

2009 FME International User Conference: Training Module
Advanced FME Workflows





Advanced Module

Advanced FME Workflows

Advanced Workflow Techniques - 1	3
Workbench and the Flow of Features	3
Variables	4
Why Use the Scope Setting?	9
Advanced Parameters	10
Controlling Data Flows Using Parameters	10
Advanced Workflow Techniques - 2	14
Multi-Pipeline Workflows	14
Startup and Shutdown Scripts	19
What are Startup and Shutdown Scripts?	19
Creating a Script	19
Accessing FME Variables	20
Halting FME after a Startup Script	20
Startup and Shutdown Facts	20

Advanced Workflow Techniques - 1



Some of the advanced techniques for handling the flow of features in a workspace can be very effective – but not always intuitive. For that reason a little background info is first required...

Workbench and the Flow of Features

One aspect of FME you may not have considered is the way in which features flow through a workspace. There are a number of rules governing this concept, knowledge of which can help you work more efficiently with FME and avoid unexpected results.

Rule 1: *Features are processed in the order that they are read.*

Features are read in order from their dataset, and the datasets are read in the order they appear in the navigation pane. This order is consistent, unless otherwise affected by the results of a transformation.

Consequences of rule 1:

Features that are read first will reach any given transformer first. Some transformers take advantage of this with a setting that lets you confirm which features will be the first to arrive.

The *Clipper* transformer's 'Clippers First' setting is an example of this. It can improve efficiency by avoiding the disadvantages rule 3 brings.

A good example of this can be found at: <http://www.fmepedia.com/index.php/Clipper>

Rule 2: *Features are individually processed as far as possible in the workflow.*

This means that once a feature is read FME will process that feature as far as it can in the workflow before reading the next. It will process all features through each transformer in turn.

Consequences of rule 2:

Unexpected results can occur when a feature passes through a second transformer before other features have passed through the first. Global Variable transformers can be affected in this way.

Rule 3: *A Group-Based transformer holds all features until they are available for processing.*

Remember how transformers can work on a feature or group basis? Since group processing requires all features, a transformer of this type holds features in memory until it receives them all.

Consequences of Rule 3:

A transformer holding features in memory is consuming system resources.

With a very large dataset you may not want to do this, if there are alternatives. However, you can use a transformer of this type to purposely avoid the consequences of rule 2.

NB: *Features may also exit a group-based transformer in a different order to which they entered it.*

If this all seems too complicated the examples and exercises that follow will help to illustrate these rules and how to work them to your advantage.



Variables

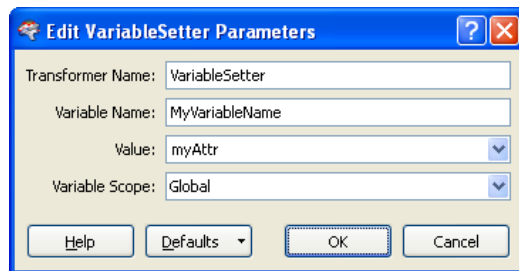
Variables – often also known as ‘global variables’ – are a feature of FME for passing information between streams of features; often from one feature to another that follows after it.

Because they are a way of managing data flows, the two variable related transformers come under the Infrastructure category of transformers.

VariableSetter

The *VariableSetter* transformer is a tool for creating or setting the value of a variable.

It was once known as the *GlobalVariableSetter*, but the word ‘global’ was dropped with the advent of custom transformers, when variables could apply to just one transformer within a workspace.



A *VariableSetter* transformer has a number of settings (left).

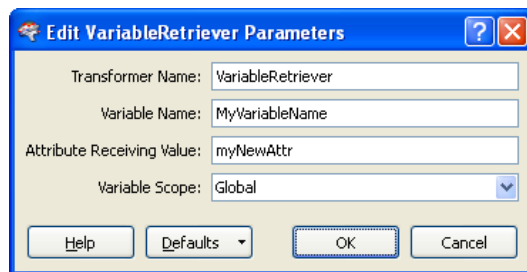
Variable Name is always a text-entry field. The names of variables are not held in such a way that they can be selected from a drop down list.

The value to which to set the variable can be either a constant value or an attribute selected from the drop down list.

Variable Scope can either be Global or Local. Global means it can be accessed anywhere within a workspace; Local is used within a Custom Transformer and means it can only be accessed within the Custom Transformer in which it is used.

VariableRetriever

The *VariableRetriever* transformer is a tool for extracting the value from a variable, and placing it inside an attribute. It too was once prefixed with the word ‘global’.



A *VariableRetriever* transformer has a similar set of settings to the *VariableSetter* (left).

Variable Name is again a text-entry field.

The attribute in which to store the retrieved value is a text-entry field and cannot be selected from a list.

Variable Scope can again either be Global or Local.

Associative Arrays

Associative Arrays are like a matrix of values that can be used as a dynamic lookup table. To create one use an attribute as the *VariableSetter* variable name using the @Value() function.

This use is complex to explain, but a *VariableRetriever* can retrieve such values (again using @Value in the name) and it’s even possible to do a reverse lookup from value to attribute!

Global Variables and the Flow of Features

The flow of features in a workspace is important for Variables because it is usually vital to make sure that the feature setting the variable does so before another feature tries to retrieve it.

The *FeatureHolder* transformer can be used to control the flow of features.



Example 1: Feature Flow and the FeatureHolder

Here we have a number of features, and want to count how many there are. Normally the *StatisticsCalculator* transformer would do this for us, but here we'll use Variables instead.

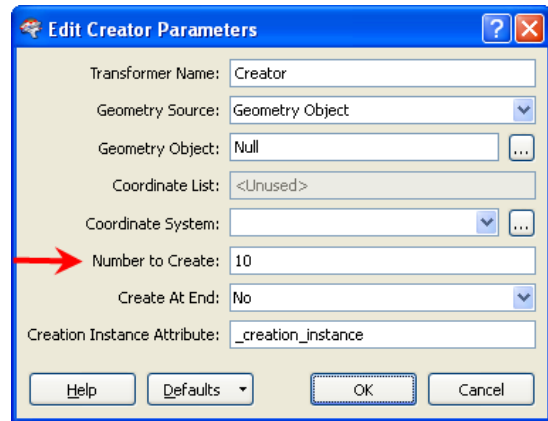
Instead of using source data we'll create our own data in the workspace itself.
Start Workbench with an empty workspace.
Place a *Creator* transformer.

Open the *Creator* settings dialog.
Change the 'number to create' setting to 10.

When each feature is created it is tagged with an ID number (*_creation_instance*).

