1.1. Example: Geometrical Transformations

This transformation example describes geometrical transformations. Input of the transformation are geometrical operations (rotate, move, explode) and geometrical data (based on a simplified DXF metamodel). The output of the transformation is the geometrical data (based on the same simplified DXF metamodel) that has been transformed according to the geometrical operations.

DXF™, Drawing eXchange Format, is the native vector file format of Autodesk's AutoCAD CAD application. DXF features include support for 3D objects, curves, text and associative dimensioning.

Particularity of the transformations is that it has two input metamodels and that one of the input metamodels is the same as the output metamodel.

1.1.1. Metamodels

The simplified DXF metamodel has an aggregation hierarchy. It consists of one DXF element that contains a number of Meshes (a 3D object based on a set of contiguous lines) which on their turn consist of a number of Points (two consecutive points represent a line). Meshes and Points have names. Points have three coordinates, namely x, y and z.

![Figure 1 DXF Metamodel](image)

The following km3 file describes the DXF metamodel:

```java
package DXF {

    class DXF {
        reference meshes[*] ordered container : Mesh;
    }

    class Mesh {
        attribute name : String;
        reference points[*] ordered container : Point oppositeOf mesh;
    }

    class Point {
        attribute name : String;
        reference mesh : Mesh oppositeOf points;
        attribute x : Double;
        attribute y : Double;
        attribute z : Double;
    }
}

package PrimitiveTypes {
```
The GeoTrans metamodel has been designed to describe geometrical operations that can be executed in sequence or parallel. The execution order is described by the means of a tree structure. Geometrical operations are executed in sequence if the output of an operation is the input of the follow-up operation. Parallel operations are independent of each other.

The GeoTrans metamodel has an aggregation hierarchy. GeoTransfos have GeoTransfo. GeoTransfo can be put in a tree structure through subGeoTransfos and supergeoTransfo. GeoTransfo may have Params (parameters).

Each model must have exactly one GeoTransfos instance in which all GeoTransfo are directly or indirectly contained. GeoTransfo represents a geometrical operation. The actual operation is determined from its name attribute. Freeze means that the output of the operation has to be captured (added to the output model). If for several operations the freeze attribute is set true, the result is a superposition of movements like in nocturnal photos with long light exposure. You may also use this feature to build a complex figure with a repetitive pattern, e.g. you build a hexagon from a single line that you turn and freeze six times. A Param has a name, a Double value param and a back reference to the operation it belongs to.

There are four basic GeoTransfo instances:

The instance Rotation with the name rotate and three Param instances with the names rotationX, rotationY, rotationZ. The Param attribute of a Param instance contains the corresponding double value (e.g. 10.0 degrees for rotationX).

The instance MovementForward with the name moveForward and three Param instances with the names x, y, z. The Param attribute contains the corresponding double values (e.g. 10.0 units to be moved forward in x direction).

The instance MovementBackward with the name moveBackward and three Param instances with the names x, y, z. The Param attribute contains the corresponding double values.

The instance Explosion with the name explode and one Param instance with the name factor which contains in param the corresponding double value (the explosion factor).

Please note that there are different ways to express metamodels. This metamodel is historically grown.

```
datatype Double;
datatype String;
}
```

```
package GeoTrans {

class GeoTransfos {
  attribute name : String;
}
```

![Figure 2 GeoTrans Metamodel](image)
class GeoTransfo {
    attribute name : String;
    attribute freeze : Boolean;
    reference subGeoTransfos[*] ordered container : GeoTransfo oppositeOf superGeoTransfo;
    reference superGeoTransfo : GeoTransfo oppositeOf subGeoTransfos;
    reference params[*] ordered container : Param oppositeOf geoTransfo;
}

class Param {
    attribute name : String;
    attribute param : Double;
    reference geoTransfo : GeoTransfo oppositeOf params;
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Double;
    datatype String;
}

1.1.2. Rules Specification

In the following the rules for geometrical transformations will be described.

Rules for the execution order of geometrical operations in general:

- A geometrical operation (GeoTransfo) that has no superordinate operation (supergeoTransfo) has to be executed with the input data of the input model as entry and to forward the result to subordinate operations (subGeoTransfos). If freeze is true the result has to be added to the output model.

