Annex A: Additional Examples

(normative)

A.1 Relations Examples

A.1.1 UML to RDBMS Mapping

A.1.1.1 Overview

This example maps persistent classes of a simple UML model to tables of a simple RDBMS model. A persistent class maps to a table, a primary key and an identifying column. Attributes of the persistent class map to columns of the table: an attribute of a primitive datatype maps to a single column; an attribute of a complex data type maps to a set of columns corresponding to its exploded set of primitive datatype attributes; attributes inherited from the class hierarchy are also mapped to the columns of the table. An association between two persistent classes maps to a foreign key relationship between the corresponding tables.

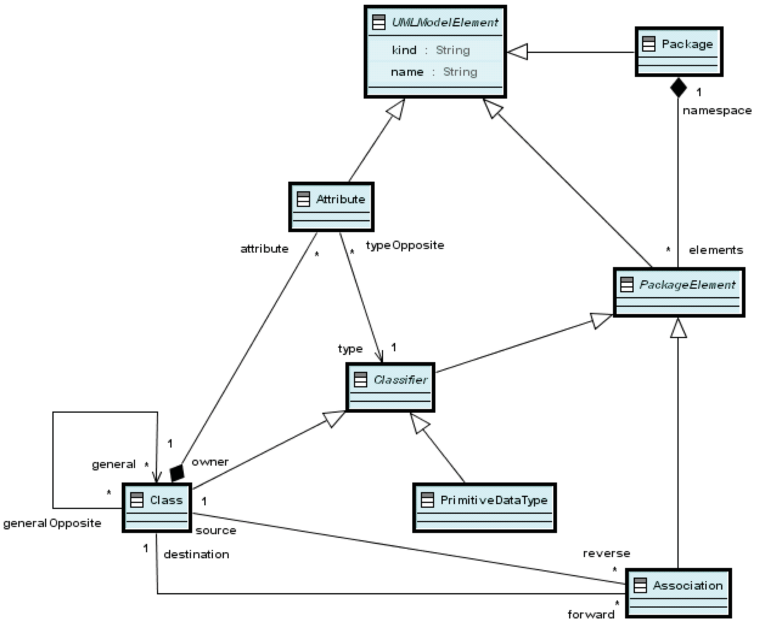

Figure A.1 - Simple UML Metamodel

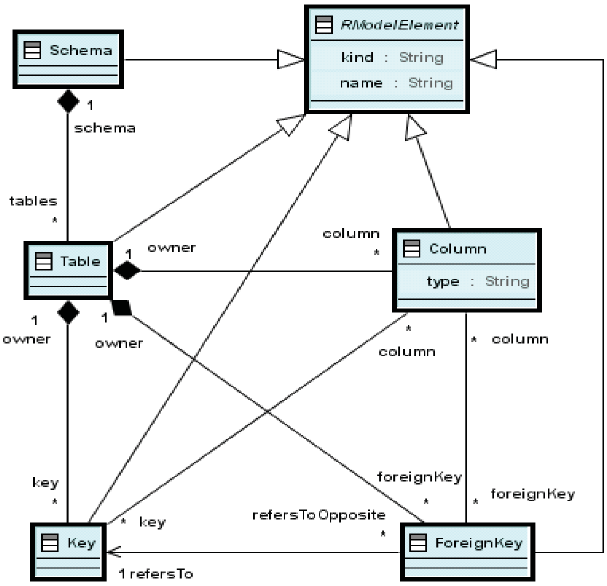

Figure A.2 - Simple RDBMS Metamodel

UML to RDBMS mapping in textual syntax

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
key Table (name, schema);
key Column (name, owner);   // owner:Table opposite column:Column
key Key (name, owner); // key of class ëKeyí;
                       // owner:Table opposite key:Key

top relation PackageToSchema   // map each package to a schema
{
   pn: String;

   checkonly domain uml p:Package {name=pn};
   enforce domain rdbms s:Schema {name=pn};
}

top relation ClassToTable   // map each persistent class to a table
{
   cn, prefix: String;

   checkonly domain uml c:Class {namespace=p:Package {},
                                kind='Persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
          column=cl:Column {name=cn+'_tid', type='NUMBER'},
          key=k:Key {name=cn+'_pk', column=cl}};
   when {
      PackageToSchema(p, s);
   }
   where {
      prefix = '';
      AttributeToColumn(c, t, prefix);
   }
}

relation AttributeToColumn
{
   checkonly domain uml c:Class {};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      PrimitiveAttributeToColumn(c, t, prefix);
      ComplexAttributeToColumn(c, t, prefix);
      SuperAttributeToColumn(c, t, prefix);
   }
}

relation PrimitiveAttributeToColumn
{
   an, pn, cn, sqltype: String;

   checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                type=p:PrimitiveDataType {name=pn}}};
   enforce domain rdbms t:Table {column=cl:Column {name=cn,
                                type=sqltype}};
   primitive domain prefix:String;
   where {
      cn = if (prefix = '') then an else prefix+'_'+an endif;
      sqltype = PrimitiveTypeToSqlType(pn);
   }
}

relation ComplexAttributeToColumn
{
   an, newPrefix: String;

   checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                type=tc:Class {}}};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      newPrefix = prefix+'_'+an;
      AttributeToColumn(tc, t, newPrefix);
   }
}

relation SuperAttributeToColumn
{
   checkonly domain uml c:Class {general=sc:Class {}};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      AttributeToColumn(sc, t, prefix);
   }
}

