The University Of Sheffield.

ReMoDeL Data Refinement

Data Transformations in ReMoDeL, Part 1

Technical Report

Revision: 1.0

Date: 25 July 2022

Anthony J H Simons
Department of Computer Science
University of Sheffield

Contents

1. Introduction

This document describes the first part of a chain of model transformations applied to the task of data refinement, using **ReMoDeL v3**, a high-level syntax for defining models and model transformations. It assumes the reader is familiar with the *ReMoDeL* metamodel language and transformation language [1]. It also assumes prior knowledge of data modelling notations, including the UML Class Diagram [2], the "Crow's Foot" Entity Relationship Diagram [3] and the SQL Data Definition Language [4].

1.1 Software Engineering Models

Model-Driven Engineering (MDE) is a general strategy in software engineering that creates and manipulates software designs at a high level using abstract models. Model-Driven Development (MDD) is the subfield which focuses specifically on generating executable software systems from high level designs. To do this, there must exist suitable design models that capture relevant views of the intended software system. Each view offers a quasi-independent perspective, an abstraction, or simplification, of some aspect of the system.

Computer Aided Software Engineering (CASE) tools have traditionally supported three main views, constructing models that highlight *data, process* and *time*:

- The *data view* is usually expressed in different kinds of structural data model, such as an Entity-Relationship Diagram [3], a simple kind of information model; or a UML Class Diagram [2], which captures more semantic relationships.
- The *process view* can be expressed using a Dataflow Diagram [5] describing processes with their inputs and outputs; or UML Activity Diagram [2] which also describes the sequencing of processes; or a Jackson Structured Program chart [6], which describes the detailed program block structure.
- The *time view* can be expressed using a traditional flowchart [7], a UML Activity Diagram [2] or State Machine Diagram [2], both of which express ordering constraints; or the UML Sequence Diagram [2] or Communication Diagram [2] which describe a more detailed call-graph.

A diagram is a graphical representation of an underlying model, which captures certain logical information. A model is constructed from elements, typically vertices, edges and attributes, that are taken from a metamodel. Each model element is an instance of some type defined in the metamodel. In this sense, a metamodel is "the type of" a model [1].

1.2 Transformation Chains

A model transformation is a collection of rules for transforming the elements of a *source* model into the elements of a *target* model. A transformation may be *endogenous*, meaning that the source and target models have the same metamodel type, or *exogenous*, meaning that the source and target metamodels are distinct and the transformation performs a translation from one type to the other [1].

Where the target type of one transformation is the source type for another, it is possible to construct *transformation chains*. Several transformations may then be applied consecutively, where the output of one transformation is used as the input for the next one in the sequence. Transformation chains are employed in MDD, in which high-level abstract models are progressively refined, via intermediate model representations, into concrete models that are

closer to executable code. Building such transformation chains is in fact the goal of MDD, which seeks to find suitable refinement rules and model representations.

In a declarative transformation language, like that of *ReMoDeL*, each transformation is a *functional mapping* from a source to a target. Therefore, when two transformations are chained together, this is equivalent to *function composition*. This provides a mathematical basis for reasoning about transformation chains. A transformation may be a simple *mapping*, with one source and target, or a more complex *merging*, with multiple sources and one target. A chain of mapping transformations is a *linear function composition*. A chain of merging transformations is a *hierarchical function composition*.

1.3 Data Refinement

The general problem of combining the different high-level views of a software system has not yet been solved. Here, we focus on the more tractable sub-problem of *data refinement*, the transformation of a high-level and semantically rich data model given by the UML Class Diagram [2], via an intermediate representation of data offered by the Entity-Relationship Diagram [3], to a low-level model corresponding to the SQL Data Definition Language used to define a relational database [4].

The data refinement problem is one of the better-understood problems in MDD, due to the existence of well-known methods for normalising a data model. The chain of transformations to be considered altogether includes the following:

- *Class Diagram to ER Diagram*: this first transformation maps each UML class to an entity. Some of these are strong and others weak, if they depend on related entities for identification. The UML semantic relationships: association, aggregation, generalisation and composition, are mapped to simpler relationships, in which the direction of dependency is correctly established.
- *ER Diagram to Normal ER Diagram:* this second transformation converts the ER Diagram to at least third normal form (3NF+). It merges one-to-one relationships, and splits many-to-many relationships by introducing an intermediate linker entity. Every entity has a natural, derived, or surrogate identifier.
- *Normal ER Diagram to Existence Dependency Graph:* this third transformation orders the entities by existence dependency and converts relationships into directed references owned by the entities. These form the basis for foreign keys.
- *Existence Dependency Graph to Database Schema:* this fourth transformation converts entities to tables, attributes to columns with database types, identifiers to primary keys and references to additional columns and foreign keys. Column names are transformed to prevent name clashes. Data deletion semantics are identified.
- *Database Schema to SQL Data Definition Language:* the final code generation step is a simple translation of the Database Model to SQL. It uses a bespoke code generator written in Java, designed according to the *Visitor Pattern* [8].

This document (part 1 of 2) covers the first half of the above transformation chain, from the *UML Class Diagram* to the *Normal ER Diagram*. Partly, this is due to the need to introduce each of the modelling notations and their *ReMoDeL* encodings, before explaining the various mapping rules involved in the first two transformations.

2. UML Class Diagram

The UML Class Diagram is a well-known notation for modelling classes and relationships in object-oriented programming [2]. The diagram can be used at many stages in the software engineering lifecycle, from initial conceptual sketches to detailed documentation of code. Here, we are interested in the early use of this diagram in information modelling, to capture entities, attributes and semantic relationships. This is assumed to be the entry point to data analysis, where the designer records initial information about data.

2.1 Class Attributes and Operations

When the UML Class Diagram is used in information modelling, the emphasis is on data, rather than behaviour. The diagram consists of classes, each containing a number of named attributes and operations (collectively known as features). Figure 1 illustrates the notation.

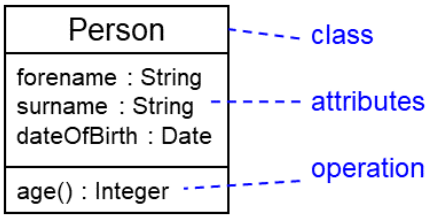

Figure 1: UML notation for class, attribute, operation

The attributes have basic types, such as *Integer, Real* or *String*. They are assigned to a class on the basis of *attribute dependency*, viz. the attribute's value is logically determined by the given class instance in question.

> For example, the *number* of an *Account* depends directly on the *Account* instance in question, so should be assigned as an attribute of *Account*. By contrast, the *forename* of the *Account's* holder should not be assigned as an attribute of *Account*, since the value of this is not determined directly by the *Account.*

A class may optionally contain operations, specified as signatures, annotated with argument and result types. When present, these may identify high-level business operations owned by the class, or *derived* features of the class, viz. whose values can be calculated from other attributes.

> For example, if the *dateOfBirth* is an attribute of a *Person*, then the *age* of a *Person* is derived, since it can be calculated from the *dateOfBirth* and the current date.

UML also uses the syntax */age* (with a prefix slash) to indicate a derived attribute, which we take as being equivalent to an operation *age()* returning the same result. Operations play no further part in data analysis, since they are not stored as data.

Class features may be annotated with markers to indicate *private, protected, public* or *package* visibility. We take the view that this is not a concern of analysis. These may be added later during design, according to rules of encapsulation. Visibility plays no further part in data analysis, which is only concerned with stored (non-derived) attributes.

2.2 Class Semantic Relationships

Classes enter into semantic relationships. There are six principal relationships, and two variants of the *association* relationship, illustrated in figure 2. All of these are binary relationships relating a source class to a target class.

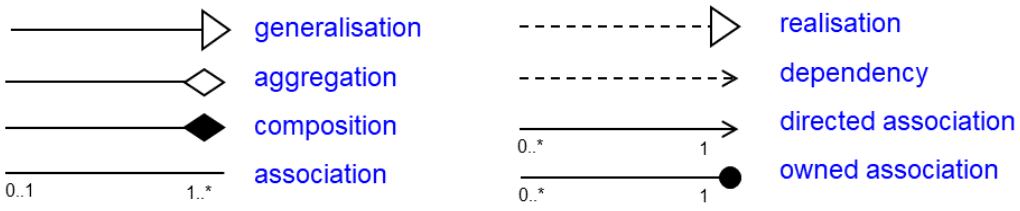

Figure 2: UML notation for semantic relationships

The first four relationships are directly relevant to an analysis of data dependency:

- *Generalisation*: relates a more specific subclass to a more general superclass. This describes an inheritance relationship, in which the subclass inherits all the features of the superclass. This also describes a type-compatibility relationship in which a subclass may be substituted where the superclass was expected.
- *Aggregation*: relates a component part class to an assembled whole class. This describes a whole-parts relationship, in which the parts exist independently, but may be included as part of the whole assembly.
- *Composition*: relates a constituent part class to a composite whole class. This describes a whole-parts relationship, in which the parts cannot exist independently from the whole, which is indivisible.
- *Association*: relates one class to another class, with a multiplicity marker at each end. This describes an associative relationship, in which each class is related to a specific multiple of the class at the other end of the association.

UML regards composition and aggregation as special cases of association, since both may have multiplicity markers at their ends. Generalisation does not show multiplicity, but this is implicit. All of these relationships have implications for data dependency.

Two of the remaining semantic relationships describe functional dependency, rather than data dependency.

- *Realisation*: relates a concrete class to an abstract interface. This indicates a type-compatibility between the class and the interface, such that the class may be substituted where the interface was expected.
- *Dependency*: relates one class to another on which it functionally depends in some way that is not already captured by other semantic relationships. This is a catch-all relationship, used where no other relationship is appropriate.

Realisation is similar to *generalisation* in its type-compatibility sense, but has no further consequences for data, since an interface is abstract. *Dependency* is used to denote functional dependency, where one class uses another class passed as an argument to an operation. This relates to behavioural coupling and has no further bearing on data analysis.

The final two relationships are variants of *association*, which should properly be used only during design, since they make concrete decisions about implementation strategy:

- *Directed association*: is an association, in which it is possible to navigate efficiently from the source class to the target class. This expresses a coding requirement, which could be, but need not be, implemented through a direct reference.
- *Owned association*: is an association, in which the source class owns a direct reference to the target class. This expresses a structural data dependency.

We take the view that these should not be the concern of analysis. Data analysis is able to resolve the direction of data dependency; and if it turns out that this conflicts with any premature decision specified using these relationships, then the model is inconsistent.

2.3 Class Diagram Examples

We shall develop two case studies in the rest of this document, to illustrate the different UML notations and how these affect the process of data analysis. Figure 3 shows the information model for a cycle shop that sells custom-built bicycles to its customers.

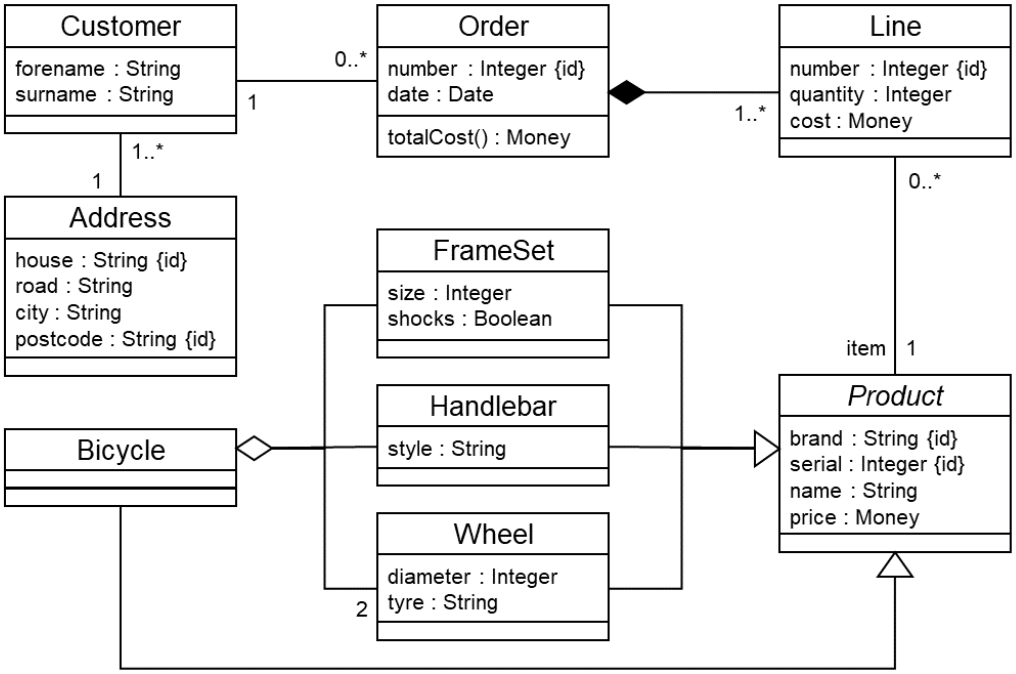

Figure 3: UML Class Diagram for the Cycle Shop

This case study contains examples of all the semantic relationships. Generalisation is used to show that *Bicycle, FrameSet, Handlebar* and *Wheel* are all kinds of *Product* sold in the shop. Aggregation is used to show that a *Bicycle* is assembled from a *FrameSet*, a *Handlebar* and two *Wheels*. Composition is used to show that an *Order* consists of multiple *Lines*. Association is used to show that every *Order* is for one *Customer* (but a *Customer* may place many *Orders*); likewise to show that every *Customer* lives at one *Address* (which may hold many *Customers*).

Some of the classes contain attributes marked with the UML constraint {*id*}, to indicate that they are identifiers. We take the view that where many attributes are so marked, they will form a compound identifier (rather than alternative candidate keys). Where no identifiers are listed, the data transformation process must later generate a surrogate key.

The diagram in figure 3 leaves a number of specifications implicit. We assume that UML default interpretations will apply. None of the associations is named (this is optional in UML). The ends of the associations sometimes contain multiplicity markers, and otherwise we assume that the multiplicity is 1 (the default in UML). The association end-roles are mostly unnamed (apart from *item*) and we assume that end-role names may be synthesised from the nearby adjoining type name.

