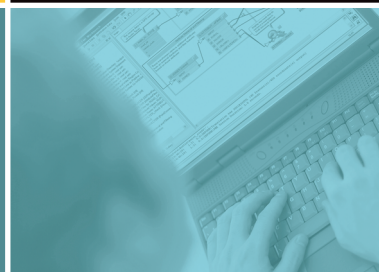


2009 FME International User Conference: Training Module
Stretching FME Boundaries





Tips and Tricks

Stretching FME Boundaries

Introduction.....	2
Training Data.....	2
01. LineJoiner.....	3
Create a Route with LineJoiner.....	3
02. Topology.....	4
Topology Example.....	4
Application Exercise.....	5
03. Segment Orientation.....	6
Orientation Example.....	6
Application Exercise.....	7
04. Measures.....	9
Measure Example.....	9
Application Exercise.....	10
Format Support for Measures.....	11
05. Geometry Traits.....	12
Trait Example.....	12
Application Example.....	13
06. Linear Referencing Transformers.....	14
NeighborFinder Exercise.....	14
Application Example.....	15
07. Paths.....	16
Path Example.....	16
Application Example.....	17
08. Bus Stop Segments and Events (Advanced).....	18
Application Exercise.....	18
The End.....	19
Appendix 1. Anatomy of an FME Feature.....	20

Introduction

This advanced training module introduces linear geometry transformations that can be used for Linear Referencing or the creation of Network Linear Elements. The topics covered should be useful for anyone undertaking more complex processing of linear geometries such as creating routes (LRS), any form of line joining or inserting points on lines. We're also going to look at the tricky issue of line orientation.

The module builds on an example application to enhance bus line segments and create bus routes. You will compare the results of the LineJoiner with other transformation approaches. You'll add measures, preserve segment attribution as traits, create a route as a path and then insert bus stop vertices on the path.

In the workspace directory, there are two sets of workspaces:

- examples that illustrate each topic being discussed, i.e. *Example_02_Begin.fmw*.
- application workspaces that build on each other to illustrate the development of the more complex bus route application, i.e. *Application_02.fmw*.

Workshop Prerequisites

Attendees should have a comfortable understanding of FME Desktop, authoring workspaces and the more common FME transformers.

Training Data

The ESRI Shape source data represents Bus Line segments and Bus Stop locations:

Bus Line Segments (lines):

```
Feature Type: BusSegments
ATTRIBUTE_NAME      ATTRIBUTE_VALUE
DIRECTION             WB2
LINE_NO               005
SEGID                 100311
SEQUENCENO            1
```

Bus Stops (points):

```
Feature Type: BusStops
ATTRIBUTE_NAME      ATTRIBUTE_VALUE
DIRECTION             WB2
LINE_NO               005
STOP_NO               50190
STOPSEQ               2
```

Have a look at the data using **FME Universal Viewer**. Inspect line number 050 – can we anticipate any potential problems when constructing a route for this Bus Line?

01. LineJoiner

LineJoiner takes lines and connects them into longer lines – if the end points match exactly. Lines remain broken at points where three or more lines converge.

LineJoiner is useful in applications where many short line segments (with similar attribute values) can be joined together to reduce the overall number of line segments and where the line orientation is not that important.

Create a Route with LineJoiner

1. Open the workspace *Application_01.fmw*

2. Connect a **LineJoiner** to the Sorter and construct a route for each Bus Line (LINE_NO) and Direction (DIRECTION) using the Group By parameter.

3. Use the Visualizer transformer to view the results (use DIRECTION & LINE_NO to group the results for display in the Visualizer properties)

Analysis:

- What do we gain and what do we lose with using the LineJoiner approach?
- Can we construct a continuous route for *Bus Line 050* using the LineJoiner?
- Can we preserve all the attribution?

4. Enter a **List Name** in the LineJoiner settings and see what output you now get.

Through the remainder of this training module, we'll compare the results of LineJoiner with some of the other route building transformers.

02. Topology

To successfully build a route, you might need to understand some of the topological relationships between the bus line segments.

We're going to experiment with the TopologyBuilder transformer. TopologyBuilder computes connectivity on sets of input point and line features. It will automatically create intersections where features overlap, unless the setting Assume Clean Data is set to Yes. TopologyBuilder will also can output the relationships between lines and areas.

TopologyBuilder is typically used to determine topological relationships to aid in decision making or to pass the topology to other transformers.

Topology Example

1. Open the workspace *Example_02_Begin.fmw*
2. Add the **TopologyBuilder** transformer, run the workspace, and view the results with FME Universal Viewer.

The TopologyBuilder adds the following attributes to the line segments:

ATTRIBUTE_NAME	ATTRIBUTE_VALUE
_arc_id	2
_from_node	3
_left_edge	2
_right_edge	-2
_to_node	4
ID	2
LineAttrs	LINE

and for the nodes:

ATTRIBUTE_NAME	ATTRIBUTE_VALUE
_node_angle{0}.fme_arc_angle	0
_node_angle{0}.fme_arc_id	2
_node_angle{1}.fme_arc_angle	70.0168934781
_node_angle{1}.fme_arc_id	4
_node_angle{2}.fme_arc_angle	180
_node_angle{2}.fme_arc_id	-1
_node_angle{3}.fme_arc_angle	250.0168934781
_node_angle{3}.fme_arc_id	-3
_node_number	2

Analysis:

- How many arcs (bus line segments) connect to this node?
- How would we know if a node was the end of a route?

3. One of the problems with this topology is that it has added nodes, and broken the lines where the lines intersect. To turn off this behavior, set:

Assume Clean Data (Advanced): Yes

Re-run the workspace and check the results in the FME Universal Viewer.

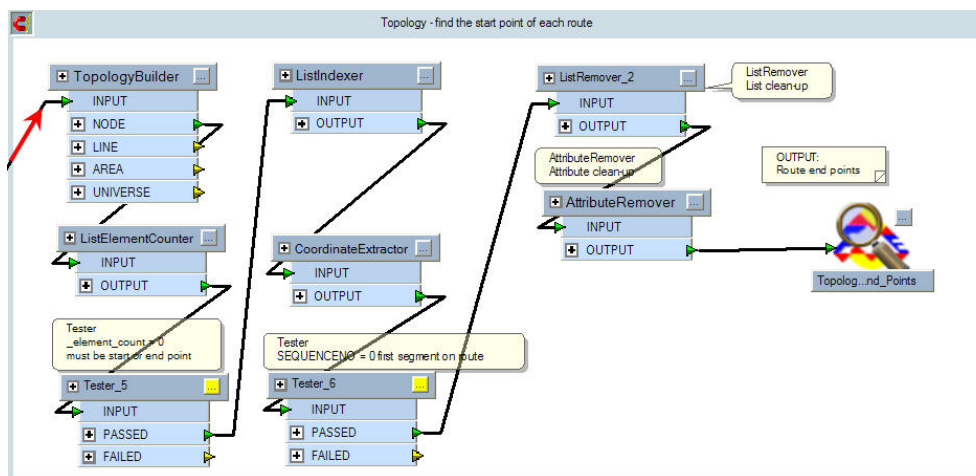
Application Exercise

For our bus line application, we want to determine the start and end points of each route. We'll see why we need this later. To do this we'll use the TopologyBuilder – with a few other supporting transformers.

