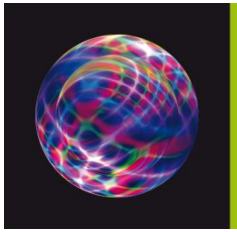
BSc. Isaac Boates (EIFER)
MSc. Ing. Alexandru Nichersu (EIFER)
Dr. rer. nat. Tatjana Kutzner (TUM)

Nanaimo Water Pipes Utility Network ADE Sample



EIFER

Supplementary Information
2017-12-08

Technical University of Munich

EDF

KIT Karlsruhe Institute of Technology

Introduction

This PDF details the specific steps that were used to improve the Nanaimo sample dataset (originally presented 2017-06-30 in Vienna), for a presentation in Karlsruhe on 2018-12-08 by Isaac Boates.

The steps included are carried out in FME Workbench. It is assumed that the reader is already familiar with FME, the UtilityNetwork ADE, and the steps involved in creating the original Nanaimo data sample. If they are not, tutorials on FME can be found at https://knowledge.safe.com/page/tutorials and information about the UtilityNetwork ADE and the Nanaimo sample can be found at https://github.com/TatjanaKutzner/CityGML-UtilityNetwork-ADE

The processes described are as follows:

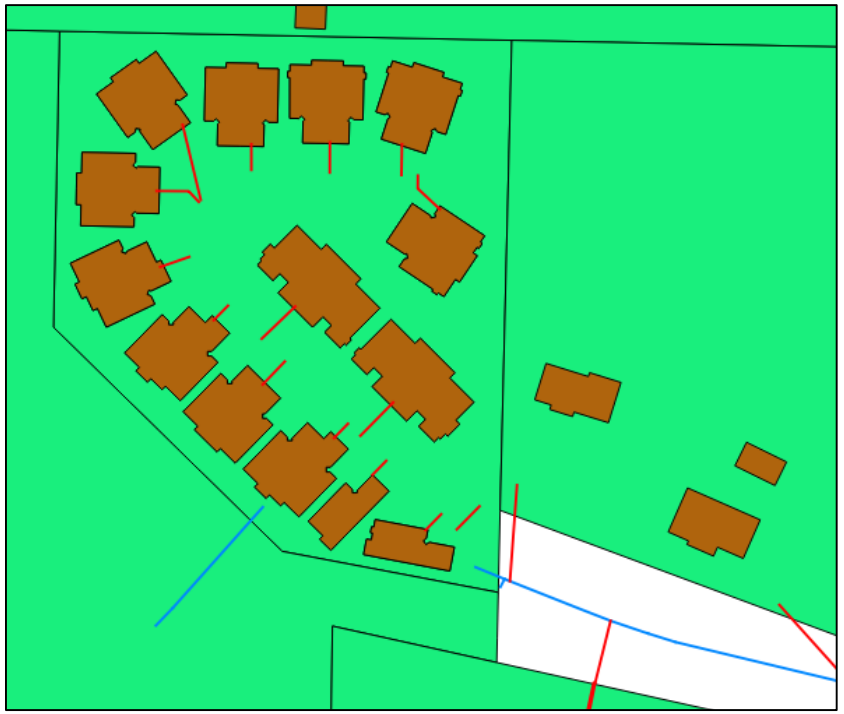
- Preprocessing Service Lines
- Preprocessing Buildings (& Parcels)
- Adding Buildings & TerminalElements
- Adding InterFeatureLinks from TerminalElements to Service Lines.