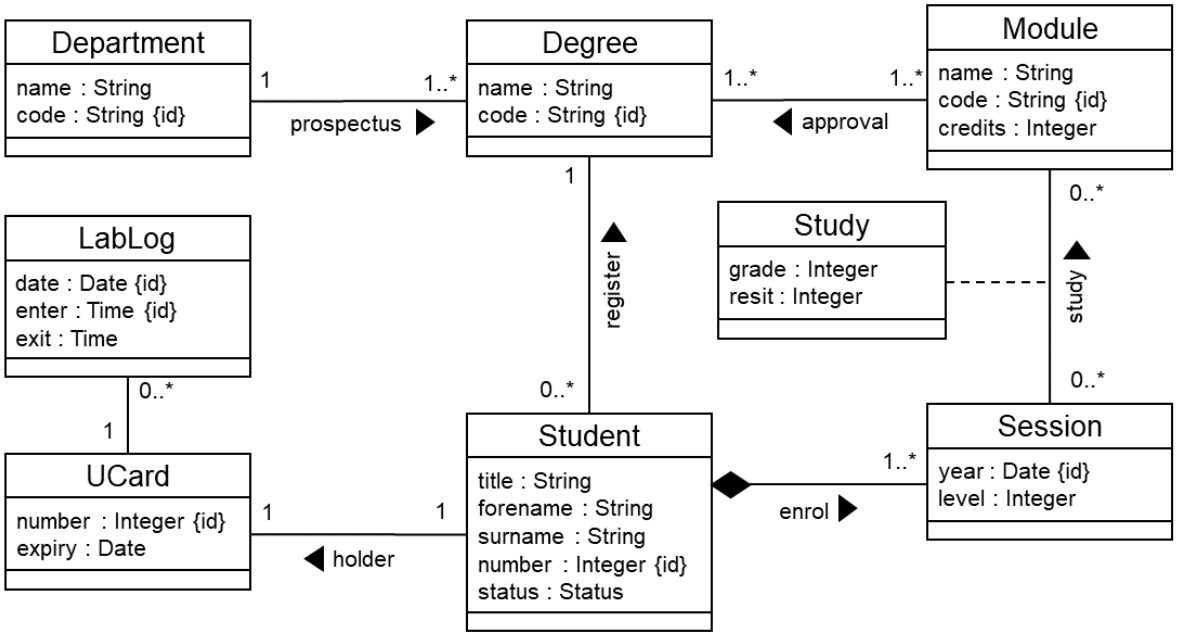

Figure 4: UML Class Diagram for Student Records

Figure 4 shows the information model for a student records system. This mostly deals with different kinds of association, which are mostly named. The *holder* association is one-to-one, and the *approval* association is many-to-many, both of which require special treatment during data normalisation. The *study* association is many-to-many and also has its own attributes, represented by the *Study* association class. The focus of the model is on the *Student* record with its dependent *Session* record that links to the programme of modules followed by the student in a given academic session. Several classes have attributes with the same names, such as *name, code* and *number*, which require special treatment in foreign key generation.

These two case studies will be created as models within *ReMoDeL*, that is, as instances of a metamodel representing the types of element in a UML class diagram. The models will be used as input to a chain of model transformations.

2.4 Metamodel for a Class Diagram

Figure 5, which extends over two pages, shows a metamodel for a UML Class Diagram, suitably simplified for information modelling. It excludes certain features irrelevant to this purpose (such as interfaces, realisation) but preserves operations.

Using the *ReMoDeL* textual syntax for metamodels [1], it describes *Named* things having a *name: String*. *Type* is a kind of *Named* thing that is subdivided into *BasicType* and *ClassType*. The *Typed* concept, a kind of *Named* thing, refers to a *type: Type*. Its derived concepts include *Variable* (inheriting *type*), *Operation* (inheriting *type*, which declares a list

of *Variable* arguments), *Attribute* (whose *type* is specialised as a *BasicType*) and *EndRole* (whose *type* is specialised as a *ClassType*).

```
metamodel UML {
   concept Named {
      attribute name : String
   }
   concept Type inherit Named {
   }
   concept BasicType inherit Type {
   }
   concept ClassType inherit Type {
      component attributes : Attribute{}
      component operations : Operation{}
      operation identifiers : Attribute{} {
         attributes.select(attrib | attrib.id)
      }
      operation dependents : Attribute{} {
         attributes.reject(attrib | attrib.id)
      }
   }
   concept Typed inherit Named {
      reference type : Type
   }
   concept Variable inherit Typed {
   }
   concept Attribute inherit Typed {
      reference type : BasicType
      attribute id : Boolean
   }
   concept Operation inherit Typed {
      attribute specification : Boolean
      component arguments : Variable[]
   }
   concept EndRole inherit Typed {
      reference type : ClassType
      attribute range : String
      operation isOne : Boolean {
         range = "" or range = "1" or range = "1..1"
      }
      operation isZeroOne : Boolean {
         range = "0..1"
      }
      operation isZeroMany : Boolean {
         range = "*" or range = "0..*"
      }
      operation isOneMany : Boolean {
         range = "1..*"
      }
      operation isMany : Boolean {
         not self.isOne
      }
      operation isOptional : Boolean {
         self.isZeroOne or self.isZeroMany
      }
      operation isMultiple : Boolean {
         not (self.isOne or self.isZeroOne)
      }
      operation getName : String {
         if name /= "" then name
            else type.name.asName
      }
   }
   concept Relationship inherit Named {
      component source : EndRole
      component target : EndRole
      operation isOneToOne : Boolean {
```

```
            source.isOne and target.isOne
         }
         operation isOneToMany : Boolean {
            source.isOne and target.isMany
         }
         operation isManyToOne : Boolean {
            source.isMany and target.isOne
         }
         operation isManyToMany : Boolean {
            source.isMany and target.isMany
         }
      }
      concept Generalisation inherit Relationship {
         operation getName : String {
            if name /= "" then name
               else source.type.name.concat("KindOf")
                  .concat(target.type.name)
         }
      }
      concept Aggregation inherit Relationship {
         operation getName : String {
            if name /= "" then name
               else source.type.name.concat("MadeOf")
                 .concat(target.type.name)
         }
      }
      concept Composition inherit Relationship {
         operation getName : String {
            if name /= "" then name
               else source.type.name.concat("PartOf")
                  .concat(target.type.name)
         }
      }
      concept Association inherit Relationship {
         component type : ClassType
         operation getName : String {
            if name /= "" then name
              else if type /= null then type.name
                 else (source.type.name).concat("To")
                    .concat(target.type.name)
         }
      }
      concept Diagram inherit Named {
         component basicTypes : BasicType{}
         component classTypes : ClassType{}
         component generalisations : Generalisation{}
         component aggregations : Aggregation{}
         component compositions : Composition{}
         component associations : Association{}
      }
   }
```

Figure 5: A metamodel for the UML Class Diagram

The *EndRole* concept plays a significant part in data analysis. It defines a *range: String* to record the multiplicity marker, which in UML can be expressed in a variety of ways. To aid in grouping the alternatives, a number of operations are defined. These in turn are the basis for the operations *isOne, isMany, isOptional, isMultiple*, which later inform the translation. If the *EndRole* is not explicitly named, a name may be synthesised from the related type (the *ReMoDeL* operation *asName* converts type case to name case).

All semantic relationships are rooted in *Relationship*, which has a source *EndRole* and a target *EndRole*. From this are derived *Generalisation, Aggregation, Composition* and *Association*. These may specify explicit multiplicities in their *EndRoles*, or leave these

undefined (default assumptions are made later). An *Association* may optionally have a component *type: ClassType* to indicate that it is a UML association class. *Relationship* plays a useful part in data analysis by offering the operations *oneToOne, oneToMany, manyToOne* and *manyToMany*, which in turn depend on the *EndRole* operations. If a *Relationship* is not explicitly named, a suitable name is generated by operation.

Finally, a *Diagram* is the top-level enclosing concept in the metamodel, containing sets of the other types of element. Basic types, class types and the four semantic relationships are stored separately as components of the diagram.

We highlight a number of *ReMoDeL* syntax features. In this metamodel:

- inheritance is used to derive concepts from more general concepts;
- reference types may be specialised simply by redeclaring them;
- property access may be controlled using explicit access operations.

Since access expressions like: *obj.prop* are translated by the compiler into Java access methods: *obj.getProp()*, it is possible to specialise property access by defining explicit *getProp()* operations in subtype concepts. This is used to provide default name rules for some concepts. The compiler redefines *getProp()* methods to retype references.

2.5 Cycle Shop Example Model

Figure 6 encodes the first example UML Class Diagram from figure 3 in the *ReMoDeL* textual syntax for models [1]. This defines seven basic types (including the five standard UML basic types, plus the two datatypes *Date* and *Money*) and nine class types.

```
model uml1 : UML {
   d1 : Diagram(name = "Cycle Shop", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
      b7 : BasicType(name = "Money")
   }, classTypes = ClassType{
      c1 : ClassType(name = "Address", attributes = Attribute{
         a1 : Attribute(name = "house", type = b5, id = true),
         a2 : Attribute(name = "road", type = b5),
         a3 : Attribute(name = "city", type = b5),
         a4 : Attribute(name = "postcode", type = b5, id = true)
      }),
      c2 : ClassType(name = "Customer", attributes = Attribute{
         a5 : Attribute(name = "forename", type = b5),
         a6 : Attribute(name = "surname", type = b5)
      }),
      c3 : ClassType(name = "Order", attributes = Attribute{
         a7 : Attribute(name = "number", type = b2, id = true),
         a8 : Attribute(name = "date", type = b6)
      }, operations = Operation{
         o1 : Operation(name = "totalCost", type = b7)
      }),
      c4 : ClassType(name = "Line", attributes = Attribute{
         a9 : Attribute(name = "number", type = b2, id = true),
         a10 : Attribute(name = "quantity", type = b2),
         a11 : Attribute(name = "cost", type = b7)
      }),
      c5 : ClassType(name = "Product", attributes = Attribute{
         a12 : Attribute(name = "brand", type = b5, id = true),
```

```
         a13 : Attribute(name = "serial", type = b2, id = true),
         a14 : Attribute(name = "name", type = b5),
         a15 : Attribute(name = "price", type = b7)
      }),
      c6 : ClassType(name = "Bicycle"),
      c7 : ClassType(name = "FrameSet", attributes = Attribute{
         a16 : Attribute(name = "size", type = b2),
         a17 : Attribute(name = "shocks", type = b1)
      }),
      c8 : ClassType(name = "Handlebar", attributes = Attribute{
         a18 : Attribute(name = "style", type = b5)
      }),
      c9 : ClassType(name = "Wheel", attributes = Attribute{
         a19 : Attribute(name = "diameter", type = b2),
         a20 : Attribute(name = "tyre", type = b5)
      })
   }, generalisations = Generalisation{
      g1 : Generalisation(source =
         e1 : EndRole(type = c6), target =
         e2 : EndRole(type = c5)),
      g2 : Generalisation(source =
         e3 : EndRole(type = c7), target =
         e4 : EndRole(type = c5)),
      g3 : Generalisation(source =
         e5 : EndRole(type = c8), target =
         e6 : EndRole(type = c5)),
      g4 : Generalisation(source =
         e7 : EndRole(type = c9), target =
         e8 : EndRole(type = c5))
   }, aggregations = Aggregation{
      a21 : Aggregation(source =
         e9 : EndRole(type = c7), target =
         e10 : EndRole(type = c6)),
      a22 : Aggregation(source =
         e11 : EndRole(type = c8), target =
         e12 : EndRole(type = c6)),
      a23 : Aggregation(source =
         e13 : EndRole(type = c9, range = "2"), target =
         e14 : EndRole(type = c6))
   }, compositions = Composition{
      c10 : Composition(source =
         e15 : EndRole(type = c4, range = "1..*"), target =
         e16 : EndRole(type = c3))
   }, associations = Association{
      a22 : Association(source =
         e17 : EndRole(type = c1, range = "1"), target =
         e18 : EndRole(type = c2, range = "1..*")),
      a23 : Association(source =
         e19 : EndRole(type = c2, range = "1"), target =
         e20 : EndRole(type = c3, range = "0..*")),
      a24 : Association(source =
         e21 : EndRole(type = c4, range = "0..*"), target =
         e22 : EndRole(name = "item", type = c5, range = "1"))
   })
}
```

Figure 6: The model for the Cycle Shop in ReMoDeL syntax.

This example is chosen to include all of the UML semantic relationships (generalisation, aggregation, composition, association), none of which are named. It offers a mix of single, multiple and missing identifiers, which will be treated during data normalisation.