// map each association between persistent classes to a foreign key
top relation AssocToFKey
{
   srcTbl, destTbl: Table;
   pKey: Key;
   an, scn, dcn, fkn, fcn: String;

   checkonly domain uml a:Association {namespace=p:Package {},
            name=an,
            source=sc:Class {kind='Persistent',name=scn},
            destination=dc:Class {kind='Persistent',name=dcn}
         };
   enforce domain rdbms fk:ForeignKey {schema=s:Schema {},
            name=fkn,
            owner=srcTbl,
            column=fc:Column {name=fcn,type='NUMBER',owner=srcTbl},
            refersTo=pKey
         };
   when {   /* when refers to pre-condition */
      PackageToSchema(p, s);
      ClassToTable(sc, srcTbl);
      ClassToTable(dc, destTbl);
      pKey = destTbl.key;
   }
   where {
      fkn=scn+'_'+an+'_'+dcn;
      fcn=fkn+'_tid';
   }
}

function PrimitiveTypeToSqlType(primitiveTpe:String):String
{
   if (primitiveType='INTEGER')
   then 'NUMBER'
   else if (primitiveType='BOOLEAN')
      then 'BOOLEAN'
      else 'VARCHAR'
      endif
   endif;
}

}
```

UML to RDBMS mapping in graphical syntax

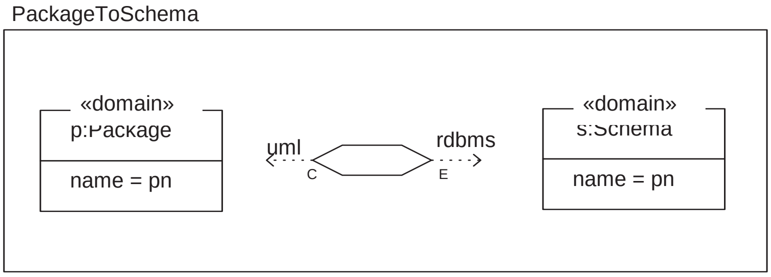

Figure A.3 - PackageToSchema relation

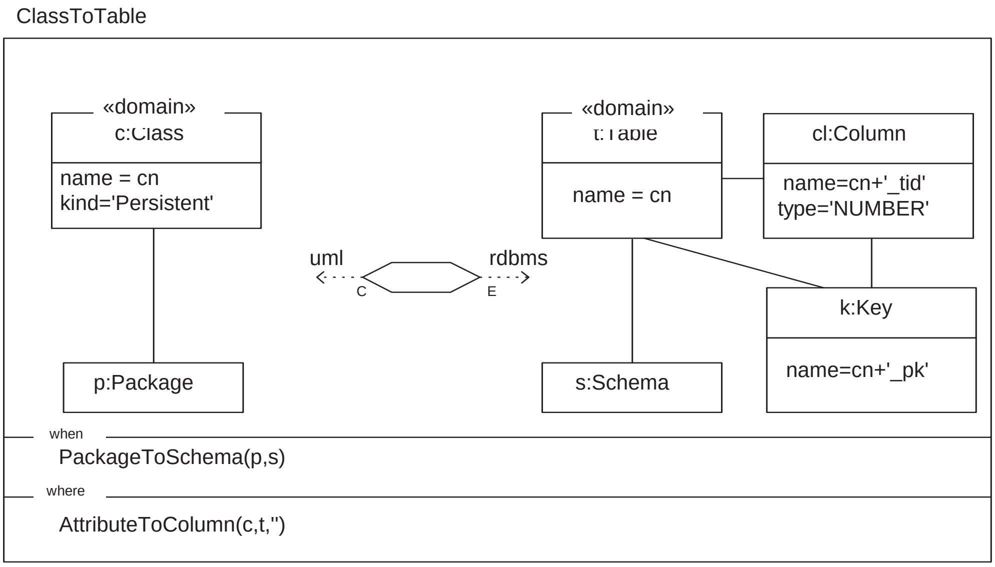

Figure A.4 - ClassToTable relation

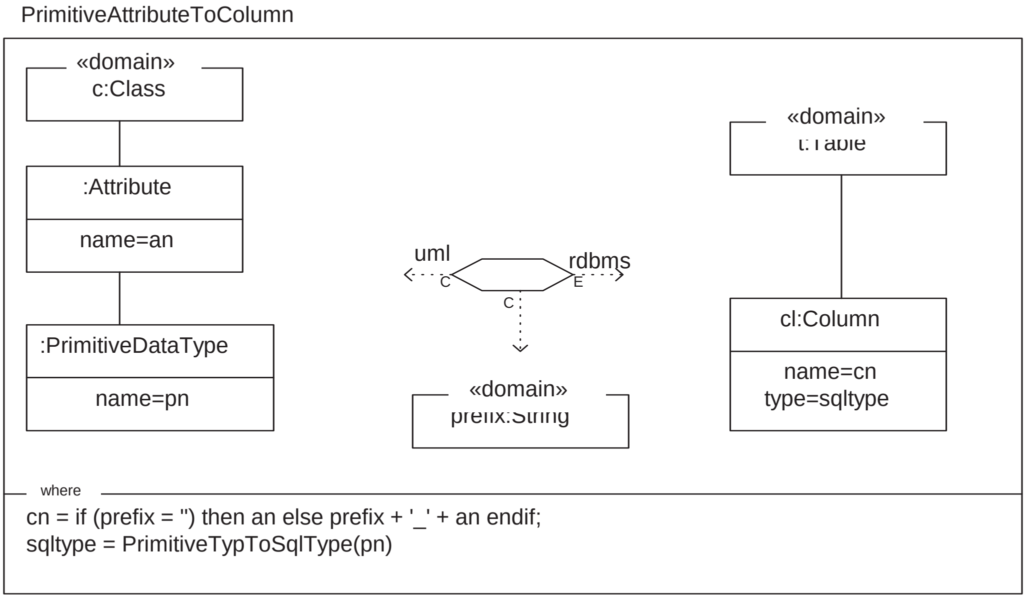

Figure A.5 - PrimitiveAttributeToColumn relation

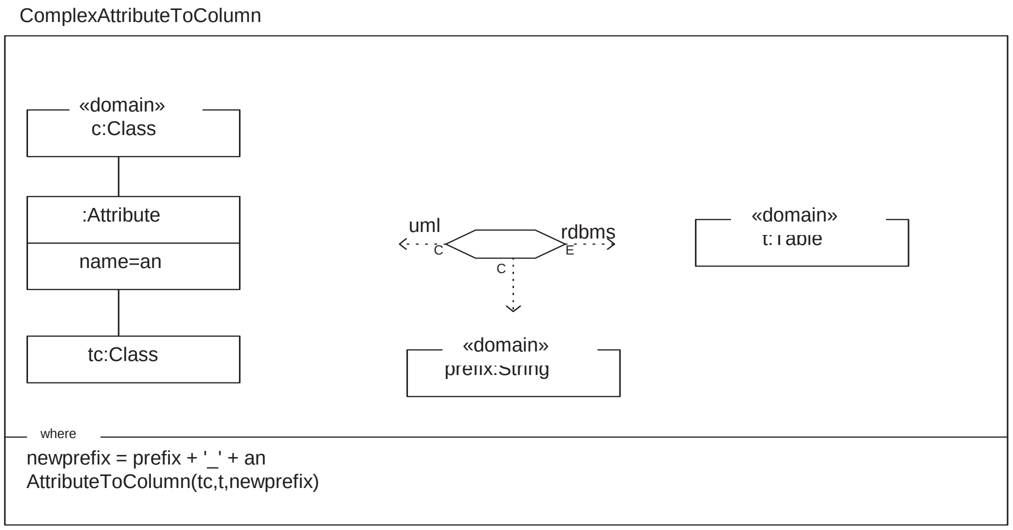

Figure A.6 - ComplexAttributeToColumn relation

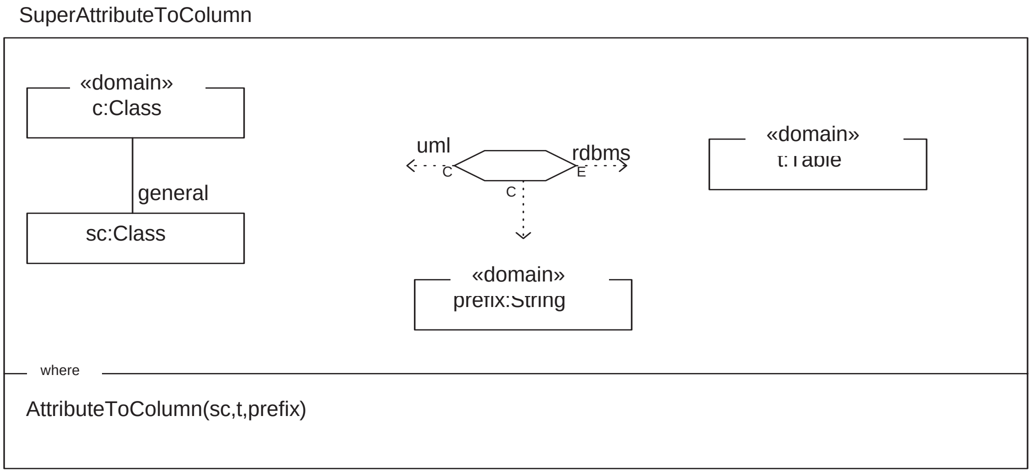

Figure A.7 - SuperAttributeToColumn relation

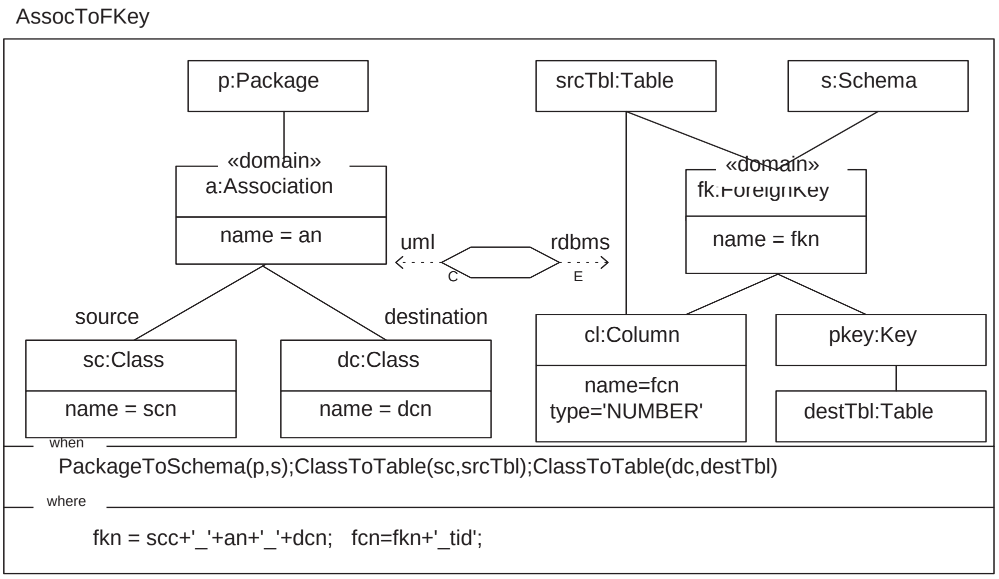

Figure A.8 - AssocToFKey relation

A.2 Operational Mapping Examples

A.2.1 Book To Publication example