- A geometrical operation (GeoTransfo) that has a superordinate operation (supergeoTransfo) has to be executed with the output data of the superordinate operation as entry and to forward the result to subordinate operations (subGeoTransfos). If freeze is true the result has to be added to the output model.

Rules for executing specific operations:

- A Rotation operation has to rotate all points of each Mesh according to the rotationX, rotationY and rotationZ values. The rotation may be around the Mesh's gravity center (average of all points of a Mesh) or around the zero point (x=y=z=0). The outcome of the rotation depends on the rotation point.

- A MovementForward operation has to move all Points by adding the x, y, z values of the operation to the coordinates of each point. Negative input values lead to a backward movement.
• A MovementBackward operation has to move all *Points* by subtracting the x, y, z values of the operation from the coordinates of each point. Negative input values lead to a forward movement.

• An Explosion operation has to calculate a movement for each *Mesh*. The further the *Mesh* is away from the explosion point (x=y=z=0) the weaker the influence. The movement of the explosion (movementX, movementY, movementZ) is calculated for each *Mesh* based on the explosionFactor and the gravitation center of each *Mesh*:

  \[
  \begin{align*}
  \text{movementX} &= e.x \times \text{explosionFactor} / (0.1 + |e.x| \times |e.x|) \\
  \text{movementY} &= e.y \times \text{explosionFactor} / (0.1 + |e.y| \times |e.y|) \\
  \text{movementZ} &= e.z \times \text{explosionFactor} / (0.1 + |e.z| \times |e.z|)
  \end{align*}
  \]

  Each *Point* of a *Mesh* has to be moved by movementX, movementY and movementZ.

### 1.1.3. Typical Test Example

A typical test example of a geometrical transformation is a Rotation 45 around the zero point (concerning x and y axes) and a forward movement on the x dimension of 20.

```xml
<?xml version="1.0" encoding="ASCII"?>
<GeoTransfos xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns="GeoTrans" name="GeoTransfos">
  <transfos name="rotate">
    <subGeoTransfos name="moveForward">
      <params name="x" param="20.0"/>
      <params name="y" param="0.0"/>
      <params name="z" param="0.0"/>
    </subGeoTransfos>
    <params name="rotationX" param="45.0"/>
    <params name="rotationY" param="45.0"/>
    <params name="rotationZ" param="0.0"/>
  </transfos>
</GeoTransfos>
```

Typical test data is a cube:

```xml
<Mesh name="Mesh #1">
  <Point name="p0" x="0.000000" y="0.000000" z="0.000000"/>
  <Point name="p1" x="1.000000" y="0.000000" z="0.000000"/>
  <Point name="p2" x="1.000000" y="1.000000" z="0.000000"/>
  <Point name="p3" x="0.000000" y="1.000000" z="0.000000"/>
</Mesh>

<Mesh name="Mesh #2">
  <Point name="p0" x="0.000000" y="1.000000" z="1.000000"/>
  <Point name="p1" x="1.000000" y="1.000000" z="1.000000"/>
  <Point name="p2" x="1.000000" y="0.000000" z="1.000000"/>
  <Point name="p3" x="0.000000" y="0.000000" z="1.000000"/>
</Mesh>

<Mesh name="Mesh #3">
  <Point name="p0" x="0.000000" y="0.000000" z="1.000000"/>
  <Point name="p1" x="0.000000" y="0.000000" z="1.000000"/>
  <Point name="p2" x="0.000000" y="1.000000" z="1.000000"/>
  <Point name="p3" x="0.000000" y="1.000000" z="1.000000"/>
</Mesh>

<Mesh name="Mesh #4">
  <Point name="p0" x="1.000000" y="0.000000" z="0.000000"/>
  <Point name="p1" x="1.000000" y="0.000000" z="0.000000"/>
  <Point name="p2" x="1.000000" y="1.000000" z="0.000000"/>
  <Point name="p3" x="1.000000" y="1.000000" z="0.000000"/>
</Mesh>
```

```xml
<DXF>
</DXF>
```
When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:

When running the example with the typical test input, the cube will be "rolled" on its edge and pushed along the x axes as follows:
Geometrical Transformations

ATL Transformation Example

<DXF>

<Mesh name="Mesh #3">
  <Point name="p0" x="20.0" y="0.0" z="0.0"/>
  <Point name="p1" x="19.5" y="-0.7071067811865475" z="0.5000000000000001"/>
  <Point name="p2" x="19.0" y="1.1102230246251565E-16" z="1.0"/>
  <Point name="p3" x="19.5" y="0.7071067811865476" z="0.5"/>
</Mesh>