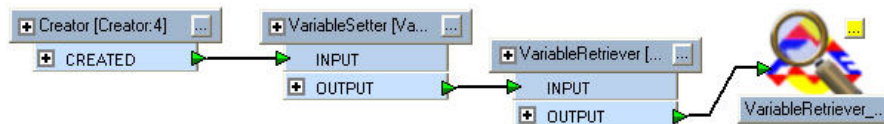
We'll use Variables to find the largest ID number.

Connect a *VariableSetter* after the *Creator*.
Select *_creation_instance* for the 'value' field.



Connect a *VariableRetriever* after the *VariableSetter*.

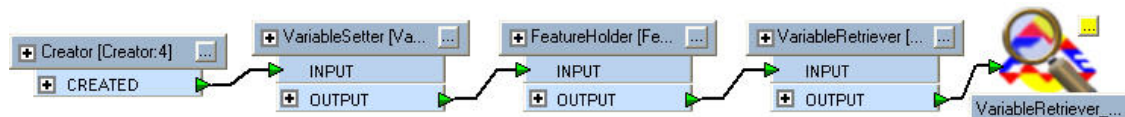
Change the 'attribute receiving value' to be *max_id*. Connect a *Visualizer*. In the *Visualizer* settings choose to group by the attribute *max_id*. Now run the workspace.



Examine the output. Notice how the *max_id* is always the same as the ID of that feature. This is emphasized by a separate output feature type for each feature.

This has proved Rule 2. If all features were processed through a transformer at the same time then the *max_id* would be the same (9) for all the features.

Insert a *FeatureHolder* transformer between the two variable transformers. This will hold features here, and stop them progressing to the *VariableRetriever* until all features are ready.



Run the workspace and examine the output.

Notice how the *max_id* value is now the same for all features; and so all features appear on the same output feature type. That's because all features passed through the *VariableSetter* first then waited at the *FeatureHolder*. This has proved rule 3.



Exercise 1

Let's do an exercise to give you practice at using Variables in a real-life situation.

In this exercise we have been provided with a plain CSV file containing X/Y coordinates and need to build line features (roads) from them. The features need to be tagged with attributes. This is a common requirement – in fact FME handles some formats (Z-Map, SEG-P1) this way.

Detailed Steps

1) Inspect the Source Data

Inspecting the source data should always be your first step before starting a translation. Since the source data is a plain CSV file you will do best to open it in a text editor.

Source Format Comma Separated Value (CSV)
Source Dataset C:\FMEData\Roads\Roads.csv

Can you uncover the structure of this data? It is not as simple as you might think. The vertices in road features are represented by consecutive records within the file, and rather than each vertex having attributes, each road has attributes stored as a header record indicated by a “!” character.

The ability to attach the header attributes to each feature – without the benefit of an associated ID number – is a clear case where variables need to be used.

2) Create a New Workspace.

Start Workbench if necessary and create a new workspace to translate the source CSV data into the format named “Generic (Any Format)”. The source setting “Has Field Names” must be set.

3) Filter out Header Records

The first thing to do will be to filter out header records from which to extract the attribute. A *StringSearcher* transformer (aka *Grepper*) will be able to do this.

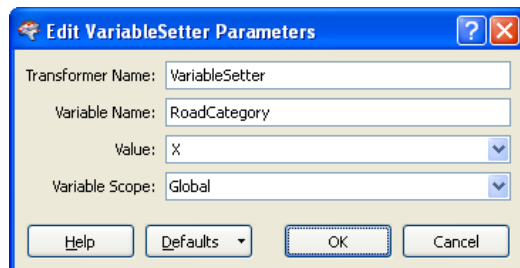
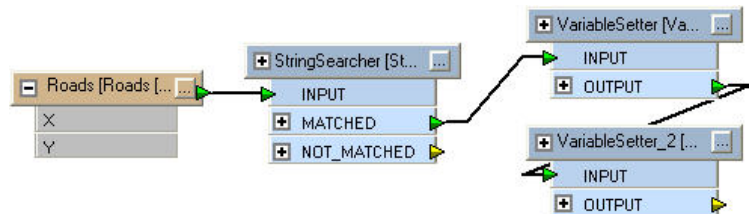
Add a *StringSearcher*. Set up an expression to find records which start with a ! character. The *StringSearcher* help tells you how to define an expression as “starts with”.

Connect the *StringSearcher* output ports to *Visualizers* and run the workspace to test your setup.

4) Set Variables

Now we've filtered off the Header records, we need to store their information as Variables.

Add two *VariableSetter* transformers (**right**):



In the first (**left**), call the variable RoadCategory and give it a value from the X attribute.

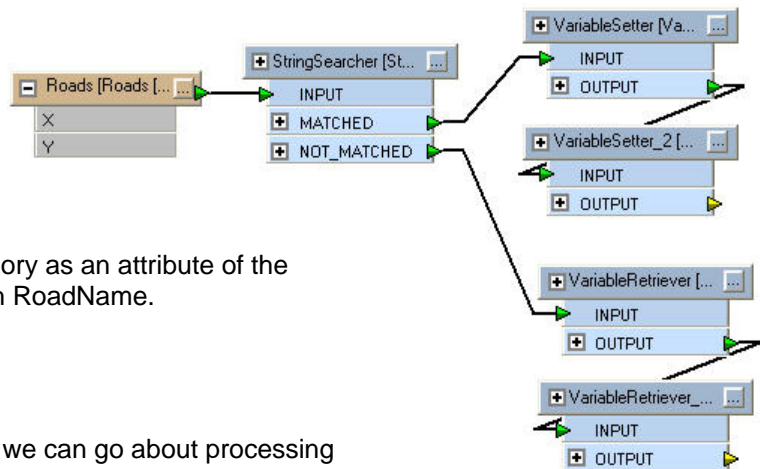
In the second, call the variable RoadName and give it a value from the Y attribute.

5) Retrieve Variables

Having set the Variables, we now have to retrieve them.

Add two *VariableRetriever* transformers to process the non-header records.

Retrieve the variable RoadCategory as an attribute of the same name, and do likewise with RoadName.



6) Process Points

Now we have the right attributes we can go about processing the GPS data and turning it into line features.

Use a *2DPointReplacer* and a *PointConnector* transformer to turn the non-header records into line features. The *PointConnector* break attribute should be RoadName.

Add a *Visualizer* transformer and run the workspace to prove the output is correct.

7) Fix Output

Does the output look correct? No.