```
metamodel BOOK {
  class Book {title: String; composes chapters: Chapter [*];}
  class Chapter {title : String; nbPages : Integer;}
}

metamodel PUB {
  class Publication {title : String; nbPages : Integer;}
}

transformation Book2Publication(in bookModel:BOOK,out pubModel:PUB);

main() {
  bookModel->objectsOfType(Book)->map book_to_publication();
}

mapping Class::book_to_publication () : Publication {
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

A.2.2 Encapsulation example

```
-- This QVT definition performs an in place transformation on
-- a UML class-diagram model by privatizing the attributes and
-- creating accessor methods
```

```
modeltype UML uses "omg.org.uml14";

transformation Encapsulation(inout classModel:UML);

// Indicating that UML1.4 Name type is to be treated as a String
tag "TypeEquivalence" UML::Name = "String";

-- entry point: selects the packages and applies the transformation
-- on each package

main() {
  classModel.objectsOfType(Package)
     ->map encapsulateAttributesInPackageClasses();
}
-- Applies the transformation to each class of the package

mapping inout Package::encapsulateAttributesInPackageClasses () {
  init {self.ownedElement->map encapsulateAttributesInClass();}
}

-- Performs the encapsulation for each attribute of the class
-- The initialization section is used to retrieve the list of attributes
-- The population section is used to add the two accessor operations
-- The end section is used to privatize each attribute

mapping inout Class::encapsulateAttributesInClass ()
{
  init { var attrs := self.feature[Attribute];}
  operation := { -- assignment with additive semantics
      attrs->object(a) Operation {
           name := "get_" + self.name.firstToUpper();
           visibility := "public";
           type := a.type;
         };
      attrs->object(a) Operation {
            name := "set_" + self.name.firstToUpper();
            visibility := "public";
            parameter := object Parameter {
               name := 'a_'+ self.name.firstToUpper();
               kind := "in";
               type := a.type;};
            };
         };
  end { attrs->map privatizeAttribute();}
}

-- in place privatization of the attribute

mapping inout Attribute::privatizeAttribute () {
   visibility := "private";
}
```

