that have an extent (nodes and edges) from those which restrict the extent of other constructs (constraints). Another advantage of the use of a hypergraph based model is that it has a natural mapping to all higher level modelling languages we have studied to date.

How are the transformations specified? The notion of transforming schemas is widely studied, and surveys of database transformations can be found in [4, 15]. A distinction that can be drawn between most previous work and ours is the level of granularity at which the transformations between schemas are presented. Most work [18, 26, 44, 16] takes a **coarse grain** approach, where a transformation will specify a semantic mapping between equivalent structures in the high level modelling language, such as the mapping between a many-many relationship in one schema to an entity with two one-many relationships. In our approach and in [10, 6], the aim is to specify transformations at a **fine grain** level. Our approach differs in that our transformations are schema oriented, in the sense that we incrementally add, delete or rename single constructs in the HDM, rather than be query oriented as in [10, 6] where they specify the query that is used to map sets of constructs in one model to sets of constructs in another. The use of BAV transformations has the advantage that it establishes a bidirectional mapping between schemas.

6 Acknowledgments, Summary and Future Work

The original HDM and its implementation in the AutoMed project was developed in collaboration between Imperial College and Birkbeck College, and the AutoMed project was funded by the EPSRC. We acknowledge the contribution of all the AutoMed project members to many discussions of the work presented in this paper, and also acknowledge the reviewers of this paper for many helpful suggestions and for detecting various errors in the early drafts of this paper.

In this paper we have extended the **hypergraph data model** (**HDM**) [39]. The HDM differs from normal graphs in that edges may connect together any number of nodes or edges rather than just two nodes. HDM nodes and edges have an extent, with the extent of an edge being constrained to take values that must appear in the extent of the nodes or other edges it connects.

The HDM in [39] allows arbitrary constraints. Our extension in this paper is to define six fundamental constraints that we believe cover the majority of features in the popular high level data modelling languages. We have used these to precisely define mappings between the HDM schema and a representative set of features from the relational, ER, ORM and UML-class data models.

We have taken an example UoD, and given schemas for that UoD in four high level data models named above. Then using our mappings we have derived an equivalent HDM representation for each. The similarity of each HDM graph with its high level data model's counterpart illustrates the fact that many of the semantics of these high level data models implied by their graphical structures are similar across the spectrum of data models, and this commonality is captured by the graphical structure of HDM. Although each HDM schema was shown to be equivalent (with the exception of some aspects of the UML model) in the sense that each has the same information capacity, they were syntactically different.

We reviewed the set of primitive reversible transformations on HDM graphs that we have used extensively elsewhere in our schema integration and evolution work [27, 39, 29, 30]. We then gave five equivalence preserving graph transformations predicated on using only our six fundamental constraints, and defined in terms of these primitive HDM transformations.

We then showed how the three equivalent HDM graphs from the ER, relational and ORM schemas can be converted into each other through a sequence of these equivalence preserving transformations. The mappings from the high level data models to their HDM representations, along with the equivalence transformations between them show that these three schemas are equivalent, by virtue of the fact that only `add`, `delete` and `rename` transformations were required to implement the mapping. As the equivalence transformations are built from HDM's reversible primitive transformations, queries can be rewritten from one schema to another. In principle, we could therefore migrate data from an instance of a schema in one data modelling language to an instance of an equivalent schema in a different data modelling language.

We also discussed how our approach identifies when two HDM schemas are not equivalent by virtue of any 'left over' constraints from one schema when we try to convert it into the other schema (after transformations are applied, there are some constraints in one schema that do not appear in the other). This was used to show exactly how the UML schema from the example UoD differs from the other three schemas. We also outlined how our approach might be used to demonstrate that some features of a high level data modelling language may not be representable in another. Note this does not prove that the two schemas are not equivalent: to realise this goal we would need to prove that our equivalence transformations are adequate to convert between any two equivalent HDM schemas, and also prove that every feature of the data models in question are mapped into an equivalent set of HDM model constructs.

Our future work will expand our approach to take account of type information in a data source, and also to model list and bag based data models, such as XML (which has list based semantics) and SQL (*i.e.* the relational model with bag semantics). We will also investigate heuristic search techniques to determine automatically which equivalence preserving transformations are required to map a schema in one modelling language into a schema in another modelling language.

References

1. M. Andersson. Extracting an entity relationship schema from a relational database through reverse engineering. In *Proc. ER'94*, LNCS, pages 403–419. Springer, 1994.
2. P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proc EDBT'96*, volume 1057 of *LNCS*, pages 79–95. Springer-Verlag, 1996.
3. L. Baresi and R. Heckel. Tutorial introduction to graph transofrmation: A software engineering perspective. In A. Corradini *et al*, editor, *Proc. ICGT*, volume 2505 of *LNCS*, pages 402–429. Springer-Verlag, 2002.
4. C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
5. P.A. Bernstein. Applying model management to classical meta data problems. In *Proc. CIDR'03*, 2003.
6. S. Bowers and L. Delcambre. On modeling conformance for flexible transformation over data models. In *Knowledge Transformation for the Semantic Web*, pages 34–48. IOS Press, 2003.
7. S. Bowers and L. Delcambre. The uni-level description: A uniform framework for representing information in multiple data models. In *Proc. ER'03*, volume 2813 of *LNCS*, pages 45–58. Springer-Verlag, 2003.
8. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE2004*, volume 3084 of *LNCS*, pages 82–97. Springer-Verlag, 2004.
9. 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.
10. K.T. Claypool and E.A. Rundensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *Proc. ICEIS 03*, pages 219–224, 2003.
11. C.J. Date. Object identifiers vs. relational keys. In *Relational Database: Selected Writings 1994–1997* [14], chapter 12, pages 457–476.
12. C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 8th edition edition, 2004.
13. C.J. Date, H. Darwen, and N.A. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2003.
14. C.J. Date, H. Darwen, and D. McGoveran. *Relational Database: Selected Writings 1994–1997*. Addison-Wesley, 1998.

