The University Of Sheffield.

ReMoDeL Data Refinement

Data Transformations in ReMoDeL, Part 2

Technical Report

Revision: 1.0

Date: 31 July 2022

Anthony J H Simons
Department of Computer Science
University of Sheffield

Contents

1. Introduction 4
   1.1 Software Engineering Models 4
   1.2 Transformation Chains 4
   1.3 Data Refinement 5
2. Existence Dependency Graph 6
   2.1 Existence Dependency 6
   2.2 Dependencies and Attributes 7
   2.3 Existence Dependency Graph Examples 7
   2.4 Metamodel for an Existence Dependency Graph 9
   2.5 Cycle Shop Example Model 11
   2.6 Student Records Example Model 12
3. SQL Database Schema 14
   3.1 Primary and Foreign Key Constraints 14
   3.2 Semantic Consistency Maintenance 15
   3.3 SQL Database Schema Examples 16
   3.4 Metamodel for an SQL Database Schema 18
   3.5 The Cycle Shop Example Model 20
   3.6 The Student Records Example Model 23
4. Normal ERM to EDG Transformation 25
   4.1 Mapping of Types 25
   4.2 Mapping of Properties 25
   4.3 The ReMoDeL Transformation Normal ERM to EDG 25
   4.4 Normal ERM to EDG Examples 27
5. EDG to SQL Transformation 28
   5.1 Mapping of Types 28
   5.2 Mapping of Properties 28
   5.3 Mapping of Identifiers 28
   5.4 The ReMoDeL EDG to SQL Transformation 29
   5.5 EDG to SQL Examples 32
6. Code Generation 33
   6.1 The Visitor Interface 33
   6.2 The MySQLWriter Implementation 34

6.3 The Generated Cycle Shop Database 37
6.4 The Generated Student Records Database 39
6.5 The Chain of Compiled Transformations 40
7. References 42

1. Introduction

This document describes the second 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 Design Pattern* [8].

This document (part 2 of 2) covers the second half of the above transformation chain, from the *Normal ER Diagram* to the *SQL Data Definition Language*. 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 latter two transformations and the code-generation step.

2. Existence Dependency Graph

The development of the Existence Dependency Graph (EDG) in software engineering is due to Snoeck and Dedene [9], who demonstrated that this viewpoint is simpler and provides greater semantic clarity than parts-wholes relationships. It is used along with an object-event table and object state machines in the software engineering method MERODE [10], which benefits from points of correspondence in these views to propagate consistency constraints across models. An EDG is an object model that is already in normal form. Whereas MERODE constructs these from first principles, we derive these after a process of traditional data normalisation.

2.1 Existence Dependency

The notion of existence dependency is simple. Entities are arranged in a dependency graph, connected by directed relationships, according to whether the entity at the tail of the arrow is existence-dependent on the entity at the head. That is, the dependent entity at the tail cannot exist, without the prior existence of the entity at the head. This also means that the lifetime of the dependent entity must be wholly contained within the lifetime(s) of the entity (or entities) at the head.

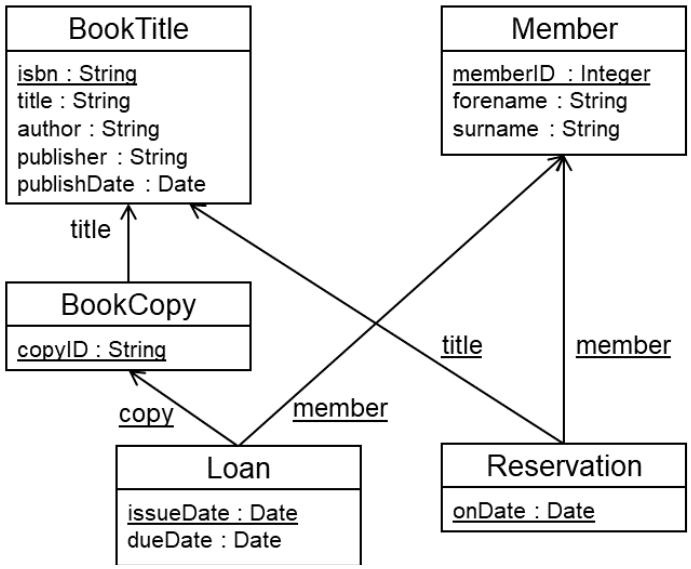

Figure 1: Existence Dependency Graph

Figure 1 illustrates the idea, depicting an EDG for the library domain, using a variant of the UML class diagram notation. The graph is a DAG (directed acyclic graph), meaning that an entity cannot depend (transitively) upon itself in a cycle, but an entity may depend on more than one other entity. Each entity is drawn as a named rectangle, containing a list of attributes. Each directed arrow represents an existence dependency.

> For example, a *BookCopy* depends on a pre-existing *BookTitle*; and a *Loan* depends on the pre-existence of both a *BookCopy* and a *Member* of the library. The lifetime of a *BookCopy* cannot exceed that of its *BookTitle*; and the lifetime of a *Loan* must be contained within the lifetimes of both the related *BookCopy* and *Member*.

2.2 Dependencies and Attributes

In our notation, the arrows flow from the dependent to the master entity. In Snoeck's notation [9], dependencies are drawn as lines with different end-annotations to denote different kinds of optionality and multiplicity. We omit this detail, partly because the head of every arrow must always have the multiplicity of *exactly one*. The tail of every arrow could have any multiplicity, but after normalisation, cannot have *exactly one* (in 3NF, the related entities would have been merged). In database design, the point of identifying multiplicity is to determine the direction of dependency, which flows from the *optional*- or *many*-side to the *one*-side. This is what the EDG makes explicit.

Each dependency arrow may be treated a reference from the source to the target entity. This is useful from the database design perspective, since it reveals into which entities a foreign key must be placed (the source) and from which entity suitable identifying attributes must be copied (the target). So, the EDG simplifies the relationships of ERM, ready for database implementation. The EDG could also be converted back into a UML Class Diagram that may more easily be mapped to and from a relational database (foreign keys map to object pointers and vice-versa). That is, the resulting class structure closely matches the table structure.

In our notation, entities contain a list of named attributes; and any identifying attributes are underlined. Our dependency arrows are also named. This name corresponds to the name of the original UML end-role on the target side. We also underline this name, if the dependency is identifying. Whereas a dependency is an indication that foreign key attributes will eventually be copied (from target to source), an identifying dependency is also an indication that these will also be part of a compound primary key (in the source). Snoeck [11] also views dependencies as short-hand for foreign keys; however, she does not distinguish those which must also serve as primary keys.

> For example, when the linker *Loan* is eventually converted into a database table, its compound primary key will consist of the attributes {*memberID, copyID, issueDate*}. The primary key of *Loan* must include at least the foreign keys {*memberID, copyID*} relating *Loan* to the entities *Member* and *BookCopy*. The inclusion of *issueDate*, a local identifying attribute, comes from domain analysis, to support identification of loans of the same book to the same borrower on different occasions.

A correspondence exists between *identifying* dependencies and *weak* entities (from ERM). In figure 1, *Loan* is related by identifying dependencies *copy, member* respectively to *Copy* and *Member* and it is also a weak associative entity. By contrast, *BookCopy* is related by a non-identifying dependency *title* to *BookTitle*, and is not a weak entity, in the sense that it is wholly identified by its local *copyID*. So, our existence dependency notation preserves the distinction between strong and weak entities.

2.3 Existence Dependency Graph Examples

We shall develop the same two case studies from part 1 of this report [11] in the rest of this document, to illustrate how the normalised ERM models are converted into EDG models. Figure 2 shows the Existence Dependency Graph for the Cycle Shop case study developed in part 1. This figure resolves all the many-to-one relationships from ERM as directed references, representing the existence dependency of the source upon the target.

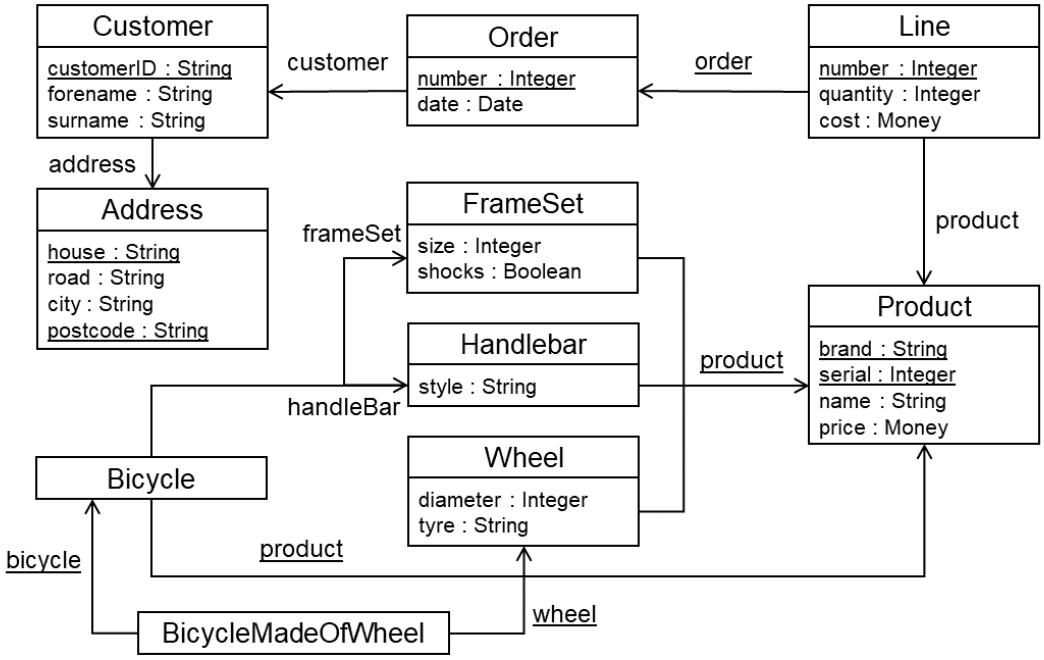

Figure 2: Existence Dependency Graph for the Cycle Shop

So, it is clear that a *Customer* will refer to an *Address*, and that an *Order* will refer to a *Customer*. The dependency of *Line* upon *Order* has an underlined name, indicating that this relationship is identifying; and this implies that *Line* is a weak entity. Earlier, in the ERM, we determined that *Line* is a detail entity, which cannot be identified solely by its local identifier *number*, but also depends on the identifier *number* of its master entity *Order*. (The clash of names is deliberate; to be resolved by a later transformation rule).

Elsewhere in the model, there are a number of weak entities that were originally derived from UML generalisation relationships. *Bicycle, FrameSet, Handlebar* and *Wheel* are all kinds of *Product* sold in the shop. These have been converted into subtype entities, related to *Product* through the identifying dependency *product*, which indicates that the entities are weak. Their primary key attributes will eventually be renamed copies of the primary key attributes of *Product* {*brand, serial*}, which also serve as the foreign key relating each subtype to the supertype *Product*.

A more interesting case is *Bicycle*, which by UML aggregation is an assembly of *FrameSet, Handlebar* and two *Wheel*s. The translation of the aggregation made *Bicycle* existence dependent upon *FrameSet* and *HandleBar*, but not upon *Wheel*. This is because there is only one each of the former, which allows the creation of a reference; however, an artificial linker *BicycleMadeOfWheel* was needed to link a *Bicycle* to more than one *Wheel*. In other respects, the *Bicycle* assembly is correctly dependent on the parts that constitute it. A *Bicycle* can be deleted without deleting any of its parts. The additional constraint is that the linker *BicycleMadeOfWheel* must also be deleted in cascading fashion.

The direction of dependency for aggregation is the reverse of that for the composition relationship between *Line* and *Order*, in which the part is existence dependent on the whole. Deleting an *Order* must result in the deletion of all related *Lines*, in a cascading fashion. This is how we determine the difference between UML aggregation and UML composition in the Existence Dependency Graph. Our solution for aggregation allows the whole to depend directly on single parts, and for these references to be made null if the part is removed or

deleted. This contrasts with Snoeck [9], who constructs associative entities to relate all parts to an aggregate whole. Snoeck's approach has some attractions, but results in entities that are related in a one-to-one fashion, which we seek to eliminate in 3NF.