2.6 Student Records Example Model

Figure 7 encodes the second example UML Class Diagram from figure 4 in the *ReMoDeL* textual syntax for models [1]. This defines eight basic types (including the five standard

UML basic types, plus the three datatypes *Date, Time* and *Status*) and seven class types with one association class. The *Status* type enumerates {*home, overseas*} status.

```
model uml2 : UML {
   d1 : Diagram(name = "Student Records", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
      b7 : BasicType(name = "Time"),
      b8 : BasicType(name = "Status")
   }, classTypes = ClassType{
      c1 : ClassType(name = "Department", attributes = Attribute{
         a1 : Attribute(name = "name", type = b5),
         a2 : Attribute(name = "code", type = b5, id = true)
      }),
      c2 : ClassType(name = "Degree", attributes = Attribute{
         a3 : Attribute(name = "name", type = b5),
         a4 : Attribute(name = "code", type = b5, id = true)
      }),
      c3 : ClassType(name = "Module", attributes = Attribute{
         a5 : Attribute(name = "name", type = b5),
         a6 : Attribute(name = "code", type = b5, id = true),
         a7 : Attribute(name = "credits", type = b2)
      }),
      c4 : ClassType(name = "Session", attributes = Attribute{
         a8 : Attribute(name = "year", type = b6, id = true),
         a9 : Attribute(name = "level", type = b2)
      }),
      c5 : ClassType(name = "Student", attributes = Attribute{
         a10 : Attribute(name = "title", type = b5),
         a11 : Attribute(name = "forename", type = b5),
         a12 : Attribute(name = "surname", type = b5),
         a13 : Attribute(name = "number", type = b2, id = true),
         a14 : Attribute(name = "status", type = b8)
      }),
      c6 : ClassType(name = "UCard", attributes = Attribute{
         a15 : Attribute(name = "number", type = b2, id = true),
         a16 : Attribute(name = "expiry", type = b6)
      }),
      c7 : ClassType(name = "LabLog", attributes = Attribute{
         a17 : Attribute(name = "date", type = b6, id = true),
         a18 : Attribute(name = "enter", type = b7, id = true),
         a19 : Attribute(name = "exit", type = b7)
      })
   }, compositions = Composition{
      c8 : Composition(name = "Enrol", source =
         e1 : EndRole(type = c4, range = "1..*"), target =
         e2 : EndRole(type = c5))
   }, associations = Association{
      a20 : Association(name = "Prospectus", source =
         e3 : EndRole(type = c1, range = "1"), target =
         e4 : EndRole(type = c2, range = "1..*")),
      a21 : Association(name = "Approval", source =
         e5 : EndRole(type = c3, range = "1..*"), target =
         e6 : EndRole(type = c2, range = "1..*")),
      a22 : Association(name = "Study", source =
         e7 : EndRole(type = c4, range = "0..*"), target =
         e8 : EndRole(type = c3, range = "0..*"), type =
         c9 : ClassType(name = "Study", attributes = Attribute{
            a23 : Attribute(name = "grade", type = b2),
            a24 : Attribute(name = "resit", type = b2)
         })),
      a25 : Association(name = "Register", source =
         e9 : EndRole(type = c5, range = "0..*"), target =
```

```
         e10 : EndRole(type = c2, range = "1")),
      a26 : Association(name = "Holder", source =
         e11 : EndRole(type = c5, range = "1"), target =
         e12 : EndRole(type = c6, range = "1")),
      a27 : Association(source =
         e13 : EndRole(type = c6, range = "1"), target =
         e14 : EndRole(type = c7, range = "0..*"))
   })
}
```

Figure 7: The model for the Student Records in ReMoDeL syntax.

This example is chosen to include many different kinds of association, having different multiplicities at their ends, including one-to-one and many-to-many, which will be treated specially during data normalisation. These associations are mostly named, to show how this may be useful. One association has attributes, expressed using an association class.

2.7 UML Dependency Semantics

Throughout the transformation chain, we will seek to preserve UML dependency semantics. Particular attention must be paid to:

- *Composition* – the parts are existence-dependent on the whole, so deleting the whole must also delete the parts;
- *Aggregation* – the whole is dependent on the parts, which exist independently, so deleting the whole must leave the parts intact; parts may also be exchanged, so removing parts should not delete the whole;
- *Generalisation* – a subclass is existence-dependent on its superclass, so deleting the superclass must also delete the subclass.

3. Entity-Relationship Diagram

The Entity-Relationship Diagram, also commonly known as the Entity-Relationship Model (ERM) is an older notation for specifying information models. The original notation was due to Chen [9], in which entities, attributes and relationships are represented as differently-styled nodes linked in various ways. A slightly more compact notation, which lists attributes inside the entity icons and shows relationships as lines with end adornments was originally due to Everest [3] and became known as "Crow's Foot" notation. Variants of this were used in a number of approaches, including SSADM [5], Barker's notation [10] and Information Engineering [11].

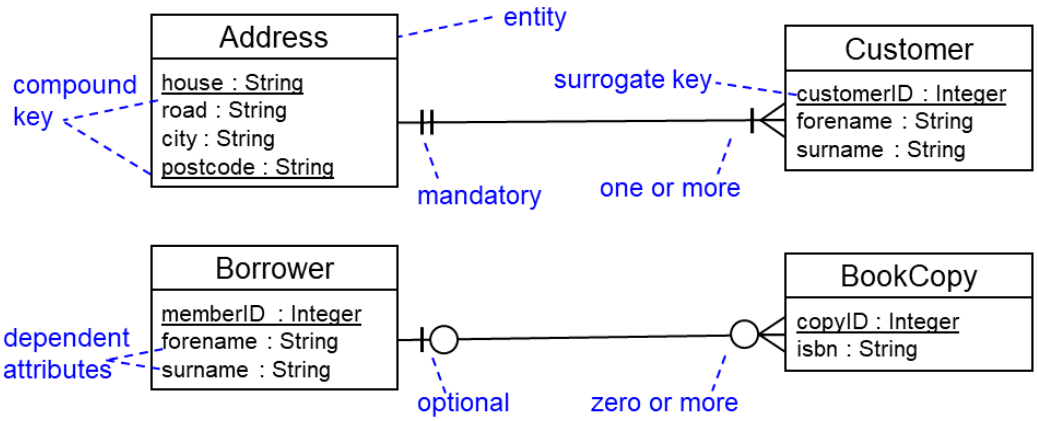

Figure 8: Crow's Foot notation for entities, attributes and relationships

Figure 8 illustrates the Crow's Foot notation, in which entities are depicted as named rectangles, each containing a list of attributes. The entities are connected by relationships whose ends are annotated to indicate how many related entities exist at each end of a relationship.

3.1 Primary and Dependent Attributes

The attributes have basic types, such as *Integer, Real* or *String*. They are assigned to an entity on the basis of *attribute dependency*, viz. the attribute's value is logically determined by the given entity instance in question.

Some attributes are underlined, indicating that they are (part of) the *primary key* for the entity in question. The primary key consists of one or more attributes whose values, taken together, uniquely identify that entity. Terminology about primary keys includes the following:

- *Candidate key* – is any attribute (or set of such) whose value uniquely identifies an entity and could be chosen to serve as the primary key (but need not be).
- *Natural key* – is an attribute naturally occurring in the domain of discourse, which uniquely identifies the entity, such as the *isbn* of a *BookTitle*.
- *Surrogate key* – is an artificially generated attribute, where no natural key exists, whose value uniquely identifies the entity, such as the *copyID* of a *BookCopy*.
- *Compound key* – is a set of attributes, whose values taken together uniquely identify the entity, such as the *house* and *postcode* of an *Address*.
- *Primary key* – is chosen from the candidate keys and may be a single natural key, a surrogate key, or a compound key, which uniquely identifies the entity.

The attributes of an entity are divided into *primary attributes* (part of the primary key) and *dependent attributes*, whose values depend on the entity, and hence on the primary key. Attribute dependency can be re-cast as: every non-key attribute must depend wholly on the primary key.

3.2 Relationship Cardinality and Optionality

Relationships express, at each end, a *cardinality* (maximum number of participating entities) and an *optionality* (minimum number of participating entities). Sometimes these are simply lumped together as the *multiplicity*. Figure 8 shows the adornments placed on the ends of a relationship. Cardinality adornments are drawn next to the entity; optionality adornments are drawn further away.

- Cardinality one – is shown as a stroke across the relationship, near the entity.
- Cardinality many – is shown as a crow's foot symbol, touching the entity.
- Optionality zero – is shown as a circle, next to the cardinality adornment.
- Optionality one – is shown as a stroke, next to the cardinality adornment.

Together, these can occur in four combinations, as shown in figure 8. We read relationships in both directions; each entity at the source-end is related to a specified number of entities at the target-end (where source, target are taken from the direction of reading).

- *Mandatory* – is at least one and at most one (exactly one). A *Customer* is related to exactly one *Address*.
- *Optional* – is at least zero and at most one. A *BookCopy* is optionally related to a *Borrower* (who loaned it).
- *Zero-Many* – is at least zero and at most many. A *Borrower* is related to zero or more *BookCopies* (which were loaned).
- *One-Many* – is at least one and at most many. An *Address* is home to one or more *Customers*.

Relationships may be characterised in a coarser way, according to the multiplicities at each end. Fixed participation (*mandatory*) is classified as *one* and variable participation (*optional, zero-many, one-many*) is classified as *many*. An alternative characterisation is possible, in which single participation (*mandatory, optional*) is classified as *one*, and multiple participation (*zero-many, one-many*) is classified as *many*.

The trade-off is: the first scheme will minimise foreign keys with null values, but requires more associative entities (see section 3.3); and the second scheme requires foreign keys with null values, but needs fewer associative entities. Proceeding with the first scheme, this gives four possibilities, three of which are shown in figure 8:

- *One-to-one* – mandatory multiplicity at each end (not shown);
- *One-to-many* – e.g. the relationship between *Address* and *Customer*;
- *Many-to-one* – e.g. the relationship between *Customer* and *Address*;
- *Many-to-many* – e.g. the relationship between *Borrower* and *BookCopy*.

One-to-one and *many-to-many* relationships are symmetrical; the others are asymmetric: *one-to-many* is the reflection of *many-to-one*. These are handled differently during data normalisation.

3.3 Strong and Weak Entities

Entities coming directly from the domain of discourse represent primary information. We refer to these as *strong entities*. Each of these must include (eventually) a primary key taken from their listed attributes. If no natural key exists, a surrogate key is created. Otherwise, we prefer to use natural keys or compound keys from the domain of discourse.

> Some advocate creating a surrogate ID for every entity, especially when mapping a UML class to an ERM entity. The argument for this is based on the idea that objects have an identity (memory address) that is independent of their attribute values. This means that naturally-unique attributes must be demoted to dependent attributes. We advocate using natural keys where these are available. The arguments for this are that this avoids bloating the data and avoids creating extra internal data dependencies. The only argument for doing otherwise is where large compound keys (three or more attributes) are used frequently in other entities as foreign keys.

Other dependent entities may be derived during the process of data analysis. These secondary entities are known as *weak entities* in contrast to the *strong entities*. There are three kinds of weak entity:

- *Associative entity* – this is an entity created to model a relationship between two entities, also known as a *linker entity*.
- *Detail entity* – this is an entity created to model a dependent part of another entity, which is known as the *master entity*.
- *Subtype entity* – this is an entity created to model an extension of another entity, which is known as the *supertype entity*.

The property of *weak entities* is that they cannot be identified solely by a local primary key, but must include the primary key of the entity (or entities) on which they depend.

> For example, a *Loan* associative entity created to model the relationship between a *Borrower* and a *BookCopy* must include the primary key of both the *Borrower* and *BookCopy* as part of its own primary key (which may include other key attributes, such as the *issueDate*).
>
> For example, the *Line* detail entity, a component part of an *Order* master entity, must include the primary key of the *Order* as part of its own primary key (which will also include a local *weak key* attribute enumerating the line *number*, which is not sufficient by itself).
>
> For example, a *StudentBorrower* subtype entity of the *Borrower* entity must include the primary key of *Borrower* as part of its own primary key (which may, but need not, include local key attributes).

This kind of information is critical to data analysis and normalisation, but is not typically captured in the popular Crow's Foot notation [3, 5, 10, 11]; although *associative* and (other) *weak* entities are distinguished in Chen's original notation [9]. Some suggestions have included:

- indicate an associative entity by marking the four corners with diagonal strokes (derived from Chen's notation for an associative entity);

- indicate an associative entity using dashed outline and a dashed connection to the many-to-many relationship (similar to the UML association class concept);

but there is no general agreement on how to depict weak entities in the Crow's Foot notation. Instead, weak entities are drawn like strong entities, but with suitable attributes added by hand to represent a copy of a remote primary key.

Our preference is to indicate weak entities visually and show the identifying relationships through which they obtain (part of) their primary key. We do not wish to replicate key attributes in weak entities at this stage, since entities may yet be merged, during normalisation, and the attributes chosen for primary keys may change as a result.

3.4 Entity-Relationship Diagram Examples

We continue with the case studies introduced in section 2.3, to show how these would look in our proposed ERM notation. Figure 9 illustrates the ERM conversion of the Cycle Shop case study, first shown as a UML class diagram in figure 3.

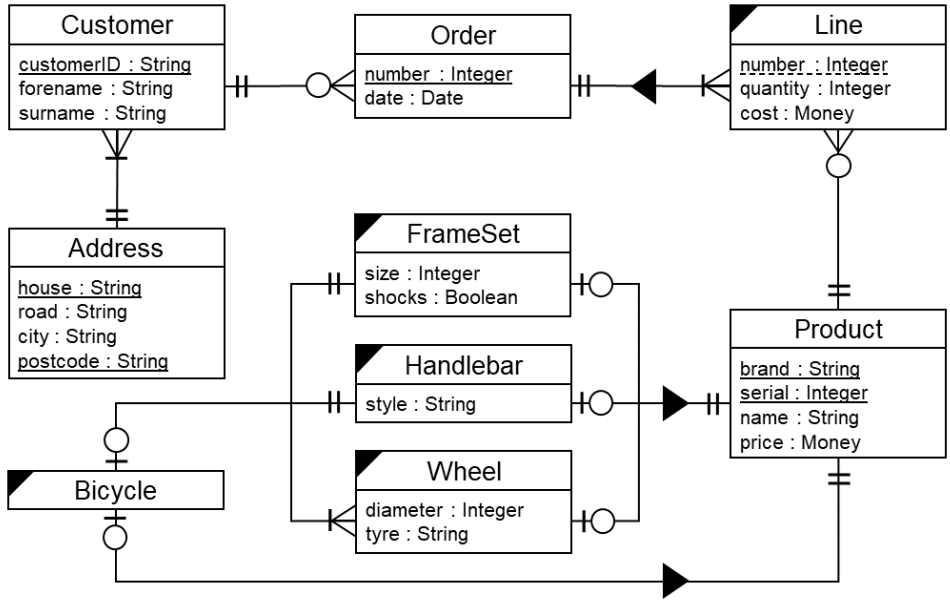

Figure 9: Entity Relationship Diagram for the Cycle Shop

All UML classes have been converted into ERM entities. All strong entities have either a simple, or a compound natural key, shown by the underlining of key attributes. Some entities have been marked as weak entities, using a black triangle in the top left corner to indicate this. A weak entity depends on another strong entity for part of its primary key. This is shown by marking some relationships as identifying relationships, using a black triangle to indicate the direction of dependency.