Preprocessing – Service Lines

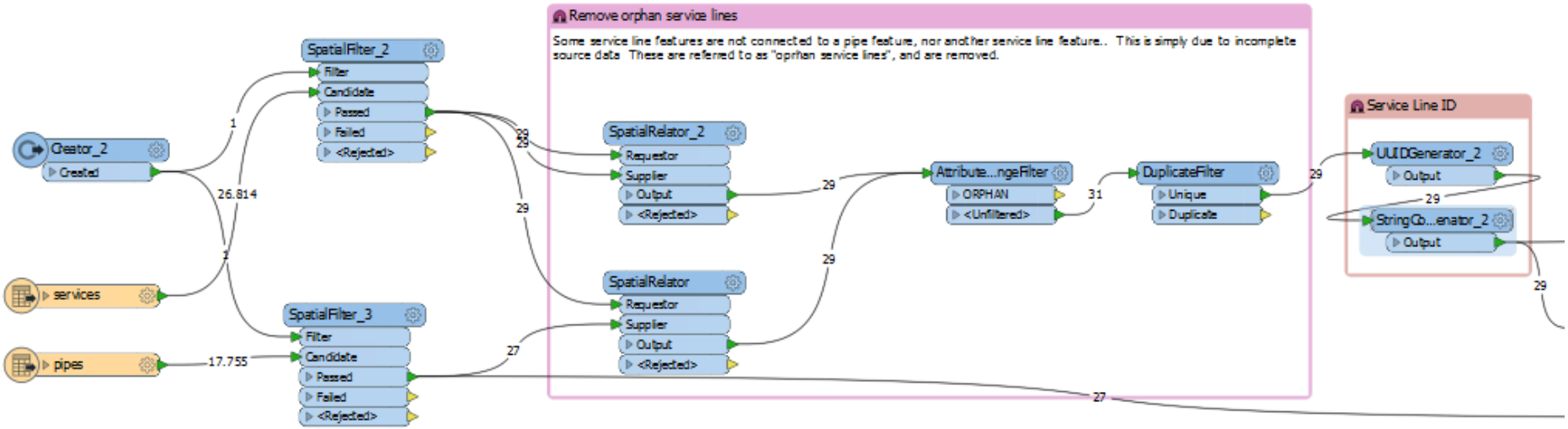
The „SERVICES“ shapefile contains the service line features, which are the pipes that lead from the main water lines towards the buildings themselves (but not necessarily alway directly to the buildings.

Unfortunately there are some missing water pipe features in the „PIPES“ shapefile, which leave some service lines orphaned.

It would be nice to find a way to rebuild the water pipe features, but due to lack of information, the current strategy is the simply remove all orphaned service lines.

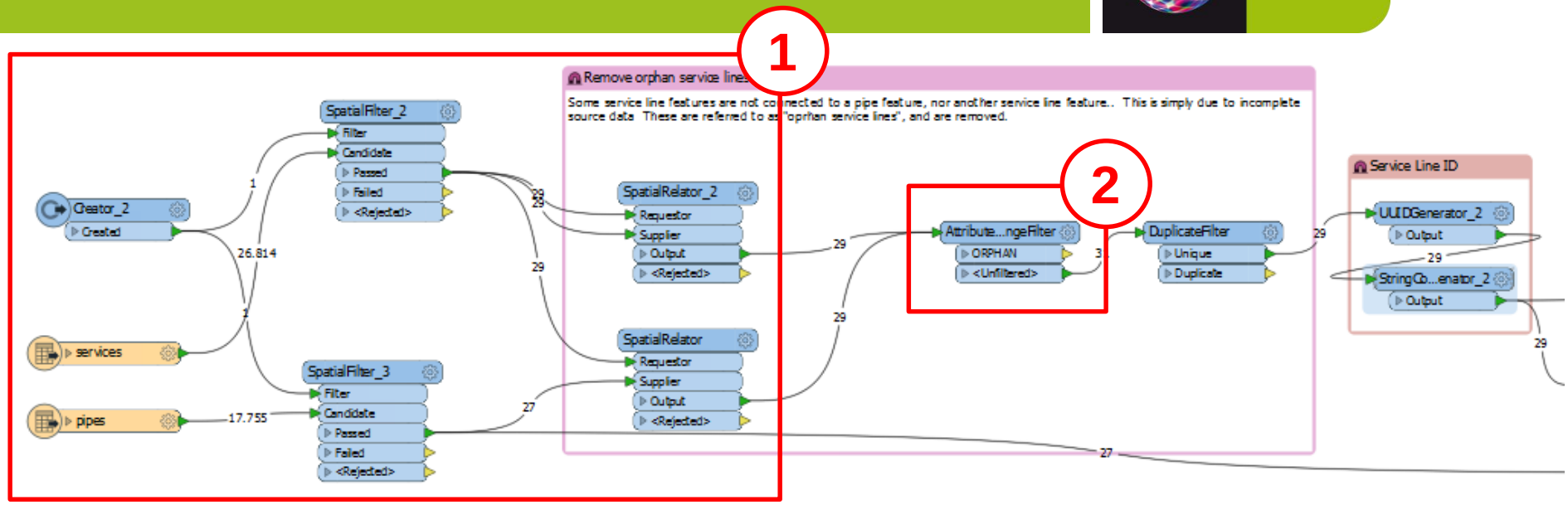


Preprocessing – Service Lines



The methodology used to achieve this goal is found in the „Remove orphan service lines“ bookmark in the sample data workbench.