1. Open the workspace *Application_02.fmw* and inspect the bookmark “Topology - find the start point of each route”.

In this bookmark we:

- use TopologyBuilder, as before, to build the relationships between the bus lines
- only use the topology nodes



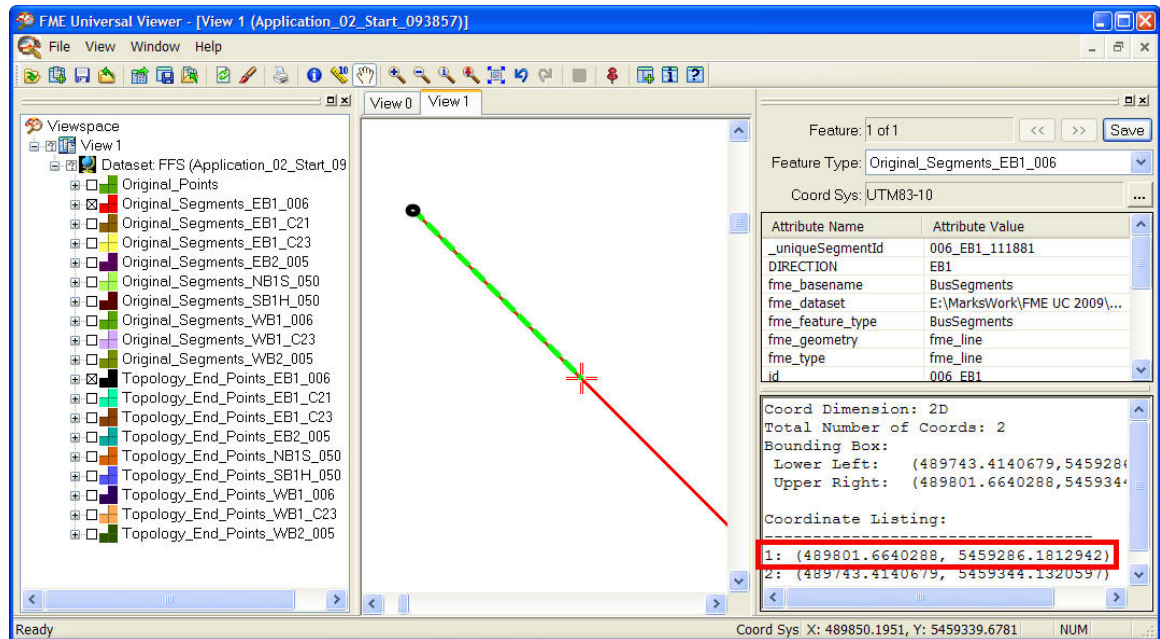
- added a couple of tests to check for the connectivity of the topology nodes and for the sequence number of the segment – we end-up keeping only the first point of each bus route
- CoordinateExtractor preserves the X/Y coordinate of the start points as attributes.

2. Check the results in FME Universal viewer. we should have nine end points – one for each of the nine bus lines.

03. Segment Orientation

The TopologyBuilder gave us the start point of each bus line, but can we be sure all the bus line segments pointed in the correct direction?

Inspect the first segment (SEQUENCENO: 0) of the LINE_NO: 06, DIRECTION: East from the previous application exercise. Does the first point of the segment match the start point of the 06 Bus Line?



Above: Route start point vs. segment start point

One of the trickiest problem with creating networks or routes is line orientation. Areas have well defined orientations based on a left-hand or right-hand rule. Line orientations are always relative to the preceding line or, perhaps, follow a rule such as “from lower-left to upper-right”.

LineJoiner automatically re-orientes the segments when it joins them. However, there is no control of the overall direction of the resulting route. For bus routes, gas networks, and river flows, the orientation of the route is probably very important.

Orientation Example

Using the previous topology example workspace or the *Example_03_Begin.fmw*

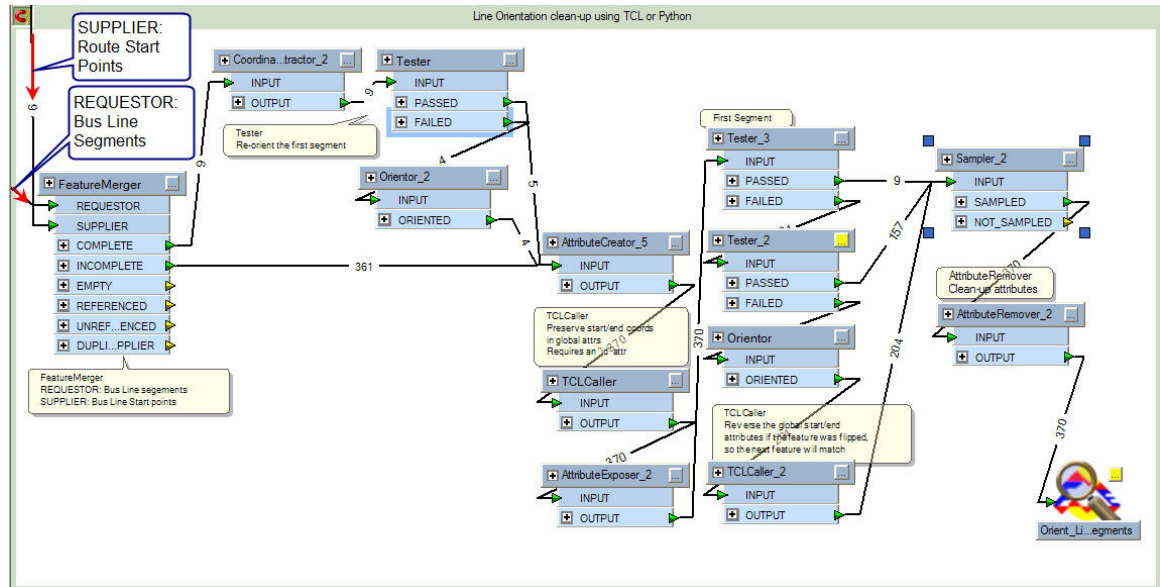
1. Connect an **Orienter** transformer to the LINE output port of the TopologyBuilder
2. Run the workspace and compare the results of the TopologyBuilder LINE with the Orienter output (click on the coordinate list in the Visualizer)

All we’ve done is reverse the orientation of the lines. We have not aligned the two connected lines so they both have the same direction relative to each other. We need a more sophisticated approach.

Application Exercise

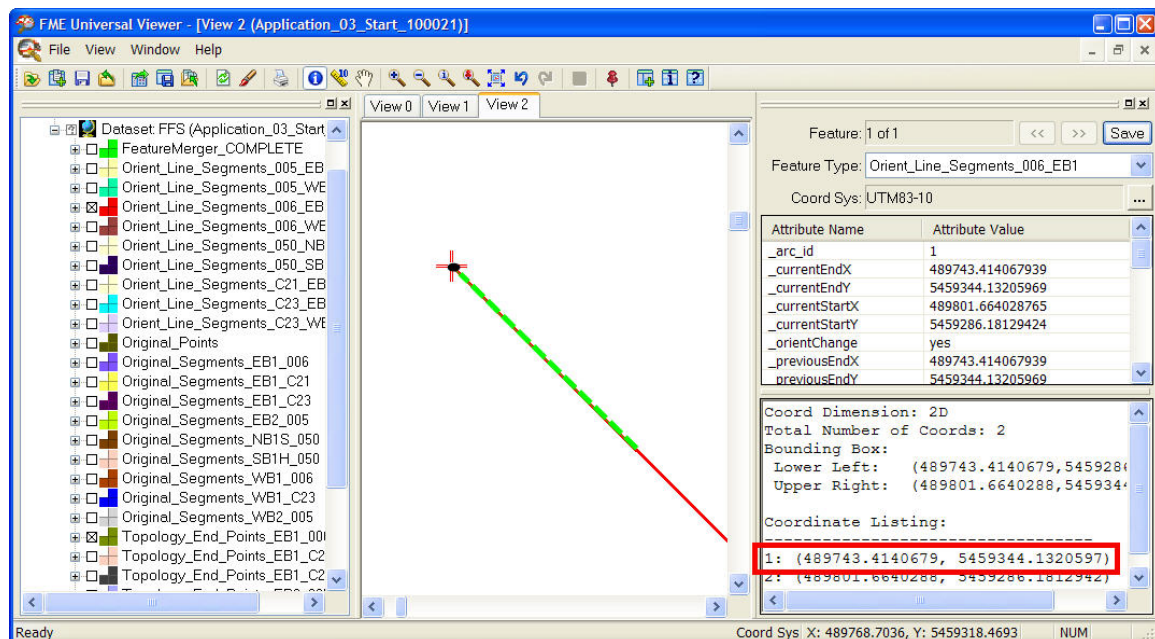
We're going to wrap the Orienter in some other transformers so it knows when to reverse the line segment, if it needs to.

1. Open the *Application_03.fmw* workspace and navigate to the "Line Orientation Clean-up" bookmark



Ouch... but it looks worse than it is!
Above: Bus Line Orientation

2. Run the workspace and compare the first segment (SEQUENCE_NO: 0) of the LINE_NO: 06, DIRECTION: East with the start point of the route.



Above: Route start point vs. segment start point

In Universal Viewer, follow Bus Line 06. Now all the segments line-up in the same direction, i.e. the start point of a line always touches the end point of the previous line.