A.2.3 Uml to Rdbms

The metamodels used here are the same metamodels used for the relational version given in Appendix A.1.1. We provide below their definition using the concrete syntax for metamodels. Note we are assuming that all multi-valued associations are ordered.

```
metamodel SimpleUml {

 abstract class UMLModelElement {
   kind : String;
   name : String;
 }

 class Package extends UMLModelElement {
   composes elements : PackageElement [*] ordered opposites namespace [1];
 }

 abstract class PackageElement extends UMLModelElement {
 }

 class Classifier extends PackageElement {
 }

 class Attribute extends UMLModelElement {
    references type : Classifier [1];
 }

 class Class extends Classifier {
    composes attribute : Attribute [*] ordered opposites owner [1];
    references general : Classifier [*] ordered;
  }

 class Association extends PackageElement {
    source : Class [1] opposites reverse [*];
    destination : Class [1] opposites forward [*];
 }

 class PrimitiveDataType extends Classifier {
 }

}

metamodel SimpleRdbms {

 abstract class RModelElement {
   kind : String;
   name : String;
 }

 class Schema extends RModelElement {
   composes tables : Table [*] ordered opposites schema [1];
 }

 class Table extends RModelElement {
    composes column : Column [*] ordered opposites owner[1];
    composes _key : Key [*] ordered  opposites owner[1];
        // '_key' is an automatic alias for 'key'
```

```
      composes foreignKey : ForeignKey [*] ordered opposites owner[1];
   }

   class Column extends RModelElement {
     type : String;
   }

   class Key extends RModelElement {
      references column : Column [*] ordered opposites _key [*];
   }

   class ForeignKey extends RModelElement {
      references refersTo : Key [1];
      references column : Column [*] ordered opposites foreignKey [*];
   }

  }
```

Below the transformation definition

```
transformation Uml2Rdb(in srcModel:UML,out dest:RDBMS);

-- Aliases to avoid name conflicts with keywords
tag "alias" RDBMS::Table::key_ = "key";
-- defining intermediate data to reference leaf attributes that may
-- appear when struct data types are used
intermediate class LeafAttribute {
  name:String;
  kind:String;
  attr:UML::Attribute;
};
intermediate property UML::Class::leafAttributes : Sequence(LeafAttribute);

-- defining specific helpers

query UML::Association::isPersistent() : Boolean {
    result = (self.source.kind='persistent' and self.destination.kind='persistent');
}

-- defining the default entry point for the module
-- first the tables are created from classes, then the tables are
-- updated with the foreign keys implied by the associations

main() {
  srcModel.objects()[Class]->map class2table(); -- first pass
  srcModel.objects()[Association]->map asso2table(); -- second pass
}

-- maps a class to a table, with a column per flattened leaf attribute

mapping Class::class2table () : Table
  when {self.kind='persistent';}
{
  init { -- performs any needed intialization
    self.leafAttributes := self.attribute
      ->map attr2LeafAttrs("",""); // ->flatten();
  }
  -- population section for the table
  name := 't_' + self.name;
  column := self.leafAttributes->map leafAttr2OrdinaryColumn("");
  key_ := object Key {  -- nested population section for a 'Key'
           name := 'k_'+ self.name; column := result.column[kind='primary'];
         };
}

-- Mapping that creates the intermediate leaf attributes data.

mapping Attribute::attr2LeafAttrs (in prefix:String,in pkind:String)
: Sequence(LeafAttribute) {
  init {
    var k := if pkind="" then self.kind else pkind endif;
    result :=
       if self.type.isKindOf(PrimitiveDataType)
       then -- creates a sequence with a LeafAttribute instance
         Sequence {
           object LeafAttribute {attr:=self;name:=prefix+self.name;kind:=k;}
         }
       else self.type.asType(Class).attribute
         ->map attr2LeafAttrs(self.name+"_",k)->asSequence()
       endif;
  }
}

-- Mapping that creates an ordinary column from a leaf attribute

mapping LeafAttribute::leafAttr2OrdinaryColumn (in prefix:String): Column {
  name := prefix+self.name;
  kind := self.kind;
  type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif;
}

-- mapping to update a Table with new columns of foreign keys

mapping Association::asso2table() : Table
  when {self.isPersistent();}
{
  init {result := self.destination.resolveone(Table);}
  foreignKey := self.map asso2ForeignKey();
  column := result.foreignKey->column ;
}

-- mapping to build the foreign keys

mapping Association::asso2ForeignKey() : ForeignKey {
   name := 'f_' + self.name;
   refersTo := self.source.resolveone(Table).key_;
   column := self.source.leafAttributes[kind='primary']
              ->map leafAttr2ForeignColumn(self.source.name+'_');
}

-- Mapping to create a Foreign key from a leaf attributes
-- Inheriting of leafAttr2OrdinaryColumn has the effect to call the
-- inherited rule before entering the property population section

mapping LeafAttribute::leafAttr2ForeignColumn (in prefix:String) : Column
  inherits leafAttr2OrdinaryColumn {
     kind := "foreign";
}
```