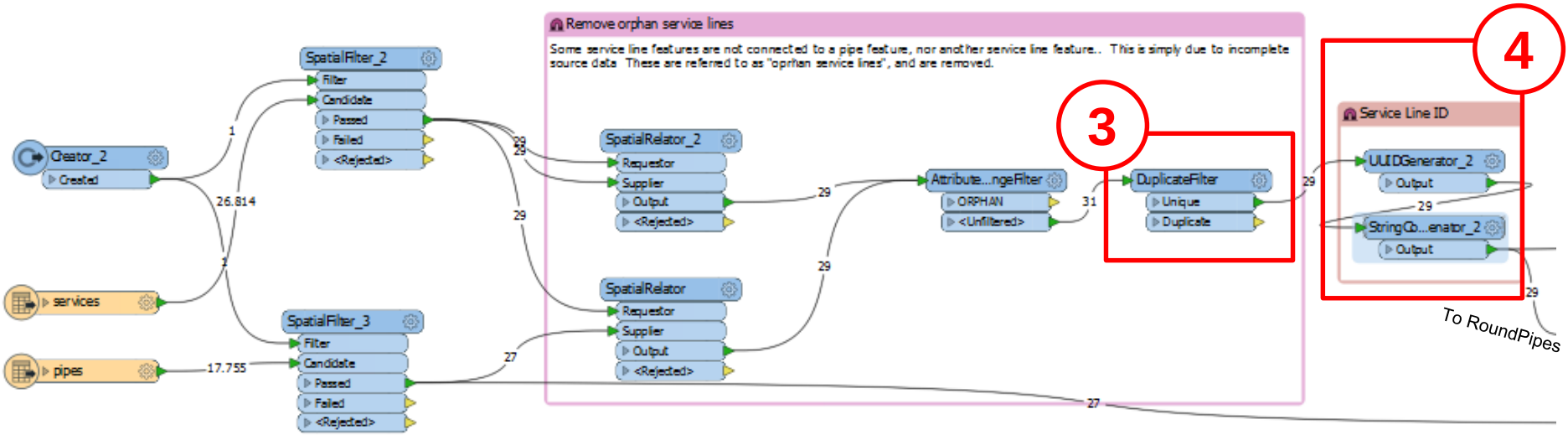
Preprocessing – Service Lines



1 First, both the service lines and the pipes passed (optionally through the SpatialFilter to transform only a subset of features) to a pair of SpatialRelators. The service lines are the requestors in both. The upper one uses service lines again as supplier and the bottom one uses pipes as suppliers, with „Requestor Touches Supplier“ as the condition in both.

2 A subsequent AttributeRangeFilter is able to count the number of spatial connections and can filter out those with none as „orphans“.

Preprocessing – Service Lines



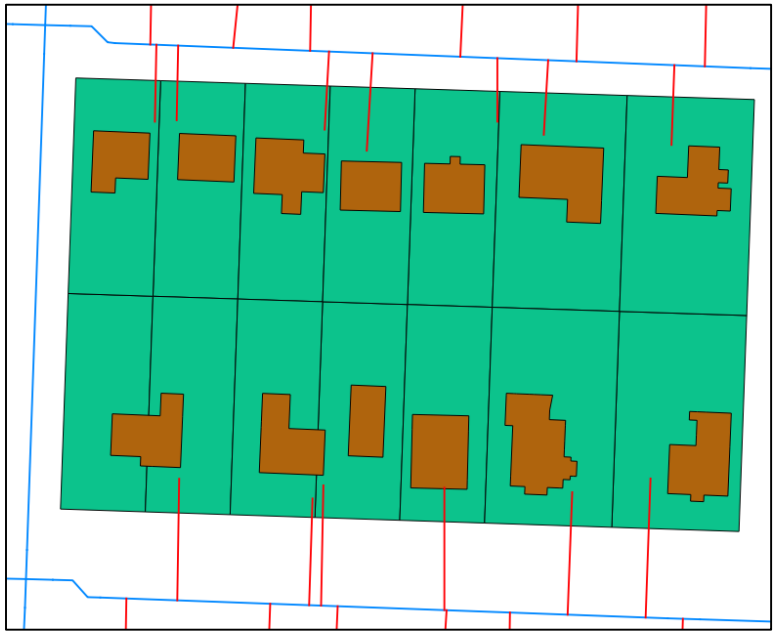
3 The features are then passed to a DuplicateFilter, which filter out all the duplicates from the pair of SpatialRelators (those which touch another service line and a pipe feature).