Both UML generalisation and UML composition have been converted into simpler ERM relationships, each relating a weak and a strong entity. The detail entity *Line* is dependent on the master entity *Order*; and the subtype entities *Bicycle*, *FrameSet*, *Handlebar* and *Wheel* are dependent on the supertype *Product*. This preserves the required existence dependency described in section 2.7. The detail entity *Line* provides a further weak key attribute *number*,

shown using dashed underlining. All key attributes of weak entities become weak key attributes. The other subtype entities did not offer any further key attributes.

Both UML generalisation and UML composition have been converted into simpler ERM relationships, each relating a weak and a strong entity. The detail entity *Line* is dependent on the master entity *Order*; and the subtype entities *Bicycle*, *FrameSet*, *Handlebar* and *Wheel* are dependent on the supertype *Product*. This preserves the required existence dependency described in section 2.7. The detail entity *Line* provides a further weak key attribute *number*, shown using dashed underlining. All key attributes of weak entities become weak key attributes. The other subtype entities did not offer any further key attributes.

All UML associations have been converted into ERM relationships with suitable multiplicity adornments at each end. These replace the ranges specified in UML. The UML aggregation, describing a *Bicycle* as an assembly of parts, has been converted into an ERM relationship with an automatic multiplicity of *optional* at the whole-end. This is to preserve the semantics of aggregation described in section 2.7. The multiplicities at the part-end are translated from the UML ranges (with the exact range of 2 becoming *one-many*). The UML generalisations have been converted into ERM relationships with an automatic multiplicity of *optional* at the subtype-end, and *one* at the supertype end (viz. capturing the fact that every *Bicycle* is always a *Product*; but any *Product* may, or may not, be a *Bicycle*).

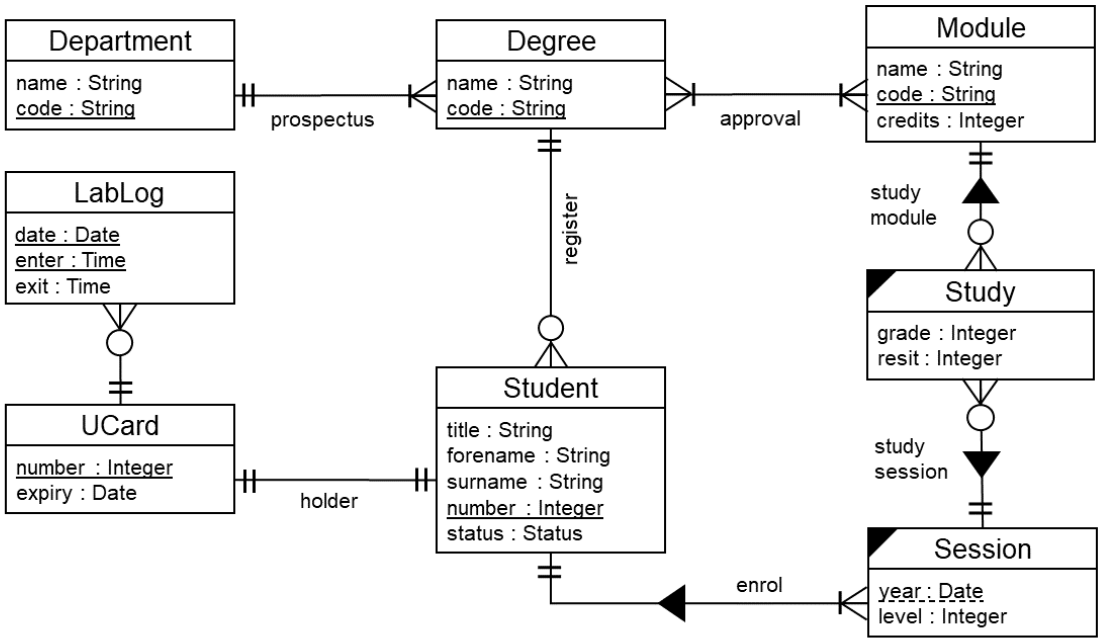

Figure 10: Entity Relationship Diagram for the Student Records

Figure 10 illustrates the ERM conversion of the Student Records case study, first shown as a UML class diagram in figure 4. All UML classes have been converted into ERM entities, including one weak detail entity *Session*. The weak associative entity *Study* is indicated using a black triangle in the top-left corner. This entity is the conversion of the UML association class, which qualified the association between *Module* and *Session* with some attributes. Its dependency on both related entities is shown by marking these relationships as identifying, using a black triangle to indicate the direction of dependency.

One point of subtlety is that *Study* depends (partly) on *Session*, which in turn depends on *Student*. Eventually, *Session* will require a compound key (including the *Student number* and the local *year*). *Study* will therefore have a compound key (*Module code, Student number, year*). However, we do not copy these attributes across, while there are still unresolved issues in data normalisation. A one-to-one relationship exists between *UCard* and *Student*, which must be eliminated in 3NF (3rd Normal Form) by merging the two entities. This could result in a different primary key being chosen for the merged result.

The UML to ERM transformation is complete once all concepts unique to UML have been converted into suitable ERM concepts; but the resulting diagrams are not yet in 3rd Normal Form. In figure 10, whereas the association class *Study* was promoted to an entity to resolve the many-to-many relationship between *Module* and *Session*, the other many-to-many relationship *approval* between *Module* and *Degree* has not yet been normalised. In figure 9, the transformation of the aggregation relationship has left an optional-to-many relationship between *Bicycle* and *Wheel*, which has not yet been normalised.

3.5 Metamodel for an Entity Relationship Diagram

Figure 11, which extends over three pages, shows a metamodel for an Entity-Relationship Diagram. Using the *ReMoDeL* textual syntax for metamodels [1], it describes *Named* things having a *name: String*. *Type* is a kind of *Named* thing that is subdivided into *BasicType* and *Entity*. The *Typed* concept, a kind of *Named* thing, refers to a *type: Type*. Its derived concepts include *Attribute* (whose *type* is specialised as a *BasicType*) and *EndRole* (whose *type* is specialised as an *Entity*).

```
metamodel ERM {
   concept Named {
      attribute name : String
   }
   concept Type inherit Named {
   }
   concept BasicType inherit Type {
   }
   concept Entity inherit Type {
      attribute linker : Boolean
      attribute detail : Boolean
      attribute subtype : Boolean
      component attributes : Attribute{}
      operation identifiers : Attribute{} {
         attributes.select(attr | attr.id)
      }
      operation dependents : Attribute{} {
         attributes.reject(attr | attr.id)
      }
      operation weak : Boolean {
         linker or detail or subtype
      }
      operation weight : Integer {
         attributes.size
      }
   }
   concept Typed inherit Named {
      reference type : Type
   }
   concept Attribute inherit Typed {
      reference type : BasicType
      attribute id : Boolean
      operation surrogate : Boolean {
         name.endsWith("ID")
      }
```

```
}
concept EndRole inherit Typed {
   reference type : Entity
   attribute optional : Boolean
   attribute multiple : Boolean
   operation isOne : Boolean {
      not (optional or multiple)
   }
   operation isMany : Boolean {
      multiple or optional
   }
   operation isZeroOne : Boolean {
      optional and (not multiple)
   }
   operation isZeroMany : Boolean {
      optional and multiple
   }
   operation isOneMany : Boolean {
      (not optional) and multiple
   }
   operation getName : String {
      if name /= "" then name
         else type.name.asName
   }
}
concept Relationship inherit Named {
   attribute id : Boolean
   attribute kindOf : Boolean
   attribute partOf : Boolean
   attribute madeOf : Boolean
   component source : EndRole
   component target : EndRole
   operation isOneToOne : Boolean {
      source.isOne and target.isOne
   }
   operation isOneToMany : Boolean {
      source.isOne and target.isMany
   }
   operation isManyToOne : Boolean {
      source.isMany and target.isOne
   }
   operation isManyToMany : Boolean {
      source.isMany and target.isMany
   }
   operation majorType : Entity {
      if source.type.weight < target.type.weight
         then target.type
         else source.type
   }
    operation minorType : Entity {
      if source.type.weight < target.type.weight
         then source.type
         else target.type
   }
   operation refersTo(entity : Entity) : Boolean {
      source.type = entity or target.type = entity
   }
   operation getName : String {
      if name /= "" then name
         else if kindOf then source.type.name.concat("KindOf")
            .concat(target.type.name)
         else if partOf then source.type.name.concat("PartOf")
            .concat(target.type.name)
         else if madeOf then source.type.name.concat("MadeOf")
            .concat(target.type.name)
         else source.type.name.concat("To")
            .concat(target.type.name)
   }
```

```
    }
    concept Diagram inherit Named {
       component basicTypes : BasicType{}
       component entities : Entity{}
       component relationships : Relationship{}
       operation strongEntities : Entity{} {
          entities.reject(entity | entity.weak)
       }
       operation weakEntities : Entity{} {
          entities.select(entity | entity.weak)
       }
       operation oneToOne : Relationship{} {
          relationships.select(rel | rel.isOneToOne)
       }
       operation manyToMany : Relationship{} {
          relationships.select(rel | rel.isManyToMany)
       }
       operation oneToMany : Relationship{} {
          relationships.select(rel | rel.isOneToMany)
       }
       operation manyToOne : Relationship{} {
          relationships.select(rel | rel.isManyToOne)
       }
    }
}
```

Figure 11: A metamodel for the Entity Relationship Diagram

An *Entity* is *weak* if it is specified as a *linker* (associative), a *detail*, or a *subtype* entity; and is *strong* otherwise. The *attributes* of an Entity can be filtered to find either *identifiers* or *dependents*. They are *surrogate* if their name ends with "ID". The *weight* of an *Entity* is the size of its attribute-set (a heuristic used later in normalisation).

An *EndRole* encodes directly whether it is *multiple* or *optional*, and from this, operations derive whether it has *one*, *many, zeroOne, zeroMany* or *oneMany* multiplicity. A *Relationship* contains *source* and *target EndRoles* and a *Boolean* attribute *id* specifies whether it is identifying. A *Relationship* also records whether it was derived from a specific UML semantic relationship: *kindOf* (subtype), *partOf* (detail) or *madeOf* (aggregate). A *Relationship* derives from its *EndRoles* whether it is *oneToOne, oneToMany, manyToOne* or *manyToMany*. It is possible to select the *majorType* or *minorType* (the *Entity* with greater or lesser *weight*), and to determine whether the *Relationship* refers to a given *Entity*.

Finally, a *Diagram* allows selection of its *BasicTypes*, *Entities* and *Relationships*, and filtered subsets of strong or weak *Entities*, and filtered subsets of the four kinds of *Relationship*. Many concepts are named, and default names are generated for *EndRole* and *Relationship*, even if these were not supplied.

3.6 The Cycle Shop Example Model

Figure 12 encodes the Entity Relationship Model from figure 9 in the *ReMoDeL* textual syntax for models. This model was translated automatically from the equivalent UML model by a transformation, to be presented below in section 4.

All named concepts have received an explicit name, as a result of being translated from UML concepts which supplied synthesised names where no explicit name was given. The basic types have been mapped to equivalent types, and the *ClassType* concepts have been mapped to *Entity* concepts, some of which are weak and marked as *detail* or *subtype* entities.

```
model erm1 : ERM {
   d1 : Diagram(name = "Cycle Shop", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
      b7 : BasicType(name = "Money")
   }, entities = Entity{
      e1 : Entity(name = "Address", attributes = Attribute{
         a1 : Attribute(name = "house", type = b5, id = true),
         a2 : Attribute(name = "postcode", type = b5, id = true),
         a3 : Attribute(name = "road", type = b5),
         a4 : Attribute(name = "city", type = b5)
      }),
      e2 : Entity(name = "Customer", attributes = Attribute{
         a5 : Attribute(name = "forename", type = b5),
         a6 : Attribute(name = "surname", type = b5)
      }),
      e3 : Entity(name = "Order", attributes = Attribute{
         a7 : Attribute(name = "number", type = b2, id = true),
         a8 : Attribute(name = "date", type = b6)
      }),
      e4 : Entity(name = "Line", detail = true, attributes = Attribute{
         a9 : Attribute(name = "number", type = b2, id = true),
         a10 : Attribute(name = "quantity", type = b2),
         a11 : Attribute(name = "cost", type = b7)
      }),
      e5 : Entity(name = "Product", attributes = Attribute{
         a12 : Attribute(name = "brand", type = b5, id = true),
         a13 : Attribute(name = "serial", type = b2, id = true),
         a14 : Attribute(name = "name", type = b5),
         a15 : Attribute(name = "price", type = b7)
      }),
      e6 : Entity(name = "Bicycle", subtype = true),
      e7 : Entity(name = "FrameSet", subtype = true,
      attributes = Attribute{
         a16 : Attribute(name = "size", type = b2),
         a17 : Attribute(name = "shocks", type = b1)
      }),
      e8 : Entity(name = "Handlebar", subtype = true,
      attributes = Attribute{
         a18 : Attribute(name = "style", type = b5)
      }),
      e9 : Entity(name = "Wheel", subtype = true, attributes = Attribute{
         a19 : Attribute(name = "diameter", type = b2),
         a20 : Attribute(name = "tyre", type = b5)
      })
   }, relationships = Relationship{
      r1 : Relationship(name = "BicycleKindOfProduct", id = true, kindOf = true,
         source = e10 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e11 : EndRole(name = "product", type = e5)
      ),
      r2 : Relationship(name = "FrameSetKindOfProduct", id = true, kindOf = true,
         source = e12 : EndRole(name = "frameSet", type = e7, optional = true),
         target = e13 : EndRole(name = "product", type = e5)
      ),
      r3 : Relationship(name = "HandlebarKindOfProduct", id = true, kindOf = true,
         source = e14 : EndRole(name = "handlebar", type = e8, optional = true),
         target = e15 : EndRole(name = "product", type = e5)
      ),
      r4 : Relationship(name = "WheelKindOfProduct", id = true, kindOf = true,
         source = e16 : EndRole(name = "wheel", type = e9, optional = true),
         target = e17 : EndRole(name = "product", type = e5)
      ),
```

```
      r5 : Relationship(name = "LinePartOfOrder", id = true, partOf = true,
         source = e18 : EndRole(name = "line", type = e4, multiple = true),
         target = e19 : EndRole(name = "order", type = e3)
      ),
      r6 : Relationship(name = "BicycleMadeOfFrameSet", madeOf = true,
         source = e20 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e21 : EndRole(name = "frameSet", type = e7)
      ),
      r7 : Relationship(name = "BicycleMadeOfHandlebar", madeOf = true,
         source = e22 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e23 : EndRole(name = "handlebar", type = e8)
      ),
      r8 : Relationship(name = "BicycleMadeOfWheel", madeOf = true,
         source = e24 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e25 : EndRole(name = "wheel", type = e9, multiple = true)
      ),
      r9 : Relationship(name = "AddressToCustomer",
         source = e26 : EndRole(name = "address", type = e1),
         target = e27 : EndRole(name = "customer", type = e2, multiple = true)
      ),
      r10 : Relationship(name = "CustomerToOrder",
         source = e28 : EndRole(name = "customer", type = e2),
         target = e29 : EndRole(name = "order", type = e3, optional = true,
            multiple = true)
      ),
      r11 : Relationship(name = "LineToProduct",
         source = e30 : EndRole(name = "line", type = e4, optional = true,
            multiple = true),
         target = e31 : EndRole(name = "item", type = e5)
      )
   })
}
```