A.2.4 SPEM UML Profile to SPEM metamodel

```
modeltype UML uses "omg.org.spem_umlprofile";
modeltype SPEM uses "omg.org.spem_metamodel";
transformation SpemProfile2Metamodel(in umlmodel:UML,out spemmodel:SPEM);

query UML::isStereotypedBy(stereotypeName:String) : Boolean;
query UML::Classifier::getOppositeAends() : Set(UML::AssociationEnd);

main () {
  -- first pass: create all the SPEM elements from UML elements
  umlmodel.rootobjects()[UML::Model]->map createDefaultPackage();
  -- second pass: add the dependencies beyween SPEM elements
  umlmodel.objects[UML::UseCase]->map addDependenciesInWorkDefinition();
}

mapping UML::Package::createDefaultPackage () : SPEM::Package {
  name := self.name;
  ownedElement := self.ownedElement->map createModelElement();
}

mapping UML::Package::createProcessComponent () : SPEM::ProcessComponent
  inherits createDefaultPackage
  when {self.isStereotypedBy("ProcessComponent");}
  {}

mapping UML::Package::createDiscipline () : SPEM::Discipline
  inherits createDefaultPackage
  when {self.isStereotypedBy("Discipline");}
  {}


mapping UML::ModelElement::createModelElement () : SPEM::ModelElement
  disjuncts
    createProcessRole, createWorkDefinition,
    createProcessComponent, createDiscipline
  {}

mapping UML::UseCase::createWorkDefinition () : SPEM::WorkDefinition
  disjuncts
    createLifeCycle, createPhase, createIteration,
    createActivity, createCompositeWorkDefinition
  {}


mapping UML::Actor::createProcessRole () : SPEM::ProcessRole
  when {self.isStereotypedBy("ProcessRole");}
  {}

-- rule to create the default process performer singleton
mapping createOrRetrieveDefaultPerformer () : SPEM::ProcessPerformer {
  init {
    result := resolveoneByRule(createOrRetrieveDefaultPerformer);
    if result then return endif;
  }
  name := "ProcessPerformer";
}

mapping abstract UML::UseCase::createCommonWorkDefinition ()
```

```
 : SPEM::WorkDefinition
{
   name := self.name;
   constraint := {
      self.constraint[isStereotypedBy("precondition")]
         ->map createPrecondition();
      self.constraint[isStereotypedBy("goal")]->map createGoal();
   };
}

mapping UML::UseCase::createActivity () : SPEM::WorkDefinition
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Activity");}
  {}

mapping UML::UseCase::createPhase () : SPEM::Phase
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Phase");}
  {}

mapping UML::UseCase::createIteration () : SPEM::Iteration
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("Iteration");}
  {}

mapping UML::UseCase::createLifeCycle () : SPEM::LifeCycle
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("LifeCycle");}
  {}

mapping UML::UseCase::createCompositeWorkDefinition () : SPEM::WorkDefinition
  inherits createCommonWorkDefinition
  when {self.isStereotypedBy("WorkDefinition");}
  {}

mapping UML::Constraint::createPrecondition () : SPEM::Precondition {
  body := self.body;
}

mapping UML::Constraint::createGoal () : SPEM::Goal {
  body := self.body;
}

mapping UML::UseCase::addDependenciesInWorkDefinition ()
 : SPEM::WorkDefinition
 merging addDependenciesInActivity
{
  init {
    result := self.resolveone(WorkDefinition);
    var performers
      := self.getOppositeAends()[i|i.association
           [isStereotypedBy("perform")]->notEmpty()];
    assert (not performers->size()>1)
       with log("A unique performer is allowed",self);
  }
  subWork := self.clientDependency[*includes].supplier
     ->resolveone(WorkDefinition);
  performer := if performers then performers->first()
               else createOrRetrieveDefaultPerformer() endif;
}
```

```
mapping UseCase::addDependenciesInActivity () : WorkDefinition
  when {self.stereotypedBy("Activity");}
  {
    assistant := self.getOppositeAends()[i|i.association
       [a|a.isStereotypedBy("assist")]->notEmpty()]->resolve();
  }
```

