Automatic Normalisation via Metamodelling

D.H.Akehurst, B.Bordbar, P.J.Rodgers, N.T.G.Dalgliesh

University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF
{ D.H.Akehurst, B.Bordbar, P.J.Rodgers, N.T.G.Dalgliesh }@ukc.ac.uk

The process of normalisation has long been accepted as a crucial part of the design for good database systems. By using a declarative approach to the specification of the normalisation rules and a precisely defined transformation, over a meta-model of a database system design language, we can automate the normalisation process. A tool supporting the normalisation of database system designs can subsequently be developed providing an invaluable aid to the software system designer.

Introduction

The Unified Modelling Language (UML [1]) is becoming the industry standard for expressing database schemas [2]. A fundamental part of schema design is the process of Normalisation. This process, which is well documented in the literature [3] [4], is designed to guarantee data integrity in the model by using formal techniques to eliminate redundancy and the possibility of data update anomalies. The process consists of the stepwise application of a sequence of transformations on the schema in order to meet defined constraints, each of which describes a *normal form* – first (1NF), second (2NF), third (3NF) and Boyce-Codd (BCNF) normal forms.

All normal form definitions make use of *functional dependencies* between attributes. A functional dependency is a pair consisting of a set of attributes (the determinant) whose values can be used to uniquely identify the values of another set of attributes (the dependant). For example, a property identification number can be used to identify a unique property address and the rent charged for that property.

Through using UML as the schema definition language we gain access to the Object Constraint Language (OCL [1]), a declarative constraint, navigation and logical expression language. This is used to construct expressions that encode functional dependencies between attributes and the above four normal form definitions on classes at a meta-level. These OCL expressions provide a concise, easily understandable and generic mechanism for reasoning about the specific normal form of any schema instantiated from the meta-model.

The transition steps from one normal form to the next can be described as a series of transformations on the model. The transformations take the form of replacing each class from the schema that is in the lower normal form with two or more new classes (and linking associations) that conform to the higher normal form. This transformation can also be encoded as a transformation over the UML meta-model, and thus be used as a generic transformation, applicable to any schema definition.

The combination of these two meta-level specifications is used to construct a schema transformation tool that takes as input a UML definition of a database system schema along with definitions of the functional dependencies, and produces as output a new UML schema that is conformant to one or more of the defined normal forms.

The rest of this paper describes: the UML profile necessary for modelling database system schemas; the meta-level encoding of the four normal forms in OCL; and finally the tool built to automate this process, including the meta-transformation used to convert schemas between normal forms. The paper concludes with an over view of the achievements and a look towards future work.

UML for Data Modelling

A number of suggestions have been made for a data modelling profile in UML [5] [6]. These profiles are well suited for the specification of models of data, both in relational and object-based databases. However, our objective, to support the normalization process, requires a more precise semantics including a set of well-formedness rules, neither of which is defined in the referenced work.

We define a set of stereotypes and tagged value that extend the UML meta-model to include the relational data modelling concepts, such as Keys, Functional Dependencies, and Table Definitions. Additionally, well-formedness rules are defined that specify the logical connections between the concepts. Subsequently we enhance the profile by formally defining the semantics of the concepts in terms of the 'instance' metamodel elements. Finally we add the normalization operations that can be used to verify if a class is in one or other of the normal forms.

The semantics and well-formedness rules are not of direct relevance to this paper, hence we simply include a summary of our UML profile, which defines the following stereotypes and tagged values:

Stereotype	UML meta-model baseClass
«Schema»	Package
«TableDefiniton»	Classifier
«Field»	Attribute
«FunctionalDependency»	Dependancy

Tag	Applies To
PK : Boolean	«Field»
CK : Set(Integer)	«Field»
FK : (Seq(«Field»), Seq(«Field»))	AssociationEnd

For examples of each of these extensions see Figure 2 (below), which contains a class diagram specifying part of a schema definition.

Schema

A schema is the collection of Table Definitions and inter-relationships that form the specification of the database system. We stereotype the UML Package element in order to identify that a particular package defines a database system. «Schema» packages contain «TableDefinition» classes and relationships between them.

Table Definition

A data class defines the type of its instances; it can therefore be taken to define the structure of a corresponding table where attributes and their types define column names and their domains. We introduce a stereotype «TableDefinition» that extends the concept of Classifier (from the UML meta-model).

Functional Dependency

To express a functional dependency we extend the 'Dependency' concept, introducing the stereotype «FunctionalDependency». The well-formedness rules constrain this to a dependency between attributes.

The notation for depicting a functional dependency between sets of attributes follows the 'dashed arrow' syntax specified for the dependency relationship in the UML standard (see section 2). However, avoid excess clutter on a diagram they can also be expressed textually as a pair of sets linked by an arrow '→', in line with the notation used in relational database modelling.