Line Orientation Clean-up Bookmark

In the “Line Orientation clean-up” bookmark we are:

- using FeatureMerger to attach the attributes from the route start points (`_xRouteStart` & `_yRouteStart`), which we created earlier, to the corresponding first segment on each route. The join is based on the `LINE_NO`, `DIRECTION`, `SEGID`
- use the Orienter to reorient the first segment so the route start matches the segment start
- use the TCLCaller to set some global values to preserve the start & end coordinates of each segment, and set them on each feature.
- Tester compares the coordinates of the previous feature with the current feature
- use the Orienter to reorient the segment
- more TCL to reset the global values if the segment orientation was switched
- AttributeRemover to clean-up

The result is that all segments along a bus line now align in the direction the bus will take.

The TCL used in this bookmark is a little tricky and we don't have time in this module to walk through the code in detail. But this is a good example of how the TCL can be used to pass attributes from one feature to the next feature. In this case we are relying on the fact that the sequence of the segments is correct.

This could also be done using the PythonCaller.

04. Measures

Measures are values attached to each vertex on a feature. Typically they represent a value that describes the distance along a route, such as a linear distance, percentage distance or in some cases a scaled distance. Measures can also be a discrete value such as an offset, a database key, a value that represents the type of vertex, or a code that describes the following segment. For example, in Aeronautical charts, a measure can be used to describe the type of arc the next segment will be built from (line segment, clockwise arc, counterclockwise arc, loxodrome, orthodrome, etc.).

Different database & GIS vendors may have their own specific definitions for measures. For example, Oracle refers to measures as a distance along an LRS geometric segment, and has a separate representation for an offset. In FME, both the Oracle measure and offset would be a measure.

In FME, measure values can be either an integer or float, but not a character string. The measure transformers are in the workbench Linear Referencing transformer category and are:

MeasureSetter, MeasureExtractor, MeasureGenerator.

MeasureGenerator is the only transformer that will create measure values.

The MeasureSetter & MeasureExtractor transformers will move values from an attribute list to a measure, and vice versa. Measures can also be given names – so you can have multiple measures on a feature.

Since the measure is part of the feature geometry, if the geometry is split (i.e. using Clipper) then the measures will be preserved for each of the split pieces of geometry.

Measure Example

Let's experiment with the measure transformers:

1. Open the workspace *Example_04_Begin.fmw*
2. Add a **MeasureGenerator** transformer and connect it to the Cloner output port.
3. Add a **Visualizer** transformer.
4. Run the workspace and inspect the results – in particular check the feature geometries.

MeasureGenerator has added a measure named <default measure> to the features:

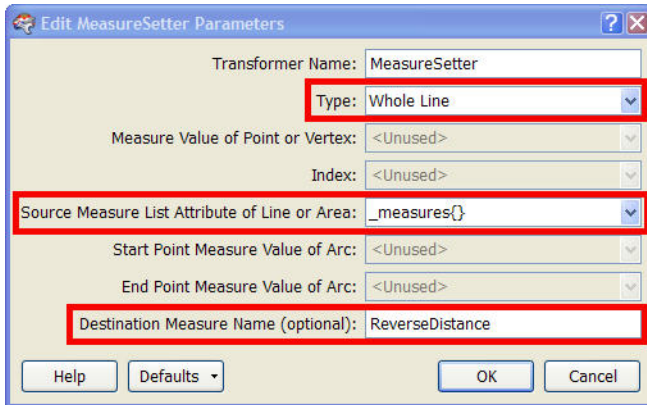
```
Geometry Type: IFMELine
Measures (1): <default measure>
Number of Coordinates: 4 -- Coordinate Dimension: 2
0: (5,2)<0>
1: (12,9)<9.89949493661167>
2: (14,20)<21.0798348241106>
3: (25,25)<33.1628807977052>
```

5. Connect a **MeasureExtractor** to the MeasureGenerator and view the results.

The MeasureExtractor converts the geometry measures into an attribute list:

ATTRIBUTE_NAME	ATTRIBUTE_VALUE
_measures{0}	0
_measures{1}	9.89949493661167
_measures{2}	21.0798348241106
_measures{3}	33.1628807977052
ID	2
LineAttrs	LINE

6. Add and connect an **Orientor** transformer



7. Add a **MeasureSetter** transformer and set the parameters as shown to the left.

Run the workspace.

We now have two measure values, one for the forward direction, one for the reversed direction

```

Geometry Type: IFMELine
Measures (2): <default measure>, ReverseDistance
Number of Coordinates: 4 -- Coordinate Dimension: 2
0: (25,25)<33.1628807977052,0>
1: (14,20)<21.0798348241106,9.89949493661167>
2: (12,9)<9.89949493661167,21.0798348241106>
3: (5,2)<0,33.1628807977052>
    
```

Those are the measure transformers – let’s have a look at how they could be applied.

Application Exercise

In our bus line exercise, instead of using distance measures, we’re going to set the measure values on the bus line data as follows:

Description	Measure
Start of a segment	1
End of segment	2
Bus stop vertex	9
Interior segment vertex	0

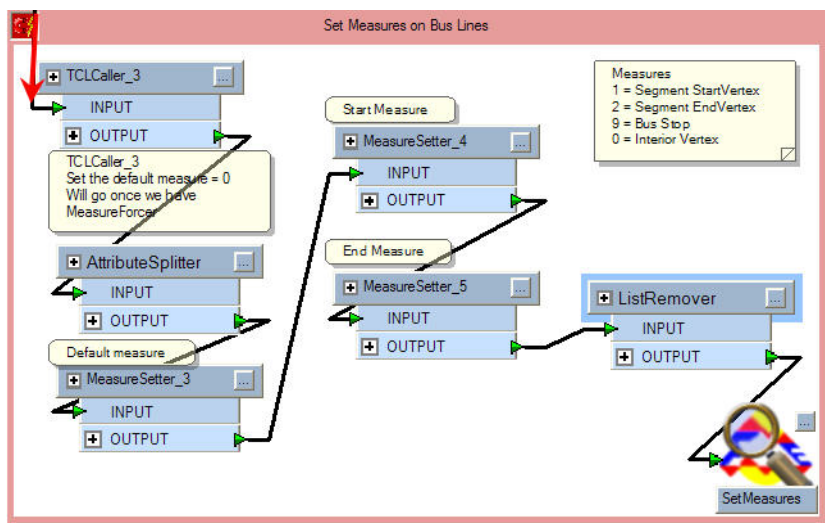
1. Open workspace *Application_04.fmw* and zoom to the “**Set Measures on Bus Lines**” bookmark.

Here we’re setting a default measure value – “0” - for all the vertices on a segment, then set the start and end values (“1” and “2”, respectively).

- Run the workspace and inspect the results.

```

Geometry Type: IFMELine
Measures (1): <default measure>
Number of Coordinates: 5 -- Coordinate Dimension: 2
0: (491233.840269268,5457061.88358017) <1>
1: (491192.313565907,5457053.72900958) <0>
2: (491098.08977129,5457052.21332575) <0>
3: (490721.756105394,5457066.55456922) <0>
4: (490698.493732392,5457071.94626054) <2>
  
```



In the “Set Measures on Bus Lines” bookmark we are:

- using TCLCaller to create a list of default measure values (“0” for each vertex)
- MeasureSetter to convert the list to a geometry measure
- a couple more MeasureSetters to set the start (“1”) and end (“2”) segment measures

This workspace also sets the measure on the bus stop points to “9” in the “Set Measures on Bus Stops” bookmark. Later, we’ll insert the bus stop locations onto the bus route, and we’ll try and preserve that measure value.

How can this be used?

Adding measures to the vertices allows you to preserve certain characteristics of the data as it is transformed by other transformers. For example, add a LineJoiner after the measures (Group By LINE_NO & DIRECTION). Notice how you can track the original segments in the new joined lines.

In older versions of FME, measures were stored as a comma separated attribute. So in the example above, you might see an attribute:
sde_measure 1,0,0,0,2
The drawn back of this approach was that if the geometry was split (i.e. using Clipper) the measure attribute list no longer lined-up with the number of vertices on the geometry.

Format Support for Measures

Not all formats support the concept of measures. In some formats, i.e. ESRI Shape, the feature class must be explicitly configured to accept a measure value. If measures are passed to a format that is not expecting measures then features might get rejected.