Cascading deletion must also be implemented for subtype entities. Deleting a *Product* should result in the automatic deletion of the related subtype instance, whose type is one of {*Bicycle, FrameSet, HandleBar, Wheel*}. This can be handled directly in the database using foreign key constraints. Deleting a *Bicycle* should also result in the deletion of the related *Product*. This can only be achieved by explicit coding in the implementation.

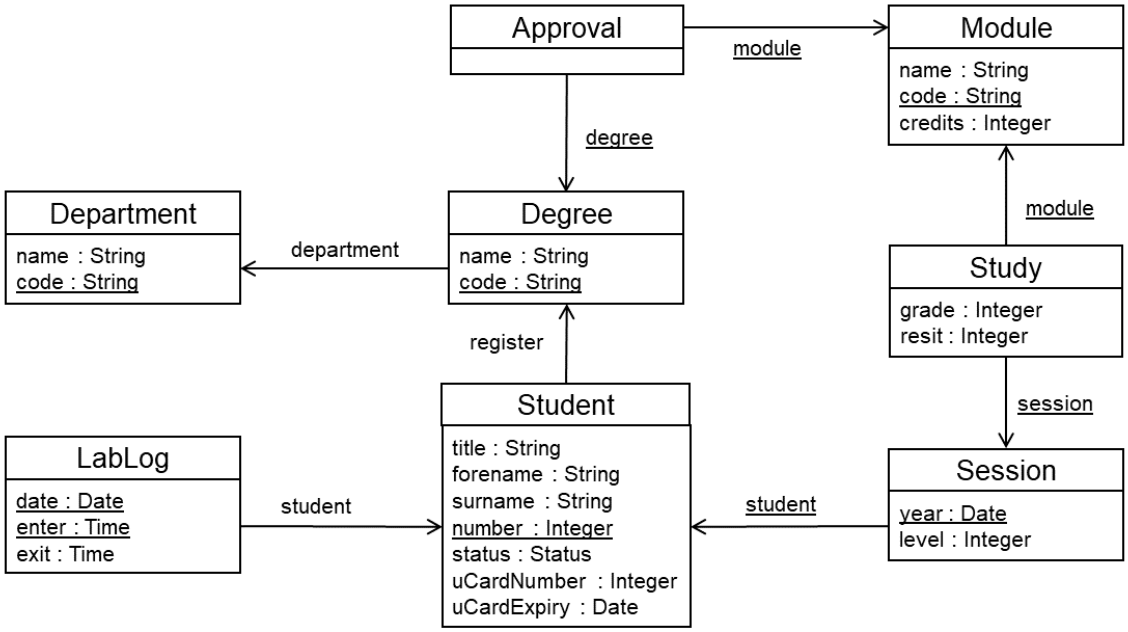

Figure 3: Existence Dependency Graph for Student Records

Figure 3 shows the Existence Dependency Graph for the Student Records system. After normalisation to 3NF+, the model contains associative entities *Approval* and *Study*, which are existence dependent on the pairs of entities that they relate. The separate *UCard* entity has been merged with *Student,* and *LabLog* is now existence dependent upon *Student* (instead of upon *UCard*).

Dependent entities *LabLog* and *Degree* are strong entities, being self-identifying; whereas the associative entities *Approval* and *Study* are weak entities (with identifying references). The *Session* entity is also weak, with an identifying reference to *Student*. This reflects how *Session* was originally a detail entity dependent on the master *Student*. There is a chain of identifying references from *Study* to *Session*, and from *Session* to *Student*. This will have implications for the creation of foreign keys that also serve as part of the primary keys for these two entities.

2.4 Metamodel for an Existence Dependency Graph

Figure 4 shows a metamodel for the Existence Dependency Graph. The main difference between this model and the earlier metamodels for UML and ERM are that all relationships are now encoded as references that are owned by an *Entity*, rather than as separate edges that refer to their source and target vertices.

```
metamodel EDG {
   concept Named {
      attribute name : String
   }
   concept Type inherit Named {
   }
   concept BasicType inherit Type {
   }
   concept Entity inherit Type {
      component properties : Property{}
      operation attributes : Attribute{} {
         properties.select(prop | prop.simple )
      }
      operation references : Reference{} {
         properties.select(prop | not prop.simple )
      }
      operation simpleIDs : Attribute{} {
         self.attributes.select(prop | prop.id)
      }
      operation complexIDs : Reference{} {
         self.references.select(prop | prop.id)
      }
   }
   concept Typed inherit Named {
      reference type : Type
   }
   concept Property inherit Typed {
      attribute id : Boolean
      operation simple : Boolean {
         false
      }
   }
   concept Attribute inherit Property {
      reference type : BasicType
      operation simple : Boolean {
         true
      }
      operation surrogate : Boolean {
         name.endsWith("ID")
      }
   }
   concept Reference inherit Property {
      reference type : Entity
      attribute kindOf : Boolean
      attribute partOf : Boolean
      attribute madeOf : Boolean
   }
   concept Diagram inherit Named {
      component basicTypes : BasicType{}
      component entities : Entity{}
   }
}
```

Figure 4: A metamodel for the Existence Dependency Graph

Using the *ReMoDeL* textual syntax for metamodels [1], figure 4 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*. A *Property* is a kind of *Typed* concept which may be identifying. From this are derived *Attribute* (whose *type* is specialised as a *BasicType*), and *Reference* (whose *type* is specialised as an *Entity*).

The generalisation of *Property* in this metamodel allows an *Entity* to consist of a collection of *Properties*; and then to provide operations that filter this to yield sets of *Attributes* or *References*, using the ability of *select* to type-cast to a more specific type. *Property* also

specifies via the *Boolean id* attribute whether the *Property* (*Attribute*, or *Reference*) is identifying.

Apart from this, the metamodel is very simple. *Reference* also specifies whether it was derived from earlier generalisation, aggregation or composition in UML, using the Boolean *kindOf, madeOf* or *partOf* attributes. The top-level enclosing *Diagram* concept simply contains sets of *BasicTypes* and *Entities*.

2.5 Cycle Shop Example Model

Figure 5 encodes the Cycle Shop case study from figure 2 in the *ReMoDeL* textual syntax for models [1]. This model was translated automatically from a normalised ERM model, using a transformation to be presented in section 4. It contains a single *Diagram* consisting of sets of *BasicTypes* and *Entities*, where each *Entity* consists of a set of heterogeneous *Properties*, some of which are identifying, and which are either *Attributes* or *References*.

```
model edg1 : EDG {
   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", properties = Property{
         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", properties = Property{
         a5 : Attribute(name = "customerID", type = b2, id = true),
         a6 : Attribute(name = "forename", type = b5),
         a7 : Attribute(name = "surname", type = b5),
         r1 : Reference(name = "address", type = e1)
      }),
      e3 : Entity(name = "Order", properties = Property{
         a8 : Attribute(name = "number", type = b2, id = true),
         a9 : Attribute(name = "date", type = b6),
         r2 : Reference(name = "customer", type = e2)
      }),
      e4 : Entity(name = "Product", properties = Property{
         a10 : Attribute(name = "brand", type = b5, id = true),
         a11 : Attribute(name = "serial", type = b2, id = true),
         a12 : Attribute(name = "name", type = b5),
         a13 : Attribute(name = "price", type = b7)
      }),
      e5 : Entity(name = "FrameSet", properties = Property{
         a14 : Attribute(name = "size", type = b2),
         a15 : Attribute(name = "shocks", type = b1),
         r3 : Reference(name = "product", type = e4, id = true, kindOf = true)
      }),
      e6 : Entity(name = "Handlebar", properties = Property{
         a16 : Attribute(name = "style", type = b5),
         r4 : Reference(name = "product", type = e4, id = true, kindOf = true)
      }),
      e7 : Entity(name = "Wheel", properties = Property{
         a17 : Attribute(name = "diameter", type = b2),
         a18 : Attribute(name = "tyre", type = b5),
         r5 : Reference(name = "product", type = e4, id = true, kindOf = true)
      }),
```

```
      e8 : Entity(name = "Line", properties = Property{
         a19 : Attribute(name = "number", type = b2, id = true),
         a20 : Attribute(name = "quantity", type = b2),
         a21 : Attribute(name = "cost", type = b7),
         r6 : Reference(name = "order", type = e3, id = true, partOf = true),
         r7 : Reference(name = "item", type = e4)
      }),
      e9 : Entity(name = "Bicycle", properties = Property{
         r8 : Reference(name = "product", type = e4, id = true, kindOf = true),
         r9 : Reference(name = "frameSet", type = e5, madeOf = true),
         r10 : Reference(name = "handlebar", type = e6, madeOf = true)
      }),
      e10 : Entity(name = "BicycleMadeOfWheel", properties = Property{
         r11 : Reference(name = "bicycle", type = e9, id = true, partOf = true),
         r12 : Reference(name = "wheel", type = e7, id = true, partOf = true)
      })
   })
}
```

Figure 5: The EDG model for the Cycle Shop in ReMoDeL syntax.

Some of the *References* are marked as encoding kind-of, made-of or part-of relationships. This shows how all of the UML semantic relationships (generalisation, aggregation, composition, association) have been appropriately tracked by the EDG. All *References* are named after the original end-role names.

2.6 Student Records Example Model

Figure 6 encodes the second case study, which is for the Student Records system shown in figure 3, in the *ReMoDeL* textual syntax for models [1]. This shows the merger of *UCard* attributes into *Student*, and the creation of the two associative entities for *Study* and *Approval*, which each have a pair of identifying references to their related entities. The detail entity *Session* has an identifying reference to *Student*. Other references are non-identifying, so will eventually yield foreign keys, rather than primary and foreign keys.

```
model edg2 : EDG {
   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", properties = Property{
         a1 : Attribute(name = "code", type = b5, id = true),
         a2 : Attribute(name = "name", type = b5)
      }),
      e2 : Entity(name = "Degree", properties = Property{
         a3 : Attribute(name = "code", type = b5, id = true),
         a4 : Attribute(name = "name", type = b5),
         r1 : Reference(name = "department", type = e1)
      }),
      e3 : Entity(name = "Module", properties = Property{
         a5 : Attribute(name = "code", type = b5, id = true),
         a6 : Attribute(name = "name", type = b5),
         a7 : Attribute(name = "credits", type = b2)
      }),
      e4 : Entity(name = "Student", properties = Property{
         a8 : Attribute(name = "number", type = b2, id = true),
         a9 : Attribute(name = "title", type = b5),
         a10 : Attribute(name = "forename", type = b5),
```

```
      a11 : Attribute(name = "surname", type = b5),
      a12 : Attribute(name = "status", type = b8),
      a13 : Attribute(name = "uCardNumber", type = b2),
      a14 : Attribute(name = "uCardExpiry", type = b6),
      r2 : Reference(name = "degree", type = e2)
   }),
   e5 : Entity(name = "LabLog", properties = Property{
      a15 : Attribute(name = "date", type = b6, id = true),
      a16 : Attribute(name = "enter", type = b7, id = true),
      a17 : Attribute(name = "exit", type = b7),
      r3 : Reference(name = "student", type = e4)
   }),
   e6 : Entity(name = "Approval", properties = Property{
      r4 : Reference(name = "module", type = e3, id = true),
      r5 : Reference(name = "degree", type = e2, id = true)
   }),
   e7 : Entity(name = "Session", properties = Property{
      a18 : Attribute(name = "year", type = b6, id = true),
      a19 : Attribute(name = "level", type = b2),
      r6 : Reference(name = "student", type = e4, id = true, partOf = true)
   }),
   e8 : Entity(name = "Study", properties = Property{
      a20 : Attribute(name = "grade", type = b2),
      a21 : Attribute(name = "resit", type = b2),
      r7 : Reference(name = "session", type = e7, id = true),
      r8 : Reference(name = "module", type = e3, id = true)
   })
})
}}
```

Figure 6: The EDG model for the Student Records in ReMoDeL syntax.

One aspect that is not immediately obvious in the two models in figures 5 and 6 is that the *Entities* have been sorted in order of existence dependency, such that later *Entities* depend on earlier *Entities*. This does not affect the semantics of the model, but it does improve the presentation. All *Reference* objects refer to their *Entity* objects via a *ReMoDeL* object identifier. On first occurrence, each object identifier is expanded to a full definition of the object in question. So, the ordering ensures that every *Entity* is encountered and defined before any *Reference* refers to it. This avoids inline expansion of *Entity* definitions inside *Reference* types.