A.3 Core Examples

A.3.1 UML to RDBMS Mapping

This example expresses the same transformation semantics, and uses the same metamodels shown in the Relations Examples in Section A.1.1.

```
-- A Transformation definition from SimpleUML to SimpleRDBMS
module UmlRdbmsTransformation imports SimpleUML, SimpleRDBMS {

    transformation umlRdbms {
        uml imports SimpleUML;
        rdbms imports SimpleRDBMS;
    }

    -- Package and Schema mapping
    class PackageToSchema {
        composite classesToTables : Set(ClassToTable) opposites owner;
        composite primitivesToNames : Set(PrimitiveToName) opposites owner;
        name : String;
        -- uml
        umlPackage : Package;
        -- rdbms
        schema : Schema;
    }

    map packageToSchema in umlRdbms {
        uml () {
            p:Package
        }
        rdbms () {
            s:Schema
        }
        where () {
            p2s:PackageToSchema|
            p2s.umlPackage = p;
            p2s.schema = s;
        }
        map {
            where () {
                p2s.name := p.name;
                p2s.name := s.name;
                p.name := p2s.name;
                s.name := p2s.name;
            }
        }
    }
```

```
-- Primitive data type marshaling
class PrimitiveToName {
    owner : PackageToSchema opposites primitivesToNames;
    name : String;
    -- uml
    primitive : PrimitiveDataType;
    -- rdbms
    typeName : String;
}

map primitiveToName in umlRdbms {
    uml (p:Package) {
        prim:PrimitiveDataType|
        prim.owner = p;
    }
    check enforce rdbms () {
        sqlType:String
    }
    where (p2s:PackageToSchema| p2s.umlPackage=p) {
        realize p2n:PrimitiveToName|
        p2n.owner := p2s;
        p2n.primitive := prim;
        p2n.typeName := sqlType;
    }
    map {
        where () {
            p2n.name := prim.name + '2' + sqlType;
        }
    }
}

map integerToNumber in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'Integer';
    }
    check enforce rdbms () {
        sqlType := 'NUMBER';
    }
}

map booleanToBoolean in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'Boolean';
    }
    check enforce rdbms () {
        sqlType := 'BOOLEAN';
    }
}

map stringToVarchar in umlRdbms refines primitiveToName {
    uml () {
        prim.name = 'String';
    }
    check enforce rdbms () {
        sqlType := 'VARCHAR';
    }
}

-- utility functions for flattening
```

```
map flattening in umlRdbms {
    getAllSupers(cls : Class) : Set(Class) {
        cls.general->collect(gen|self.getAllSupers(gen))->
            including(cls)->asSet()
    }
    getAllAttributes(cls : Class) : Set(Attribute) {
        getAllSupers(cls).attribute
    }
    getAllForwards(cls : Class) : Set(Association) {
        getAllSupers(cls).forward
    }
}

-- Class and Table mapping
class ClassToTable extends FromAttributeOwner, ToColumn {
    owner : PackageToSchema opposites classesToTables;
    composite associationToForeignKeys :
        OrderedSet(AssociationToForeignKey) opposites owner;
    name : String;
    -- uml
    umlClass : Class;
    -- rdbms
    table : Table;
    primaryKey : Key;
}

map classToTable in umlRdbms {
    check enforce uml (p:Package) {
        realize c:Class|
        c.kind := 'persistent';
        c.namespace := p;
    }
    check enforce rdms (s:Schema) {
        realize t:Table|
        t.kind <> 'meta';
        default t.kind := 'base';
        t.schema := s;
    }
    where (p2s:Package2Schema| p2s.umlPackage=p; p2s.schema=s;) {
        realize c2t:ClassToTable|
        c2t.owner := p2s;
        c2t.umlClass := c;
        c2t.table := t;
    }
    map {
        where () {
            c2t.name := c.name;
            c2t.name := t.name;
            c.name := c2t.name;
            t.name := c2t.name;
        }
    }
    map {
        check enforce rdbms () {
            realize pk:Key,
            realize pc:Column|
            pk.owner := t;
            pk.kind := 'primary';
            pc.owner := t;
            pc.key->includes(pk);
            default pc.key := Set(Key){pk};
            default pc.type := 'NUMBER';
        }
        where () {
            c2t.primaryKey := pk;
            c2t.column := pc;
        }
        map {
            check enforce rdbms () {
                pc.name := t.name+'_tid';
                pk.name := t.name+'_pk';
            }
        }
    }
}

-- Association and ForeignKey mapping
class AssociationToForeignKey extends ToColumn {
    referenced : ClassToTable;
    owner : ClassToTable opposites associationToForeignKeys;
    name : String;
    -- uml
    association : Association;
    -- rdbms
    foreignKey : ForeignKey;
}

map associationToForeignKey in umlRdbms refines flattening {
    check enforce uml (p:Package, sc:Class, dc:Class| sc.namespace = p;) {
        realize a:Association|
        getAllForwards(sc)->includes(a);
        default a.source := sc;
        getAllSupers(dc)->includes(a.destination);
        default a.destination := dc;
        default a.namespace := p;
    }
    check enforce rdbms (s:Schema, st:Table, dt:Table, rk:Key|