4 The features are then finally passed to a UUIDGenerator and StringConcatenator transformer, which gives each a unique ServiceLine_ID attribute value. The ServiceLines are also passed towards the RoundPipe production chain, since they are also physical pipes.

Preprocessing – Buildings & Parcels

To make the buildings seen in the Nanaimo Water Network data sample, as well as their connection to the RoundPipe‘s FeatureGraphs via InterFeatureLink, the building footprints must first receive some information from their respective parcels.

On the left we see the „BUILDINGS“ (brown), „PARCELS“ (teal), „PIPES“ (blue) & „SERVICES“ shapefiles.



As we can see, in most cases, the service lines do not actually reach the building footprint, so we will use the spatial relationship between the service line and the building footprint to get the necessary information.

Preprocessing – Buildings & Parcels

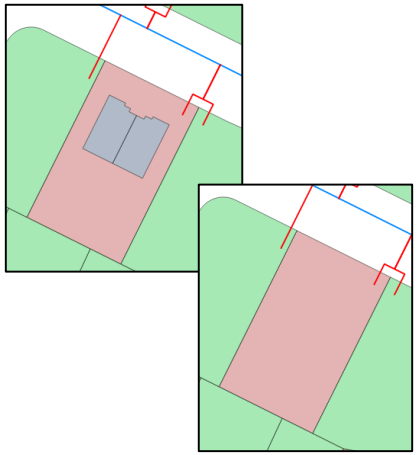
The parcels, however, need some processing. The „TYPE“ attribute gives information as to the type of parcel. Some parcel types seem to have special relationships.

For instance, in many cases, „Strata Lot“ parcels often appear fully enclosed & overlapped by „Strata“ parcels.

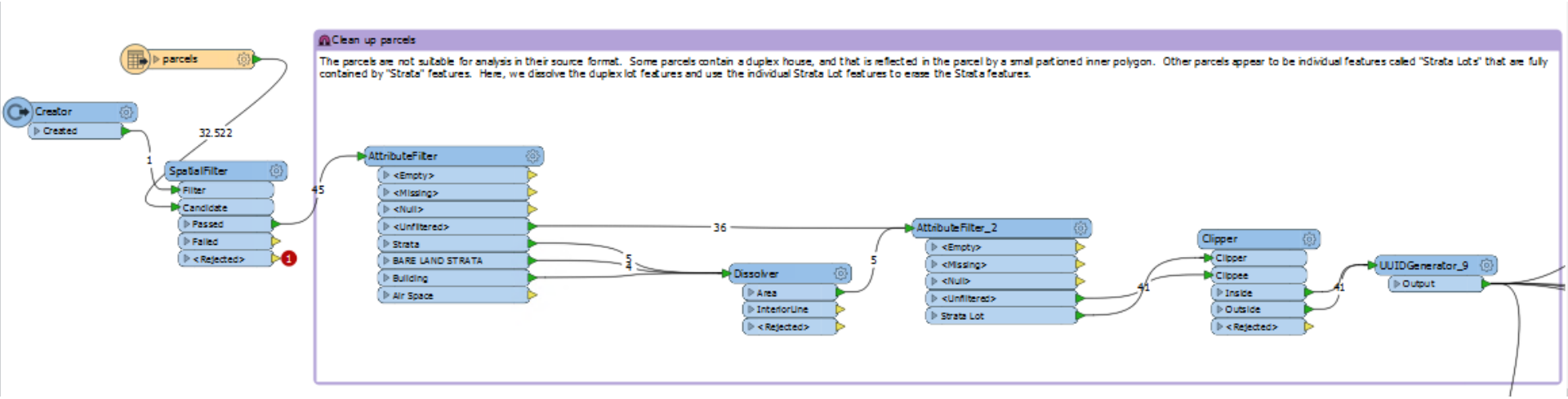
Furthermore, there are some parcels that account for a duplex building on them, despite the feature from the „BUILDINGS“ shapefile being one contiguous feature, and sharing a service line.