3. SQL Database Schema

The SQL Database Schema is a novel diagrammatic representation of the SQL data definition language [4]. It is intended as the model from which a simple code generator could generate the statements in SQL to define a complete database, with suitable primary keys and foreign keys, with their related constraints. We base the SQL Database Schema on the Existence Dependency Graph, with additional details of the names and types of foreign key columns.

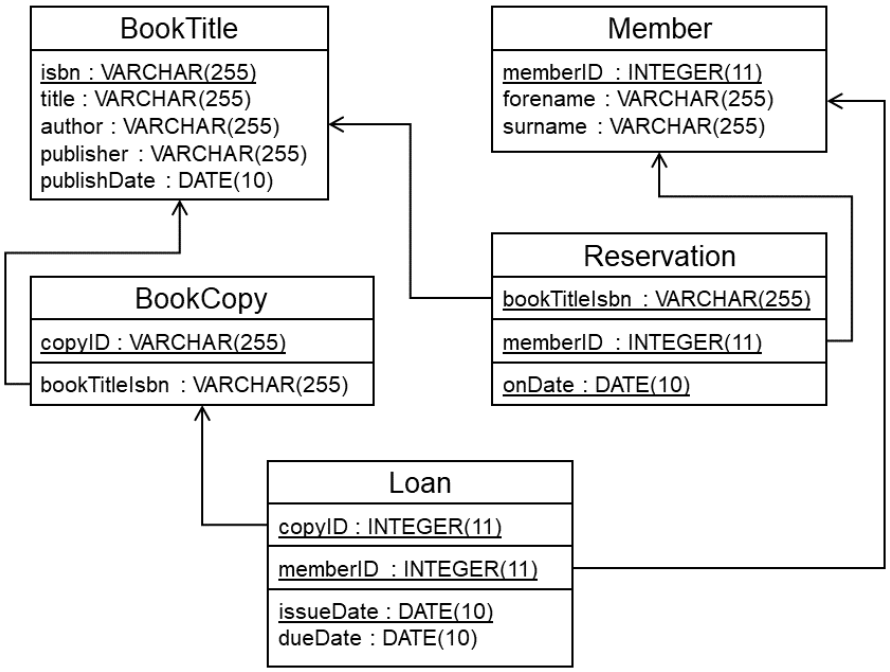

Figure 7: SQL Database Schema

Figure 7 illustrates the idea for this notation. The tables of the database are depicted as named rectangles, containing lists of columns (attributes), which now have basic types commonly found in databases, along with a maximum field width. Underlined columns will form part of the primary key for the table. Foreign key columns are grouped in separate partitions, from which references flow to target the table which is referenced. The foreign key columns so grouped must correspond to a similar set of primary key columns found in the referenced table. Any given column may be part of a primary key, part of a foreign key, or both. Only one partition exists, with no outgoing references. This contains the local columns of the table which are not part of any foreign key.

3.1 Primary and Foreign Key Constraints

The primary key consists of one or more columns whose values, taken together, uniquely identify that entity, which corresponds to a row in a database table. A foreign key consists of a copied set of columns that correspond to a primary key in another table. Terminology about primary and foreign keys includes the following:

- *Natural key* – is a column 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 column, where no natural key exists, whose value uniquely identifies the entity, such as the *copyID* of a *BookCopy*.

- *Compound key* – is a set of columns, whose values taken together uniquely identify the entity, such as the *house* and *postcode* of an *Address*.
- *Primary key* – may be a single natural key, a surrogate key, or a compound key, which uniquely identifies the entity (or table row).
- *Foreign key* – is a (possibly renamed) copy of a primary key in another table, whose values correspond to the values of the primary key.

A primary key is subject to two constraints (entity integrity):

- No column used as part of the primary key may have a *null* value.
- No two distinct table rows may have the same value for the primary key.

The first constraint can be enforced by code generation. The second constraint can only be applied during usage of the generated database, so is out of scope here. (If violated, the duplicated table rows with the same key would be merged).

A foreign key is subject to a constraint (referential integrity):

- If a foreign key's columns have non-null values, there must be a corresponding primary key, whose columns have the same values, in the referenced table.

Databases seek to prevent deleting a row in a master table, while there are still foreign keys in dependent tables which refer to it. If this were allowed, the foreign keys would become invalid, dangling references to data that no longer exists. By default, databases prevent this by rolling back transactions that would break referential integrity.

Normally, the rows in such dependent tables represent entities that are existence dependent on the row in the master table. It is normally expected that their lifetime is shorter than that of the master row. So, the behaviour of blocking early deletion of the master row is, in general, the right constraint to enforce. But there are other possibilities.

3.2 Semantic Consistency Maintenance

There are occasions when it is desirable for a database to take a different action. In a composition relationship, many detail entities depend for their existence on a master entity. We may wish to delete the whole cluster of instances at the same time. When deleting a master table row on which one or more detail table rows depend, then these should also be deleted in a cascading fashion.

Similarly, in a generalisation relationship, a subtype entity depends for its existence on a supertype entity. We may wish to delete the chain of subtype instances at the same time. When deleting a supertype table row on which some subtype table row depends, the subtype row should also be deleted in a cascading fashion (and this would proceed transitively, in more elaborate generalisation hierarchies).

However, in an aggregation relationship, in which a whole depends upon a collection of parts, we will not want to delete the parts along with the whole, nor the whole if one or more parts were deleted. We may wish to disassemble an aggregate whole and reassemble it with other parts, which exist independently. This means that it must be possible to set foreign keys to *null* in a table row denoting an aggregate entity.

These considerations give rise to the following database constraints:

- *Generalisation* – in a subtype entity, the foreign key referring to the supertype entity must have the constraint "on delete cascade" to ensure that the subtype is deleted when the supertype is deleted.
- *Composition* – in a detail entity, the foreign key referring to the master entity must have the constraint "on delete cascade" to ensure that the detail is deleted when the master is deleted.
- *Aggregation* – in an aggregate entity, the foreign keys referring to the parts must have the constraint "on delete set null" to maintain the referential integrity of the aggregate entity.
- *Association (1)* – in an associative entity, no explicit constraint is placed on foreign keys, which is equivalent to the "on delete restrict" option, which ensures that no master entity can be deleted while the associative entity references it.
- *Association (2)* – in an associative entity linking an aggregate whole to a part, the constraint "on delete cascade" is placed on both foreign keys, to ensure that if either the part or the whole is deleted, the associative entity will also be deleted.

The difference between the two rules for associations comes from different expectations about lifetimes. A regular associative entity is expected to have a shorter lifetime than its masters, whereas this is not the case for an aggregation, in which either the part or whole could be detached, resulting in a need to delete the associative entity.

3.3 SQL Database Schema Examples

We continue with the case studies introduced in section 2.3, to show how these would look in our proposed SQL Database Schema notation. Figure 8 illustrates the SQL conversion of the Cycle Shop case study, shown earlier as the Existence Dependency Graph in figure 2.

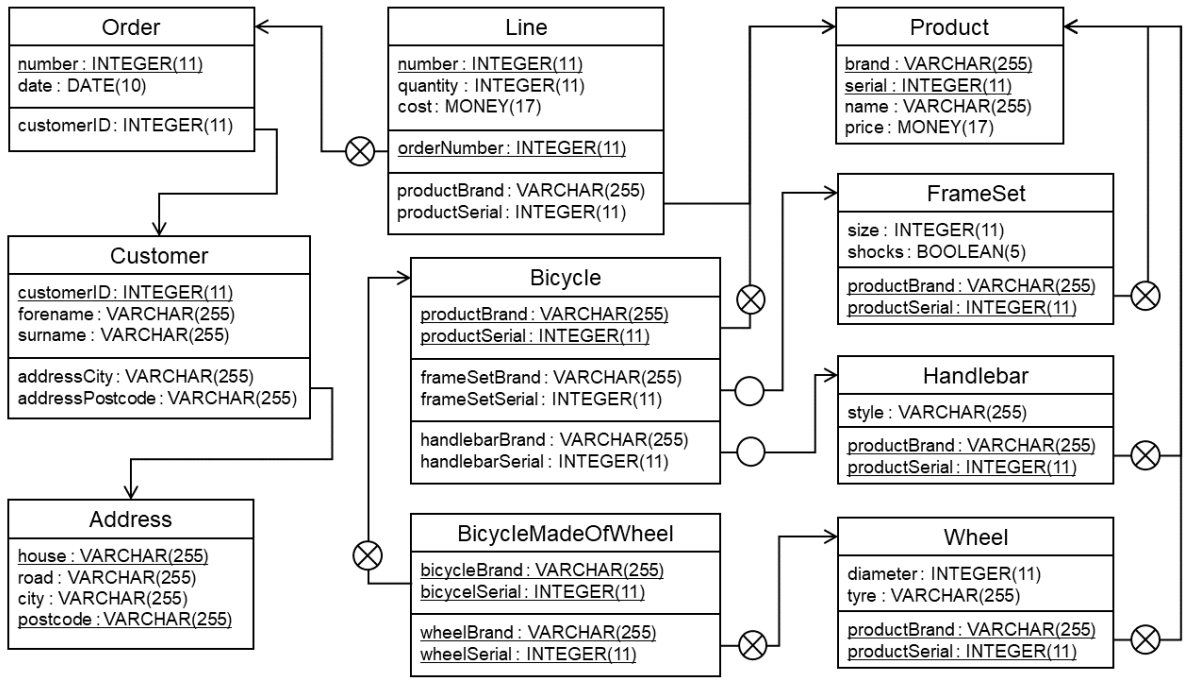

Figure 8: SQL Database Schema for the Cycle Shop

All EDG entities have been converted into SQL tables, with all primary and foreign keys now explicit. All simple types are now database types. All strong entities have either a simple, or

a compound key, shown by the underlining of key columns. Some primary keys are natural, such as the compound key in *Address*. Other primary keys are surrogate, such as the simple key in *Customer*. All weak entities have primary keys that depend in part on the primary key of a referenced entity. These key columns are both part of the foreign key referring to the other entity, and part of the primary key of the weak entity.

In particular, the detail entity *Line* has a compound primary key consisting of the foreign key *orderNumber* and the local key column *number*. The subtype entity *Bicycle* has a primary key consisting of the compound foreign key {*productName, productSerial*} relating it to its supertype *Product*. The linker entity *BicycleMadeOfWheel* has a compound primary key consisting of the two compound foreign keys relating it to *Bicycle* and *Wheel*.

In figure 9, we introduce a notation for consistency maintenance rules. Where a reference to another table has a circle adornment containing a cross, this indicates cascading deletion. If the entity instance (viz. table row) at the head is deleted, then the corresponding entity instance(s) (viz. table rows) at the tail of the reference must also be deleted. Where a reference to another table has a plain circle adornment, this indicates setting the reference to null. If the entity instance (viz. table row) at the head is deleted, then the corresponding foreign key columns at the tail of the reference must be set to null. Where a reference has no adornment, this indicates the default semantics, meaning that the head entity may not be deleted while any tail entity exists.