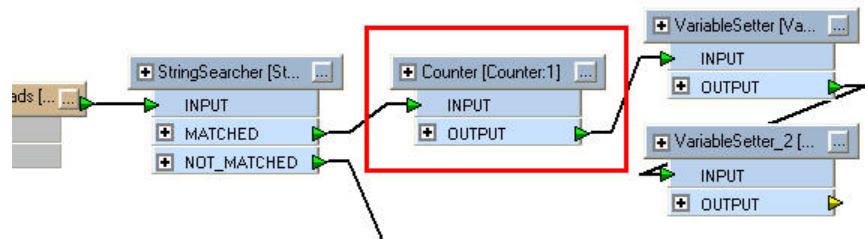
The problem is that there are several features with the same road name; we would be better off to create an ID number for each road and use that as the break attribute.

A *Counter* transformer can create an ID number for us.

However – where should the *Counter* be added? We want to count each road feature (not each point) so it cannot come before the *PointConnector*, but if it cannot be before the *PointConnector*, how can we use it as the break attribute?

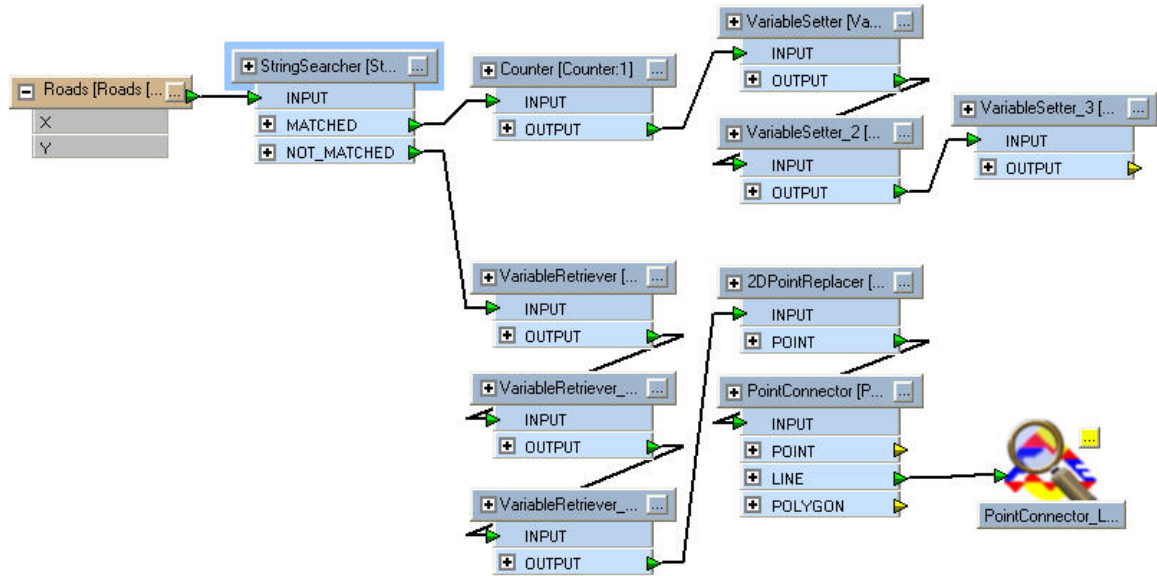
The answer to these questions is to count the header features and transfer the ID number using Variables again.

Add a *Counter* into the header processing pipeline, before the first *VariableSetter*. Set the count output attribute to be RoadID.



Add another *VariableSetter* and *VariableRetriever* transformer to transfer RoadID from the header onto the road features.

Change the *PointConnector* break attribute to be RoadID (rather than RoadName).



At this point your workspace should look like this (**above**) and your output like this (**below**):

8) Fix RoadCategory

Query a feature in the FME Viewer.

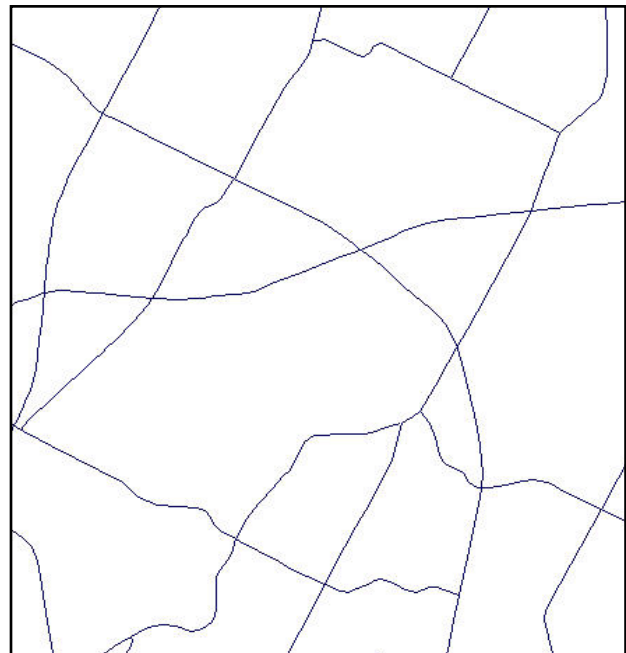
Look at the value for RoadCategory.

It still has the ! marker character in front of it. We should remove that.

Remove the extra character using either the *SubStringExtractor* or *StringReplacer* transformer.

Because the value of this attribute may vary in length you cannot rely on using fixed length strings.

Run the workspace again to test your solution.



9) Save Workspace

The following exercises will build on this workspace, so save a good copy of it.

Do you see how this workspace works?

We are relying on features being read in the order of the data, and staying in that same order as they are processed. We are also relying on features being processed one at a time, through the entire workspace, rather than in a group at every transformer.

Discussion Topics: There are a couple of ways in which this workspace could be broken.

- If the CSV structure were different (for example, records not in sequential order)
- If a group-based transformer were used before the Variable transformers; for example the PointConnector before the VariableRetrievers – do you see why this is the case?

Why Use the Scope Setting?

The ability to set the scope of a transformer occurs not just in the *VariableSetter* and *VariableRetriever*, but also in transformers such as the *Counter*, where multiple instances of the transformer can interfere with each other.

Scope applies when the transformer is used within a Custom Transformer that is intended for use in a number of locations in a workspace.

It should be set to local when each instance of the Custom Transformer needs to be independent of all the other instances. It should be set to global when all instances of the Custom Transformer are intended to operate in tandem.



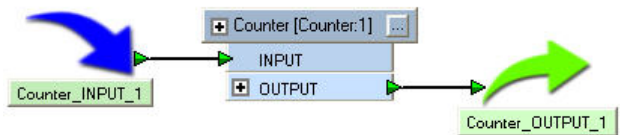
Example 2: Local or Global?

Here we want to create a Custom Transformer containing a *Counter* transformer.

Start Workbench. Place a *Creator* transformer. Change the 'number to create' setting to 10.