<Mesh name="Mesh #4">
  <Point name="p0" x="20.207106781186546" y="0.7071067811865476" z="1.2071067811865475"/>
  <Point name="p1" x="19.707106781186546" y="1.1102230246251565E-16" z="1.7071067811865475"/>
  <Point name="p2" x="20.207106781186546" y="-0.7071067811865475" z="1.2071067811865475"/>
  <Point name="p3" x="20.707106781186546" y="0.0" z="0.7071067811865475"/>
</Mesh>

<Mesh name="Mesh #5">
  <Point name="p0" x="20.0" y="0.0" z="0.0"/>
  <Point name="p1" x="20.707106781186546" y="0.0" z="0.7071067811865475"/>
  <Point name="p2" x="20.207106781186546" y="-0.7071067811865475" z="1.2071067811865475"/>
  <Point name="p3" x="19.5" y="-0.7071067811865475" z="0.5000000000000001"/>
</Mesh>

<Mesh name="Mesh #6">
  <Point name="p0" x="19.5" y="0.7071067811865476" z="0.5"/>
  <Point name="p1" x="20.207106781186546" y="0.7071067811865476" z="1.2071067811865475"/>
  <Point name="p2" x="19.707106781186546" y="1.1102230246251565E-16" z="1.7071067811865475"/>
  <Point name="p3" x="19.0" y="1.1102230246251565E-16" z="1.0"/>
</Mesh>
</DXF>
1.1.4. ATL Code

This ATL code for geometrical transformation consists of a program that executes the geometrical transformations and a library that contains the actual implementation of the geometrical operations. Please note that because of shortage of time the current implementation does not fully implement the described rules. In this version “freeze” is ignored, only the final result is displayed. Therefore it makes not sense to have parallel operations. GeoTrans input models are restricted to sequentially ordered operations.

The geometrical transformation module:

```atl
module GeometricalTransformations;
create OUT : DXF2 from IN1 : DXF1, IN2 : GeoTrans;
uses GeometryLib;

helper def : getRealParam( command : GeoTrans!GeoTransfo , paramName : String ) : Real =
  if (command.params->select(c | c.name = paramName)->size()=0) then
    0.0
  else
    (command.params->select(c | c.name = paramName)->first()).param
  endif;

-- calculates the gravity center (=average) of all points of a mesh
helper def : getNewGC(s : DXF1!Mesh) : TupleType(x : Real, y : Real, z : Real) =
  let nbPoints : Integer = s.points->size() in
  Tuple{
    x = s.points->iterate(e; acc : Real = 0.0 | acc + e.x) / nbPoints,
    y = s.points->iterate(e; acc : Real = 0.0 | acc + e.y) / nbPoints,
    z = s.points->iterate(e; acc : Real = 0.0 | acc + e.z) / nbPoints,
  }
```


z = s.points->iterate(e; acc : Real = 0.0 | acc + e.z) / nbPoints
;

helper def : execute(point : DXF1!Point, -- calculatedSoFar
    a : TupleType(x : Real, y : Real, z : Real),
    command : GeoTrans!GeoTransfo) :
    TupleType(x : Real, y : Real, z : Real) =
    if (command.name='explode') then
        let p : Real = thisModule.getRealParam(command, 'factor') in
        thisModule.explode( p , a )
    else
        if (command.name='rotate') then
            let rotationAngle : TupleType(x : Real, y : Real, z : Real) =
                Tuple { x = thisModule.getRealParam(command, 'rotationX'),
                         y = thisModule.getRealParam(command, 'rotationY'),
                         z = thisModule.getRealParam(command, 'rotationZ')} in
                thisModule.rotate( rotationAngle ,
                                   thisModule.getNewGC(point.mesh) , a )
        else
            if (command.name='moveForward') then
                let vector : TupleType(x : Real, y : Real, z : Real) =
                    Tuple { x = thisModule.getRealParam(command, 'x'),
                             y = thisModule.getRealParam(command, 'y'),
                             z = thisModule.getRealParam(command, 'z')} in
                thisModule.moveForward( a, vector )
            else
                if (command.name='moveBackward') then
                    let vector : TupleType(x : Real, y : Real, z : Real) =
                        Tuple { x = thisModule.getRealParam(command, 'x'),
                                 y = thisModule.getRealParam(command, 'y'),
                                 z = thisModule.getRealParam(command, 'z')} in
                        thisModule.moveBackward( a, vector )
                else
                    a
                endif
            endif
        endif
    endif
    endif
;