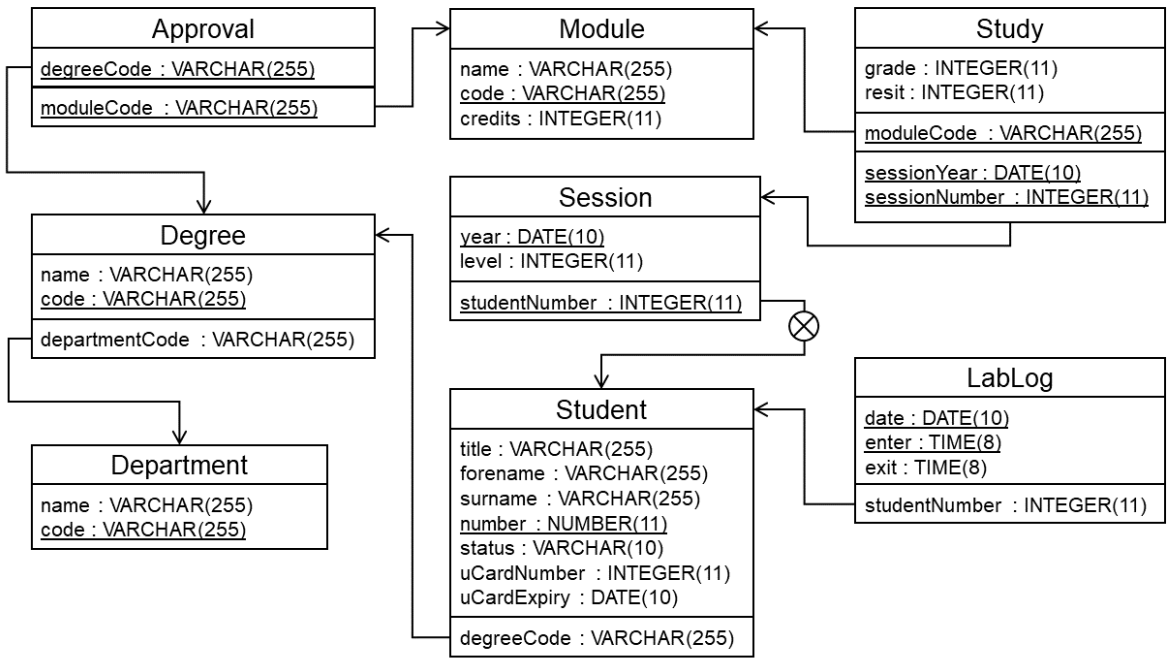

Figure 9: SQL Database Schema for the Student Records

Figure 9 illustrates the SQL Database Schema conversion of the Student Records case study, shown earlier as the Existence Dependency Graph in figure 3. All EDG entities have been converted into SQL tables, with all primary and foreign keys now explicit. Basic types have been converted into suitable SQL basic types.

In particular, the *String* and *Integer* types have been mapped respectively to the default SQL choices *VARCHAR(255)* and *INTEGER(11)*, to allow generous width strings and maximum length integers. A mapping rule that was more sensitive to field names, as well as their types,

might be able to select a shorter field width for attributes such as *code: String*. The other mapped types *DATE(10)* and *TIME(8)* types have been given suitable field widths, and the enumerated type *Status* has been mapped to a short string-representation, *VARCHAR(10).* SQL does support enumerated types, but these must be declared with all enumerated values, which our *BasicType* has not captured.

Foreign keys are correctly generated in *Student, Degree* and *LabLog*, which are distinctly renamed copies of the primary keys of the related entities. The renaming rule creates a name for each foreign key column by prefixing it with the name of the reference to the entity owning the copied primary key. In figure 8, this proved critical when distinguishing the *brand* and *serial* of a *FrameSet* and those of a *Handlebar* in *Bicycle*. We discuss this further in section 3.5.

Foreign keys are also correctly marked as primary keys in associative entities *Approval* and *Study*. One point of subtlety is that the associative entity *Study* depends on *Session* and on *Module*; and the detail entity *Session* depends in turn on *Student*. *Session* has correctly acquired the compound key for a detail entity {*studentNumber, year*}, which includes one foreign key. *Study* has also correctly acquired the compound key for an associative entity, consisting of both the foreign keys {*moduleCode, sessionNumber, sessionYear*}. All references from associative entities have the default restriction preventing deletion of their related entities. The reference from the detail entity *Session* to *Student* is marked with a cascading deletion adornment.

3.4 Metamodel for an SQL Database Schema

Figure 10, which extends over two pages, shows a metamodel for an SQL Database Schema. 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 *TableType*. The *Typed* concept, a kind of *Named* thing, refers to a *type: Type*. Its derived concepts include *Column* (whose *type* is specialised as a *BasicType*) and *SearchKey* (whose *type* is specialised as a *TableType*). The specialisations of *SearchKey* include *PrimaryKey* and *ForeignKey*.

A *TableType* consists of a set of *Columns* and a set of *SearchKeys*, which is a heterogeneous set containing exactly one *PrimaryKey* and zero or more *ForeignKeys*. *TableType* provides filtering operations to access the only *PrimaryKey*, and to select a (possibly empty) set of *ForeignKeys*. To aid in this, the *SearchKey* defines default operations *isPrimary*, *isForeign*, which are selectively redefined in each subtype concept to discriminate keys of the appropriate type.

The *Column* concept is a named and typed component of a *TableType*. It defines a field *width* attribute, which specifies the maximum number of characters allocated to the text representation of the column's value. Furthermore, it defines two *Boolean* valued attributes: *notNull*, to specify when a non-null value must be given for this column, and *autoInc*, to specify when the integer value of this column should be automatically incremented.

SearchKey defines features common to *PrimaryKey* and *ForeignKey*, including a reference to the *type* to which the key refers, and a reference to a set of *columns* that constitute the key. In *PrimaryKey*, the *columns* refer to the primary key identifiers. In *ForeignKey*, the *type* refers to the referenced *TableType*, and the *columns* refer to the foreign key columns. *ForeignKey* also refers to a *remote* set of columns, the matching primary key identifiers in the referenced

TableType. *ForeignKey* defines two Boolean valued attributes: *cascade*, to specify that a cascading delete is required after deleting the referenced table row, and *setNull*, to specify that the local foreign key columns should have their values set to null.

Database defines the top-level root concept of this model. It consists of a set of *BasicTypes* and a set of *TableTypes*. All of the inter-table referencing has now been captured in *ForeignKeys* that are components of a given *TableType*.

```
metamodel SQL {
   concept Named {
      attribute name : String
   }
   concept Typed inherit Named {
      reference type : Type
   }
   concept Type inherit Named {
   }
   concept BasicType inherit Type {
      operation accept(visitor : Visitor) : Boolean {
         visitor.visitBasic(self)
      }
   }
   concept TableType inherit Type {
      component columns : Column{}
      component keys : SearchKey{}
      operation primary : PrimaryKey {
         keys.detect(key | key.isPrimary)
      }
      operation foreign : ForeignKey{} {
         keys.select(key | key.isForeign)
      }
      operation accept(visitor : Visitor) : Boolean {
         visitor.visitTable(self)
      }
   }
   concept Column inherit Typed {
      reference type : BasicType
      attribute width : Integer
      attribute notNull : Boolean
      attribute autoInc : Boolean
      operation accept(visitor : Visitor) : Boolean {
         visitor.visitColumn(self)
      }
   }
   concept SearchKey inherit Typed {
      reference type : TableType
      reference columns : Column{}
      operation isPrimary : Boolean {
         false
      }
      operation isForeign : Boolean {
         false
      }
      operation accept(visitor : Visitor) : Boolean {
         false
      }
   }
   concept PrimaryKey inherit SearchKey {
      operation isPrimary : Boolean {
         true
      }
      operation accept(visitor : Visitor) : Boolean {
         visitor.visitPrimary(self)
      }
   }
```

```
    concept ForeignKey inherit SearchKey {
       reference remote : Column{}
       attribute cascade : Boolean
       attribute setNull : Boolean
       operation isForeign : Boolean {
          true
       }
       operation accept(visitor : Visitor) : Boolean {
          visitor.visitForeign(self)
       }
    }
    concept Database inherit Named {
       component basicTypes : BasicType{}
       component tableTypes : TableType{}
       operation accept(visitor : Visitor) : Boolean {
          visitor.visitDatabase(self)
       }
    }
    concept Visitor {
       operation visitDatabase(db : Database) : Boolean {
          false
       }
       operation visitBasic(type : BasicType) : Boolean {
          false
       }
       operation visitTable(type : TableType) : Boolean {
          false
       }
       operation visitColumn(col : Column) : Boolean {
          false
       }
       operation visitPrimary(key : PrimaryKey) : Boolean {
          false
       }
       operation visitForeign(key : ForeignKey) : Boolean {
          false
       }
    }
}
```

Figure 10: A metamodel for the SQL Database Schema

Finally, this metamodel defines a concept called *Visitor*, a feature which is present in any metamodel from which it is expected that code will be generated. This follows the Visitor Design Pattern [8] for traversing a tree-like data structure, in this case for the sake of code generation. This pattern is a collaboration between the metamodel elements and the *Visitor*. Metamodel elements define a standard operation with the signature: *accept(Visitor): Boolean*, which in turn invokes a specific operation of the *Visitor* to process that kind of element. The operations are placeholders; *Visitor* specifies the interface to be used by the eventual code generator, which will generate actual SQL data definitions from the model. We return to this in section 6 which discusses code generation.

3.5 The Cycle Shop Example Model

Figure 11 encodes the *Cycle Shop* database schema, shown above in figure 8, in the *ReMoDeL* textual syntax for models [1]. This model was translated automatically from an Existence Dependency Graph model, using a transformation to be presented in section 5. The database schema is presented in order of existence dependency, which aids in the legibility of the model. All *TableTypes* are defined before being referenced as *types* in *ForeignKeys*. All *Columns* are defined, before being referenced as *columns* or *remote* columns in *PrimaryKey*

or *ForeignKey* structures. This is achieved partly through ordering the source EDG model, and partly through the ordered structure of *TableType*.

```
model sql1 : SQL {
   d1 : Database(name = "Cycle Shop", basicTypes = BasicType{
      b1 : BasicType(name = "BOOLEAN"),
      b2 : BasicType(name = "INTEGER"),
      b3 : BasicType(name = "INT UNSIGNED"),
      b4 : BasicType(name = "DOUBLE"),
      b5 : BasicType(name = "VARCHAR"),
      b6 : BasicType(name = "DATE"),
      b7 : BasicType(name = "MONEY")
   }, tableTypes = TableType{
      t1 : TableType(name = "Address", columns = Column{
         c1 : Column(name = "house", type = b5, width = 255, notNull = true),
         c2 : Column(name = "postcode", type = b5, width = 255, notNull = true),
         c3 : Column(name = "road", type = b5, width = 255),
         c4 : Column(name = "city", type = b5, width = 255)
      }, keys = SearchKey{
         p1 : PrimaryKey(columns = Column{c1, c2})
      }),
      t2 : TableType(name = "Customer", columns = Column{
         c5 : Column(name = "addressHouse", type = b5, width = 255),
         c6 : Column(name = "addressPostcode", type = b5, width = 255),
         c7 : Column(name = "customerID", type = b2, width = 11,
            notNull = true, autoInc = true),
         c8 : Column(name = "forename", type = b5, width = 255),
         c9 : Column(name = "surname", type = b5, width = 255)
      }, keys = SearchKey{
         p2 : PrimaryKey(columns = Column{c7}),
         f1 : ForeignKey(type = t1, columns = Column{c5, c6},
            remote = Column{c1, c2})
      }),
      t3 : TableType(name = "Order", columns = Column{
         c10 : Column(name = "customerID", type = b2, width = 11),
         c11 : Column(name = "number", type = b2, width = 11, notNull = true),
         c12 : Column(name = "date", type = b6, width = 10)
      }, keys = SearchKey{
         p3 : PrimaryKey(columns = Column{c11}),
         f2 : ForeignKey(type = t2, columns = Column{c10}, remote = Column{c7})
      }),
      t4 : TableType(name = "Product", columns = Column{
         c13 : Column(name = "brand", type = b5, width = 255, notNull = true),
         c14 : Column(name = "serial", type = b2, width = 11, notNull = true),
         c15 : Column(name = "name", type = b5, width = 255),
         c16 : Column(name = "price", type = b7, width = 17)
      }, keys = SearchKey{
         p4 : PrimaryKey(columns = Column{c13, c14})
      }),
      t5 : TableType(name = "FrameSet", columns = Column{
         c17 : Column(name = "productBrand", type = b5, width = 255,
            notNull = true),
         c18 : Column(name = "productSerial", type = b2, width = 11,
            notNull = true),
         c19 : Column(name = "size", type = b2, width = 11),
         c20 : Column(name = "shocks", type = b1, width = 5)
      }, keys = SearchKey{
         p5 : PrimaryKey(columns = Column{c17, c18}),
         f3 : ForeignKey(type = t4, columns = Column{c17, c18},
            remote = Column{c13, c14}, cascade = true)
      }),
      t6 : TableType(name = "Handlebar", columns = Column{
         c21 : Column(name = "productBrand", type = b5, width = 255,
            notNull = true),
         c22 : Column(name = "productSerial", type = b2, width = 11,
            notNull = true),
         c23 : Column(name = "style", type = b5, width = 255)
```

```
      }, keys = SearchKey{
         p6 : PrimaryKey(columns = Column{c21, c22}),
         f4 : ForeignKey(type = t4, columns = Column{c21, c22},
            remote = Column{c13, c14}, cascade = true)
      }),
      t7 : TableType(name = "Wheel", columns = Column{
         c24 : Column(name = "productBrand", type = b5, width = 255,
            notNull = true),
         c25 : Column(name = "productSerial", type = b2, width = 11,
            notNull = true),
         c26 : Column(name = "diameter", type = b2, width = 11),
         c27 : Column(name = "tyre", type = b5, width = 255)
      }, keys = SearchKey{
         p7 : PrimaryKey(columns = Column{c24, c25}),
         f5 : ForeignKey(type = t4, columns = Column{c24, c25},
            remote = Column{c13, c14}, cascade = true)
      }),
      t8 : TableType(name = "Line", columns = Column{
         c28 : Column(name = "orderNumber", type = b2, width = 11, notNull = true),
         c29 : Column(name = "itemBrand", type = b5, width = 255),
         c30 : Column(name = "itemSerial", type = b2, width = 11),
         c31 : Column(name = "number", type = b2, width = 11, notNull = true),
         c32 : Column(name = "quantity", type = b2, width = 11),
         c33 : Column(name = "cost", type = b7, width = 17)
      }, keys = SearchKey{
         p8 : PrimaryKey(columns = Column{c28, c31}),
         f6 : ForeignKey(type = t3, columns = Column{c28},
            remote = Column{c11}, cascade = true),
         f7 : ForeignKey(type = t4, columns = Column{c29, c30},
            remote = Column{c13, c14})
      }),
      t9 : TableType(name = "Bicycle", columns = Column{
         c34 : Column(name = "productBrand", type = b5, width = 255,
            notNull = true),
         c35 : Column(name = "productSerial", type = b2, width = 11,
            notNull = true),
         c36 : Column(name = "frameSetBrand", type = b5, width = 255),
         c37 : Column(name = "frameSetSerial", type = b2, width = 11),
         c38 : Column(name = "handlebarBrand", type = b5, width = 255),
         c39 : Column(name = "handlebarSerial", type = b2, width = 11)
      }, keys = SearchKey{
         p9 : PrimaryKey(columns = Column{c34, c35}),
         f8 : ForeignKey(type = t4, columns = Column{c34, c35},
            remote = Column{c13, c14}, cascade = true),
         f9 : ForeignKey(type = t5, columns = Column{c36, c37},
            remote = Column{c17, c18}, setNull = true),
         f10 : ForeignKey(type = t6, columns = Column{c38, c39},
            remote = Column{c21, c22}, setNull = true)
      }),
      t10 : TableType(name = "BicycleMadeOfWheel", columns = Column{
         c40 : Column(name = "bicycleBrand", type = b5, width = 255,
            notNull = true),
         c41 : Column(name = "bicycleSerial", type = b2, width = 11,
            notNull = true),
         c42 : Column(name = "wheelBrand", type = b5, width = 255, notNull = true),
         c43 : Column(name = "wheelSerial", type = b2, width = 11, notNull = true)
      }, keys = SearchKey{
         p10 : PrimaryKey(columns = Column{c40, c41, c42, c43}),
         f11 : ForeignKey(type = t9, columns = Column{c40, c41},
            remote = Column{c34, c35}, cascade = true),
         f12 : ForeignKey(type = t7, columns = Column{c42, c43},
            remote = Column{c24, c25}, cascade = true)
      })
   })
}
```