Left: Strata Lot features totally covered by single Strata feature.

Right: Duplex building (BARE_LAND_STRATA) totally covered by single Strata Feature, but sharing a single service line.



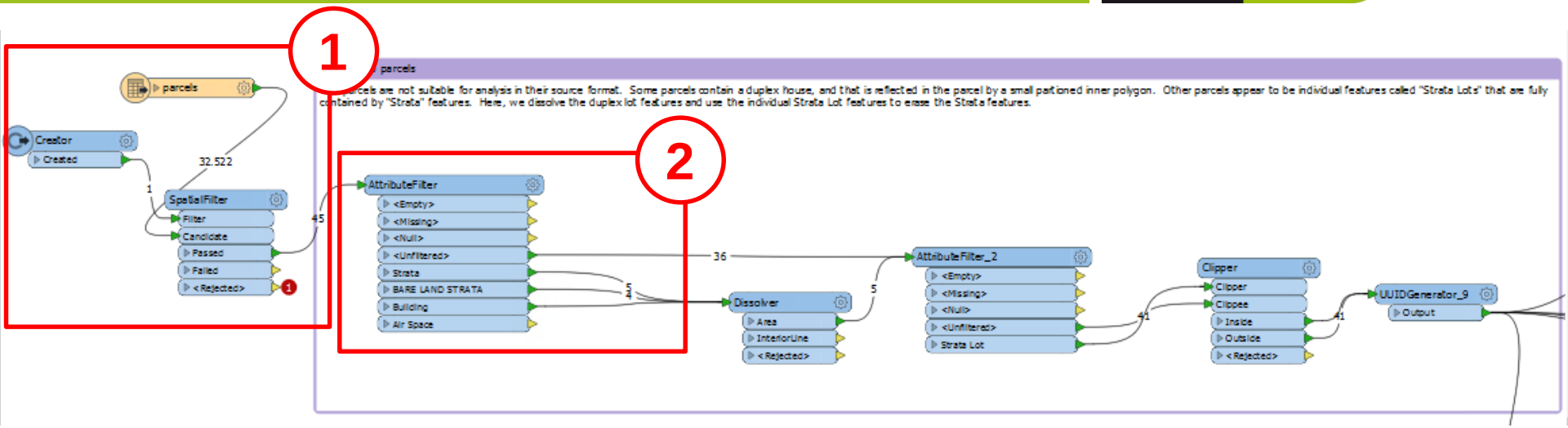
Preprocessing – Buildings & Parcels



It would be much more helpful to use if we removed all these overlaps, so that we were left only with individual parcels. A parcel can have 1..n buildings, but wherever possible, a building should only have 1 parcel.

The methodology used to achieve this goal is found in the „Clean up parcels“ bookmark in the sample data workbench.