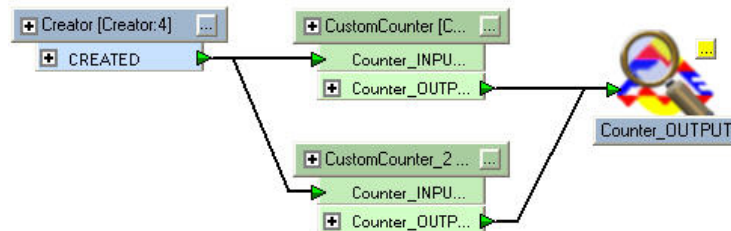
Connect up a *Counter* transformer and then a *Visualizer*.

Right-click the *Counter* transformer, choose the option to Create a Custom Transformer, and name that new transformer *CustomCounter*.



Run the workspace. You will get ten features numbered from 0 to 9.

Back in the main canvas place another copy of the *CustomCounter* transformer – you can either copy and paste the original or look up the transformer in the transformer gallery.



Create a second pipeline of data by connecting the *Creator* to the second *CustomCounter* and connecting that on to the *Visualizer* transformer (**left**):

Now re-run the workspace. You will get 20 features in the output... but will they be one set of features numbered 0 to 19, or two sets of features numbered 0 to 9?

The answer depends on the Scope setting of the *Counter* inside the Custom Transformer.

If Scope is set to Global then the same counter name will be used for every instance of the custom transformer, giving one set of features numbered 0 to 19.

If Scope is set to Local then a different counter name will be used for every instance of the custom transformer, giving two different sets of features each numbered 0 to 9.

This setting is very important in workspaces that use *Counters* to set feature IDs for writing to a database: using Local instead of Global can result in duplicate ID numbers in the same table, whereas using Global instead of Local can result in missing ID sequences in multiple tables.

Advanced Parameters



You may be familiar with using published parameters to manage run-time settings, but did you know it's possible to use them to control the flow of data?

Controlling Data Flows Using Parameters

Controlling the flow of data through a workspace often depends on decisions relating to the state of geometry or attributes at various points. However, various items of FME functionality allow the user to make certain decisions before the workspace is carried out.

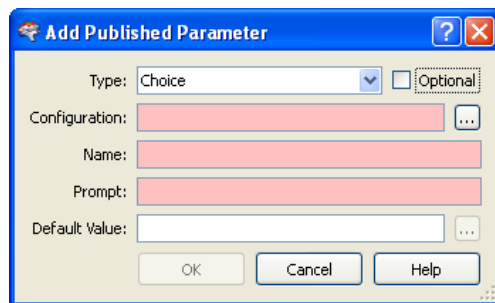
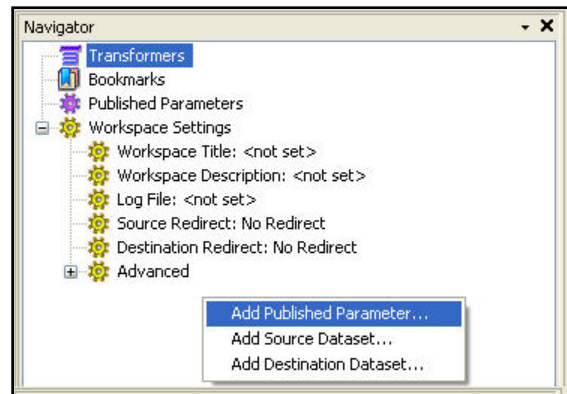
The first new item that helps this aim is Custom (aka Standalone or Independent) Published Parameters.

Creating Custom Published Parameters

“Custom” parameters are different to the usual form of published parameter in that they are created independently of any transformer, reader, writer or feature type. In effect the user creates them to use however he or she desires within the workspace.

Right: Custom parameters can be created in a number of ways:

- Right-clicking on the Published Parameters title
- Right-clicking any area of blank space in the Navigator pane (**right**)
- Using the Insert option on the menubar



Left: The dialog box for creating a published parameter has a number of required fields. These are highlighted in red and will remain that colour until satisfactorily completed.

The name and prompt fields are exactly the same as are used for regular published parameters; a name to identify the parameter and a short phrase to use for prompting a user to enter a value.

The default value field is also similar, and is where a user can set the parameter’s default value.



A parameter – published or otherwise – can be described as a global constant. In other words its value is set when the translation is started, and remains the same throughout the process.

Custom Published Parameters – Other Settings

Parameter Type can be used to create various types of parameter:

Choice:	A single choice from a pick list of values
Multiple:	Multiple choices from a pick list of values
Choice or Text:	A choice from a pick list, or the ability to enter a plain text value
Color Picker:	A colour selection chart that returns values in FME RGB notation
Coordinate System Name:	A choice from the complete list of FME coordinate systems
Directory:	The selection of a directory or folder
File Name:	The selection of a single file
Float:	A floating point number
Integer:	An integer (number with zero decimal places)
Password:	A text entry field whose value is masked by asterisk characters
Text:	A text entry field

The Configuration field is where extra information can be provided for some of these parameter types. The supported types are:

Choice:	Dialog to configure the pick list of choices
Multiple:	Dialog to configure the pick list of choices
Choice or Text:	Dialog to configure the pick list of choices
File Name:	Dialog to configure a file description and filter (for example “My GPS Files” - *.gps)

The Optional field is also new. This is a check box that identifies whether this parameter is optional, or whether it must be given a value before the workspace can be run.

Fetching Parameters

Because an independent parameter has no form of link to an existing transformer, reader or writer, there needs to be a method by which a workspace can access its value. This is achieved through the *ParameterFetcher* transformer.

The *ParameterFetcher* provides a list of all independent published parameters.

The workspace creator defines the parameter to be fetched, and the attribute into which to fetch it.

When the translation reaches this part of the workspace, the value of the chosen parameter is passed into the target attribute.



That attribute can then be used within the workspace in the same way as any user attribute; searched, concatenated, tested, etc.

Remember – the value of a parameter is fixed. You cannot change it, and nor can you read it into an attribute, update it and write it back. There is no such transformer as the *ParameterWriter*!



The list of parameters includes all those attached to an existing transformer, reader, or writer, thus allowing a user to obtain the value of these parameters too. The ParameterFetcher also permits the definition of a parameter name as a piece of text. This lets a user obtain the values of pre-defined macros such as FME_BUILD_NUM and FME_HOME by entering their name manually.

Testing and directing to different sections

The combination of standalone published parameters, and *ParameterFetcher* transformer, allows a workspace designer to set up a workspace to route features through different transformers, depending on the choice of the user at run time.

For example, a user may be prompted as to whether he or she wants labels on the output data. If the answer is yes then the features are routed through a *Labeller* transformer. If the answer is no that section of workspace is bypassed.