Figure 12: the SQL Database Schema for the Cycle Shop

One aspect worth highlighting is the need for a sophisticated renaming strategy, where specific columns are referred to in multiple locations. The most complex example of this is in the table *Bicycle*, which refers in three different ways to columns {*brand, serial*} originally defined in *Product*. Since *Bicycle* is a subtype of *Product*, its primary key is also the foreign key {*productBrand, productSerial*} referring to the supertype. The one-step renaming rule prefixes the column names accordingly, to indicate their provenance. This is also the case for *FrameSet* and *Handlebar*, which are also subtypes of *Product*.

Now, *Bicycle* is an aggregate of *FrameSet* and *HandleBar*, so a two-step renaming rule is required, to distinguish the additional foreign keys {*frameSetBrand, frameSetSerial*} referring to *FrameSet*, and {*handelbarBrand, handlebarSerial*} referring to *Handlebar*, from each other and from{*productBrand, productSerial*} referring to *Product*. The two-step rule derives the references to *Product*, but prefixes the column names according to their local (rather than distant) provenance. This can only be achieved because we preserved the notion of named references in the EDG right up until the transformation to SQL (rather than expanding out foreign key columns in earlier transformations).

3.6 The Student Records Example Model

Figure 13 encodes the *Student Records* database schema, shown above in figure 9, in the *ReMoDeL* textual syntax for models [1]. The main feature here is the way in which the linker tables *Study* and *Approval* make use of their columns appropriately as foreign keys and again as compound primary keys.

```
model sql2 : SQL {
   d1 : Database(name = "Student Records", basicTypes = BasicType{
      b1 : BasicType(name = "BOOLEAN"),
      b2 : BasicType(name = "INTEGER"),
      b3 : BasicType(name = "INT UNSIGNED"),
      b4 : BasicType(name = "DOUBLE"),
      b5 : BasicType(name = "VARCHAR"),
      b6 : BasicType(name = "DATE"),
      b7 : BasicType(name = "TIME"),
      b8 : BasicType(name = "VARCHAR")
   }, tableTypes = TableType{
      t1 : TableType(name = "Department", columns = Column{
         c1 : Column(name = "code", type = b5, width = 255, notNull = true),
         c2 : Column(name = "name", type = b5, width = 255)
      }, keys = SearchKey{
         p1 : PrimaryKey(columns = Column{c1})
      }),
      t2 : TableType(name = "Degree", columns = Column{
         c3 : Column(name = "departmentCode", type = b5, width = 255),
         c4 : Column(name = "code", type = b5, width = 255, notNull = true),
         c5 : Column(name = "name", type = b5, width = 255)
      }, keys = SearchKey{
         p2 : PrimaryKey(columns = Column{c4}),
         f1 : ForeignKey(type = t1, columns = Column{c3}, remote = Column{c1})
      }),
      t3 : TableType(name = "Module", columns = Column{
         c6 : Column(name = "code", type = b5, width = 255, notNull = true),
         c7 : Column(name = "name", type = b5, width = 255),
         c8 : Column(name = "credits", type = b2, width = 11)
      }, keys = SearchKey{
         p3 : PrimaryKey(columns = Column{c6})
      }),
      t4 : TableType(name = "Student", columns = Column{
         c9 : Column(name = "degreeCode", type = b5, width = 255),
         c10 : Column(name = "number", type = b2, width = 11, notNull = true),
         c11 : Column(name = "title", type = b5, width = 255),
```

```
      c12 : Column(name = "forename", type = b5, width = 255),
      c13 : Column(name = "surname", type = b5, width = 255),
      c14 : Column(name = "status", type = b8, width = 10),
      c15 : Column(name = "uCardNumber", type = b2, width = 11),
      c16 : Column(name = "uCardExpiry", type = b6, width = 10)
   }, keys = SearchKey{
      p4 : PrimaryKey(columns = Column{c10}),
      f2 : ForeignKey(type = t2, columns = Column{c9}, remote = Column{c4})
   }),
   t5 : TableType(name = "LabLog", columns = Column{
      c17 : Column(name = "studentNumber", type = b2, width = 11),
      c18 : Column(name = "date", type = b6, width = 10, notNull = true),
      c19 : Column(name = "enter", type = b7, width = 8, notNull = true),
      c20 : Column(name = "exit", type = b7, width = 8)
   }, keys = SearchKey{
      p5 : PrimaryKey(columns = Column{c18, c19}),
      f3 : ForeignKey(type = t4, columns = Column{c17}, remote = Column{c10})
   }),
   t6 : TableType(name = "Approval", columns = Column{
      c21 : Column(name = "moduleCode", type = b5, width = 255, notNull = true),
      c22 : Column(name = "degreeCode", type = b5, width = 255, notNull = true)
   }, keys = SearchKey{
      p6 : PrimaryKey(columns = Column{c21, c22}),
      f4 : ForeignKey(type = t3, columns = Column{c21}, remote = Column{c6}),
      f5 : ForeignKey(type = t2, columns = Column{c22}, remote = Column{c4})
   }),
   t7 : TableType(name = "Session", columns = Column{
      c23 : Column(name = "studentNumber", type = b2, width = 11,
         notNull = true),
      c24 : Column(name = "year", type = b6, width = 10, notNull = true),
      c25 : Column(name = "level", type = b2, width = 11)
   }, keys = SearchKey{
      p7 : PrimaryKey(columns = Column{c23, c24}),
      f6 : ForeignKey(type = t4, columns = Column{c23}, remote = Column{c10},
         cascade = true)
   }),
   t8 : TableType(name = "Study", columns = Column{
      c26 : Column(name = "sessionNumber", type = b2, width = 11,
         notNull = true),
      c27 : Column(name = "sessionYear", type = b6, width = 10, notNull = true),
      c28 : Column(name = "moduleCode", type = b5, width = 255, notNull = true),
      c29 : Column(name = "grade", type = b2, width = 11),
      c30 : Column(name = "resit", type = b2, width = 11)
   }, keys = SearchKey{
      p8 : PrimaryKey(columns = Column{c26, c27, c28}),
      f7 : ForeignKey(type = t7, columns = Column{c26, c27},
         remote = Column{c23, c24}),
      f8 : ForeignKey(type = t3, columns = Column{c28}, remote = Column{c6})
   })
})
}
```

Figure 13: the SQL Database Schema for Student Records

In figures 12 and 13, every detail type (*Line, Session*) annotates its foreign key reference to its master type with a *cascade* instruction. Likewise, every subtype (*Bicycle, FrameSet, Handlebar* and *Wheel*) annotates its foreign key reference to its supertype with a *cascade* instruction. The aggregate references from *Bicycle* to *FrameSet* and *Handlebar* are annotated with a *set null* instruction. Associative linker tables (*Approval, Study*) have no annotations, meaning that they will enforce the default restrict constraint, but the aggregating linker *BicycleMadeOfWheel* has references annotated with a *cascade* instruction.

4. Normal ERM to EDG Transformation

The transformation from a normalised Entity Relationship Model to an Existence Dependency Graph must perform a number of mappings from the ERM metamodel to the EDG metamodel. This kind of transformation is *exogeneous*, also described as a *translation*. The source metamodel for an Entity Relationship Model was defined in part 1 of this report [11] in section 3; and the EDG metamodel was presented in section 2 of part 2 (the current document). This transformation is fairly simple and straightforward.

4.1 Mapping of Types

Every ERM basic type must be mapped to a similar basic type in the EDG. This follows a similar strategy to previous basic type transformations. Types with the same names are created, although these are distinct and belong to the target metamodel.

Every ERM entity must be mapped to an EDG entity, which has the same name, a mapped set of attributes, and possibly a mapped set of references. The latter are created by filtering a subset of the ERM diagram's relationships that refer to the entity as its source, and mapping these to references (see section 4.2), which are added to the entity.

We also wish to order the set of entities in the EDG diagram by existence dependency. This is to make the resulting model more legible for a human reader, avoiding inline expansion of referenced definitions, when it is presented in the *ReMoDeL* textual syntax for models.

4.2 Mapping of Properties

The properties of an EDG entity include its attributes and references. Every ERM attribute must be mapped to a similar EDG attribute, with the same name and a mapped basic type. The mapped EDG attribute must preserve whether the attribute is *identifying*.

Every ERM relationship must be mapped to a simpler ERM reference, which is named and has an entity type. We already know that every relationship in the normal ERM model is many-to-one and is ordered with the source being dependent upon the target. So, it is simply a matter of creating an EDG reference from the mapped source entity to the mapped target entity.

All the semantic properties of relationships must be preserved in references. This includes whether they are *identifying*, and whether they are marked as *part-of* (detail to master), *kind-of* (subtype to supertype) or *made-of* (aggregate to component). The mapped EDG reference is given the same name as the ERM end-role target. This will prove important later, when expanding references to foreign keys in the SQL metamodel.