        st.schema = s;
        rk.owner = dt;
        rk.kind = 'primary';
    ) {
        realize fk:ForeignKey,
        realize fc:Column|
        fk.owner := st;
        fc.owner := st;
        fk.refersTo := rk;
        fc.foreignKey->includes(fk);
        default fc.foreignKey := Set(ForeignKey){fk};
    }
    where (p2s:PackageToSchema, sc2t:ClassToTable, dc2t:ClassToTable|
        sc2t.owner = p2s;
        p2s.umlPackage = p;
        p2s.schema = s;
        sc2t.table = st;
        dc2t.table = dt;
        sc2t.umlClass = sc;
        dc2t.umlClass = dc;
    ) {
        realize a2f:AssociationToForeignKey|
        a2f.owner := sc2t;
        a2f.referenced := dc2t;
        a2f.association := a;
        a2f.foreignKey := fk;
        a2f.column := fc;
    }
    map {
        where () {
            a2f.name := if a.destination=dc and a.source=sc
                        then a.name
                        else if a.destination<>dc and a.source=sc
                        then dc.name+'_'+a.name
                        else if a.destination=dc and a.source<>sc
                        then a.name+'_'+sc.name
                        else dc.name+'_'+a.name+'_'+sc.name
                        endif endif endif;
            a.name := if a.destination=dc and a.source=sc
                      then a2f.name
                      else a.name
                      endif;
            fk.name := name;
            name := fk.name;
            fc.name := name+'_tid';
        }
    }
    map {
        where () {
            fc.type := rk.column->first().type;
        }
    }
}

-- attribute mapping
abstract class FromAttributeOwner {
    composite fromAttributes : Set(FromAttribute) opposites owner;
}

abstract class FromAttribute {
    name : String;
    kind : String;
    owner : FromAttributeOwner opposites fromAttributes;
    leafs : Set(AttributeToColumn);
    -- uml
    attribute : Attribute to uml;
}

abstract class ToColumn {
    -- rdbms
    column : Column;
}

class NonLeafAttribute extends FromAttributeOwner, FromAttribute {
    leafs := fromAttributes.leafs;
}

class AttributeToColumn extends FromAttribute, ToColumn {
    type : PrimitiveToName;
}

map attributes in umlRdbms refines flattening {
    check enforce uml (c:Class) {
        realize a:Attribute|
        default a.owner := c;
        getAllAttributes(c)->includes(a);
    }
    where (fao:FromAttributeOwner) {
        fa : FromAttribute|
        fa.attribute := a;
        fa.owner := fao;
    }
    map {
        where {
            fa.kind := a.kind;
            a.kind := fa.kind;
        }
    }
}

map classAttributes in umlRdbms refines attributes {
    where (fao:ClassToTable| fao.umlClass=c) {}
    map {
        where {
            fa.name := a.name;
            a.name := fa.name;
        }
    }
}

map primitiveAttribute in umlRdbms refines attributes {
    check enforce uml (t:PrimitiveDataType) {
        a.type := t;
    }
    where (p2n:PrimitiveToName|p2n.primitive=t) {
        realize fa:AttributeToColumn|
        fa.type := p2n;
    }
    map {
        where {
            fa.leafs := Set(AttributeToColumn) {fa};
        }
    }
}

map complexAttributeAttributes in umlRdbms refines attributes {
    check uml (ca:Attribute|ca.type=c) {}
    where (fao:NonLeafAttribute | fao.attribute=ca) {}
    map {
        where {
            fa.name := fao.name+'_'+a.name;
        }
    }
}

map complexAttribute in umlRdbms refines attributes {
    check uml (t:Class) {
        a.type = t;
    }
    where () {
        realize fa:NonLeafAttribute
    }
    map {
            where {
                fa.leafs := fromAttributes.leafs;
            }
        }
    }

    map classPrimitiveAttributes in umlRdbms refines classAttributes, primitiveAttribute {}

    map classComplexAttributes in umlRdbms refines classAttributes, complexAttribute {}

    map complexAttributePrimitiveAttributes in umlRdbms refines complexAttributeAttributes,
primitiveAttribute {}

     map complexAttributeComplexAttributes in umlRdbms refines complexAttributeAttributes,
complexAttribute {}

    -- column mapping
    map attributeColumns in umlRdbms {
        check enforce rdbms (t:Table) {
            realize c:Column|
            c.owner := t;
            c.key->size()=0;
            c.foreignKey->size()=0;
        }
        where (c2t:ClassToTable| c2t.table=t;) {
            realize a2c:AttributeToColumn|
            a2c.column := c;
            c2t.fromAttribute.leafs->include(a2c);
            default a2c.owner := c2t;
        }
        map {
            check enforce rdbms (ct:String) {
                c.type := ct;
            }
            where (p2n:PrimitiveToName) {
                a2c.type := p2n;
                p2n.typeName := ct;
            }
        }
        map {
            where () {
                c.name := a2c.name;
                a2c.name := c.name;
            }
        }
        map {
            where () {
                c.kind := a2c.kind;
                a2c.kind := c.kind;
            }
        }
    }


} -- end of module UmlRdbmsTransformation
```