Candidate Key

We use a tagged value named 'CK' to indicate that an attribute is part of a candidate key for a Table Definition. An attribute may form part of more than one candidate key, hence we specify the data value of the tagged value to be a set of integers. Each integer indicates that the attribute is part of a different key.

The UML standard does not give any indication of syntax for the definition of tagged values on attribute stereotypes; we conform to the notation for tagged values defined on Classes, by adding the name and value within braces (e.g. '{CK = 1,2 }'), alongside the relevant attribute.

Primary Key

It is possible for a table definition to have more than one candidate key. In such a case the relational model has historically required that exactly one of those keys to be chosen as the *primary key*. To indicate the primary key we define an addition tagged value with a Boolean data type named 'PK'.

To depict that an attribute forms part of the primary key the name 'PK' is added to the defined tagged values for the attribute; e.g. '{PK, CK = 2,3 }'. (Boolean tagged values do not require an explicit value if being set to true.)

Foreign Key

A foreign Key is a set of attributes that map to the primary key of an associated Table Definition. Associations indicate the relationships between Table Definitions. Standard UML semantics assume the presence of a unique object identifier, and this is used to semantically distinguish between different objects. However, in Relational database semantics, the primary key forms this distinction. Thus, for an Association instance (Link) to be navigated it is necessary to know which attributes contain the foreign key values that identify the 'object' at the other end of an association. We indicate this by using a tagged value 'FK' on an AssociationEnd. This tag defines the attribute names that form the foreign key, e.g. {FK=p#,c#}.

Meta Encoding of the Normal Forms

The following operations are defined for the «TableDef» stereotype. They all return a Boolean value indicating whether or not the table is in one of the defined Normal Forms. For a Schema to be in one of the normal forms, all of the contained Table definitions must be in that normal form. It is necessary for a particular Table definition to contain the specification of Primary Keys, Functional Dependencies, and Candidate Keys in order for these operations to be successful.

is1NF() : Boolean

UML class diagrams with PKs are in 1NF. We define an OCL expression that fails if there are no PKs defined, i.e. a table is not in 1NF if there are no PKs defined.

```
result:
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
attributes->exists(a | a.taggedValue->contains(tg |
                               tg.name = 'PK' and tg.dataValue=true) );
```

is2NF() : Boolean

A table is in 2NF if it is in 1NF, plus every non-key attribute is functionally dependent on the full key.

```
result: is1NF and
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                          d.stereotype.name=`«FunctionalDependency»') in
Let pk = attributes->select(a|a.taggedValue->contains(tg |
                               tg.name = 'PK' and tg.dataValue=true)) in
Let npks = attributes - pk in
  npks->forAll( na |
         dependencies->exists( d | d.supplier->contains(na) and d.client = pk )
```

is3NF() : Boolean

Class must be in 2NF, plus every non-key attribute should be functionally dependent on the full key and nothing but the full key

```
result: is2NF and
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                          d.stereotype.name=`«FunctionalDependency»') in
Let pk = attributes->select(a|a.taggedValue->contains(tg |
                               tg.name = 'PK' and tg.dataValue=true)) in
Let npks = attributes - pk in
  npks->forAll( na |
     dependencies->select(d|d.supplier.contains(na))->forAll(d|d.client = pk) )
```

This constraint says, that for all dependencies containing a non-PK attribute in the client part, the supplier part of that dependency must be the PK.

isBCNF() : Boolean

A relation is in Boyce-Codd Normal Form (BCNF) if every determinant is a candidate key.

```
result:
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                          d.stereotype.name=`«FunctionalDependency»') in
Let cks = attributes.taggedValue->select(name=`CK').dataValue->
           flatten->collect( i |
              attributes->select( a | a.taggedValue->contains(tg |
                 tg.name=`CK' and tg.dataValue=i) ) in
  dependencies->forAll(d | cks->contains(d.client) )
```

Tool Support

Work at UKC is almost complete regarding the production of a tool (illustrated in Figure 1) to support the automatic normalisation process described in this paper. The tool accepts an XMI encoding of the UML schema, and a (possibly separate) XMI encoding of the functional dependencies. These are converted into a combined java object model of the schema definition using IBM's XMI framework [7] and additional libraries developed at UKC [8].