4.3 The ReMoDeL Transformation Normal ERM to EDG

The ReMoDeL transformation for converting a normalised ERM diagram to an Existence Dependency Graph is shown below as figure 14. The transformation is called *NormToEdg* and belongs to the transformation group *UmlDB* (UML and databases).

```
transform NormToEdg : UmlDB {
   metamodel source : ERM
   metamodel target : EDG
```

```
   mapping ermToEdgDiagram(diagram : ERM_Diagram) : EDG_Diagram {
      create EDG_Diagram(name := diagram.name,
         basicTypes := ermToEdgBasicTypes(diagram),
            entities := ermToEdgEntities(diagram)
        )
      }
   mapping ermToEdgBasicTypes(diagram : ERM_Diagram) : EDG_BasicType{} {
      diagram.basicTypes.collect(type : ERM_BasicType |
         basicToEdgBasicType(type))
   }
   mapping ermToEdgEntities(diagram : ERM_Diagram) : EDG_Entity{} {
      sortEntities(diagram.entities.collect(entity : ERM_Entity |
         entityToEdgEntity(entity, diagram)).asList,
         create EDG_Entity[]()).asSet
   }
   function sortEntities(open : EDG_Entity[],
            closed : EDG_Entity[]) : EDG_Entity[] {
      if open.isEmpty then closed
          else if entityRefers(open.first, open.rest)
             then sortEntities(open.rest.with(open.first), closed)
             else sortEntities(open.rest, closed.with(open.first))
   }
   function entityRefers(first : EDG_Entity, rest : EDG_Entity[]) : Boolean {
       first.references.exists(ref : EDG_Reference | rest.has(ref.type))
   }
   mapping basicToEdgBasicType(type : ERM_BasicType) : EDG_BasicType {
      create EDG_BasicType(name := type.name)
   }
   mapping entityToEdgEntity(entity : ERM_Entity,
            diagram : ERM_Diagram) : EDG_Entity {
      create EDG_Entity(name := entity.name,
         properties := create EDG_Property{}()
            .union(entityToEdgAttributes(entity))
            .union(entityToEdgReferences(entity, diagram))
      )
   }
   mapping entityToEdgAttributes(entity : ERM_Entity) : EDG_Attribute{} {
      entity.attributes.collect(attrib : ERM_Attribute |
         attribToEdgAttribute(attrib))
   }
   mapping attribToEdgAttribute(attrib : ERM_Attribute) : EDG_Attribute {
      create EDG_Attribute(name := attrib.name,
          type := basicToEdgBasicType(attrib.type),
          id := attrib.id
       )
   }
   mapping entityToEdgReferences(entity : ERM_Entity,
            diagram : ERM_Diagram) : EDG_Reference{} {
      diagram.relationships.select(rel : ERM_Relationship |
         rel.source.type = entity).collect(rel : ERM_Relationship |
            relToEdgReference(rel, diagram))
   }
   mapping relToEdgReference(rel : ERM_Relationship,
             diagram : ERM_Diagram) : EDG_Reference {
      create EDG_Reference(name := rel.target.name,
         type := entityToEdgEntity(rel.target.type, diagram),
         id := rel.id,
         kindOf := rel.kindOf,
         partOf := rel.partOf,
         madeOf := rel.madeOf
      )
   }
}
```

Figure 14: the NormToEdg model transformation

The breakdown of this transformation shorter than others, consisting of 9 separate mapping rules and 2 auxiliary functions, and may be summarised:

- To map the normal ERM diagram to an EDG diagram you create a diagram with the same name, map the ERM basic types to corresponding EDG basic types; then map all the ERM entities to EDG entities, while sorting the EDG entities by existence dependency.
- To map an ERM entity to an EDG entity, you create an EDG entity of the same name, whose properties are found as the union of mapping the ERM entity's attributes and mapping any of the ERM diagram's relationships whose source refers to this entity.
- To map an ERM attribute to an EDG attribute, you create an EDG attribute with the same name and mapped type, preserving whether it is identifying.
- To map an ERM relationship to an EDG reference in a given EDG entity, create an EDG reference, whose name is that of the ERM relationship's target end-role, and whose type is found by mapping the type of the target end-role. All identifying, part-of, kind-of or made-of constraints are preserved.

The ordering of the set of EDG entities by existence dependency is achieved using the recursive function *sortEntities()*. This has an *open* list of entities to sort, and a *closed* list of entities in sorted order. On each recursion, the first entity from the *open* list is considered. If it refers to any other entity in the *open* list, it is added to the tail of the *open* list. Otherwise, it is moved to the *closed* list.

4.4 Normal ERM to EDG Examples

We have already given examples of this transformation in use. In part 1 of this report [11], section 5.5 listed the normal ERM model for the *Cycle Shop* case study in figure 16; and section 6.6 listed the normal ERM model for the *Student Records* case study in figure 17.

Executing the *NormToEdg* transformation on the source ERM model for the *Cycle Shop* example creates the target EDG model listed in figure 5 in section 2.5 of the current report. Executing the *NormToEdg* transformation on the source ERM model for the *Student Records* example creates the target EDG model listed in figure 6 in section 2.6 of the current report.

5. EDG to SQL Transformation

The transformation of an Existence Dependency Graph to an SQL Database Schema must perform a number of mappings from the EDG metamodel to the SQL metamodel. Once again, this is an *exogeneous* transformation, also known as a *translation*. We consider separately the mapping of types to tables, the mapping of attributes and references to columns, and the mapping of identifiers to search keys.

5.1 Mapping of Types

EDG basic types are mapped to SQL basic types, which have different type names. Each SQL basic type is also associated with a field width; so, there are two mappings from basic type to basic type, and from basic type to field width.

EDG entities are mapped to SQL tables having the same names. Each table is populated with columns that have been mapped from the EDG entity's attributes and references (see section 5.2). The tables also contain one primary key and possibly many foreign keys. These refer to the mapped columns in the same table. The primary key refers to all columns originating from any *identifying* properties; whereas a foreign key refers only to the columns originating from one reference, whether or not this is *identifying* (see section 5.3).

5.2 Mapping of Properties

EDG properties are either attributes or references. All properties are eventually mapped to columns. An EDG attribute is mapped to a single SQL column having the same name, but with a mapped SQL basic type and also a mapped field width. Some *identifying* attributes will be mapped to columns that form part of the primary key (see section 5.3).

An EDG reference is mapped to a set of one or more SQL columns, which will form part of a foreign key. The reference refers to another EDG entity, and it is the identifiers of this target entity that are of interest here. These identifiers are mapped to renamed columns, whose modified names are formed by prefixing the (type-cased) old name by the reference name (derived from an end-role name). These renamed columns are distinct from the identifying columns created in the mapped target table. They are essentially renamed copies of these columns, having the same mapped SQL basic type and field width.

Mapping an EDG reference may result in the transitive expansion of references into identifiers. For example, a dependency chain of weak entities will require recursive expansion of the identifying references referring to their respective master entities. This requires careful construction of the mapping rules to track which referenced entity is to be treated as the provenance of the remote attribute, when renaming.

5.3 Mapping of Identifiers

Any EDG property may be identifying, whether an attribute or a reference. An identifying attribute will be mapped to a column forming part of a primary key in the owning table. An identifying reference will be expanded to a set of identifying attributes, which are mapped to one or more renamed columns forming part of the same primary key. Furthermore, any reference (whether identifying or not) will be mapped to renamed columns forming part of a foreign key (see section 5.2).

Each EDG entity is mapped to a primary key for the corresponding SQL table, by constructing a primary key and populating it with suitable columns. These are found by mapping the entity's *identifying* properties to columns, as described above.

For each primary key, the SQL entity integrity constraint is added. For each column forming part of the primary key, the *not null* constraint is added. For any such column which was derived from a surrogate identifier, the *auto-increment* constraint is added.

Each EDG entity is mapped to zero or more foreign keys in the corresponding SQL table, depending on how many references it has. For each EDG reference in the entity, a new foreign key is constructed. It refers to a remote SQL table, which is found by mapping the target type of the EDG reference. It is populated with two sets of columns. The remote columns are the identifying columns in the mapped target table. The local columns are renamed copies of these.

For each foreign key, it is determined whether this should have an SQL deletion constraint; otherwise, the SQL *restrict* constraint is assumed to apply by default. If the EDG reference has the semantic tags *part-of* or *kind-of*, then a *cascade* constraint is added. If the EDG reference has the semantic tag *made-of*, then a *set null* constraint is added.

5.4 The ReMoDeL EDG to SQL Transformation

The ReMoDeL transformation for converting an Existence Dependency Graph into an SQL Database Schema is shown over the next few pages as figure 15. The transformation is called *EdgToSql* and belongs to the transformation group *UmlDB* (UML and databases).

```
transform EdgToSql : UmlDB {
   metamodel source : EDG
   metamodel target : SQL

   mapping edgToSqlDatabase(diagram : EDG_Diagram) : SQL_Database {
      create SQL_Database(name := diagram.name,
         basicTypes := edgToSqlBasicTypes(diagram),
         tableTypes := edgToSqlTableTypes(diagram)
      )
   }
   mapping edgToSqlBasicTypes(diagram : EDG_Diagram) : SQL_BasicType{} {
      diagram.basicTypes.collect(type : EDG_BasicType | basicToBasicType(type))
   }
   mapping edgToSqlTableTypes(diagram : EDG_Diagram) : SQL_TableType{} {
      diagram.entities.collect(entity : EDG_Entity | entityToTableType(entity))
   }
   mapping basicToBasicType(type : EDG_BasicType) : SQL_BasicType {
      create SQL_BasicType(name := typeToDataType(type.name))
   }
   function typeToDataType(name : String) : String {
      if name = "Boolean"
      then "BOOLEAN"
      else if name = "Integer"
      then "INTEGER"
      else if name = "Natural"
      then "INT UNSIGNED"
      else if name = "Real"
      then "DOUBLE"
      else if name = "String"
      then "VARCHAR"
      else if name = "Date"
      then "DATE"
      else if name = "Time"
```

```
   then "TIME"
   else if name = "Money"
   then "MONEY"
   else "VARCHAR"
}
function typeToFieldWidth(name : String) : Integer {
   if name = "Boolean"
   then 5
   else if name = "Integer"
   then 11
   else if name = "Natural"
   then 11
   else if name = "Real"
   then 17
   else if name = "String"
   then 255
   else if name = "Date"
   then 10
   else if name = "Time"
   then 8
   else if name = "Money"
   then 17
   else 10
}
mapping entityToTableType(entity : EDG_Entity) : SQL_TableType {
   create SQL_TableType(name := entity.name,
      columns := entityToColumns(entity),
      keys := create SQL_SearchKey{}()
         .with(entityToPrimaryKey(entity))
         .union(entityToForeignKeys(entity))
   )
}
mapping entityToColumns(entity : EDG_Entity) : SQL_Column{} {
   create SQL_Column{}()
      .union(entity.references.collate(ref : EDG_Reference |
         referenceToColumns(ref)))
      .union(entity.attributes.collect(attrib : EDG_Attribute |
         attributeToColumn(attrib)))
}
mapping attributeToColumn(attrib : EDG_Attribute) : SQL_Column {
   create SQL_Column(name := attrib.name,
      type := basicToBasicType(attrib.type),
      width := typeToFieldWidth(attrib.type.name),
      notNull := attrib.id,
      autoInc := attrib.surrogate
   )
}
mapping referenceToColumns(ref : EDG_Reference) : SQL_Column{} {
   create SQL_Column{}()
      .union(ref.type.complexIDs.collate(remote : EDG_Reference |
         remoteRefToColumns(remote, ref)))
      .union(ref.type.simpleIDs.collect(attrib : EDG_Attribute |
         remoteAttribToColumn(attrib, ref)))
}
mapping remoteRefToColumns(remote : EDG_Reference,
         ref : EDG_Reference) : SQL_Column{} {
   create SQL_Column{}()
      .union(remote.type.complexIDs.collate(next : EDG_Reference |
         remoteRefToColumns(next, ref)))
      .union(remote.type.simpleIDs.collect(attrib : EDG_Attribute |
         remoteAttribToColumn(attrib, ref)))
}
mapping remoteAttribToColumn(attrib : EDG_Attribute,
         ref : EDG_Reference) : SQL_Column {
   create SQL_Column(name := if attrib.surrogate
                             then ref.name.concat("ID")
                             else ref.name.concat(attrib.name.asType),
      type := basicToBasicType(attrib.type),
```

```
            width := typeToFieldWidth(attrib.type.name),
            notNull := ref.id
        )
    }
    mapping entityToPrimaryKey(entity : EDG_Entity) : SQL_PrimaryKey {
        create SQL_PrimaryKey(
            columns := create SQL_Column{}()
                .union(entity.complexIDs.collate(ref : EDG_Reference |
                    referenceToColumns(ref)))
                .union(entity.simpleIDs.collect(attrib : EDG_Attribute |
                    attributeToColumn(attrib)))
        )
    }
    mapping entityToForeignKeys(entity : EDG_Entity) : SQL_ForeignKey{} {
        create SQL_ForeignKey{}()
        .union(entity.references.collect(ref : EDG_Reference | refToForeignKey(ref)))
    }
    mapping refToForeignKey(ref : EDG_Reference) : SQL_ForeignKey {
        create SQL_ForeignKey(type := entityToTableType(ref.type),
            columns := referenceToColumns(ref),
            remote := entityToTableType(ref.type).primary.columns,
            cascade := ref.partOf or ref.kindOf,
            setNull := ref.madeOf
        )
    }
}
```