05. Geometry Traits

Geometry traits are similar to attributes on a feature; they are any user-defined characteristic. The difference is that they are stored on each **geometry** of the feature. Previously data such as this could only be stored at the feature level (as an attribute). This capability is useful for situations where geometries are combined or split apart during workspace transformations and you need to preserve some specific characteristic when merging the geometries – such as an original segment ID.

There are three transformers to manipulate geometry traits:

GeometryTraitSetter, GeometryTraitExtractor and GeometryTraitRemover.

Trait Example

1. Open workspace *Example_05_Begin.fmw*
2. Add a **GeometryTraitSetter** transformer to the workspace and connect it to the Cloner transformer.
3. Set the **Source Attributes** parameter to the “ID” attribute.
4. Inspect the results using the Visualizer. Notice how the trait information is now part of the geometry.


```
Number of Geometry Traits: 1
GeometryTrait(string): `ID' has value `3'
Number of Coordinates: 3 -- Coordinate Dimension: 2
0: (0,0)
1: (2,5)
2: (10,10)
```
5. Connect an **Aggregator** transformer to the GeometryTraitSetter and set the **Group By** to be LineID. Notice how the traits are preserved on each part of the geometry.

```
Geometry Type: IFMEMultiCurve
Number of Curves: 2
-----
Curve Number: 0
  Geometry Type: IFMELine
  Number of Geometry Traits: 1
  GeometryTrait(string): `ID' has value `3'
  Number of Coordinates: 3 -- Coordinate Dimension: 2
  0: (0,0)
  1: (2,5)
  2: (10,10)
-----
Curve Number: 1
  Geometry Type: IFMELine
  Number of Geometry Traits: 1
  GeometryTrait(string): `ID' has value `1'
  Number of Coordinates: 6 -- Coordinate Dimension: 2
  0: (10,10)
  1: (15,10)
  2: ...
```

6. Connect a second **GeometryTraitSetter** to the Aggregator output port and set the **Source Attributes** parameter to the “LineID” attribute. The traits can be recursive:

```

Geometry Type: IFMEMultiCurve
Number of Geometry Traits: 1
GeometryTrait(string): `LineID' has value `A'
Number of Curves: 2
-----
Curve Number: 0
  Geometry Type: IFMELine
  Number of Geometry Traits: 1
  GeometryTrait(string): `ID' has value `3'
  Number of Coordinates: 3 -- Coordinate Dimension: 2
  0: (0,0)
  1: (2,5)
  2: (10,10)
-----
Curve Number: 1
  Geometry Type: IFMELine
  Number of Geometry Traits: 1
  GeometryTrait(string): `ID' has value `1'
  Number of Coordinates: 6 -- Coordinate Dimension: 2
  0: (10,10)
  1: (15,10)
  2: (20,15)
  3: (30,20)
  4: (35,25)
  5: (40,25)

```

The feature with LineID = A now has three traits – one trait (LineID) for the entire aggregate geometry, and two traits (ID) on each geometry part of the aggregate.

Deaggregate the features and use **GeometryTraitExtractor** to recover the original “ID” value for each geometry.

Application Example

1. Open workspace *Application_05.fmw* and run it. In this exercise we are preserving the segment ID (SEGID) and the segment sequence number (SEQUENCENO).
2. Check the results in the Visualizer.

We don’t really need traits for our bus line application, so we’ll be dropping them later on using the **GeometryTraitRemover**.

In older versions of FME, you could try and use list attributes to try and preserve the characteristics of individual features that have been merged. In the previous example, you could add a list name to the Aggregator and each list element would preserve the attributes for each of the features (and geometries) that merged. The problem is associating the correct list element to the corresponding geometry part when, say, you de-aggregate.

06. Linear Referencing Transformers

In our bus route application, we want to add the location of the bus stops onto the routes. We've already tagged the bus stops with a measure ("9") so the inserted points should be recognizable as bus stop locations along the route.

The way to insert a point on a line in FME is to use the NeighborFinder. The NeighborFinder transformer is quite complex and can be used for a wide range of tasks associated with building spatial relationships between features in proximity of each other – the underlying FME factory is called the ProximityFactory. But the basic task of the NeighborFinder is to find the closest CANDIDATE feature within some maximum distance of each BASE feature.

Check the NeighborFinder transformer documentation for all the options. One option is to insert the nearest vertex of the CANDIDATE into the BASE feature.

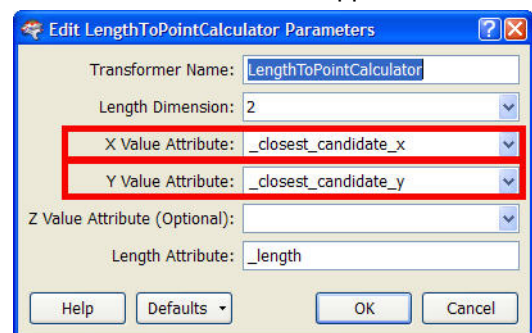
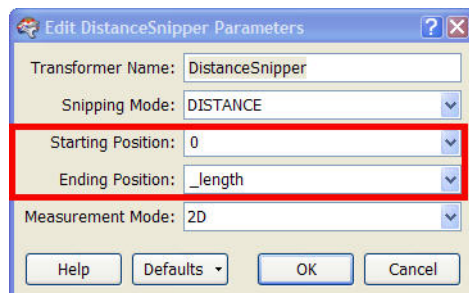
NeighborFinder Exercise

Using the NeighborFinder transformer:

1. Open *Example_06_Begin.fmw* and review the sample data.
2. Add a **NeighborFinder** transformer and connect the:
 - points to the CANDIDATE input port
 - lines to the BASE input port
 Use a Maximum Distance: 4
3. Run the workspace and inspect the results in FME Visualizer. NeighborFinder adds quite a lot of attributes
 Note that one of the segments does not match a point – that's OK.
4. Set the NeighborFinder parameter **Insert vertex on BASE feature: Yes** and rerun the workspace
 Notice how a new point has been inserted on the lines.

In some LRS applications, the bus stops would be considered to be an event and you might need to calculate the distance to the location of the bus stop and populate an event table. This can be achieved with a combination of the LengthToPointCalculator and the DistanceSnipper.

5. Connect the **LengthToPointCalculator** to the NeighborFinder MATCHED port. Set the Parameters as shown.



6. Connect the **DistanceSnipper** to the LengthToPointCalculator and set the Starting and Ending Position parameters.

7. Inspect the results.
It would be relatively easy to restructure these features and write them to an event table.

Application Example

In our bus route application, we're not going to create events. But we do want to insert the bus stops onto the bus line segments.

1. Open workspace *Application_06.fmw* and inspect the NeighborFinder bookmark.

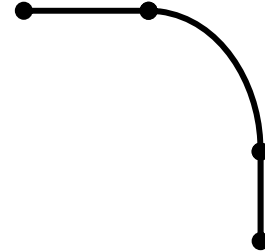
This bookmark is more complex than it should be. Currently the NeighbourFinder drops the measure value on the inserted point (it's set to an undefined value). Hence the transformers in the "Clean-up Measure"s bookmark. The TCLCaller loops through the measure values and sets any undefined value to "9". Hopefully this bookmark can be removed with a future update to FME 2010.

The output from this bookmark is all the bus line segments with bus stop locations added and tagged with a measure value of "9".

07. Paths

An FME geometry path is a multi-part line or area feature. The geometry parts are lines or arcs that connect end-to-end to form a continuous line or area. Paths may also be referred to as chains.

Right: Path of three parts – line, arc, line



Not all formats support path geometries. If a path geometry is written to a format that does not support paths – or the equivalent – FME will stroke the path into a continuous linear feature (replacing arcs with vertices if necessary)

Paths add more flexibility to the geometry representation. **Also, paths allow you to preserve certain characteristics of the individual geometry parts as traits or measures.**

Paths are different to aggregates - multi-part geometries such as multi-line or multi-polygons. Paths have a definitive structure in that the parts join end to end, whereas there is no requirement for geometric connectivity in aggregates.

The transformers which can be used to handle path geometries are:
PathBuilder, PathSplitter and GeometryRefiner.