This becomes particularly useful when used in conjunction with the Generic Writer; for example with Shape format there is no point in creating labels, as Shape does not support text geometry.



Exercise 2

This exercise will continue where exercise 1 left off. We will now ask for user input as to how to style the output data.

Detailed Steps

1) Open Workspace

Open the completed workspace from exercise 1, or use the workspace *Exercise2-Begin.fmw*

2) Create Custom Parameter

Create a new published parameter.

Type: Choice
 Optional: No (i.e. don't tick the checkbox)
 Configuration: Two options – Yes and No
 Name: LabelRoads
 Prompt: Create labels from road names?

3) Place ParameterFetcher Transformer

Place a *ParameterFetcher* transformer just before the final output to fetch the LabelRoads parameter into an attribute called LabelRoads

4) Place Tester Transformer

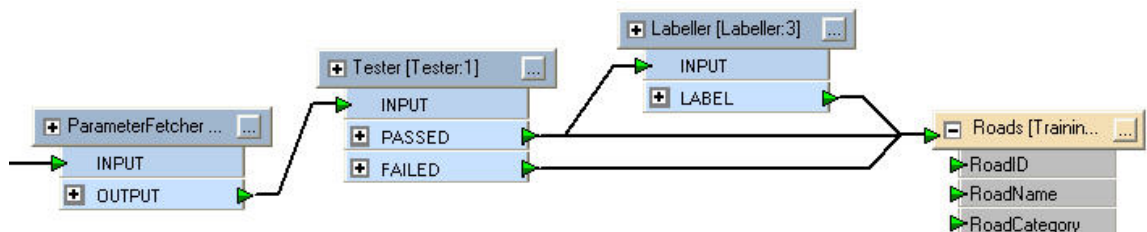
Place a *Tester* transformer after the *ParameterFetcher*. Test if the value of LabelRoads is Yes.

5) Place Labeller Transformer

Place a *Labeller* transformer. Connect it to the PASSED port of the *Tester*. Connect the *Labeller* output and the *Tester* FAILED ports to the Roads destination feature type.

Because the *Labeller* will **replace** the road features, you will also need a connection that from the PASSED port to the output but that bypasses the *Labeller*

Below: At this point the final part of the workspace should look something like this.



6) Check Labeller Settings

Update the *Labeller* settings as follows:

Label Attribute: RoadName
Label Offset: 1
Label Height: 500
Label Spacing: 9999

7) Redirect Output

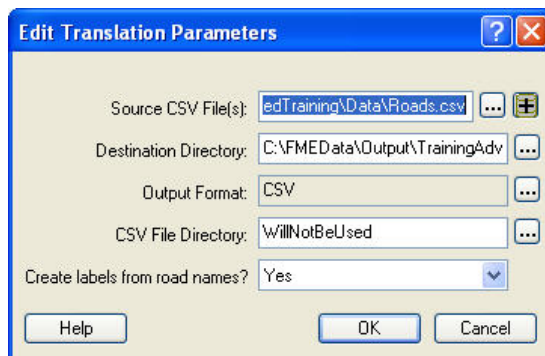
Redirect the workspace output to the FME Viewer.
 Run the workspace using File > Prompt and Run Translation, or use Ctrl+R as a shortcut.

You should be prompted as to whether or not to create labels.
 Try running the workspace with this parameter alternatively set to Yes and No.

8) Clean Dialog

By now you should be able to turn labels on and off at will.

However, the dialog that opens at run time is not very neat; it may have any of the extra parameters shown here (*right*). Clean it up by deleting any parameters that you feel no longer require to be published (*below*)



Advanced Task

Publish the Generic Writer Output Format parameter (if it is not already published).

You will now also be able to set that in the run time dialog, and to fetch the chosen format with the *ParameterFetcher*.

For this task fetch the format parameter and test to see if it is KML. If it is then use the *KMLStyler* transformer to set some symbology settings for line colour and line width.

Incidentally, the short name for this format is OGCKML.

Are there any other parameters you need to set to write to KML?
 Running the translation without this parameter should reveal the problem!
 Which transformer could you use to fix this issue?

Remember to turn off the output redirection, if you actually want a KML output file!

Advanced Workflow Techniques - 2



Dividing data up into multiple flows can result in some creative ways to carry out processing.

Multi-Pipeline Workflows

In most cases you won't want to duplicate data within a workspace, because adding extra features adds to the burden on system resources.

However, using multi-pipeline (aka stream or flow) workspaces creatively can be of great benefit, because you can leave a set of features in their original state, and merely attach the results of processes carried out on the duplicated features.

Example 3: SelfIntersectionFilter

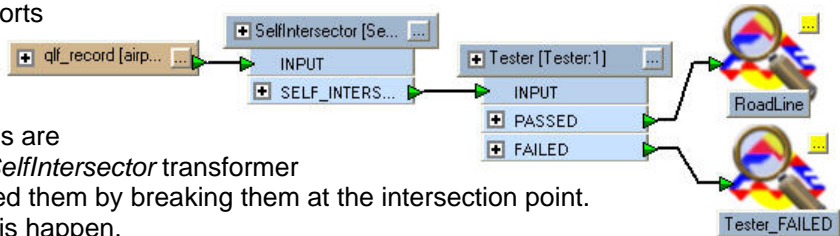
As an example of this technique, let's create a SelfIntersectionFilter custom transformer. Here we wish to find self-intersecting lines, but not try to fix them.

In an empty workspace add the following source data:

Source Format CITS Data Transfer Format (QLF)
Source Dataset C:\FMEData\Data\Airport\airport.qlf

Add a *SelfIntersector* transformer, then a *Tester* to test where `_segments > 1`. This will help us to divide self-intersecting from non-self-intersecting lines.

Attach the *Tester* output ports to *Visualizers* and run the workspace.



You will see which features are self-intersecting, but the *SelfIntersector* transformer will have automatically fixed them by breaking them at the intersection point. We do not wish to have this happen.

The simple solution is to keep the original source features and merely attach flags to indicate they self-intersect by merging that information from the intersected copy of the features.