Preprocessing – Buildings & Parcels

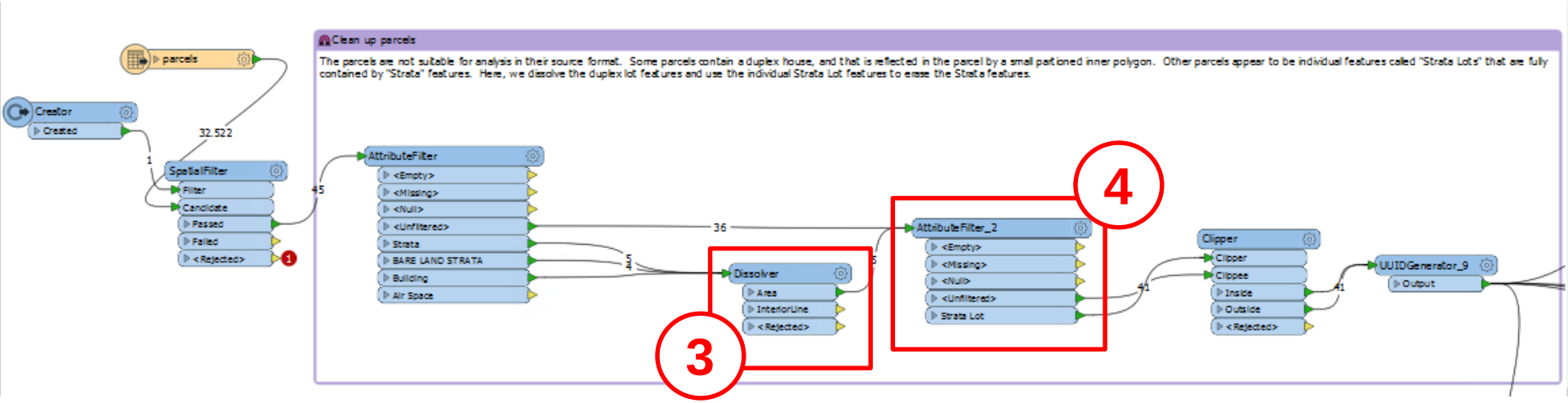


1 The parcels are passed (optionally through the SpatialFilter to transform only a subset of features) to the AttributeFilter.

2 The Attribute Filters on the „TYPE“ attribute, letting all features pass except for those with TYPE IN (‚Strata‘, ‚BARE LAND STRATA‘, ‚Building‘, ‚Air Space‘)

‚Air Space‘ is simply discarded because it is not interesting to us.

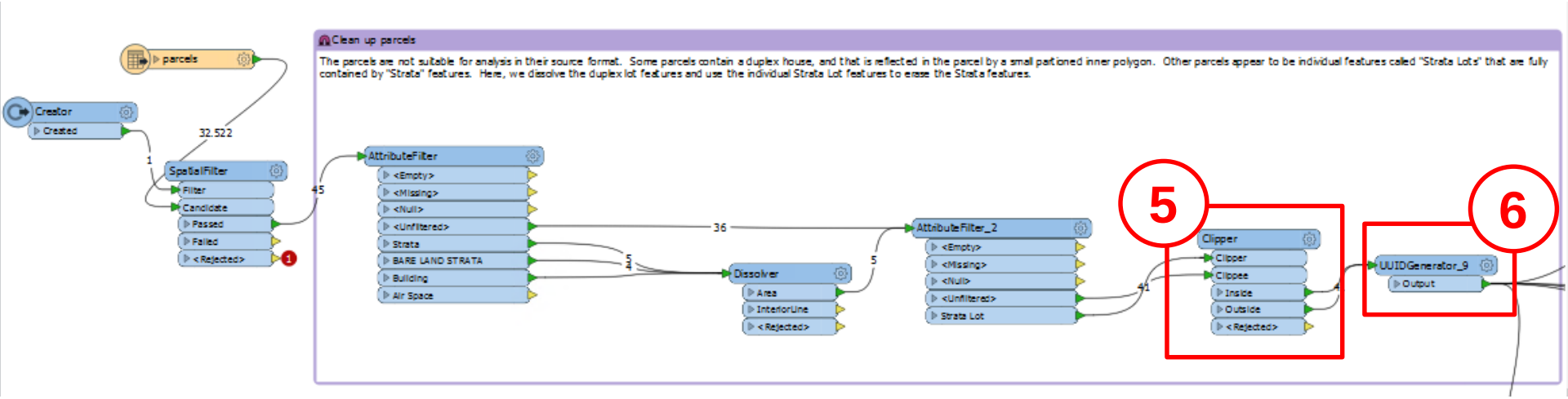
Preprocessing – Buildings & Parcels



3 The ‚Strata', ‚BARE LAND STRATA', & ‚Building' parcels are passed to the Dissolver, which dissolved based on the „PLAN" attribute. This attribute appears to be unique for the features around each individual building or building agglomeration.

4 Now, the unfiltered features from earlier and the newly-dissolved features are passed to the next AttributeFilter transformer, which sorts them into ‚Strata Lot' and Others.

Preprocessing – Buildings & Parcels



5 The ‚Strata Lot' features are then used to clip chunks out of the rest of the features. The „inside" features are therefore the individual building parcels, while the „outside" features are the areas around them. We might as well keep both, even though we are mostly interested in the „inside" areas.