Figure 15: the EdgToSql model transformation

The breakdown of this transformation is moderately complex, consisting of 13 separate short mapping rules and 2 auxiliary functions, and may be summarised:

- To map the EDG diagram to an SQL database, you create a database with the same name, map the EDG basic types to SQL basic types, and map the EDG entities to SQL tables.
- To map an EDG basic type to an SQL basic type, you create an SQL basic type with the appropriate SQL type name (using an auxiliary function).
- To map an EDG entity to an SQL table type, you create an SQL table type with the same name, and map all the entity's attributes and references to columns, and then map the entity to a primary key, and finally map the entity to a (possibly empty) set of foreign keys.
- To map an EDG attribute to a column, you create an SQL column with the same name, with the mapped basic type, and with a mapped field width (using an auxiliary function). If the attribute was *identifying*, set the *not null* constraint; and if the attribute was a *surrogate*, set the *auto-increment* constraint.
- To map an EDG reference to a (renamed) set of columns, you map all the identifiers of the one-step referenced entity to columns. This involves mapping the *remote* identifying references and *remote* identifying attributes to columns, which is different from mapping local properties, since the columns will be renamed.
- To map a *remote* EDG reference to a (renamed) set of columns, you map all the remote identifiers of the two-step referenced entity to columns. Mapping the *remote* identifying references is a recursive application of this rule; mapping the *remote* identifying attributes will create renamed columns.
- To map a *remote* EDG attribute to a (renamed) column, you create an SQL column with a name synthesised from the name of the first traversed EDG reference and the attribute name, with the mapped basic type and field width. If the attribute was *identifying*, set the *not null* constraint (but never set the *auto-increment* constraint, since this column is part of a foreign key).

- To map an EDG entity to an SQL primary key, create a primary key containing the result of mapping all the identifying properties of the entity to columns. Columns resulting from mapped references precede columns from mapped attributes.
- To map an EDG entity to a (possibly empty) set of foreign keys, map each of the entity's EDG references to an SQL foreign key. To do this, create a foreign key whose type refers to the mapped table type of the EDG reference, whose local columns refer to the result of mapping this reference to columns, and whose remote columns refer to the primary key columns selected from the mapped table type. If the reference was tagged *part-of* or *kind-of*, set the *cascade* constraint. If the reference was tagged *made-of*, set the *set null* constraint.

Perhaps the most difficult aspect of this transformation is dealing with column naming. If a column is derived from a local attribute, its name is the same as that of the attribute. If a column is derived from a local reference, its name is prefixed with the name of the reference, which is typically the name-case version of the referenced type. Surrogate identifiers are likewise recreated to use the local reference-name concatenated with "ID". When a column is derived from a remote reference, the alternative rule keeps track of the first reference to be traversed, so that this may be used in renaming.

5.5 EDG to SQL Examples

We have already given examples of this transformation in use. Section 2.5 described a source EDG model for the *Cycle Shop* case study, listed in figure 5, which was depicted as a diagram in figure 2. Section 2.6 described a source EDG model for the *Student Records* case study, listed in figure 6, which was depicted as a diagram in figure 3.

Executing the *EdgToSql* transformation on the source EDG model for the *Cycle Shop* case study, listed in figure 5, creates the target SQL model listed in figure 12, in section 3.5. This target model was depicted as a diagram in figure 8, in section 3.3.

Executing the EdgToSql transformation on the source EDG model for the *Student Records* case study, listed in figure 6, creates the target SQL model listed in figure 13, in section 3.6. This target model was depicted as a diagram in figure 9, in section 3.3.

6. Code Generation

The code generation step is usually not considered part of the chain of model transformations, which operate in the domain of models. Typically, code generation requires a bespoke program that is not expressed in the same model transformation language. Instead, it is considered as a separate step that converts the final model to some form of executable code. This final step is sometimes called a *model-to-text* translation.

In our case, the code that we wish to generate is *SQL data definition language*, which may be executed in an SQL environment to create a database. The final SQL Database Schema model was constructed in such a way as to facilitate code generation. The approach that we use follows the Visitor Design Pattern [8] for traversing a tree-like structure, in this case, in order to translate it into executable SQL instructions.

We implement the code generator in Java, in order to link this with the compilation model for the *ReMoDeL* language [12], which cross-compiles metamodels and model transformations into Java, using the Java compiler to generate executable bytecodes.

6.1 The Visitor Interface

The SQL Database Schema model provided a concept called *Visitor*, which defined a number of operations. *Visitor* was shown at the end of the metamodel in figure 10, but we repeat it below in figure 17 for ease of reference.

```
concept Visitor {
   operation visitDatabase(db : Database) : Boolean {
      false
   }
   operation visitBasic(type : BasicType) : Boolean {
      false
   }
   operation visitTable(type : TableType) : Boolean {
      false
   }
   operation visitColumn(col : Column) : Boolean {
      false
   }
   operation visitPrimary(key : PrimaryKey) : Boolean {
      false
   }
   operation visitForeign(key : ForeignKey) : Boolean {
      false
   }
}
```

Figure 17: The Visitor concept from the SQL Database Schema

This concept will be compiled following the usual translation scheme into Java [12], creating a class called *Visitor* in the package *meta.sql*. Each of the operations is translated into a trivial Java method returning the value *false*. This is illustrated in figure 18.

While it is tempting to think that we could implement the code generator by adding suitable printing instructions by hand to the body of each of these methods, this strategy will not survive recompilation of the metamodel. Every time that the *ReMoDeL* compiler processes the SQL metamodel, it will regenerate the class in figure 18, with trivial methods (again), overwriting any manually edited version of the class.

```
package meta.sql;

import remodel.util.*;

class Visitor extends Top {
    public boolean visitDatabase(db : Database) {
       return false;
    }
    public boolean visitBasic(type : BasicType) {
       return false;
    }
    public boolean visitTable(type : TableType) {
       return false;
    }
    public boolean visitColumn(col : Column) {
       return false;
    }
    public boolean visitPrimary(key : PrimaryKey) {
       return false;
    }
    public boolean visitForeign(key : ForeignKey) {
       return false;
    }
   public Visitor() {
    }
}
```

Figure 18: The compiled Visitor concept in Java

The *Visitor* is similar in purpose to an interface in Java. However, the *ReMoDeL* compiler does not support the creation of interfaces with abstract methods. All compiled operations in *ReMoDeL* are functions which return a result. So, each of *Visitor's* operations compiles to an outline method, which returns a trivial value. We chose *false* arbitrarily.

Instead, it is better to think of *Visitor* as fulfilling the role of an abstract superclass in Java, which we can extend with our own concrete visitor, which provides all the desired printing methods. These will override the outline methods of *Visitor*.

6.2 The MySQLWriter Implementation

We may implement this derived class by hand, without fear of its being overwritten. It is placed in the same metamodel package *meta.sql* as the generated *Visitor*. We call the class *MySQLWriter* to highlight the fact that it will generate SQL code for a particular database *MySQL*. We could write specialised generators for *Postgres, MS Access* or *Oracle*, which may expect small differences in the SQL code style.

The *MySQLWriter* class is shown over the next few pages as figure 19.

```
package meta.sql;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class MySQLWriter extends Visitor {
       private File file;
       private Writer writer;

       public MySQLWriter() {
       }
```

```
public boolean visitDatabase(Database db) {
	String dbName = db.getName().replace(' ', '_');
	String fileName = "code/" + dbName + ".sql";
	file = new File(fileName);
	try (BufferedWriter bufWriter =
			new BufferedWriter(new FileWriter(file))) {
		writer = bufWriter;
		writer.write("CREATE DATABASE " + dbName + ";\n\n");
		writer.write("USE " + dbName + ";\n\n");
		for (TableType tbl : db.getTableTypes()) {
			tbl.accept(this);
		}
	} catch (IOException e) {
		e.printStackTrace();
	}
	return false;
}
public boolean visitBasic(BasicType type) {
	try {
		writer.write(type.getName());
	} catch (IOException e) {
		e.printStackTrace();
	}
	return false;
}
public boolean visitTable(TableType type) {
	String tblName = type.getName();
	try {
		writer.write("CREATE TABLE " + tblName + " (\n");
		boolean comma = false;
		for (Column col : type.getColumns()) {
			if (comma)
				writer.write(",\n");
			writer.write("   ");
			col.accept(this);
			comma = true;
		}
		for (SearchKey key : type.getKeys()) {
			if (comma)
				writer.write(",\n");
			writer.write("   ");
			key.accept(this);
		}
		writer.write("\n);\n\n");
	} catch (IOException e) {
		e.printStackTrace();
	}
	return false;
}
public boolean visitColumn(Column col) {
	try {
		writer.write(col.getName() + " ");
		col.getType().accept(this);
		writer.write("(" + col.getWidth() + ")");
		if (col.getNotNull())
			writer.write(" NOT NULL");
		if (col.getAutoInc())
			writer.write(" AUTO_INCREMENT");
	} catch (IOException e) {
		e.printStackTrace();
	}
	return false;
}
public boolean visitPrimary(PrimaryKey key) {
	try {
		writer.write("PRIMARY KEY (");
		boolean comma = false;
		for (Column col : key.getColumns()) {
```

```
                        if (comma)
                                writer.write(", ");
                        writer.write(col.getName());
                        comma = true;
                }
                writer.write(")");
        } catch (IOException e) {
                e.printStackTrace();
        }
        return false;
}
public boolean visitForeign(ForeignKey key) {
        try {
                writer.write("FOREIGN KEY (");
                boolean comma = false;
                for (Column col : key.getColumns()) {
                        if (comma)
                                writer.write(", ");
                        writer.write(col.getName());
                        comma = true;
                }
                writer.write(") REFERENCES ");
                writer.write(key.getType().getName() + " (");
                comma = false;
                for (Column col : key.getRemote()) {
                        if (comma)
                                writer.write(", ");
                        writer.write(col.getName());
                        comma = true;
                }
                writer.write(")");
                if (key.getCascade())
                        writer.write("\n      ON DELETE CASCADE");
                else if (key.getSetNull())
                        writer.write("\n      ON DELETE SET NULL");
        } catch (IOException e) {
                e.printStackTrace();
        }
        return false;
}
public String toString() {
        return file.toString();
}
}
```

Figure 19: The MySQLWriter Code Generator

This class extends *Visitor* and reimplements all of *Visitor's* methods, so is a compatible kind of visitor. That is, we can pass an object of this kind to any method with the Java type signature: *boolean accept(Visitor visitor).* This method is present in each of the relevant Java classes that were compiled from concepts in the SQL metamodel (see figure 10).

To execute the *MySQLWriter*, an instance is created and is passed to the *accept()* method of the root *Database* node of an SQL model. This triggers the first *visitDatabase()* method. Thereafter, code generation proceeds as follows:

- *visitDatabase* – creates a file in a standard location, *code/dbname.sql,* where *dbname* is the name of the *Database* node, and then opens a buffered output stream onto this file. It prints the semicolon-terminated *CREATE DATABASE* and *USE dbname* instructions to set up the database, then delegates to each *TableType* node, to receive the visitor. When this is finished, the output stream is automatically closed, flushing the file to disk.