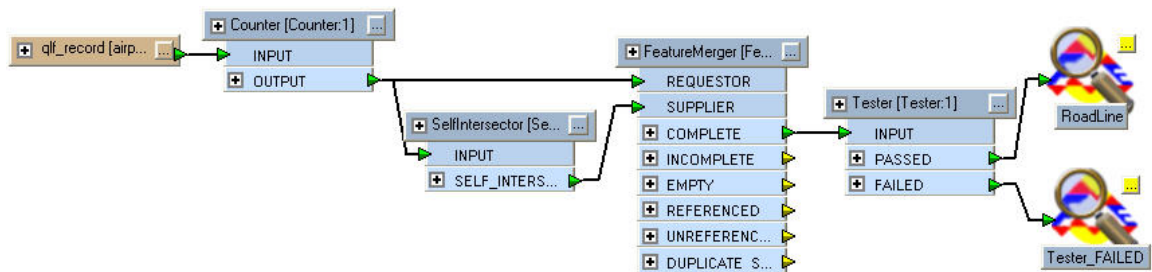
Add a *Counter* transformer before the *SelfIntersector*, to give each feature a unique ID.

Add a *FeatureMerger* transformer.

Create a new connection out of the *Counter* into the *FeatureMerger* REQUESTOR port. Connect the intersected features to the SUPPLIER port. Use the new ID attribute to join the intersected info back on to the non-intersected features.

Connect the *FeatureMerger* COMPLETE port to the *Tester*.

At this point the workspace will look something like this:



Re-run the workspace.

You will now be able to identify the intersecting features, but they will be untouched from the original geometry.

Discussion Items

As an advanced task, why not try to turn this into a Custom Transformer called the *SelfIntersectionFilter*? If you do, remember to consider whether to make the Counter Scope global or local.

This workspace is an interesting example of QA (or QC) versus Data Cleaning. FME's instinct is to clean data, in order to ensure it will load into a destination dataset. However, the user often does not want automatic data cleaning, but just an indication of where he/she needs to fix problems.

Also, are there any other FME transformers that could be used to carry out this task? The *SpatialRelator* can be set up to test for intersections, but can you turn this into a test for self-intersections? (NB: I haven't tried this – so don't automatically take this as an advanced task!)



Exercise 3

This exercise will continue where exercise 2 left off.

We want to find out which roads are crossed by rivers or streams.

The *LineOnLineOverlay* transformer will do this, but will have the effect of cutting the road at each intersection point, which is obviously unacceptable in a topologically correct network.

So we will use the previously described method to give the correct result.

Detailed Steps

1) Open Workspace

Open the completed workspace from exercise 2, or use the workspace [Exercise3-Begin.fmw](#)

2) Add Hydrography Data

Add the rivers and streams dataset to the workspace as a new source reader.

Source Format

MapInfo MIF/MID

Source Dataset

[C:\FMEData\Data\Hydrography\HydrographyLine.mif](#)

3) Add LineOnLineOverlay Transformer

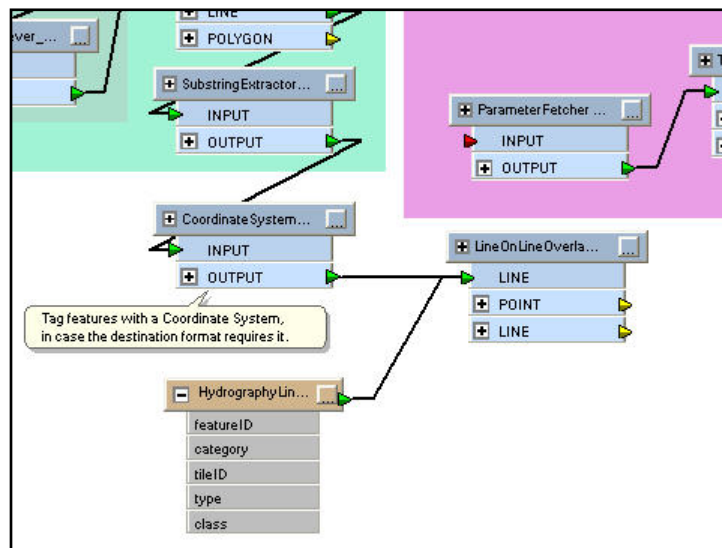
Add a *LineOnLineOverlay* transformer.

It needs to be inserted after the linework has been created, and probably before the user-defined sections, in case it needs to be used there at a later point.

Connect the *LineOnLineOverlay* before the first *ParameterFetcher*.

Also connect in the Hydrography features.

Right: At this point your workspace should look something like this:



4) Add Tester Transformer

A *Tester* transformer will help to check which features overlap a hydrography feature, and to separate out any road-on-road intersections. The test should be carried out on the intersection points, which emerge from the overlayer's POINT port (not the line – we will identify intersections by a set of attributes from both datasets; only the point output features will have these).

Add a *Tester* connected to the *LineOnLineOverlay*'s POINT output port.

Set up the *Tester* to check for points that include a hydrography overlap. These are all features that have a set of the Hydrography dataset's attributes on them, so you can test for values of featureID, category, etc.

5) Add FeatureMerger Transformer

Add a *FeatureMerger* transformer to get the overlay results back onto the road features. Information on the overlap points is SUPPLIED and the road features are REQUESTING.

The join attribute is RoadID.

This part will also filter out stream-on-stream intersections, as these won't match a RoadID and will be ignored.

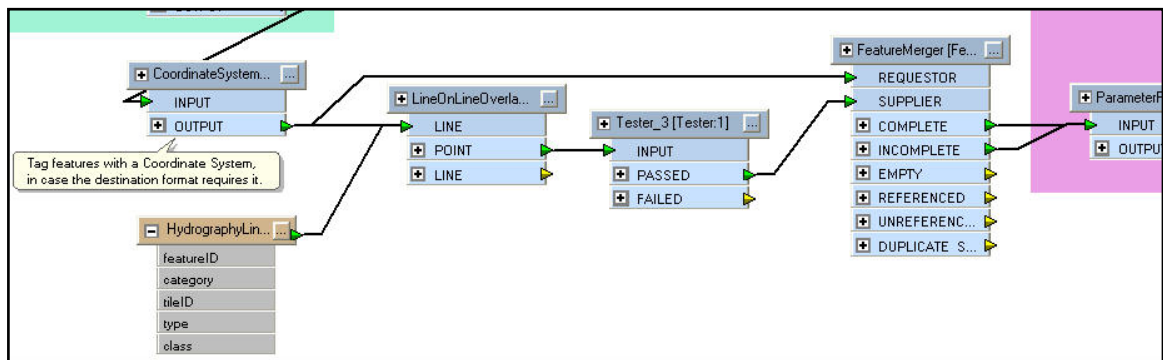
6) Re-Connect Road Features

The COMPLETE features from the *FeatureMerger* are road features that have been tagged with a stream intersection.

The INCOMPLETE features from the *FeatureMerger* are road features that do not intersect streams.

Both of these need to be reconnected to the original ParameterFetcher.