Figure 12: the Entity Relationship Model for the Cycle Shop

Certain attributes and relationships are marked with *id*, to indicate that they are identifying. Most entities have natural identifiers; however, *Customer* does not. Since this model has not yet been normalised, a surrogate key has not yet been synthesised for it. Each identifying relationship is used to link a dependent weak entity with its associated strong entity.

Certain relationships are marked as being *kindOf, partOf* or *madeOf* relationships. This is also reflected in the names (synthesised in the UML model, and assigned explicitly in this model). This provides a kind of traceability back to the UML, but also helps to determine whether cascading deletion is required in the database.

3.7 The Student Records Example Model

Figure 13 encodes the Entity Relationship Model from figure 10 in the *ReMoDeL* textual syntax for models. This model was also translated automatically from the equivalent UML model by a transformation, to be presented below in section 4. The treatment of entities, attributes and relationships is similar to the previous case study. One difference is that many relationships were explicitly named in this case study, so these names were preserved in the translation from UML to ERM.

```
model erm2 : ERM {
   d1 : Diagram(name = "Student Records", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
```

```
   b7 : BasicType(name = "Time"),
   b8 : BasicType(name = "Status")
}, entities = Entity{
   e1 : Entity(name = "Department", attributes = Attribute{
      a1 : Attribute(name = "code", type = b5, id = true),
      a2 : Attribute(name = "name", type = b5)
   }),
   e2 : Entity(name = "Degree", attributes = Attribute{
      a3 : Attribute(name = "code", type = b5, id = true),
      a4 : Attribute(name = "name", type = b5)
   }),
   e3 : Entity(name = "Module", attributes = Attribute{
      a5 : Attribute(name = "code", type = b5, id = true),
      a6 : Attribute(name = "name", type = b5),
      a7 : Attribute(name = "credits", type = b2)
   }),
   e4 : Entity(name = "Session", detail = true, attributes = Attribute{
      a8 : Attribute(name = "year", type = b6, id = true),
      a9 : Attribute(name = "level", type = b2)
   }),
   e5 : Entity(name = "Student", attributes = Attribute{
      a10 : Attribute(name = "number", type = b2, id = true),
      a11 : Attribute(name = "title", type = b5),
      a12 : Attribute(name = "forename", type = b5),
      a13 : Attribute(name = "surname", type = b5),
      a14 : Attribute(name = "status", type = b8)
   }),
   e6 : Entity(name = "UCard", attributes = Attribute{
      a15 : Attribute(name = "number", type = b2, id = true),
      a16 : Attribute(name = "expiry", type = b6)
   }),
   e7 : Entity(name = "LabLog", attributes = Attribute{
      a17 : Attribute(name = "date", type = b6, id = true),
      a18 : Attribute(name = "enter", type = b7, id = true),
      a19 : Attribute(name = "exit", type = b7)
   }),
   e8 : Entity(name = "Study", linker = true, attributes = Attribute{
      a20 : Attribute(name = "grade", type = b2),
      a21 : Attribute(name = "resit", type = b2)
   })
}, relationships = Relationship{
   r1 : Relationship(name = "Enrol", id = true, partOf = true, source =
      e9 : EndRole(name = "session", type = e4, multiple = true), target =
      e10 : EndRole(name = "student", type = e5)
   ),
   r2 : Relationship(name = "StudyToSession", id = true, source =
      e11 : EndRole(name = "study", type = e8, optional = true,
         multiple = true), target =
      e12 : EndRole(name = "session", type = e4)
   ),
   r3 : Relationship(name = "StudyToModule", id = true, source =
      e13 : EndRole(name = "study", type = e8, optional = true,
         multiple = true), target =
      e14 : EndRole(name = "module", type = e3)
   ),
   r4 : Relationship(name = "Prospectus", source =
      e15 : EndRole(name = "department", type = e1), target =
      e16 : EndRole(name = "degree", type = e2, multiple = true)
   ),
   r5 : Relationship(name = "Approval", source =
      e17 : EndRole(name = "module", type = e3, multiple = true), target =
      e18 : EndRole(name = "degree", type = e2, multiple = true)
   ),
   r6 : Relationship(name = "Register", source =
      e19 : EndRole(name = "student", type = e5, optional = true,
         multiple = true), target =
      e20 : EndRole(name = "degree", type = e2)
   ),
```

```
      r7 : Relationship(name = "Holder", source =
         e21 : EndRole(name = "student", type = e5), target =
         e22 : EndRole(name = "uCard", type = e6)
      ),
      r8 : Relationship(name = "UCardToLabLog", source =
         e23 : EndRole(name = "uCard", type = e6), target =
         e24 : EndRole(name = "labLog", type = e7, optional = true,
            multiple = true)
      )
   })
}
```

Figure 13: the Entity Relationship Model for Student Records

The UML association class *Study* has been promoted to a weak entity of the same name, marked as a *linker*, and storing the attributes of that association. As part of this, the many-to-many association between *Module* and *Session* has also been converted into a pair of many-to-one relationships, respectively linking the promoted *Study* to *Module*, and to *Session*. The multiplicities at the ends of these relationships have been suitably translated, so that each *Study* instance relates to exactly one *Module* and one *Session*.

Other relationships have so far not been normalised. These include the many-to-many *Approval* relationship between *Degree* and *Module*; and the one-to-one *Holder* relationship between *Student* and *UCard*. The normalisation of the ERM will be handled by a later transformation.

4. UML to ERM Transformation

The transformation from UML to ERM must perform a number of mappings from the UML metamodel to the ERM metamodel. This kind of transformation is *exogeneous*, also described as a *translation*. We consider separately the mapping of types, the mapping of relationships and the mapping of attributes and identifiers.

4.1 Mapping of Types

Every UML basic type must be mapped to a similar basic type in the ERM. We assume that the basic types include the predefined UML basic types (*Boolean, Integer, Real, String, Natural*); and UML data types that may be declared, such as *Date, Money, Time*; and also, UML enumerated types, such as *Status* (in figure 10).

Every UML class must be mapped to an ERM entity, which is either strong or weak, depending on the UML semantic relationships in which it participates. In particular, the following map to some kind of weak entity:

- *UML association class* – is always mapped to a *linker entity*, since it describes attributes of an association, and is promoted to an associative entity.
- *UML subclass* – is always mapped to a *subtype entity*, since it describes additional attributes added to those of a superclass. A subclass is any class related by generalisation to another class.
- *UML detail class* – is always mapped to a *detail entity*, which is existence dependent on a master entity. A detail class is any class related by composition to another class.

All other classes map to strong entities. The aggregation relationship does not affect strong/weak entity decisions.

4.2 Mapping of Relationships

All UML semantic relationships must be mapped to the simpler ERM relationship. The multiplicities at the ends of each relationship are either derived from the UML end role ranges, or they are dictated by the kind of UML semantic relationship. In particular:

- *UML generalisation* – is always mapped to an *optional-to-one* ERM relationship, where the subtype is at the optional end. Every subtype instance has a corresponding supertype, whereas every supertype instance may or may not be related to the given subtype in question. This relationship is *identifying*.
- *UML aggregation* – is always mapped to an *optional-to-one*, or *optional-to-many* relationship, where the aggregate class is at the optional end. Every aggregate is an optional assembly of its parts, which may exist in isolation.
- *UML composition* – is always mapped to a *many-to-one*, or *one-to-one relationship*, where the parts can only depend on one whole. The indivisible parts cannot exist without the whole. This relationship is *identifying*.

UML associations may either be qualified (having an association class), or unqualified (without any association class). They are transformed in different ways:

- UML unqualified association – is mapped directly to an ERM relationship with similar multiplicities (derived from ranges). At this stage, all combinations of multiplicity may be expected.

- UML qualified association - is split into two distinct ERM relationships connecting on the source and target sides to the linker entity that was mapped from an association class. Each of these relationships is *identifying*.

The rule for splitting associations must map the old source and target multiplicities carefully:

- The linker-to-source relationship has the old target multiplicity on the linker-side and a multiplicity of exactly one on the source-side.
- The linker-to-target relationship has the old source multiplicity on the linker-side and a multiplicity of exactly one on the target-side.

This reflects the fact that whereas the source mapped to M target instances, it now maps to M linker instances; and whereas the target mapped to N source instances, it now maps to N linker instances. Every linker instance maps to exactly one instance of each related entity.

4.3 Mapping of Attributes and Identifiers

The attributes of each UML class are mapped to attributes of the corresponding ERM entities. UML attributes that were marked as *{id}* identifiers are mapped to *identifying attributes* (with underlined names) in ERM; or to *weak identifiers* (with dashed underlining), if their owning entity is a weak entity. The basic types of the UML attributes are mapped to corresponding basic types in ERM.

If a strong, or detail entity has no identifying attribute, then a surrogate identifier must be synthesised. The rule for this is to create an ERM attribute, whose name is formed by concatenating the name-case version of the entity's name with the string "ID", and whose type is found by mapping the UML basic *Integer* type. For example, a surrogate identifier for the *Customer* entity will be *customerID: Integer*. Other weak entities (*linker, subtype*) do not need surrogate identifiers (but may have identifying attributes).

Weak entities must eventually have one or more *identifying* relationships. That is, they will be identified in part by the entity at the target of the relationship. Later, this will trigger the copying of identifying attributes from the target entity back to the (weak) source entity.

- A subtype entity has one identifying relationship pointing to the supertype entity;
- A detail entity has one identifying relationship pointing to the master entity;
- A linker entity has two identifying relationships, each pointing to one of the linked entities.

We delay copying identifying attributes until after the transformation to an existence dependency graph, when all relationships have been converted into references.

4.4 Alternative Mappings for Generalisation

UML generalisations may be treated in more than one way, when converting a UML class diagram to an ERM diagram. These have different advantages and disadvantages further down the line for database implementation. The possible treatments are:

- Collapse all classes related by generalisation into a single monolithic class (the so-called "fat superclass" approach).
- Copy all abstract superclass attributes into each of the concrete subclasses (the so-called "disjoint subclass" approach).

- Map all classes to distinct entities, and map all generalisations to optional-to-one relationships (the so-called "structure preserving" approach).

For example, a *Person* class (with *forename, surname*) has two subclasses *Student* (with a unique *registration*) and *Lecturer* (with a unique *employeeID*). If any represented person is either a lecturer, or a student, but not both, then the disjoint subclass approach may be used. If the domain allows someone who can be both a student and a lecturer, viz. a teaching assistant (TA), then one of the other approaches must be used. The fat superclass approach will only normalise to 2NF (because *Student* attributes depend on *registration*; *Lecturer* attributes depend on *employeeID*; and these eventually depend transitively on a surrogate *PersonID*). The structure-preserving approach will create related instances of each entity for a TA.

The "fat superclass" approach may be taken where generalisation is only used to indicate small variations of a principal concrete class type. The superclass must have a primary key. The merging of subclass attributes into the superclass results in an entity, some of whose fields will be null in each instance. However, the gain is fewer database join operations required to relate super- and subclass instances. If subclasses have their own key attributes, the merged result only satisfies 2NF (transitive dependencies remain).

The "disjoint subclass" approach is used where generalisation is used to share an abstract class's attributes with several concrete subclasses, and all instances belong exclusively either to one, or other subclass. The subclasses may have distinct primary keys. The cost is in duplicated superclass attributes, and having to replace each relationship to the superclass by separate relationships to each subclass. The benefit is fewer database join operations to relate super- and subclass instances.

The "structure preserving" approach is the most general transformation, converting each super- and subclass into distinct entities. The superclass may also be concrete (instantiable), and the subclasses may also be overlapping (viz. conceptually related instances may exist). The cost is in more database joins, when relating super- and subclass instances. We have adopted this approach.