helper def : doCommands(point : DXF1!Point,
    calculated : TupleType(x : Real, y : Real, z : Real),
    command : GeoTrans!GeoTransfo) :
    TupleType(x : Real, y : Real, z : Real) =
    if command.superGeoTransfo.oclIsUndefined() then
        thisModule.execute(a, calculated, command)
    else
        let newlyCalculated : DXF1!Point =
            thisModule.doCommands(a, calculated, command.superGeoTransfo) in
        thisModule.execute(a,newlyCalculated, command)
    endif
;

helper context DXF1!Point def : getPoint() :
    TupleType(x : Real, y : Real, z : Real) =
    Tuple { x = self.x,
             y = self.y,
             z = self.z
    };

rule DXF2DXF {
    from f : DXF1!DXF
    to out : DXF2!DXF { meshes <- f.meshes
    }
The geometrical transformation library:

**library** GeometryLib;

**helper def**: PIDiv180 : Real = 0.017453292519943295769236907684886;

**helper def** : rotate( rotationAngle : TupleType(x : Real, y : Real, z : Real),
pointOfOrigin : TupleType(x : Real, y : Real, z : Real),
rotationPoint : TupleType(x : Real, y : Real, z : Real)) :
TupleType(x : Real, y : Real, z : Real) =

```
let rotationPointO : TupleType(x : Real, y : Real, z : Real) = rotationPoint in
let XRadAng : Real = rotationAngle.x * thisModule.PIDiv180 in
let YRadAng : Real = rotationAngle.y * thisModule.PIDiv180 in
let ZRadAng : Real = rotationAngle.z * thisModule.PIDiv180 in

let SinX : Real = XRadAng.sin() in
let SinY : Real = YRadAng.sin() in
let SinZ : Real = ZRadAng.sin() in

let CosX : Real = XRadAng.cos() in
let CosY : Real = YRadAng.cos() in
let CosZ : Real = ZRadAng.cos() in

let TempY : Real = rotationPointO.y * CosY - rotationPointO.z * SinY in
let TempZ : Real = rotationPointO.y * SinY + rotationPointO.z * CosY in
let TempX : Real = rotationPointO.x * CosX - TempZ * SinX in

let Nz : Real = rotationPointO.x * SinX + TempZ * CosX in
let Nx : Real = TempX * CosZ - TempY * SinZ in
let Ny : Real = TempX * SinZ + TempY * CosZ in
```
let rotated : TupleType(x : Real, y : Real, z : Real) =
    Tuple {x = Nx, y = Ny, z = Nz} in
rotated;

helper def : moveForward( a : TupleType(x : Real, y : Real, z : Real),
    b : TupleType(x : Real, y : Real, z : Real)) :
    TupleType(x : Real, y : Real, z : Real) =
Tuple {
    x = a.x + b.x,
    y = a.y + b.y,
    z = a.z + b.z
};

helper def : moveBackward( a : TupleType(x : Real, y : Real, z : Real),
    b : TupleType(x : Real, y : Real, z : Real)) :
    TupleType(x : Real, y : Real, z : Real) =
Tuple {
    x = a.x - b.x,
    y = a.y - b.y,
    z = a.z - b.z
};

helper def: sign(s : Real): Real = -- returns absolute value
if { s < 0.0 } then
    0-1.0
else
    1.0
endif;

helper def: explode( explosionFactor : Real, e : TupleType(x : Real, y : Real, z : Real) ) :
    TupleType(x : Real, y : Real, z : Real) =
Tuple {
    x = e.x * (1 + explosionFactor /
        ( e.x + thisModule.sign(e.x)) * thisModule.sign(e.x)),
    y = e.y * (1 + explosionFactor /
        ( e.y + thisModule.sign(e.y)) * thisModule.sign(e.y)),
    z = e.z * (1 + explosionFactor /
        ( e.z + thisModule.sign(e.z)) * thisModule.sign(e.z))
};