6 All the resulting features are then passed to a UUIDGenerator, which generates a „Parcel_ID" attribute. This attribute is never actually written to the final CityGML file, but it is necessary to make the link between service line and building later.

Buildings & Associated TerminalElements

The „BUILDINGS“ shapefile has building footprint information. In some cases, it also has building height information, however this is quite rare. The end goal is to have LoD1 extrusions for buildings.

But before detailing the process of creating buildings, let us first consider the necessary components and properties that will need to be created and interrelated.

A Building element will need the following information:

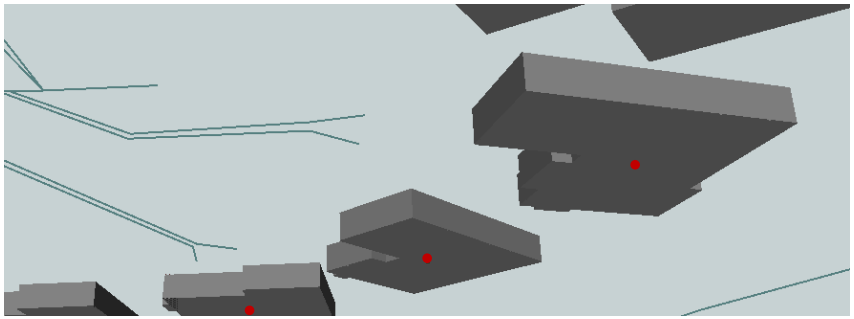
- Height
- Elevation (since the data sample uses a local DEM for height)
- An associated TerminalElement
  - This TerminalElement must also have an associated Featuregraph (a single Node), which itself must be connected to the pipe network.

Buildings & Associated TerminalElements

We can therefore envision the entire Building Feature Assemblage in two ways:

Geometrically:

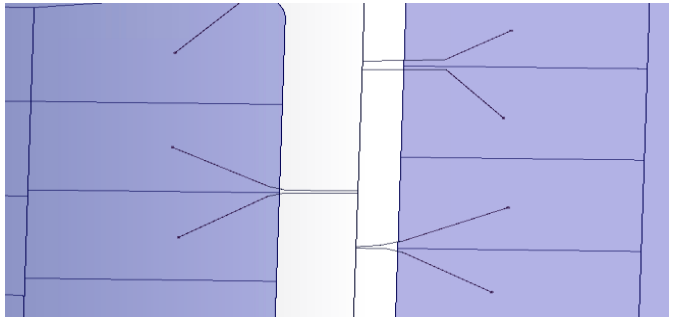
An LoD1 exstrusion, risen to the elevation of the DEM with a TerminalElement point geometry at the center of its base.



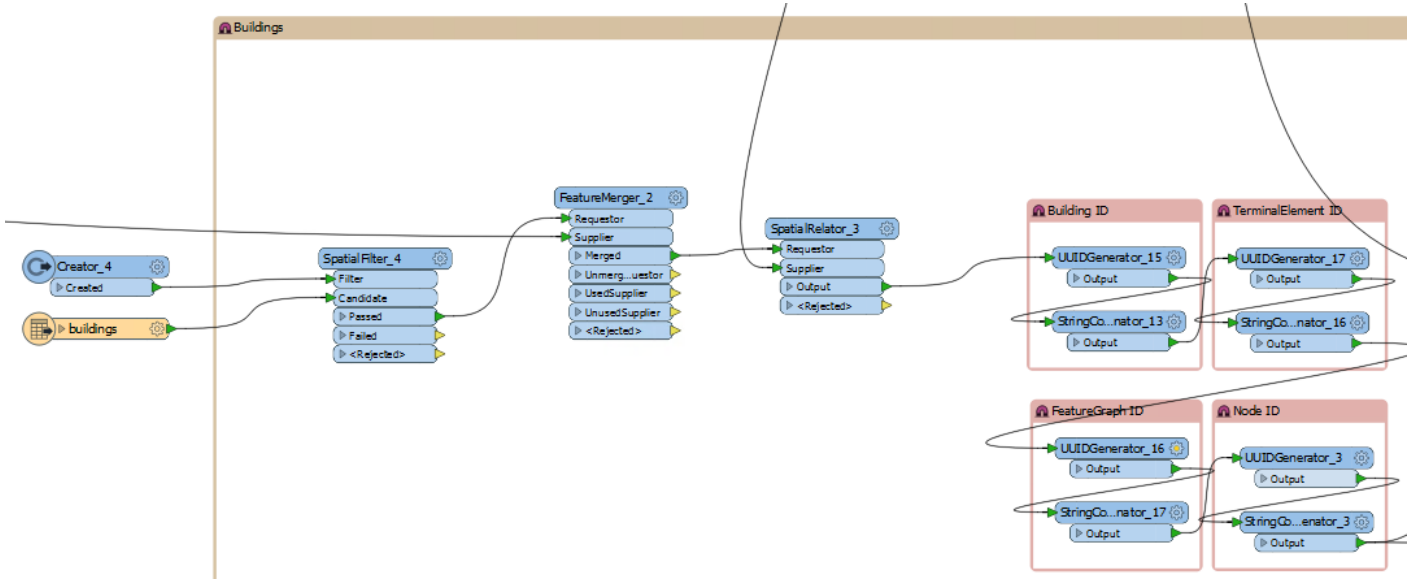
(This image is looking up at the buildings from underneath, simply for the sake of showing the position of the TerminalElements)