Below: After a bit of tidying this section of workspace should look something like this:

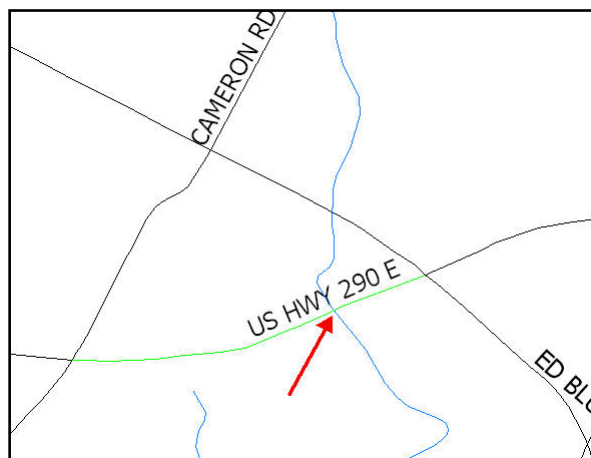


7) Run the Workspace

Run the workspace and inspect the output.

All road features that intersect a stream should have an attribute called `_overlaps` that has a value greater than 1.

Overlay the hydrography features in the Viewer to check the results of the workspace.



Left: This section of HWY290E crosses a stream at the highlighted point, and therefore has an `_overlaps` attribute (**below**):

Feature: 1 of 1	
Feature Type: Roads	
Coord Sys: TX83-CF	
Attribute Name	Attribute Value
<code>__wb_out_feat_type__</code>	Roads
<code>_overlaps</code>	2
<code>category</code>	Hydrography
<code>class</code>	Intermittent
<code>destFormat</code>	KML21
<code>featureID</code>	4434
<code>fme_basename</code>	Roads
<code>fme_color</code>	0.1.0



Advanced Tasks

This workspace is all but complete now, but here are some additional tasks in case you are looking for extra practice.

1) Hydrography Overlap Flag

A simple one this. Use an *AttributeCreator* to create an attribute `OverlapsHydrography`, to better flag road features that overlap hydrography features.

2) Output Overlap Points

To show the location of bridges, create a new destination feature type which will hold the intersection points of roads and hydrography.

Connect up the correct data to this feature type (this might take some thinking about).

Right: You may find that this particular bridge is missing. Can you figure out why and fix the problem?

Add a new published parameter to ask the workspace user whether this output is required, and filter the data according to the potential results of this question.



3) Clean Up Dialog

Clean up the run-time dialog by removing the mid/mif Hydrography prompt. Can you adjust the order in which the prompts appear in this dialog?

4) GeometryReplacer

One other technique that is similar to the Duplicate Data–FeatureMerger method, is to use the *GeometryExtractor* and *GeometryReplacer* transformer.

The *GeometryExtractor* stores the original geometry of a feature, and the *GeometryReplacer* returns it when the required processing has taken place.

This is a very difficult challenge, but can you adjust this workspace to use this technique? The problems include how to identify overlapped features, and how to extract the overlaps.

When you run the workspace, is it quicker or slower than the original? Does it use more memory?

If you weren't able to do this task, then check the completed workspace to see how it is done.

Startup and Shutdown Scripts



Some of the advanced techniques for handling the flow of features in a workspace can be very effective – but not always intuitive. For that reason a little background info is first required...

What are Startup and Shutdown Scripts?

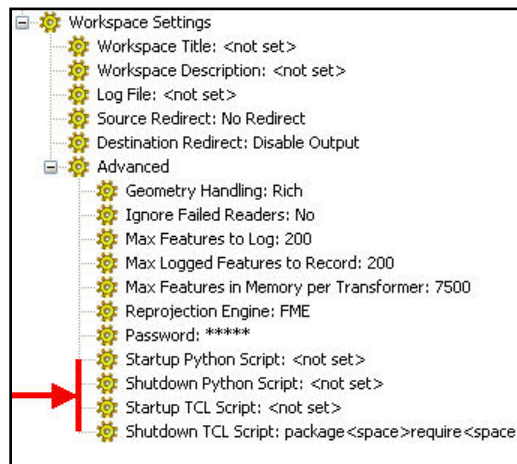
Startup and Shutdown scripts are an important component of FME because they allow you to run a script either before or after the execution of an FME workspace.

Typically a startup script would be used to check for some potential problem that might cause the workspace to fail. Finding out a problem before the translation starts is much more desirable than spending a long time translating data in a process that ultimately will fail. The method can also do something simple like maneuvering source data, or checking if an output dataset already exists.

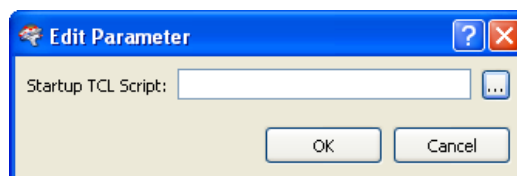
A shutdown script is often used to process the translation output in an additional way – for example copying output files to a new location – or to create some form of custom logging; for example sending an email (or Tweet!) to an administrator if the FME translation has failed.

Scripts can be in either TCL (Tool Command Language) or Python.

FME has built-in interpreters for both TCL and Python, so that scripting language can be used with no further installation.



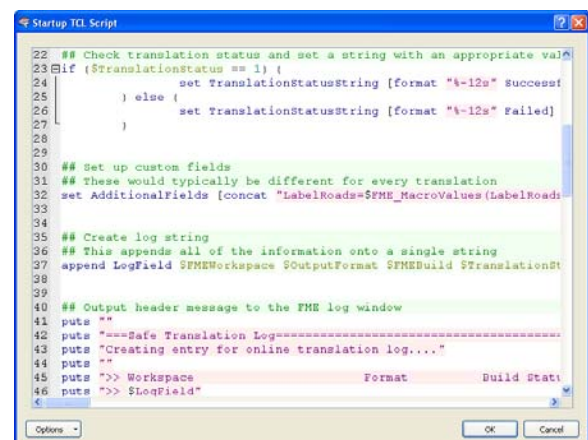
Below: Double-clicking a script setting opens this dialog. Clicking the ellipsis button opens a script editing dialog (right):



Creating a Script

Although scripts can be created externally and then accessed by FME (use the Tcl *source* or Python *import* command), or embedded into a mapping file, the easiest way to create and distribute a script is by using tools available under the advanced workspace settings section of the Navigator pane.

Left: Script creation settings in the Navigator. There are settings in which to define both startup and shutdown scripts for both Tcl and Python.



Accessing FME Variables

Shutdown and startup scripts allow access to various FME global variables, published parameters, and predefined macros. For this section we'll concentrate on Tcl rather than Python.

FME Global Variables

Tcl shutdown scripts have access to a number of global variables that represent translation statistics. For example FME_Status tells the user if the translation failed (0) or was successful (1), and FME_ElapsedTime provides the time taken to complete the translation.