4.5 The ReMoDeL Transformation UML to ERM

The ReMoDeL transformation for converting a UML class diagram to an ERM Diagram is shown over the next few pages as figure 14. The transformation is called *UmlToErm* and belongs to the transformation group *UmlDB* (UML and databases).

```
transform UmlToErm : UmlDB {
   metamodel source : UML
   metamodel target : ERM

   mapping classToErmDiagram(diagram : UML_Diagram) : ERM_Diagram {
      create ERM_Diagram(name := diagram.name,
         basicTypes := umlTypesToBasicTypes(diagram),
         entities := umlTypesToEntities(diagram),
         relationships := umlArrowsToRelationships(diagram)
            .union(umlAssocsToRelationships(diagram))
      )
   }
   mapping umlTypesToBasicTypes(diagram : UML_Diagram) : ERM_BasicType{} {
      diagram.basicTypes.collect(type : UML_BasicType | basicToBasicType(type))
   }
   mapping umlTypesToEntities(diagram : UML_Diagram) : ERM_Entity{} {
```

```
   create ERM_Entity{}()
      .union(diagram.classTypes.collect(type : UML_ClassType |
         classTypeToEntity(type, diagram)))
      .union(diagram.associations.select(assoc : UML_Association |
         assoc.type /= null).collect(assoc : UML_Association |
            assocClassToEntity(assoc, diagram)))
}
mapping umlArrowsToRelationships(diagram : UML_Diagram) : ERM_Relationship{} {
   create ERM_Relationship{}()
      .union(diagram.generalisations.collect(gen : UML_Generalisation |
         genToRelationship(gen, diagram)))
      .union(diagram.compositions.collect(comp : UML_Composition |
         compToRelationship(comp, diagram)))
      .union(diagram.aggregations.collect(aggr : UML_Aggregation |
         aggrToRelationship(aggr, diagram)))
}
mapping umlAssocsToRelationships(diagram : UML_Diagram) : ERM_Relationship{} {
   create ERM_Relationship{}()
      .union(diagram.associations.select(assoc : UML_Association |
         assoc.type /= null).collate(assoc : UML_Association |
            assocToSplitRelationships(assoc, diagram)))
      .union(diagram.associations.select(assoc : UML_Association |
         assoc.type = null).collect(assoc : UML_Association |
            assocToRelationship(assoc, diagram)))
}

mapping basicToBasicType(type : UML_BasicType) : ERM_BasicType {
   create ERM_BasicType(name := type.name)
}
mapping classTypeToEntity(type : UML_ClassType,
      diagram : UML_Diagram) : ERM_Entity {
   create ERM_Entity(name := type.name,
      detail := diagram.compositions.exists(comp : UML_Composition |
         comp.source.type = type),
      subtype := diagram.generalisations.exists(gen : UML_Generalisation |
         gen.source.type = type),
      attributes := if type.identifiers.isEmpty and not
         diagram.generalisations.exists(gen : UML_Generalisation |
            gen.source.type = type)
         then classTypeToSurrogate(type, diagram).asSet
            .union(classTypeToAttributes(type))
         else classTypeToAttributes(type)
   )
}
mapping assocClassToEntity(assoc : UML_Association,
      diagram : UML_Diagram) : ERM_Entity {
   create ERM_Entity(name := assoc.name,
      linker := true,
      attributes := classTypeToAttributes(assoc.type)
   )
}
mapping classTypeToAttributes(type : UML_ClassType) : ERM_Attribute{} {
   create ERM_Attribute{}()
   .union(type.identifiers.collect(attr | attribToAttribute(attr)))
   .union(type.dependents.collect(attr | attribToAttribute(attr)))
}
mapping classTypeToSurrogate(type : UML_ClassType,
      diagram : UML_Diagram) : ERM_Attribute {
   create ERM_Attribute(
      name := type.name.asName.concat("ID"),
      type := basicToBasicType(diagram.basicTypes.detect(basic : UML_BasicType |
         basic.name = "Integer")),
      id := true
   )
}
mapping attribToAttribute(attrib : UML_Attribute) : ERM_Attribute {
   create ERM_Attribute(
      name := attrib.name,
```

```
      type := basicToBasicType(attrib.type),
      id := attrib.id
   )
}
mapping assocToRelationship(assoc : UML_Association,
      diagram : UML_Diagram) : ERM_Relationship {
   create ERM_Relationship(name := assoc.name,
      source := umlRoleToErmRole(assoc.source, diagram),
      target := umlRoleToErmRole(assoc.target, diagram)
   )
}
mapping assocToSplitRelationships(assoc : UML_Association,
      diagram : UML_Diagram) : ERM_Relationship{} {
   create ERM_Relationship{}()
   .with(assocSourceToRelationship(assoc, diagram))
   .with(assocTargetToRelationship(assoc, diagram))
}
mapping assocSourceToRelationship(assoc : UML_Association,
      diagram : UML_Diagram) : ERM_Relationship {
   create ERM_Relationship(
      name := assoc.name.concat("To").concat(assoc.source.type.name),
      id := true,
      source := create ERM_EndRole(
         name := assoc.name.asName,
         type := assocClassToEntity(assoc, diagram),
         optional := assoc.target.isOptional or assoc.target.isZeroMany,
         multiple := assoc.target.isZeroMany or assoc.target.isOneMany
      ),
      target := create ERM_EndRole(
         name := assoc.source.name,
         type := classTypeToEntity(assoc.source.type, diagram)
      )
   )
}
mapping assocTargetToRelationship(assoc : UML_Association,
      diagram : UML_Diagram) : ERM_Relationship {
   create ERM_Relationship(
      name := assoc.name.concat("To").concat(assoc.target.type.name),
      id := true,
      source := create ERM_EndRole(
         name := assoc.name.asName,
         type := assocClassToEntity(assoc, diagram),
         optional := assoc.source.isOptional or assoc.source.isZeroMany,
         multiple := assoc.source.isZeroMany or assoc.source.isOneMany
      ),
      target := create ERM_EndRole(
         name := assoc.target.name,
         type := classTypeToEntity(assoc.target.type, diagram)
      )
   )
}
mapping umlRoleToErmRole(role : UML_EndRole,
      diagram : UML_Diagram) : ERM_EndRole {
   create ERM_EndRole(
      name := role.name,
      type := classTypeToEntity(role.type, diagram),
      optional := role.isOptional,
      multiple := role.isMultiple
   )
}
mapping aggrToRelationship(aggr : UML_Aggregation,
      diagram : UML_Diagram) : ERM_Relationship {
   create ERM_Relationship(
      name := aggr.name,
      madeOf := true,
      source := create ERM_EndRole(
         name := aggr.target.name,
         type := classTypeToEntity(aggr.target.type, diagram),
```

```
            optional := true,
            multiple := aggr.target.isMultiple
         ),
         target := umlRoleToErmRole(aggr.source, diagram)
      )
   }
   mapping compToRelationship(aggr : UML_Composition,
         diagram : UML_Diagram) : ERM_Relationship {
      create ERM_Relationship(
         name := comp.name,
         id := true,
         partOf := true,
         source := umlRoleToErmRole(comp.source, diagram),
         target := umlRoleToErmRole(comp.target, diagram)
      )
   }
   mapping genToRelationship(gen : UML_Generalisation,
         diagram : UML_Diagram) : ERM_Relationship {
      create ERM_Relationship(
         name := gen.name,
         id := true,
         kindOf := true,
         source := create ERM_EndRole(
            name := gen.source.name,
            type := classTypeToEntity(gen.source.type, diagram),
            optional := true
         ),
         target := create ERM_EndRole(
            name := gen.target.name,
            type := classTypeToEntity(gen.target.type, diagram)
         )
      )
   }
}
```

Figure 14: the UmlToErm model transformation

The breakdown of this transformation is quite long, consisting of 18 separate mapping rules, but may be summarised:

- To map the UML class diagram to an ERM diagram, you create a diagram with the same name, map the UML basic types to corresponding ERM basic types; then map all the UML class types and association classes to ERM entities; then map all the UML special semantic relationships (generalisation, composition, aggregation) and UML associations to ERM relationships.
- To map a UML class type to an entity, you create an entity with the same name, map the UML class attributes to ERM attributes, preserving any identifiers and adding a surrogate ID if needed. The entity is a *detail*, if there exists a UML composition with the class at its source. The entity is a *subtype*, if there exists a UML generalisation with the class at its source. To map a UML association class to an entity, you create a *linker* entity with the name of the promoted association class, map all the UML attributes to ERM attributes, preserving any identifiers.
- To map all UML semantic relationships to ERM relationships, you map all UML generalisations, map all UML compositions, and map all UML aggregations to corresponding relationships, also mapping their UML end-roles to ERM end-roles. The rule for mapping generalisations ensures the relationship is identifying, *kind-of* and optional-to-one. The rule for mapping compositions ensures the relationship is identifying and *part-of*. The rule for mapping aggregations ensures the relationship is *made-of* and optional at the whole-end.

- To map all UML associations to ERM relationships, you map all unqualified UML associations (without any association class) to equivalent ERM relationships; but you map all qualified UML associations (having an association class) to a pair of ERM relationships, targeting the mapped linker entity and respectively the mapped source, or target entity. You map all UML end-roles to ERM end-roles in each case.
- To map UML source and target end-roles to corresponding ERM source and target end-roles, you create ERM end-roles that are similarly named and have multiplicities derived from the UML end-role ranges, and refer to entities that were mapped from the UML classes.

One aspect worth highlighting is the number of times that some of the rules are invoked (in an idempotent fashion [1]), as part of other rules. The rule for mapping a UML class type is invoked when creating the ERM entity, but also when mapping the types of UML end-roles to corresponding ERM entity types. Similarly, the rule for mapping a UML basic type is invoked when creating the ERM basic type, but also when mapping the types of UML attributes to ERM types. This is handled efficiently.

Another aspect worth recalling is that mapping rules map one principal source object to a target object. However, many of the transformation rules in figure 14 have more than one source argument; and the second argument is usually the source diagram. This is required, since the entire source model is indexed within the diagram, and some rules need to access related source elements not directly reachable from the source object that is the principal subject of the rule.

A third aspect worth mentioning is that the transformation work may be shared out in different ways between the transformation rules, and the operations provided by the source metamodel. In this example, the UML end roles provide useful operations that derive the *{optional, multiple}* values to be stored in the ERM end roles. We could instead have provided an extra layer of mapping rules to convert UML ranges to *{multiple, optional}* flags. In this case, we preferred the first approach, since this also supported writing characterising operations for UML associations.

4.6 UML to ERM Examples

We have already given examples of this transformation in use. Section 2.3 first introduced a couple of UML examples, in the figures 4 (*Cycle Shop*) and 5 (*Student Records*). Section 3.4 introduced some equivalent ERM examples, in the figures 9 (*Cycle Shop*) and 10 (*Student Records*).

Section 2.5 describes how figure 4, the *Cycle Shop* example, is encoded in the *ReMoDeL* syntax for models, listed in figure 6. Executing the *UmlToErm* transformation on this source model creates the target model listed in figure 12, in section 3.6. This may be visualised as the ERM diagram shown as figure 9 in section 3.4.

Similarly, section 2.6 describes how figure 5, the *Student Records* example, is encoded in the *ReMoDeL* syntax for models, listed in figure 7. Executing the *UmlToErm* transformation on this source model creates the target model listed in figure 13, in section 3.7. This may be visualised as the ERM diagram shown as figure 10 in section 3.4.

5. ERM To Normal Transformation

The ERM to normal ERM transformation performs several structural modifications to the ERM metamodel, resulting in a model that is at least in 3NF (Third Normal Form). The result could be in 4NF or higher; but to guarantee this is impossible without detailed domain knowledge (e.g. to identify multivalued dependencies between sets of attributes). So, the transformation is based solely on structural information. This kind of transformation is *endogenous*, also described as a *normalisation*. We consider separately the mapping of types, the mapping of relationships and the mapping of attributes and identifiers.

5.1 Mapping of Types

ERM basic types are left unchanged by normalisation. The normal set of entities may both grow, if certain relationships give rise to new linker entities, and shrink, if sets of entities are merged into one entity. Other entities are transferred unchanged.

ERM entities must be merged, if they are related by a one-to-one relationship. This is a more difficult problem than is initially supposed, since in general, sets of entities could be related by chains of such relationships, requiring all of them to be merged. Part of the problem involves deciding what to call the merged entity; and another part involves deciding which attributes to treat as identifiers. We take the following approach.

- Entities may be major or minor concepts in the domain, depending on their weight. As a heuristic, we judge weight by a count of the attributes in the entity. A major entity has more weight than a minor entity. We determine that minor entities should be merged into the major entity; and the result takes the name of the major entity.
- When merging minor entities, their identifying attributes are now subordinate to the identifier of the major entity with which they were merged. Accordingly, all identifiers in merged entities must be demoted to dependent attributes.

A subsidiary issue is that when attributes are transferred from one entity to another, there is the risk that name-clashes could occur. Therefore, a unique renaming scheme must be adopted to ensure that local and merged attributes do not clash on their names.

5.2 Mapping of Relationships

ERM relationships are normalised by ensuring that the only kind of relationship remaining in the model is a many-to-one relationship. Other kinds of relationship are specially treated, but all relationships must be transformed:

- *One-to-one*: these relationships are eliminated, after merging the related entities.
- *Many-to-one*: the direction of dependency is good; but the source and target entities must be normalised, in case of mergers.
- *One-to-many*: these relationships are reversed, so that the direction of dependency runs from the many-side to the one-side; and the source and target entities are also normalised.
- *Many-to-many*: these relationships are split into two many-to-one relationships, each relating a linker entity (on the many-side) to one of the original source and target entities (on the one-side), which are also normalised.

Normalising a relationship has a transitive obligation to normalise the end-roles and the entities referred to by these end-roles. This may have the effect of transferring a relationship end onto a different entity type, after mergers are considered. Otherwise, the special properties of a relationship must be preserved, where appropriate.

- *Many-to-one*: after normalising, the *identifying* property is preserved; and the *part-of, kind-of,* or *made-of* property is preserved;
- *One-to-many*: after reversing, the relationship cannot be identifying; and it has no other semantic properties (since it cannot have been created from any UML semantic relationship, but only as an association).
- *Many-to-many*: after splitting, if the original relationship was *made-of* (created from a UML aggregation), each of the new split relationships must be *part-of* (strong dependency of the linker on the related entities); and both split relationships are also *identifying*.