15. S.B. Davidson, P. Buneman, and A.S. Kosky. Semantics of database transformations. In *Semantics in Databases*, LNCS, 1998.
16. S.B. Davidson and A.S. Kosky. WOL: A language for database transformations and constraints. In *Proc. ICDE97*, pages 55–65, 1997.
17. J-L. Hainaut. Transformation-based database engineerig. In *Transformation of Knowledge, Information, and Data* [48], chapter 1, pages 1–28.
18. J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. Database evolution: the DB-MAIN approach. In *Proc. ER'94*, LNCS, pages 112–131. Springer, 1994.
19. P. Hall, J. Owlett, and S.J.P. Todd. Relations and entities. In G.M. Nijssen, editor, *Modelling in Data Base Management Systems*. North-Holland, 1975.
20. T. Halpin. *Information Modeling and Relational Databases*. Academic Press, 2001.
21. S. Hartmann. Reasoning about participation constraints and Chen's constraints. In *Proc. 14th Australasian database conference*, pages 105–113. Australian Computer Society, 2003.
22. J-M. Hick and J-L. Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *Proc. ER'03*, volume 2813 of *LNCS*, pages 291–306. Springer-Verlag, 2003.
23. J.H. Jahnke and A. Zündorf. Apply graph transformations to database re-engineering. In *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 2, chapter 6. World Scientific, 1999.
24. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report No. 20, AutoMed, 2003.
25. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.
26. L. Mark and N. Roussopoulos. Integration of data, schema and meta-schema. In *Proc. ER83*, pages 585–602, 1983.
27. P.J. McBrien and A. Poulovassilis. A formalisation of semantic schema integration. *Information Systems*, 23(5):307–334, 1998.
28. 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.
29. 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.
30. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, pages 227–238. IEEE, 2003.
31. P.J. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P, at VLDB'03*, pages 91–107, 2003.
32. R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19(1):3–31, 1994.
33. M. Munch. Programmed graph rewriting system PROGRES. In *In Proc. AGTIVE'99*, volume 1779 of *LNCS*, pages 441–448. Springer-Verlag, 2000.
34. M. Munch, A. Schurr, and A.J. Winter. Integrity constraints in the multi-paradigm language progres. In H. Ehrig *et al*, editor, *Graph Transformation*, volume 1764 of *LNCS*, pages 338–352. Springer-Verlag, 2000.
35. S. Patig. Measuring expressiveness in conceptual modeling. In *Proc. CAiSE2004*, volume 3084 of *LNCS*, pages 127–141. Springer-Verlag, 2004.
36. J-M. Petit, F. Toumani, J-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proc. ICDE'96*, pages 218–227, 1996.
37. L. Popa, M.A. Hernandez, and Y. Velegrakis *et al*. Mapping XML and relational schemas with Clio. In *Proc. ICDE'02*, pages 498–499, 2002.
38. A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. on Information Systems*, 12(1):35–68, 1994.

39. A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.
40. N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. of 6th ICEIS*, 2004.
41. N. Rizopoulos and P.J. McBrien. A general approach to the generation of conceptual model transformations. In *Proc. CAiSE'05*, volume 3520 of *LNCS*. Springer-Verlag, 2005.
42. K. Schewe. Design theory for advanced datamodels. In *Proc. 12th Australasian Conf. on Database Technologies*, pages 3–9, 2001.
43. A. Sheth and J. Larson. Federated database systems. *ACM Computing Surveys*, 22(3):183–236, 1990.
44. G. Song, K. Zhang, and J. Kong. Model management through graph transformations. In *Proc. Visual Languages and Human-Centric Computing*, pages 75–82. IEEE, 2004.
45. I.Y. Song, M. Evans, and E.K. Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer & Software Engineering*, 3(4):427–459, 1995.
46. B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer, 2000.
47. N. Tong. Database schema transformation optimisation techniques for the AutoMed system. In *Proc. BNCOD'03*, volume 2712 of *LNCS*, pages 157–171. Springer, 2003.
48. P. van Bommel. *Transformation of Knowledge, Information, and Data*. Idea Group, 2005.
49. R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.
50. L.L. Yan, R.J. Miller, L.M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proc. SIGMOD'01*, pages 485–496, 2001.
51. C. Zaniolo and M. Melkanoff. A formal approach to the definition and the design of conceptual schemata for database systems. *ACM TODS*, 7(1):24–59, 1982.