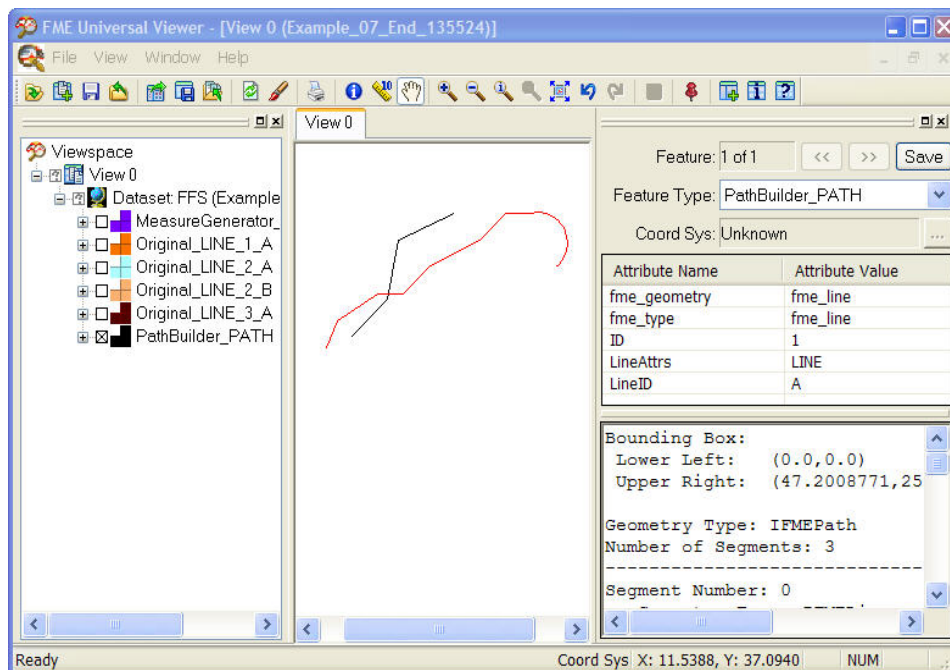
Path Example

1. Open the workspace *Example_07_Begin.fmw*, run it and check the features. Notice the arc geometry.
2. Connect a **PathBuilder** to the MeasureGenerator and inspect the results. FME has generated a polygon instead of a line.

PathBuilder is similar to PointConnector, all features must be ordered and in the correct orientation before they enter the transformer. The line segments should also connect, end-to-end. If they do not, then PathBuilder will insert new segments – which might not be what you want.

3. Add a **Sorter** to reorder the features by LineID and ID. Check the results.

In the Visualizer information panel, click on the feature coordinates to see how the individual path segments connect. Notice how the measures are a little scrambled since they were created for each of the original geometries.



4. Connect a second **MeasureGenerator** after the PathBulder and compare the results.
5. Connect a **PathSplitter** to the MeasureGenerator and run the workspace. PathSplitter takes a path geometry and creates a new set of features, one for each path part. Attributes are the same for each part.
6. Replace the PathSplitter with a **GeometryRefiner**. What changes? GeometryRefiner simplifies the geometry of a feature – in this case merging the linear parts of the path. the arc is untouched. To eliminate the arc you would have to add an ArcStroker before the GeometryRefiner

Application Example

This is pretty well our final step. We're going to use the PathBuilder to consolidate all the Bus Line segments into routes – so each bus line (LINE_NO & DIRECTION) will be a single feature with the orientation matching the bus travel direction.

1. Open workspace *Application_07.fmw*
2. Navigate to the "Path Builder" bookmark.
3. Run the workspace and check the results. You can see that the PathBuilder has reduced the number of features to nine – one for each bus route

How do these results compare to the original results we obtained from the LineJoiner? Compare the path of LINE_NO: 50 with the equivalent feature from the LineJoiner.

08. Bus Stop Segments and Events (Advanced)

The last challenge for our bus route application is to split the bus route into separate segments that connect the bus stops. We'll also create some bus stop events – the distance along the route to each bus stop.

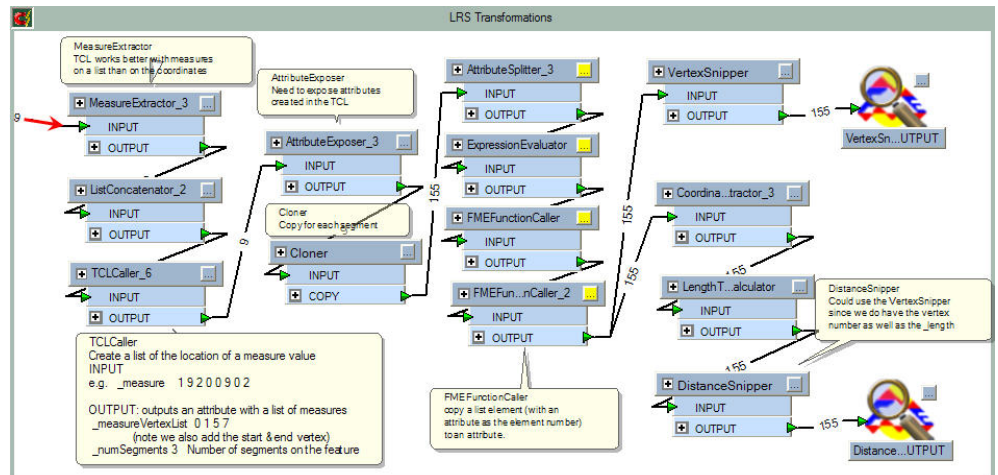
We'll use a couple of the LRS transformers we looked at before – LengthToPointCalculator and DistanceSnipper – as well as some tricky TCL!

Application Exercise

1. Open the workspace *Application_08.fmw* and zoom to the “LRS Transformations” bookmark.

This is one of the more complex bookmarks and creates two different outputs. New features that represent the bus stop to bus stop segments and a second output that is the bus line start to bus stop segments (events).

In the bookmark we have:



- TCLCaller – this TCL checks the measures and creates an attribute list of the vertex numbers, where a specific measure value occurs – 9 for the bus stops. For example, if we have measure values:
1 9 2 0 0 9 0 2
the TCL will find the vertex numbers for the measures with a value of “9”:
1 5
the TCL then adds the vertex number of the first & last segment, so the final result is:
0 1 5 7
- Cloner – makes a copy of the feature for each segment – in this case 3 clones.
- ExpressionEvaluator and FunctionCallers – sort out the vertex numbers from the list
- VertexSnipper – creates a new feature that represents the bus stop to bus stop segment
- LengthToDistanceCalculator and DistanceSnipper – create a new feature that is the route distance to the bus stop – it's this data that could be written to an LRS event table.

The End

This concludes the advanced training module Stretching FME Boundaries, held as part of the FME International User Conference 2009.