We also provide default names for any unnamed relationships, generating conventional names in the style: *SourceToTarget*, where *Source, Target* are the names of the related entities.

5.3 Mapping of Attributes and Identifiers

When mapping attributes to normalised attributes, we re-order attributes such that identifying attributes precede dependent attributes. Otherwise, the existing attributes of each ERM entity are unchanged by normalisation, having the same names and types. However, all strong entities must have at least one identifying attribute (this is also true for *detail* entities – see below).

If a strong entity has no identifying attribute, then a surrogate identifier must be synthesised. The rule for this is to create an attribute of the ERM Integer type, whose name is formed by concatenating the name-case version of the entity's name with the string "ID". For example, a surrogate identifier for the *Customer* entity may be named *customerID*.

Where attributes are transferred from a minor entity to a major entity during a merger, these attributes must be renamed. The rule for this is to create a new ERM attribute, whose name is formed by concatenating the name-case version of the minor entity's name with the type-case version of the attribute's name. The attribute's type is preserved. Furthermore, if the merged attribute was identifying, it is demoted to a dependent attribute, since the major entity already has identifying attributes.

> For example, if the minor entity *UCard* has an attribute *number: Integer*, when *UCard* is merged with the major entity *Student*, this is mapped to a new attribute *uCardNumber: Integer*. This avoids any possible name clash with other attributes called *number* in *Student*. Furthermore, whereas this attribute was formerly an identifier in *UCard*, it is demoted to a dependent attribute of *Student*.

Weak entities have identifying relationships which are preserved (see section 5.2) and may also have further local identifiers. These are normalised in the following way:

- *Linker entities* – since these are partly, or wholly identifiable through their identifying relationships, and may have additional local identifiers (only if carried over from a UML association class), their attributes are unchanged by normalisation and any local identifiers are preserved.

- *Detail entities* – since these are only partly identified through a *part-of* identifying relationship, these must have at least one local weak identifier. If none exists, then a surrogate identifier is added to the list of attributes, which otherwise are preserved unchanged.
- *Subtype entities* – since these must already be wholly identifiable through a *kind-of* relationship, there is no advantage to be gained by creating more complex compound keys for subtypes including local identifiers. Therefore, any local identifiers are demoted to dependent attributes and other attributes are preserved unchanged.

We preserve identifying relationships after normalisation, and delay copying identifying attributes from the referenced entities until after the transformation of the ERM to an existence dependency graph.

5.4 The ReMoDeL ERM to Normal Transformation

The ReMoDeL transformation for converting an ERM diagram to normal form is shown over the next few pages as figure 15. The transformation is called *ErmToNorm* and belongs to the transformation group *UmlDB* (UML and databases).

```
transform ErmToNorm : UmlDB {
   metamodel source : ERM
   metamodel target : ERM

   mapping ermToNormalDiagram(diagram : ERM_Diagram) : ERM_Diagram {
      create Erm_Diagram(name := diagram.name,
         basicTypes := diagram.basicTypes,
         entities := ermToNormalEntities(diagram),
         relationships := ermToNormalRelationships(diagram)
      )
   }
   mapping ermToNormalEntities(diagram : ERM_Diagram) : ERM_Entity{} {
      create ERM_Entity{}()
      .union(diagram.entities.collect(entity |
         entityToNormalEntity(entity, diagram)))
      .union(diagram.manyToMany.collect(rel | relToLinkerEntity(rel, diagram)))
   }
   mapping ermToNormalRelationships(diagram : ERM_Diagram) : ERM_Relationship{} {
      create ERM_Relationship{}()
      .union(diagram.manyToOne.collect(rel |
         relToForwardRelationship(rel, diagram)))
      .union(diagram.oneToMany.collect(rel |
         relToReverseRelationship(rel, diagram)))
      .union(diagram.manyToMany.collate(rel |
         relToSplitRelationships(rel, diagram)))
   }
   mapping entityToNormalEntity(entity : ERM_Entity,
         diagram : ERM_Diagram) : ERM_Entity {
      if diagram.oneToOne.exists(rel | rel.refersTo(entity))
         then entityToMergedEntity(entity, diagram)
         else entityToIdentifiedEntity(entity, diagram)
   }
   mapping entityToIdentifiedEntity(entity : ERM_Entity,
         diagram : ERM_Diagram) : ERM_Entity {
      if entity.linker
         then entity
         else create ERM_Entity(name := entity.name,
               subtype := entity.subtype,
               detail := entity.detail,
               attributes := if entity.subtype
                  then entityToDependentAttributes(entity, diagram)
                  else entityToNormalAttributes(entity, diagram)
```

```
     )
}
mapping entityToMergedEntity(entity : ERM_Entity,
      diagram : ERM_Diagram) : ERM_Entity {
   mergeEntities(
      findClosure(diagram.oneToOne.asList,
         diagram.oneToOne.select(rel | rel.refersTo(entity)).asList,
         create ERM_Relationship{}()),
      diagram
   )
}
mapping mergeEntities(entities : ERM_Entity{},
      diagram : ERM_Diagram) : ERM_Entity {
   create ERM_Entity(name := majorEntity(entities).name,
      linker := majorEntity(entities).linker,
      subtype := majorEntity(entities).subtype,
      detail := majorEntity(entities).detail,
      attributes := entityToNormalAttributes(majorEntity(entities), diagram)
         .union(entities.without(majorEntity(entities))
            .collate(entity : ERM_Entity | entityToMergedAttributes(entity)))
   )
}
mapping majorEntity(entities : ERM_Entity{}) : ERM_Entity {
   entities.combine(firstEntity, secondEntity : ERM_Entity |
      if firstEntity.weight < secondEntity.weight
         then secondEntity
         else firstEntity
   )
}
function findClosure(oneToOne : ERM_Relationship[],
      open : ERM_Relationship[], closed : ERM_Relationship{}) : ERM_Entity{} {
   if open.isEmpty
      then closed.collate(rel |
         rel.source.type.asSet.with(rel.target.type))
      else if closed.has(open.first)
         then findClosure(oneToOne, open.rest, closed)
         else findClosure(oneToOne, open.rest.append(
            oneToOne.select(rel | rel.refersTo(open.first.source.type) or
               rel.refersTo(open.first.target.type)) ),
            closed.with(open.first)
         )
}
mapping relToLinkerEntity(rel : ERM_Relationship,
      diagram : ERM_Diagram) : ERM_Entity {
   create ERM_Entity(name := rel.name, linker := true)
}
mapping entityToNormalAttributes(entity: ERM_Entity,
      diagram : ERM_Diagram) : ERM_Attribute{} {
   if entity.linker or entity.subtype
      then entity.attributes
      else entityToIdentifiers(entity, diagram).union(entity.dependents)
}
mapping entityToDependentAttributes(entity : ERM_Entity,
      diagram : ERM_Diagram) : ERM_Attribute{} {
   entity.attributes.collect(attrib : ERM_Attribute |
      create ERM_Attribute(name := attrib.name, type := attrib.type)
   )
}
mapping entityToIdentifiers(entity: ERM_Entity,
      diagram : ERM_Diagram) : ERM_Attribute{} {
   if entity.identifiers.isEmpty
      then entityToSurrogate(entity, diagram).asSet
      else entity.identifiers
}
mapping entityToSurrogate(entity: ERM_Entity,
      diagram : ERM_Diagram) : ERM_Attribute {
   create ERM_Attribute(
      name := entity.name.asName.concat("ID"),
```

```
      type := diagram.basicTypes.detect(basic : ERM_BasicType |
         basic.name = "Integer"),
       id := true
   )
}
mapping entityToMergedAttributes(entity: ERM_Entity,
      diagram : ERM_Diagram) : ERM_Attribute{} {
   entity.attributes.collect(attrib : ERM_Attribute |
      attribToMergedAttrib(attrib, entity))
}
mapping attribToMergedAttrib(attrib : ERM_Attribute,
      entity : ERM_Entity) : ERM_Attribute {
   create ERM_Attribute(
      name := if attrib.surrogate
                then attrib.name
                else entity.name.asName.concat(attrib.name.asType),
      type := attrib.type,
   )
}
mapping relToForwardRelationship(rel : ERM_Relationship,
      diagram : ERM_Diagram) : ERM_Relationship {
   create ERM_Relationship(name := relToForwardRelName(rel, diagram),
      id := rel.id,
      kindOf := rel.kindOf,
      partOf := rel.partOf,
      madeOf := rel.madeOf,
      source := roleToNormalRole(rel.source, diagram),
      target := roleToNormalRole(rel.target, diagram)
   )
}
mapping relToForwardRelName(rel : ERM_Relationship,
      diagram : ERM_Diagram) : String {
   if not rel.name.startsWith(rel.source.type.name)
      then rel.name
      else entityToNormalEntity(rel.source.type, diagram).name.concat("To")
          .concat(entityToNormalEntity(rel.target.type, diagram).name)
}
mapping relToReverseRelationship(rel : ERM_Relationship,
      diagram : ERM_Diagram) : ERM_Relationship {
   create ERM_Relationship(name := relToReverseRelName(rel, diagram),
      source := roleToNormalRole(rel.target, diagram),
      target := roleToNormalRole(rel.source, diagram)
   )
}
mapping relToReverseRelName(rel : ERM_Relationship,
      diagram : ERM_Diagram) : String {
   if not rel.name.startsWith(rel.source.type.name)
      then rel.name
      else entityToNormalEntity(rel.target.type, diagram).name.concat("To")
         .concat(entityToNormalEntity(rel.source.type, diagram).name)
}
mapping relToSplitRelationships(rel : ERM_Relationship,
      diagram : ERM_Diagram) : ERM_Relationship{} {
   create ERM_Relationship{}()
   .with(relSourceToRelationship(rel, diagram))
   .with(relTargetToRelationship(rel, diagram))
}
mapping relSourceToRelationship(rel : ERM_Relationship,
      diagram : ERM_Diagram) : ERM_Relationship {
   create ERM_Relationship(
      name := rel.name.concat("To").concat(
         entityToNormalEntity(rel.source.type, diagram).name),
      id := true,
      partOf := rel.madeOf,
      source := create ERM_EndRole(
         name := rel.name.asName,
         type := relToLinkerEntity(rel, diagram),
         optional := rel.target.optional,
```

```
            multiple := rel.target.multiple
         ),
         target := create ERM_EndRole(
            name := roleToNormalRoleName(rel.source, diagram),
            type := entityToNormalEntity(rel.source.type, diagram)
         )
      )
   }
   mapping relTargetToRelationship(rel : ERM_Relationship,
         diagram : ERM_Diagram) : ERM_Relationship {
      create ERM_Relationship(
         name := rel.name.concat("To").concat(
            entityToNormalEntity(rel.target.type, diagram).name),
         id := true,
         partOf := rel.madeOf,
         source := create ERM_EndRole(
            name := rel.name.asName,
            type := relToLinkerEntity(rel, diagram),
            optional := rel.source.optional,
            multiple := rel.source.multiple
         ),
         target := create ERM_EndRole(
            name := roleToNormalRoleName(rel.target, diagram),
            type := entityToNormalEntity(rel.target.type, diagram)
         )
      )
   }
    mapping roleToNormalRole(role : ERM_EndRole,
         diagram : ERM_Diagram) : ERM_EndRole {
       create ERM_EndRole(
          name := roleToNormalRoleName(role, diagram),
          type := entityToNormalEntity(role.type, diagram),
          optional := role.optional,
          multiple := role.multiple
       )
    }
    mapping roleToNormalRoleName(role : ERM_EndRole,
         diagram : ERM_Diagram) : String {
       if role.name /= role.type.name.asName then role.name
          else entityToNormalEntity(role.type, diagram).name.asName
    }
}
```

Figure 15: the ErmToNorm model transformation

The breakdown of this transformation is quite long, consisting of 24 separate mapping rules and one auxiliary recursive function, but may be summarised:

- To map the ERM diagram to a normal ERM diagram, you create a diagram with the same name, transfer the ERM basic types; then map all ERM entities to normal entities and extend this set by mapping all *many-to-many* relationships to extra linker entities; and then map all the ERM relationships to normal relationships, using separate rules for *one-to-many*, *many-to-one* and *many-to-many* relationships.
- To map an ERM entity to a normal ERM entity, if the diagram contains any *one-to-one* relationships referring to this entity, then create a merged entity; otherwise create a suitably identified entity.
- To create a merged ERM entity, first find the transitive closure of *one-to-one* relationships linking this entity with any others, project out the set of related entities, find the largest major entity, then merge into this entity all the attributes of the remaining entities, demoting any of their identifiers to dependent attributes.
- To create a suitably identified ERM entity, copy the entity and adjust its identifiers. If it is a *linker*, copy it unchanged. If it is a *subtype*, demote any identifiers to dependent

attributes. If it is either a *detail*, or a strong entity, check whether it has at least one identifying attribute, and if not, create a surrogate identifier.
- When merging attributes, if the attribute is already a surrogate (its name is prefixed by the name of the owning entity), leave it unchanged; otherwise create a new attribute having the same type, whose name is prefixed by the name of its old minor entity.
- When normalising relationships, map all the end-roles to normalised end-roles, in case these refer to merged entities. Otherwise, preserve all the identifying and semantic properties of *many-to-one* relationships, reverse all *one-to-many* relationships (losing any such property) and split all *many-to-many* relationships into a pair of *identifying* relationships, each joining a linker with one of the related entities, and which must be *part-of,* if the original association was *made-of*.
- When normalising an end-role, if its old name was previously derived from the name of the related type, then create a new name that is derived from the name of the normalised, and possibly merged, entity type. The normalised end-role also refers to the mapped normal type.

Perhaps the most difficult aspect of this transformation is dealing with mergers. Not only does this result in the loss of certain entities, but it also requires redirection of every relationship that referred to one of the lost entities. For this reason, the transformation must trigger the rule *entityToNormalEntity()* for every entity referenced anywhere in the pre-normal ERM, and this relies on the idempotence of rules to return the same entity.