Global variables (in this case FME_Status) can be accessed very easily using the syntax:

```
set myVariable $FME_Status
```

All available global variables are listed in the FME Fundamentals help manual. Plus, startup and shutdown Tcl scripts share a common interpreter, which means that you can create your own global variables during startup, and access them during shutdown.

Published Parameters

In Tcl, published parameters are accessed using a global variable called FME_MacroValues:

```
set myVariable $FME_MacroValues(myPublishedParameter)
```

Predefined Macros

You may be familiar with predefined macros such as FME_MF_DIR, and in fact there are others which can be similarly accessed within a Tcl script, for example:

```
set myVariable $FME_MacroValues(FME_BUILD_NUM)
```

For Python, you previously needed to import `__main__` and use `__main__.FME_MacroValues`; but for FME2009 onwards you can simply use `pyfme`, for example:

```
import pyfme
myVariable=pyfme.FME_MacroValues[ 'FME_BUILD_NUM' ]
```

Halting FME after a Startup Script

FME will not continue with a translation after a startup script has failed.

To continue a translation – after a failed script – you would need to catch the error as it occurs. A user can also purposely stop a translation by ending a Tcl script with the *error* command.

Startup and Shutdown Facts

- ESRI users can use Python scripts to access ArcObjects, meaning they can run scripts to drop topology before a translation, and rebuild it afterwards.
- Startup/Shutdown scripts cannot be used inside a custom format (fds) file
- Tcl scripts can't access FME Objects functionality, which means no use of functions or factories. Python can, but only functions that do not require a feature.
- Multiple scripts can be executed, but is not particularly recommended.
- A Tcl script can call another workspace or FME process using the `exec` command – but you should direct stdout away from the `exec` call by appending `2> NUL`:

More information about startup and shutdown scripts is available in the FME Workbench help, the FME Fundamentals manual and on fmeopedia.com



Exercise 4

This exercise will continue where exercise 3 left off.

When the translation is complete we want to run a Tcl script that creates a custom translation log and alerts an administrator to any problems.

Don't worry – you don't need any knowledge of Tcl or programming to do this exercise.

Detailed Steps

1) Open Workspace

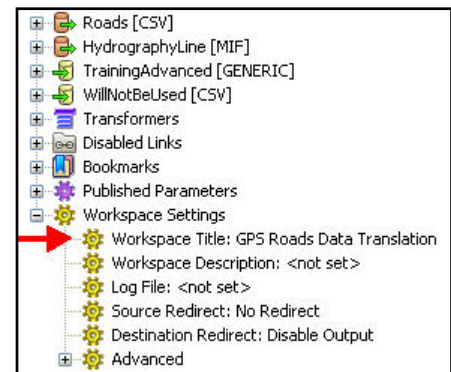
Open the completed workspace from exercise 3, or use the workspace [Exercise4-Begin.fmw](#)

2) Add Workspace Name

To create a log entry our script will need to know a name for the workspace. The workspace title in the navigator pane has been linked to the directive `MAPPING_FILE_ID`.

So if we set a title, we are setting `MAPPING_FILE_ID`, which we can then get the value of using the global variable `MappingFileId`.

In the navigator pane locate the `Workspace Title` setting and set it to: **GPS Roads Data Translation (right):**



3) Create Shutdown Script

Now let's create a shutdown script. Actually, this is already created, we just need to copy and paste it into the correct location.

In the Navigator pane locate the setting `Shutdown TCL Script`. Double click that setting. Open the script editor window.

Open the text file `C:\FMEData\Resources\Miscellaneous\TclScript.txt` and copy/paste the contents into the Shutdown Tcl script editor dialog; or if you know how, reference it in with the Tcl `source` command.

NB: For Python users an equivalent script is located at:

[C:\FMEData\Resources\Miscellaneous\PythonScript.txt](#)

4) Examine Script

Before we run the translation let's take a look at the script and see what it does.

In general the script reads a number of translation parameters, and creates a single string out of them to act as an entry to a translation log. The translation log will hold a summary of all the translations carried out (for example at a particular organization) in a fixed column format.

The translation log we are going to create/use here is an online log that accepts HTTP POST requests, which we create and send with the Tcl script.

You can look at the online log by browsing the web site: www.safe.com/results

The PHP script and CSS style sheet that make up this web site can be found in the folder `C:\FMEData\Resources\Miscellaneous` – the PHP script includes a schema of the underlying MySQL database in its comments; only the server name, username, and password are missing.

Having posted details of the translation to this log, the script then checks to see if the translation was successful or whether it failed. In the case of a failed translation it sends an email off to a system administrator to inform them of the problem.

For detailed information see the script itself – it is well documented and fairly self-explanatory. The only unfamiliar details might be the http and mime/smtp commands, although these are a part of the Tcl library that ships with FME and do not need to be separately installed.

5) Run Translation

OK, let's run the translation. Notice the extra information output to the foot of the FME log window.



NB: Should the process fail check that the names for your published parameters are the same as the ones referenced in the Tcl/Python script. This problem is indicated by an error like “no such element in array”.

Check www.safe.com/results (refresh the page if necessary)
You should notice your translation at the top of the list (**below**).



Date	IP Address	Result	
2008-02-19 10:43:50	66.119.171.18	GPS Roads Data Translation KML21 5175 Successful LabelRoads=Yes ShowBridges=No	Delete

6) Break the Translation

That's a successful translation – now think up an ingenious way to break it and run it again. (*hint: a transformer with bad settings – eg the 2DPointReplacer – or a format that with unsupported geometry types, is a sure way to cause a failure*)

Notice how the translation now reports that it Failed (instead of “Successful”).

Of course the attempt to email the administrator will also fail because the account information included in the script is bogus (but note how the script caught the email failure and didn't crash).

If you know account information for an email account or server that you have access to, then you may wish to enter that into the script and run it, to prove that this works.

7) Congratulations

Well done – this set of exercises is complete. You now have a workspace that translates some custom source data into a format of the user's choosing, with user-defined content and styling, whilst carrying out a set of spatial processes, and which keeps a record of the translation for admin and metadata purposes.



Advanced Tasks

If you desire one final task, why not create a parameter to let the user choose whether or not to write to the online log. Such a setting would allow the workspace to be tested without continually flagging failures. However, you will need to edit the Tcl script to accomplish this.

Also, if you're both a Python guru and Twitter addict, try getting FME to tweet messages. You'll need to install a Python Twitter API such as <http://pypi.python.org/pypi/twitter/1.2.1> and use a *PythonCaller* transformer to issue the correct commands.



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.