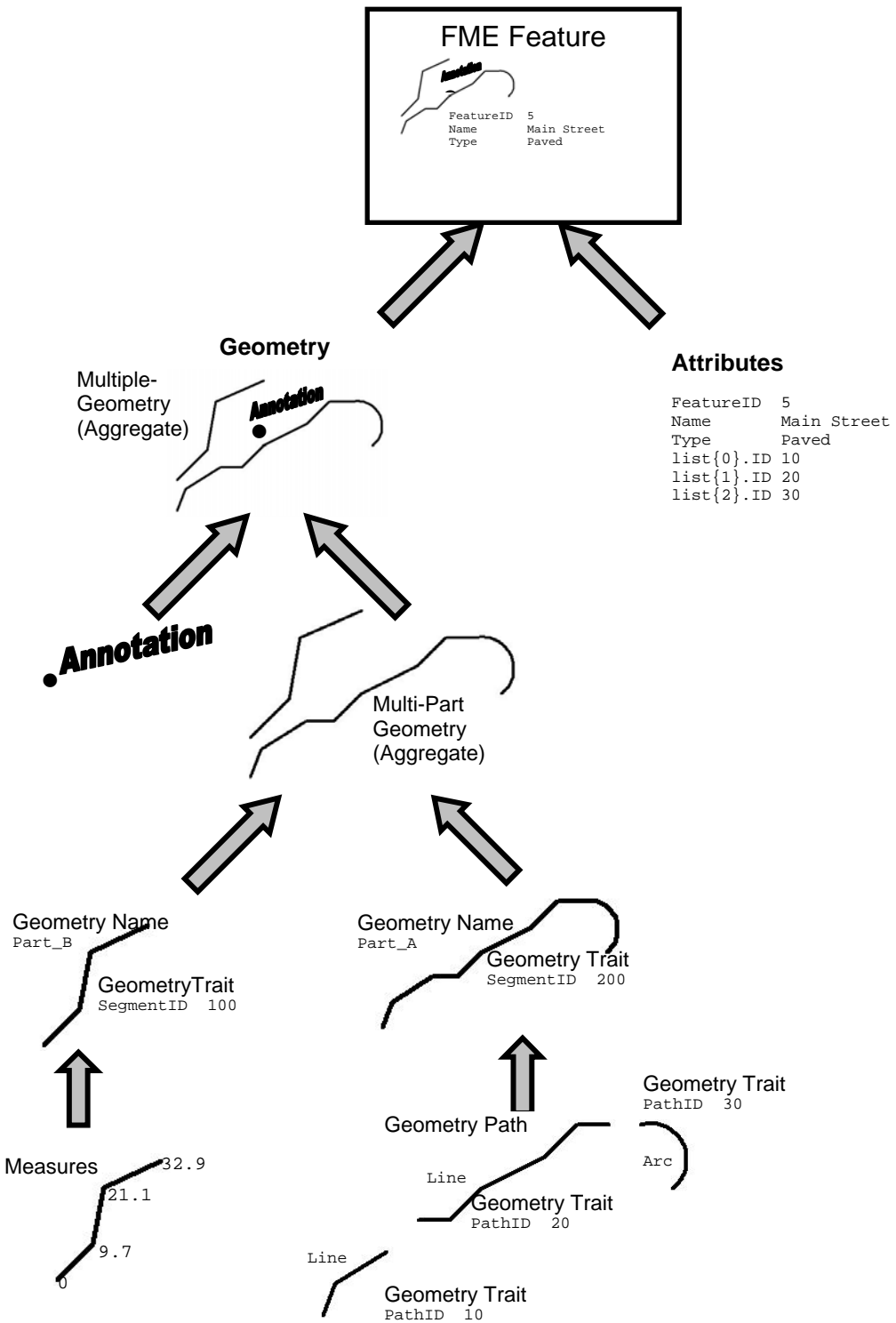
With your new-found knowledge you should now:

- have an understanding of some of the geometry structures – measures, traits paths.
- be able to use those geometry structures in some of your workspaces
- be able to take home a comprehensive example application of constructing a route.

Be sure to visit Safe Software's FMEUserCentral web site. This has links to other resources which may be of use in your study of this functionality.

On behalf of Safe Software, thank you for taking this training course.
I hope you found it useful and hope to see you in the other courses at this conference.

Appendix 1. Anatomy of an FME Feature





Advanced Module

SchemaMapper

Schema Mapping Review.....	2
Overview.....	2
Session Goal	2
The SchemaMapper.....	3
What is the SchemaMapper?.....	3
SchemaMapper Examples.....	3
Example 1: Feature Mapping.....	4
Example 2: Attribute Mapping.....	7
Example 3: Schema Mapping Variations.....	10
Schema Mapping References.....	10

Schema Mapping Review

Overview

In the context of FME Workbench, schema mapping is the process of transforming the source schema (what we have) to the destination schema (what we want). Schema mapping operates at all levels of the source and destination schemas – feature type mapping, attribute mapping and domain mapping.

Feature Type Mapping

Feature type mapping is the transformation of the source feature types to the destination feature types, e.g.:

Roads ⇒ CenterLines

Feature type mapping can be a one to one (1:1) mapping, as above, one to many (1:M), many to one (M:1) or even many to many (M:N), e.g.:

Hydrography ⇒ Rivers
Hydrography ⇒ Waterbodies

Usually this requires an attribute to be used as part of the mapping, e.g.:

Hydrography + type = Rivers | Streams ⇒ Rivers
Hydrography + type = LeftBank | RightBank ⇒ Waterbodies

The reverse is also possible.

In a typical workspace, feature type mapping is achieved using the source and destination feature type name, with the possible help of a *Tester* or *AttributeFilter* transformer.

Attribute Mapping

Attribute mapping is the transformation of the source attributes to the destination attributes, e.g.:

type ⇒ FCode

In a typical workspace, attribute mapping is achieved using the *AttributeCopier* or the *AttributeRenamer* transformers.

Domain Mapping

Domain mapping is the remapping of the attribute values to meet a well defined domain or enumerated list definition, e.g.:

type = Primary Route ⇒ US Highway
type = Secondary Route ⇒ Interstate Highway
type = Road ⇒ County Road

In some formats, the domains are represented as an integer code.

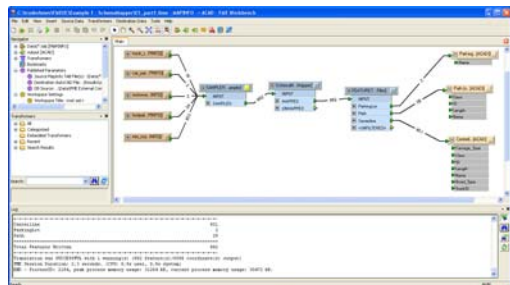
In a typical workspace, domain mapping is achieved using the *ValueMapper* transformer.

Session Goal

This session will cover some basic methods of schema mapping using the FME SchemaMapper Transformer. The topics we'll cover are:

- Feature type mapping
- Attribute mapping
- Schema mapping types

The SchemaMapper



The SchemaMapper transformer is an alternate method of carrying out schema mapping during a translation.

What is the SchemaMapper?

The *SchemaMapper* is an FME transformer designed for carrying out schema mapping dynamically during a translation. For this reason it will be of great use to users creating workspaces with dynamic writers.

The *SchemaMapper* maps features based on a Schema Mapping Table. This table can be stored within a database, but is often held as a CSV format file.

The table defines a set of conditions and a set of mapping rules to be carried out when the conditions are met. For example, if a feature is colored blue (the condition) then it should be written as a river feature (the action).

This way, a user does not have to define all of the schema mapping manually within the workspace. It also means that updates to the schema rules can be made within the Schema Mapping Table, and that the workspace does not need to be edited.

SchemaMapper Examples

In this brief tutorial, we will go through two practical exercises; one mapping feature types and the other mapping attribute values. In the process, we'll try to understand the dialogs in the *SchemaMapper* Wizard.

In Example 1, we are given a set of five MapInfo datasets. Three of them contain road information, and two contain other information. The goal for example 1 is to merge the three road feature types together into a single feature type.

In Example 2, we realize that the three feature types we've merged together should have some sort of attribute explaining where they came from. In this example, we create a new attribute, and assign it a value based on which feature type it came from.



Tip: A schema mapping table can be derived from a database metadata document such as ESRI's XML database schema description which can be exported from ArcCatalog for any selected geodatabase (Export - XML Workspace Document - Schema Only).



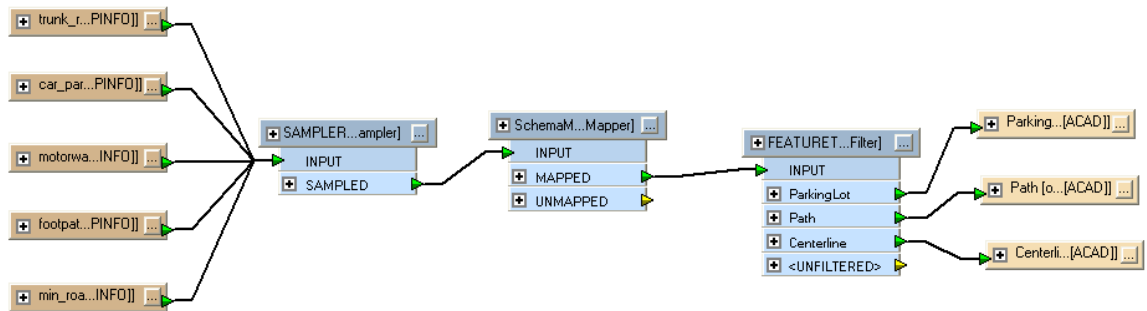
Example 1: Feature Mapping

In this example – the first of two related examples – we’ll set up the workspace to carry out some Feature mapping.

1) Open the FME Workspace

Open the workspace:

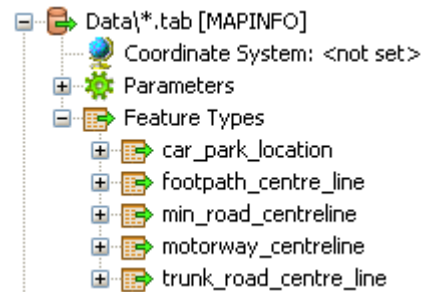
C:\FMEData\Workspaces\AdvancedTrainingWorkspaces\SchemaMapper - Example1 - Complete.fmw



We start with five MapInfo source feature types. Three are related to road data; Min_Road_Centreline, Trunk_Road_Centre_line, and Motorway_centreline.