Merging relies on computing a transitive closure. This is obtained by following chains of *one-to-one* relationships. The function *findClosure()* is a classic breadth-first search function, which starts with an *open* list of relationships to explore, and builds a *closed* set of visited relationships. On each recursion, one relationship is removed from *open* to *closed*, and *open* is expanded by adding any *one-to-one* relationships that are joined on common entities with the removed relationship. Once *open* is exhausted, the set of related entities is projected from the *closed* set of relationships.

The *mergeEntities()* rule uses a rule *majorEntity()* that performs a reduction on all the entities in the set, returning the entity with the greatest weight. This will always yield the same major entity, no matter in what order the set of entities is presented (due to rule idempotence). So, no matter how many times *majorEntity()* is called, it always returns the same major entity (even if two entities compete for greatest weight). Therefore *mergeEntities()* always returns the same merged entity for all of the original *one-to-one* related entities.

5.5 The Normal Cycle Shop Example

Figure 16 illustrates the normalised ERM for the *Cycle Shop* case study. The only difference between this and the pre-normal ERM from figure 12 is that the *optional-to-many* relationship between *Bicycle* and *Wheel* has been transformed into a *linker* entity, with the generated name *BicycleMadeOfWheel*. Other entities and relationships have not been affected, since they were already in normal form.

Since the *Bicycle* was originally an aggregate assembly of its parts, we expect the domain to support the deletion of the *Bicycle*, without deleting its parts. Where the *Bicycle* references a single *FrameSet* or a single *HandleBar*, there is no issue in deleting a *Bicycle*. However, the *Bicycle* originally referenced multiple *Wheels*, giving rise to the linker. Deleting the *Bicycle* would cause the linker to contain a dangling reference to its *Bicycle*. For this reason, the

linker has been marked as existence dependent on the *Bicycle*, through the *part-of* annotation. This annotation is used later to generate an automatic cascading deletion in the database.

This is the only case where a *linker* entity is also marked *part-of*. In other cases, we expect the domain to support the deletion of *linker* entities (representing temporary relationships) before the deletion of the related entities is required.

```
model norm1 : ERM {
   d1 : Diagram(name = "Cycle Shop", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
      b7 : BasicType(name = "Money")
   }, entities = Entity{
      e1 : Entity(name = "Address", attributes = Attribute{
         a1 : Attribute(name = "house", type = b5, id = true),
         a2 : Attribute(name = "postcode", type = b5, id = true),
         a3 : Attribute(name = "road", type = b5),
         a4 : Attribute(name = "city", type = b5)
      }),
      e2 : Entity(name = "Customer", attributes = Attribute{
         a5 : Attribute(name = "customerID", type = b2, id = true),
         a6 : Attribute(name = "forename", type = b5),
         a7 : Attribute(name = "surname", type = b5)
      }),
      e3 : Entity(name = "Order", attributes = Attribute{
         a8 : Attribute(name = "number", type = b2, id = true),
         a9 : Attribute(name = "date", type = b6)
      }),
      e4 : Entity(name = "Line", detail = true, attributes = Attribute{
         a10 : Attribute(name = "number", type = b2, id = true),
         a11 : Attribute(name = "quantity", type = b2),
         a12 : Attribute(name = "cost", type = b7)
      }),
      e5 : Entity(name = "Product", attributes = Attribute{
         a13 : Attribute(name = "brand", type = b5, id = true),
         a14 : Attribute(name = "serial", type = b2, id = true),
         a15 : Attribute(name = "name", type = b5),
         a16 : Attribute(name = "price", type = b7)
      }),
      e6 : Entity(name = "Bicycle", subtype = true),
      e7 : Entity(name = "FrameSet", subtype = true, attributes = Attribute{
         a17 : Attribute(name = "size", type = b2),
         a18 : Attribute(name = "shocks", type = b1)
      }),
      e8 : Entity(name = "Handlebar", subtype = true, attributes = Attribute{
         a19 : Attribute(name = "style", type = b5)
      }),
      e9 : Entity(name = "Wheel", subtype = true, attributes = Attribute{
         a20 : Attribute(name = "diameter", type = b2),
         a21 : Attribute(name = "tyre", type = b5)
      }),
      e10 : Entity(name = "BicycleMadeOfWheel", linker = true)
   }, relationships = Relationship{
      r1 : Relationship(name = "BicycleToProduct", id = true, kindOf = true,
         source = e11 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e12 : EndRole(name = "product", type = e5)
      ),
      r2 : Relationship(name = "FrameSetToProduct", id = true, kindOf = true,
         source = e13 : EndRole(name = "frameSet", type = e7, optional = true),
         target = e14 : EndRole(name = "product", type = e5)
      ),
```

```
      r3 : Relationship(name = "HandlebarToProduct", id = true, kindOf = true,
         source = e15 : EndRole(name = "handlebar", type = e8, optional = true),
         target = e16 : EndRole(name = "product", type = e5)
      ),
      r4 : Relationship(name = "WheelToProduct", id = true, kindOf = true,
         source = e17 : EndRole(name = "wheel", type = e9, optional = true),
         target = e18 : EndRole(name = "product", type = e5)
      ),
      r5 : Relationship(name = "LineToOrder", id = true, partOf = true,
         source = e19 : EndRole(name = "line", type = e4, multiple = true),
         target = e20 : EndRole(name = "order", type = e3)
      ),
      r6 : Relationship(name = "BicycleToFrameSet", madeOf = true,
         source = e21 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e22 : EndRole(name = "frameSet", type = e7)
      ),
      r7 : Relationship(name = "BicycleToHandlebar", madeOf = true,
         source = e23 : EndRole(name = "bicycle", type = e6, optional = true),
         target = e24 : EndRole(name = "handlebar", type = e8)
      ),
      r8 : Relationship(name = "LineToProduct",
         source = e25 : EndRole(name = "line", type = e4, optional = true,
            multiple = true),
         target = e26 : EndRole(name = "item", type = e5)
      ),
      r9 : Relationship(name = "CustomerToAddress",
         source =  e27 : EndRole(name = "customer", type = e2, multiple = true),
         target = e28 : EndRole(name = "address", type = e1)
      ),
      r10 : Relationship(name = "OrderToCustomer",
         source = e29 : EndRole(name = "order", type = e3, optional = true,
            multiple = true),
         target = e30 : EndRole(name = "customer", type = e2)
      ),
      r11 : Relationship(name = "BicycleMadeOfWheelToBicycle", id = true,
            partOf = true,
         source = e31 : EndRole(type = e10, multiple = true),
         target = e32 : EndRole(name = "bicycle", type = e6)
      ),
      r12 : Relationship(name = "BicycleMadeOfWheelToWheel", id = true,
            partOf = true,
         source = e33 : EndRole(type = e10, optional = true),
         target = e34 : EndRole(name = "wheel", type = e9)
      )
   })
}
```

Figure 16: the normalised Cycle Shop example

5.6 The Normal Student Records Example

Figure 17 illustrates the normalised ERM for the *Student Records* case study. There are several differences between this and the pre-normal ERM from figure 13.

The *many-to-many* relationship between *Degree* and *Module* has been promoted into a *linker* entity *Approval*, with *many-to-one* relationships *ApprovalToModule* and *ApprovalToDegree* linking this to their respective entities. This was the only remaining many-to-many relationship in the model. The linker entity *Study* was derived previously from a UML association class.

The *one-to-one* relationship between *Student* and *UCard* has been removed, and the attributes of *UCard* have been merged into *Student*. The renaming rules proved useful here, since both *Student* and *UCard* have an attribute called *number*. *Student* was determined to be the major

entity during the merger. When transferred to *Student*, the old attributes of *UCard* have been renamed as: *uCardNumber, uCardExpiry*. The former identifying attribute *number* has been demoted to the dependent attribute *uCardNumber*, which no longer clashes with *Student's* identifying attribute *number*.

A consequence of merging *one-to-one* relationships is that the relationship from *UCard* to *LabLog* has been replaced by a relationship between the merged entity *Student* and *LabLog* (since *UCard* no longer exists). This is an example of transferring the types of end-roles. Furthermore, this relationship has been reversed: formerly, it was a *one-to-many* relationship called *LabLogToUCard*, and now it is a *many-to-one* relationship called *StudentToLabLog*. This is an example of normalising all remaining relationships to *many-to-one.*

```
model norm2 : ERM {
   d1 : Diagram(name = "Student Records", basicTypes = BasicType{
      b1 : BasicType(name = "Boolean"),
      b2 : BasicType(name = "Integer"),
      b3 : BasicType(name = "Natural"),
      b4 : BasicType(name = "Real"),
      b5 : BasicType(name = "String"),
      b6 : BasicType(name = "Date"),
      b7 : BasicType(name = "Time"),
      b8 : BasicType(name = "Status")
   }, entities = Entity{
      e1 : Entity(name = "Department", attributes = Attribute{
         a1 : Attribute(name = "code", type = b5, id = true),
         a2 : Attribute(name = "name", type = b5)
      }),
      e2 : Entity(name = "Degree", attributes = Attribute{
         a3 : Attribute(name = "code", type = b5, id = true),
         a4 : Attribute(name = "name", type = b5)
      }),
      e3 : Entity(name = "Module", attributes = Attribute{
         a5 : Attribute(name = "code", type = b5, id = true),
         a6 : Attribute(name = "name", type = b5),
         a7 : Attribute(name = "credits", type = b2)
      }),
      e4 : Entity(name = "Session", detail = true, attributes = Attribute{
         a8 : Attribute(name = "year", type = b6, id = true),
         a9 : Attribute(name = "level", type = b2)
      }),
      e5 : Entity(name = "Student", attributes = Attribute{
         a10 : Attribute(name = "number", type = b2, id = true),
         a11 : Attribute(name = "title", type = b5),
         a12 : Attribute(name = "forename", type = b5),
         a13 : Attribute(name = "surname", type = b5),
         a14 : Attribute(name = "status", type = b8),
         a15 : Attribute(name = "uCardNumber", type = b2),
         a16 : Attribute(name = "uCardExpiry", type = b6)
      }),
      e6 : Entity(name = "LabLog", attributes = Attribute{
         a17 : Attribute(name = "date", type = b6, id = true),
         a18 : Attribute(name = "enter", type = b7, id = true),
         a19 : Attribute(name = "exit", type = b7)
      }),
      e7 : Entity(name = "Study", linker = true, attributes = Attribute{
         a20 : Attribute(name = "grade", type = b2),
         a21 : Attribute(name = "resit", type = b2)
      }),
      e8 : Entity(name = "Approval", linker = true)
   }, relationships = Relationship{
      r1 : Relationship(name = "Enrol", id = true, partOf = true,
         source = e9 : EndRole(name = "session", type = e4, multiple = true),
         target = e10 : EndRole(name = "student", type = e5)
      ),
```

```
      r2 : Relationship(name = "StudyToSession", id = true,
         source = e11 : EndRole(name = "study", type = e7,
            optional = true, multiple = true),
         target = e12 : EndRole(name = "session", type = e4)
      ),
      r3 : Relationship(name = "StudyToModule", id = true,
         source = e13 : EndRole(name = "study", type = e7,
            optional = true, multiple = true),
         target = e14 : EndRole(name = "module", type = e3)
      ),
      r4 : Relationship(name = "Register",
         source = e15 : EndRole(name = "student", type = e5,
            optional = true, multiple = true),
         target = e16 : EndRole(name = "degree", type = e2)
      ),
      r5 : Relationship(name = "Prospectus",
         source = e17 : EndRole(name = "degree", type = e2, multiple = true),
         target = e18 : EndRole(name = "department", type = e1)
      ),
      r6 : Relationship(name = "LabLogToStudent",
         source = e19 : EndRole(name = "labLog", type = e6,
            optional = true, multiple = true),
         target = e20 : EndRole(name = "student", type = e5)
      ),
      r7 : Relationship(name = "ApprovalToModule", id = true,
         source = e21 : EndRole(name = "approval", type = e8, multiple = true),
         target = e22 : EndRole(name = "module", type = e3)
      ),
      r8 : Relationship(name = "ApprovalToDegree", id = true,
         source = e23 : EndRole(name = "approval", type = e8, multiple = true),
         target = e24 : EndRole(name = "degree", type = e2)
      )
   })
}
```

Figure 17: the normalised Student Records example

5.7 Interim Conclusion

This concludes the first two transformations, from UML to ERM and from ERM to a normal ERM. The remaining transformations, and the final code-generation stage are described in part 2 of this report [12].

6. References

[1] A J H Simons. ReMoDeL Explained (rev. 2.1): An Introduction to ReMoDeL by Example. Technical Report, 12 July (University of Sheffield, 2022).

[2] M Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. (Addison-Wesley, 2003).

[3] G Everest. "Basic data structures explained with a common example", Proc. 5th Texas Conf. Computing Systems (1976), 39-46. Chapter 4 in: *Database Management: Objectives, System Functions, and Administration* (McGraw-Hill, 1986).

[4] A Zhao. *SQL Pocket Guide. A guide to SQL usage*, 4th ed. (O'Reilly Media, 2021).

[5] E Downs, P Claire and I Coe. *Structured Systems Analysis and Design Method: Application and Context,* 2nd Ed., (Prentice Hall, 1991).

[6] M A Jackson. *Principles of Program Design*, (Academic Press, 1975).

[7] H R Myler. "Flowcharts", Chapter 2.3 in: *Fundamentals of Engineering Programming with C and Fortran*, (Cambridge University Press, 1998), 32–36.

[8] E Gamma, R Helm, R Johnson and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1994).

[9] P Chen. "The Entity-Relationship Model - Toward a Unified View of Data". ACM Transactions on Database Systems. 1 (1): 9–36.

[10] R Barker. *CASE Method: Entity Relationship Modelling*, (Addison-Wesley Professional, 1990).

[11] J Martin. *Information Engineering* (3 volumes). (Prentice Hall Inc., 1989).

[12] A J H Simons. ReMoDeL Data Refinement (rev. 1.0): Data Transformations in Remodel, Part 2. Technical Report, 31 July (University of Sheffield, 2022).