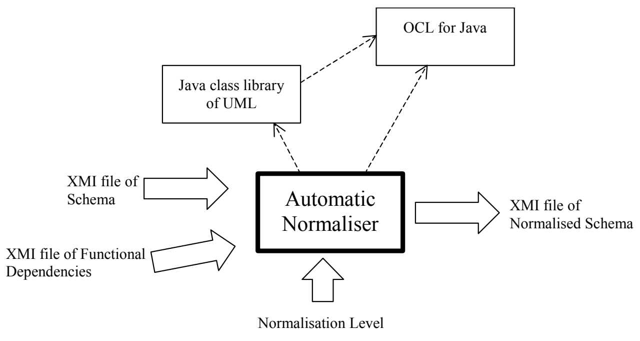
OCL for Java
Java class library of UML
XMI file of Schema
Automatic Normaliser
XMI file of Normalised Schema
XMI file of Functional Dependencies
Normalisation Level

Figure 1

The java object model is based on a set of java classes that encode the standard UML meta model. This set of classes has been automatically generated from an XMI encoding of the UML meta-model (using a simpler form of the generator to boot strap the process).

The generated java classes are built on an additional library that implements the OCL types and functions in such a way that OCL expressions can be evaluated over a java object model. The normalisation level of the java representation of schema can thus be deduced by evaluating the OCL expressions (defined above) using the OCL library.

A transformation algorithm is applied to the combined model, generating a new output model defining a schema that conforms to the required normal form. The core part of the transformation algorithm is the replacement of one class with two others as described below:

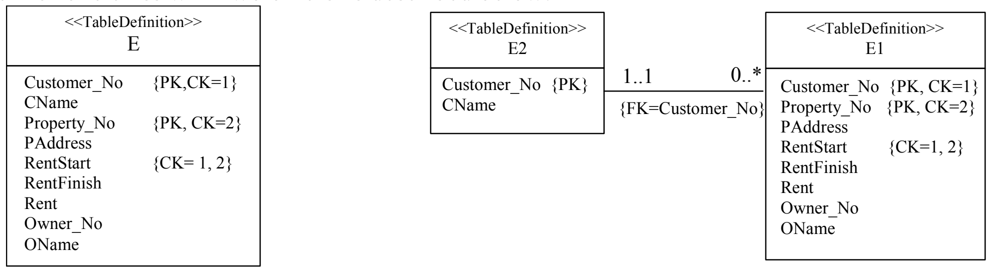

Figure 2

For a class ('E' from Figure 2) and a functional dependency A→B (e.g. {Customer_No} → {Cname}) we can perform a rewrite that replaces the original class with two new ones ('E1' and 'E2' from Figure 2). The re-writing rule for a class E into classes E1 and E2 is described as follows:

- Class E2 will contain the set of elements A (the determinant of the dependency we are decomposing) and any elements wholly dependent on A, which trivially includes B. The primary key of this class is the set of attributes A.
- Class E1 contains the elements not in class E2, except for elements of A. The set A remains in both classes, representing the association between them: as the foreign key of E1 linking to the primary key of E2. The primary key of class E1 is the same primary key of the original class E.

This algorithm is repeatedly applied to the classes and functional dependencies of the model, in order to transform the schema, as a whole, into the required normal form.

Conclusion

The work described in this paper has shown a method of automating the process of normalisation as defined within the context of database systems. The automation is achieved through the use of two declarative specifications over the UML meta-model.

1. The use of OCL declarations over a data-modelling profile of UML to encode the definitions of the four normal forms (1NF, 2NF, 3NF and BCNF).
2. The definition of a graph rewriting rule to specify the transformation steps required for converting a data model from one normal form to a higher normal form.

These specifications have been used to construct a tool that implements the transformation thus automating the normalisation process.

We plan to formalise the transformation process as a specification over the UML meta-model using graph transformations [9] or some other appropriate technique. We also plan to extend the tool support to include automatic generation of SQL Schema definitions from the UML models.

References

[1] OMG, "The Unified Modeling Language Version 1.4," Object Management Group formal/01-09-67, 2001.
[2] E. J. Naiburg and R. A. Maksimchuk, *UML for Database Design*: Addison Wesley, ISBN 0-201-72163-5, 2001.
[3] J. Paredaens, P. De-Bra, M. Gyssens, and D. Van-Gucht, *The Structure of the Relational Database Model*: Springer-Verlag, ISBN 3-540-13714-9, 1989.
[4] C. J. Date, *An Introduction to Database Systems (Introduction to Database Systems 7th Ed)*: Addison Wesley Publishing Company, ISBN 0201385902, 1999.
[5] S. Ambler, "Persistence Modeling in the UML," 1999, http://www.sdmagazine.com
[6] Rational, "The UML and Data modelling," 2000, www.rational.com
[7] IBM, "alphaWorks XMI Framework," 2001, http://www.alphaworks.ibm.com/tech/xmiframework
[8] D. H. Akehurst, "OCL for Java," 2002, http://www.cs.ukc.ac.uk/people/staff/dha/xInterests/UMLandOCL/
[9] *Handbook of graph grammars and computing by graph transformation*, Vol. 1, Foundations, edited by G. Rozenberg. World Scientific, Singapore, 1997.