These are the three feature types we would like to merge together. The sampler is added to visually simplify the workspace

Inspect the MapInfo TAB files in the FME Universal Viewer



2) Examine the CSV file for the SchemaMapper.

Let’s take a quick look at what we are working with.

Open up the file *C:\FMEData\Resources\SchemaMapper\Feature_type_mapping.csv* in a text editor.

This file acts as the Schema Mapping Table for our examples.

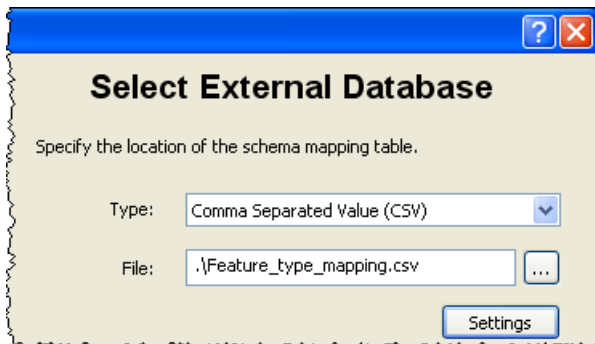
Mapping type	Source feature	Destination feature
Feature	car_park_location	ParkingLot
Feature	footpath_centre_line	Path
Feature	min_road_centreline	Centerline
Feature	motorway_centreline	Centerline
Feature	trunk_road_centre_line	Centerline

Here we see three columns. The first column is simply metadata to let us know what type of mapping we’ll be doing in the *SchemaMapper* – this column is not required. The second column defines our source feature, and the third column is where we specify what the new destination feature should be named.

Now let’s take a look at the *SchemaMapper* Wizard to see how these columns are used.

3) Examine the SchemaMapper settings.

Open the *SchemaMapper* settings by clicking the Properties button. This will launch the *SchemaMapper Wizard*.

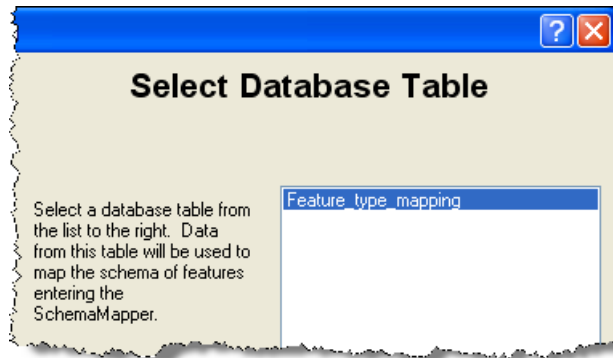


Left: The Select External Database Pane.

Note that relative paths are valid. Also, don't forget to check the settings. This can affect how the database is read.

Right: The Select Database Table Pane

In this pane, we select out database table. In the case of a CSV file, the file name is used. In a Microsoft Excel spreadsheet, each Worksheet is considered its own table.



The **Select Filter Fields Pane** is not used in this part of the exercise and can be skipped with the next button. It is used in example 2 below.

Right: The Index Mapping Pane

This pane in the Wizard should really be called the Feature Type Mapping pane – although it can be used for more than feature mapping.

By checking the **Do Index Mapping** box, we are specifying that there are feature types to map.

The Source and Destination Fields are read from the CSV file, while the Source and Destination Attributes are read from the feature.



To simplify, this pane tells FME to go to a CSV file, and get the next values in the Source Field (in this case the Source_feature column) and Destination Field (the Destination_feature column). These are the ‘What We Have’ and ‘What We Want’ parts of a schema mapping.

Then FME looks on each feature to see if the value of the source attribute specified (in this case fme_feature_type) matches the value found in the Source Field column.

If there is a match (i.e. the feature matches the ‘What We Have’ part) then the destination attribute specified (again, fme_feature_type) has its value replaced with the value of the Destination Field column.

So now we have taken a feature and mapped it from ‘What We Have’ to ‘What We Want’ based on conditions in our Schema Mapping Table.

In this case we’re using fme_feature_type as both the source and destination attribute. Since this comes from the Reader and is used by the Writer, we are literally mapping from ‘What We Have’ to ‘What We Want’ and FME will write the data accordingly. However, there’s nothing to prevent use of different attributes when you don’t want FME to automatically carry out the mapping.



In order to find fme_feature_type in the list for the Index Mapping pane, it must be exposed in at least one (but not necessarily all) of the source feature type definitions.

The **Select Mapping Fields Pane** is not used in this exercise. It is described below in example 2.

4) Run the workspace.

On the Destination Data menu, select Redirect to Visualizer. Run the workspace.

In the output notice how the five source MapInfo tables have been simplified to 3 output Feature Types. If you query a Centerline feature type in the FME Universal Viewer you can see how the *SchemaMapper* has renamed the fme_feature_type attribute. Compare this to the source data.

Re-run the workspace, this time writing directly to the AutoCAD file and again inspect the results using FME Universal Viewer.



Example 2: Attribute Mapping

This example continues on from the first.

Here we'll use the *SchemaMapper* to undertake some simple attribute mapping.

1) Inspect the schema mapping CSV file

We've successfully mapped the three road feature types to a common centerline feature type, but perhaps we should record as an attribute what the original feature type was for each feature.

We'll create a new attribute to hold the values of the original `fme_feature_type`. But since we're at it, let's change the values to be something a little more Canadian.

Open the CSV file `C:\FMEData\Resources\SchemaMapper\Attribute_mapping.csv`

Mapping type	Source Attribute	Source feature	Destination attribute	Destination Value
Attribute	<code>fme_feature_type</code>	<code>min_road_centrelines</code>	<code>CanadianRoadType</code>	Minor
Attribute	<code>fme_feature_type</code>	<code>motorway_centrelines</code>	<code>CanadianRoadType</code>	Highway
Attribute	<code>fme_feature_type</code>	<code>trunk_road_centre_line</code>	<code>CanadianRoadType</code>	Major

Here we see five columns. The first column (Mapping Type) is simply metadata to let us know what type of mapping we'll be doing; this column is not required or used by the *SchemaMapper*.

The second & third columns (Source Attribute & Source Feature) are attribute name & attribute value pairs that may be on the incoming feature. This pair of columns will act as a filter or query:

if the value of the attribute `fme_feature_type` is `min_road_centrelines` then do something...

The last two columns (Destination Attribute & Destination Value) are also attribute name & value pairs – this time used for the results of our mapping. We're going to set the attribute named in the Destination Attribute field to the value of the Destination Value field. So continuing our pseudo code from the example in the previous paragraph:

if the value of the attribute `fme_feature_type` is `min_road_centrelines` then create the attribute `CanadianRoadType` and set the value to be `Minor`.

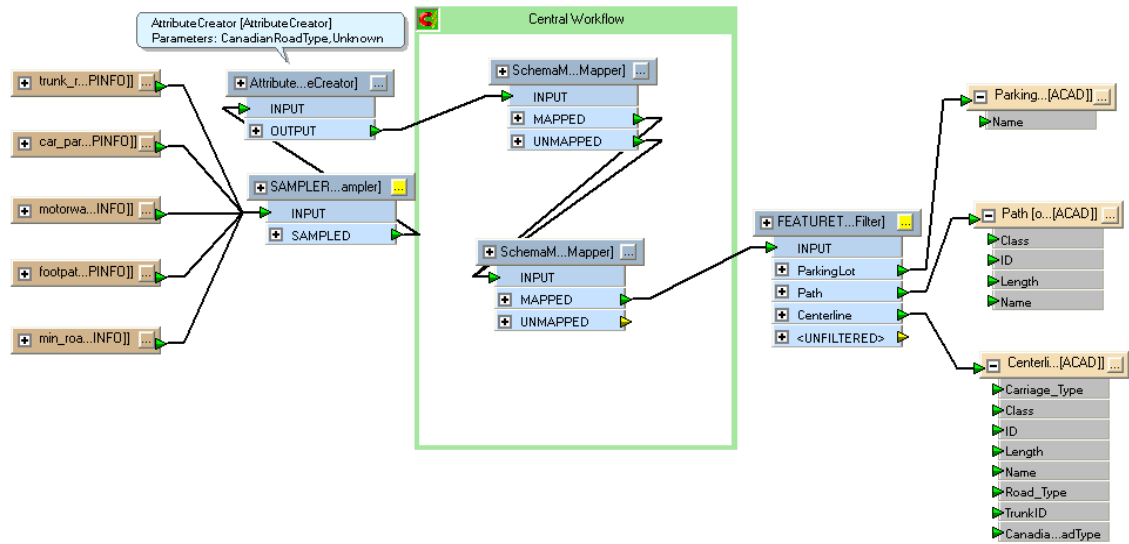
Note: you can call the field headings in the CSV file anything you want, and the fields can be in any order.