- *visitTable* – prints a *CREATE TABLE* instruction, with opening parenthesis, and then delegates to each *Column* node to receive the visitor, inserting comma separators. After this, it delegates to each *SearchKey* node to receive the visitor, inserting comma separators. After this, it prints a closing parenthesis and terminating semicolon.
- *visitColumn* – prints the column's name, delegates to the *BasicType* node to receive the visitor and print the type, then prints a field width in parentheses. If the column has the relevant constraints, prints *NOT NULL* and *AUTO_INCREMENT*.
- *visitType* – prints the SQL basic type name.
- *visitPrimary* – prints the *PRIMARY KEY* instruction and an opening parenthesis, then prints the names of the local columns, with comma separators. Prints a closing parenthesis.
- *visitForeign* – prints the *FOREIGN KEY* instruction and an opening parenthesis, then prints the names of the local columns, with comma separators, ending with a closing parenthesis. Prints the *REFERENCES* instruction, followed by the name of the referenced table, and then a similar list of remote column names, separated by commas, inside parentheses. If the foreign key has the relevant constraints, prints *ON DELETE CASCADE* or *ON DELETE SET NULL*.

We chose a certain Java coding style. We adopt a *try-catch* style that catches all exceptions (in the event of output failure) so that the methods are not obliged to declare that they *throw IOException*, since this cannot be predicted by the *ReMoDeL* compiler that generated the code for *Visitor*. Instead, we print the stack trace anyway in the event of failure. We also use a *try-with-resources* style to open the output stream in a protected *try-block*, to ensure that this is automatically closed on exit from the *visitDatabase()* method. This visitor has its own *toString()* method that prints out the name of the output file.

6.3 The Generated Cycle Shop Database

Figure 20 illustrates the MySQL data definition language instructions generated for the *Cycle Shop* case study. This is a direct model-to-text translation of the SQL database schema model listed in figure 12 and depicted in figure 8. It needs no further comment.

```
CREATE DATABASE Cycle_Shop;

USE Cycle_Shop;

CREATE TABLE Address (
   house VARCHAR(255) NOT NULL,
   postcode VARCHAR(255) NOT NULL,
   road VARCHAR(255),
   city VARCHAR(255),
   PRIMARY KEY (house, postcode)
);

CREATE TABLE Customer (
   addressHouse VARCHAR(255),
   addressPostcode VARCHAR(255),
   customerID INTEGER(11) NOT NULL AUTO_INCREMENT,
   forename VARCHAR(255),
   surname VARCHAR(255),
   PRIMARY KEY (customerID),
   FOREIGN KEY (addressHouse, addressPostcode)
      REFERENCES Address (house, postcode)
);

CREATE TABLE Order (
   customerID INTEGER(11),
```

```
   number INTEGER(11) NOT NULL,
   date DATE(10),
   PRIMARY KEY (number),
   FOREIGN KEY (customerID) REFERENCES Customer (customerID)
);

CREATE TABLE Product (
   brand VARCHAR(255) NOT NULL,
   serial INTEGER(11) NOT NULL,
   name VARCHAR(255),
   price MONEY(17),
   PRIMARY KEY (brand, serial)
);

CREATE TABLE FrameSet (
   productBrand VARCHAR(255) NOT NULL,
   productSerial INTEGER(11) NOT NULL,
   size INTEGER(11),
   shocks BOOLEAN(5),
   PRIMARY KEY (productBrand, productSerial),
   FOREIGN KEY (productBrand, productSerial)
      REFERENCES Product (brand, serial)
      ON DELETE CASCADE
);

CREATE TABLE Handlebar (
   productBrand VARCHAR(255) NOT NULL,
   productSerial INTEGER(11) NOT NULL,
   style VARCHAR(255),
   PRIMARY KEY (productBrand, productSerial),
   FOREIGN KEY (productBrand, productSerial)
      REFERENCES Product (brand, serial)
      ON DELETE CASCADE
);

CREATE TABLE Wheel (
   productBrand VARCHAR(255) NOT NULL,
   productSerial INTEGER(11) NOT NULL,
   diameter INTEGER(11),
   tyre VARCHAR(255),
   PRIMARY KEY (productBrand, productSerial),
   FOREIGN KEY (productBrand, productSerial) |
      REFERENCES Product (brand, serial)
      ON DELETE CASCADE
);

CREATE TABLE Line (
   orderNumber INTEGER(11) NOT NULL,
   itemBrand VARCHAR(255),
   itemSerial INTEGER(11),
   number INTEGER(11) NOT NULL,
   quantity INTEGER(11),
   cost MONEY(17),
   PRIMARY KEY (orderNumber, number),
   FOREIGN KEY (orderNumber) REFERENCES Order (number)
      ON DELETE CASCADE,
   FOREIGN KEY (itemBrand, itemSerial)
      REFERENCES Product (brand, serial)
);

CREATE TABLE Bicycle (
   productBrand VARCHAR(255) NOT NULL,
   productSerial INTEGER(11) NOT NULL,
   frameSetBrand VARCHAR(255),
   frameSetSerial INTEGER(11),
   handlebarBrand VARCHAR(255),
   handlebarSerial INTEGER(11),
   PRIMARY KEY (productBrand, productSerial),
```

```
   FOREIGN KEY (productBrand, productSerial)
      REFERENCES Product (brand, serial)
      ON DELETE CASCADE,
   FOREIGN KEY (frameSetBrand, frameSetSerial)
      REFERENCES FrameSet (productBrand, productSerial)
      ON DELETE SET NULL,
   FOREIGN KEY (handlebarBrand, handlebarSerial)
      REFERENCES Handlebar (productBrand, productSerial)
      ON DELETE SET NULL
);

CREATE TABLE BicycleMadeOfWheel (
   bicycleBrand VARCHAR(255) NOT NULL,
   bicycleSerial INTEGER(11) NOT NULL,
   wheelBrand VARCHAR(255) NOT NULL,
   wheelSerial INTEGER(11) NOT NULL,
   PRIMARY KEY (bicycleBrand, bicycleSerial, wheelBrand, wheelSerial),
   FOREIGN KEY (bicycleBrand, bicycleSerial)
      REFERENCES Bicycle (productBrand, productSerial)
      ON DELETE CASCADE,
   FOREIGN KEY (wheelBrand, wheelSerial)
      REFERENCES Wheel (productBrand, productSerial)
      ON DELETE CASCADE
);
```

Figure 20: The MySQL Code for the Cycle Shop

6.4 The Generated Student Records Database

Figure 21 illustrates the MySQL data definition language instructions generated for the *Student Records* case study. This is a direct model-to-text translation of the SQL database schema model listed in figure 13 and depicted in figure 9. It needs no further comment.

```
CREATE DATABASE Student_Records;

USE Student_Records;

CREATE TABLE Department (
   code VARCHAR(255) NOT NULL,
   name VARCHAR(255),
   PRIMARY KEY (code)
);

CREATE TABLE Degree (
   departmentCode VARCHAR(255),
   code VARCHAR(255) NOT NULL,
   name VARCHAR(255),
   PRIMARY KEY (code),
   FOREIGN KEY (departmentCode) REFERENCES Department (code)
);

CREATE TABLE Module (
   code VARCHAR(255) NOT NULL,
   name VARCHAR(255),
   credits INTEGER(11),
   PRIMARY KEY (code)
);

CREATE TABLE Student (
   degreeCode VARCHAR(255),
   number INTEGER(11) NOT NULL,
   title VARCHAR(255),
   forename VARCHAR(255),
```

```
   surname VARCHAR(255),
   status VARCHAR(10),
   uCardNumber INTEGER(11),
   uCardExpiry DATE(10),
   PRIMARY KEY (number),
   FOREIGN KEY (degreeCode) REFERENCES Degree (code)
);

CREATE TABLE LabLog (
   studentNumber INTEGER(11),
   date DATE(10) NOT NULL,
   enter TIME(8) NOT NULL,
   exit TIME(8),
   PRIMARY KEY (date, enter),
   FOREIGN KEY (studentNumber) REFERENCES Student (number)
);

CREATE TABLE Approval (
   moduleCode VARCHAR(255) NOT NULL,
   degreeCode VARCHAR(255) NOT NULL,
   PRIMARY KEY (moduleCode, degreeCode),
   FOREIGN KEY (moduleCode) REFERENCES Module (code),
   FOREIGN KEY (degreeCode) REFERENCES Degree (code)
);

CREATE TABLE Session (
   studentNumber INTEGER(11) NOT NULL,
   year DATE(10) NOT NULL,
   level INTEGER(11),
   PRIMARY KEY (studentNumber, year),
   FOREIGN KEY (studentNumber) REFERENCES Student (number)
      ON DELETE CASCADE
);

CREATE TABLE Study (
   sessionNumber INTEGER(11) NOT NULL,
   sessionYear DATE(10) NOT NULL,
   moduleCode VARCHAR(255) NOT NULL,
   grade INTEGER(11),
   resit INTEGER(11),
   PRIMARY KEY (sessionNumber, sessionYear, moduleCode),
   FOREIGN KEY (sessionNumber, sessionYear)
      REFERENCES Session (studentNumber, year),
   FOREIGN KEY (moduleCode) REFERENCES Module (code)
);
```

Figure 21: The MySQL Code for the Student Records

6.5 The Chain of Compiled Transformations

The entire chain of model transformations presented in parts 1 and 2 of this report [11] can be built using a simple Java program. To summarise, the four linked transformations and fifth code generation step were:

1. from a UML Class Diagram to a pre-normal Entity Relationship Model;
2. from a pre-normal ERM to a normal (3NF+) Entity Relationship Model;
3. from a normal ERM to an Existence Dependency Graph;
4. from an Existence Dependency Graph to an SQL Database Schema;
5. from an SQL Database Schema to MySQL data definition language.

The program describing this chain is called *UmlToSqlChain.java*, listed in figure 22. For convenience, we have placed it in the same package *rule.umldb* that is used for each of the four described model transformation programs.

```
package rule.umldb;

import java.io.File;
import java.io.IOException;

import meta.sql.MySQLWriter;
import meta.sql.Visitor;
import remodel.meta.Model;

/**
 * UmlToSqlChain is a transformation chain from a UML Class
 * Diagram, to a prenormal Entity-Relationship Model, to a
 * normalised Entity-Relationship Model, to a SQL Database
 * Model, followed by a code-generation step to MySQL code.
 */
public class UmlToSqlChain {

	private static File file = new File("models/uml1.mod");

	public static void main(String[] args) throws IOException {

		Model<meta.uml.Diagram> model1 = new Model<>("uml", "UML");
		model1.read(file);
		System.out.println("Successfully read: " + file);

		UmlToErm transform1 = new UmlToErm();
		Model<meta.erm.Diagram> model2 = transform1.apply(model1);
		model2.write(new File("models/erm1.mod"));

		ErmToNorm transform2 = new ErmToNorm();
		Model<meta.erm.Diagram> model3 = transform2.apply(model2);
		model3.write(new File("models/norm1.mod"));

		NormToEdg transform3 = new NormToEdg();
		Model<meta.edg.Diagram> model4 = transform3.apply(model3);
		model4.write(new File("models/edg1.mod"));

		EdgToSql transform4 = new EdgToSql();
		Model<meta.sql.Database> model5 = transform4.apply(model4);
		model5.write(new File("models/sql1.mod"));

		Visitor visitor = new MySQLWriter();
		model5.getRoot().accept(visitor);

		System.out.println("Successfully written: " + visitor);
	}
}
```

Figure 22: The UmlToSqlChain Transformation Chain.

While the figure shows how each transformation can be invoked as a client, applying it to one model to generate the next model in the chain , it is also possible to invoke a transformation as a standalone program on an input model file [12]. Here, we use a model's ability to serialise itself to a file in order to capture all the in-between models.

7. 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] M Snoeck and G Dedene. "Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types", *IEEE Transactions on Software Engineering 24(4),* (IEEE, 1998), 233–251.

[10] M Snoeck. *Enterprise Information Systems Engineering: The MERODE approach* (Springer, 2014).

[11] A J H Simons. ReMoDeL Data Refinement (rev. 1.0): Data Transformations in Remodel, Part 1. Technical Report, 25 July (University of Sheffield, 2022).

[12] A J H Simons. ReMoDeL Compiled (rev. 2.1): The Cross-Compilation of ReMoDeL to Java by Example. Technical Report, 12 July (University of Sheffield, 2022).