2) Open the FME Workspace

Now let's take a look at the SchemaMapper Wizard to see how these columns are used.

Open the workspace:

C:\FMEData\Workspaces\AdvancedTrainingWorkspaces\SchemaMapper - Example2 - Complete.fmw



We can see that in the *AttributeCreator*, we are creating a new attribute called *CanadianRoadType*, with a default value of *Unknown*. After this, we will introduce a new *SchemaMapper*.

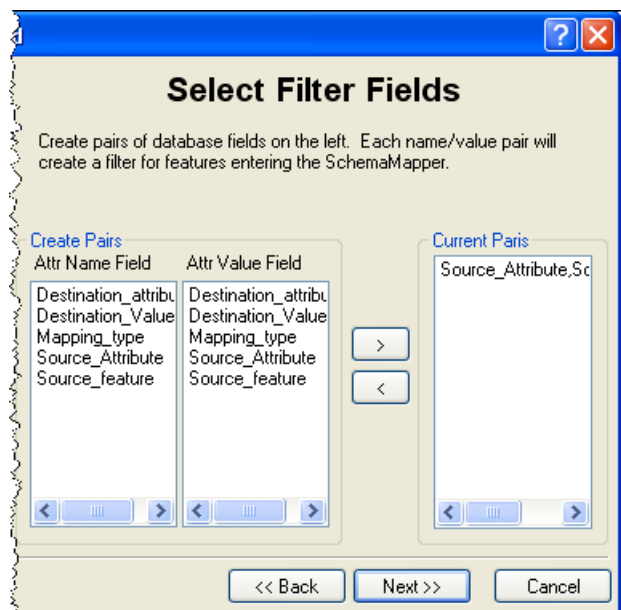
Again, it is necessary to expose the *fme_feature_type* on at least one of the source features.

This *SchemaMapper* will search for the *fme_feature_type* in our three MapInfo sources, then using the CSV file, it will search to find the corresponding attribute, and set the value of the new attribute to the value specified in the CSV file.

3) Examine the SchemaMapper settings.

Let's take a look at the new *SchemaMapper's* settings:

This time we specify the source database at *Attribute_mapping.csv* in the *Select External Database* pane. When prompted to *Select Data Table*, we pick *Attribute_mapping*.



Left: Select Filter Fields pane

This pane allows us to specify a *Attr Name Field* and *Attr Value Field* pair. What does this mean? You are defining the fields (columns) in the CSV file that are going to act as the filter or query on the CSV – as we described above:

*if the name of the attribute named in the field **Source_Attribute** equals the value in the field **Source_Feature** then ...*

Because there is no *Feature Type* mapping happening in this *SchemaMapper*, the **Do Index Mapping** pane is left unchecked and the fields blank.

Right: Select Mapping Fields pane

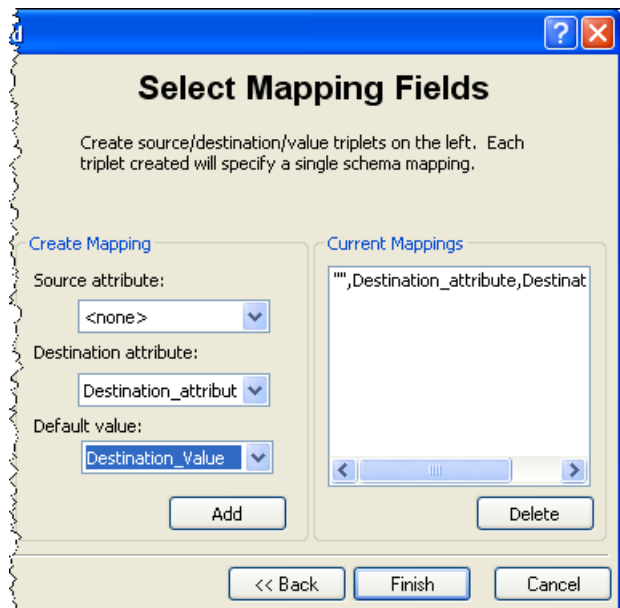
In the *Select Mapping Fields* pane, we select the CSV field names that hold the attributes to be mapped.

There are many variations of mapping. In this case we are creating a new attribute and giving it a value so there is no source attribute.

*... create the attribute named in the field **Destination Attribute** and set the value to the field **Destination Value**.*

4) Run the workspace.

Add *Visualizers* or use *Redirect to Visualizer* and run the workspace. Inspect the results.



Could you eliminate the *AttributeCreator* and set a default for the *CanadianRoadType* using the *SchemaMapper* and the CSV file?

**Example 3: Schema Mapping Variations.**

There are many variations of schema mapping: renaming feature types, renaming attributes, setting the value of attributes based on the value of another attribute and so on.

Mappings may also be one-to-one (1:1), one-to-many (1:M) or many-to-one (M:1).

In all we have identified about thirteen different types of mapping and these are summarized on the final page of this document and in the spreadsheet:

C:\FMEData\Resources\SchemaMapper\Examples\SchemaMapper_MappingTypes.xls

The workspace *SchemaMapping_Examples.fmw* illustrates how to configure the *SchemaMapper* transformer for these different mapping types, using the CSV files in the directory:

C:\FMEData\Resources\SchemaMapper\Examples\SchemaMapping_Examples.fmw

Try and identify the specific type of schema mappings your organisation could use.

Schema Mapping References

The *SchemaMapper* can be used in many different ways. Several examples are available at:

<http://www.fmepedia.com/index.php/SchemaMapper>

http://www.fmepedia.com/index.php/SchemaMapper_Example

http://www.fmepedia.com/index.php/SchemaMapper_Example_2

Type	Mapping type	Source feature	Source Attribute	Source Value	Destination feature	Destination Attribute	Destination value	Default value	Comments
1	Feature	x			x				1:0. 1 feature mapping One entry per feature
2	Feature	x	x	x	x				One table maps to many tables, based on attribute value Many entries per feature
3	Feature	x			x	x	x		Many tables map to a single table and attr value is set One entry per feature
4	Attribute		x			x		default	1:0. 1 attribute One entry per attribute
5	Attribute	x	x			x		default	The same attribute (type) in different tables map to different attributes in the destination feature type i.e. Pole/type > pole_type Manhole/type > manhole_type multiple entries per attribute, based on feature
6	Attribute	x	x	x		x		default	attributes in a table map to the different attributes based on value of the source attribute i.e. Pole/type = 1 > pole_type Pole/type = 2 > material Multiple entries per attribute, based on feature & value
7	Attribute		x	x		x		default	attributes in different tables map to different attributes based on value i.e. type = 1 > pole_type type = 2 > material so no need to filter on feature type
8	Attribute	x	x		x	x		default	Table + attribute mapping
9	Value			x			x	default	Simple dynamic value mapper. On lookup table per domain in the configuration, requires separate transformer for each domain
10	Value		x	x			x	default	configure the mapper to support several different attributes type X > A type Y > B material X > A material Y > B Also should allow changing of the attribute name as well as the value, i.e. type 8
11	Value	x	x	x			x	default	configure the mapper to support several different attributes for several tables. Still simple domain mapping.
12	Attribute & Value	x	x	x		x	x	default	This configuration allows one lookup table for all the features combination of attribute & value mapping
13	Feature & Attribute & Value	x	x	x	x	x	x	default	combination of feature & attribute & value mapping



www.safe.com

Safe Software values its customers' opinions very highly.
To provide feedback on FME training, access the feedback form at: www.safe.com/feedback.
For questions and concerns not covered by that form, please use the general feedback page at:
www.safe.com/company/contact/form.php.
...or email the Training Manager directly at: training@safe.com.