Topologically:

A FeatureGraph consisting of a single Node. It is connected via an InterFeatureLink to the FeatureGraph of the service line RoundPipe terminates on the Building‘s parcel.

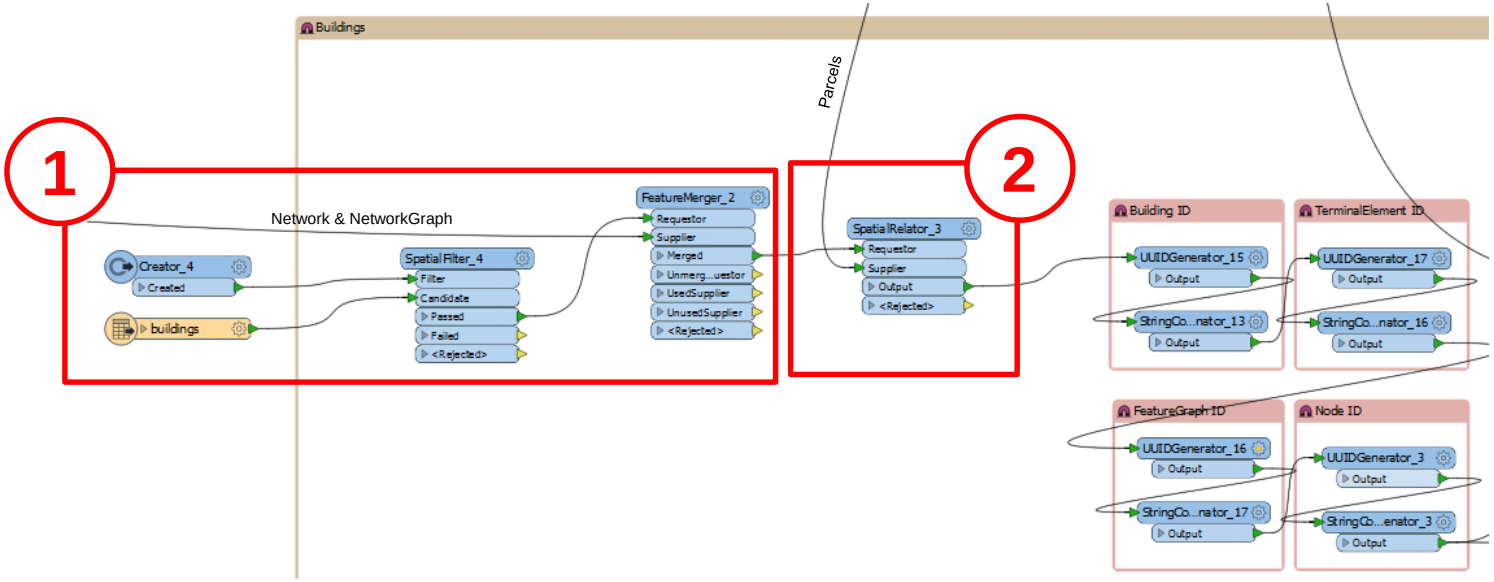


Buildings



The methodology used to create the buildings can be found in the „Buildings“ bookmark in the workbench. It will be presented in several parts since it is very large.

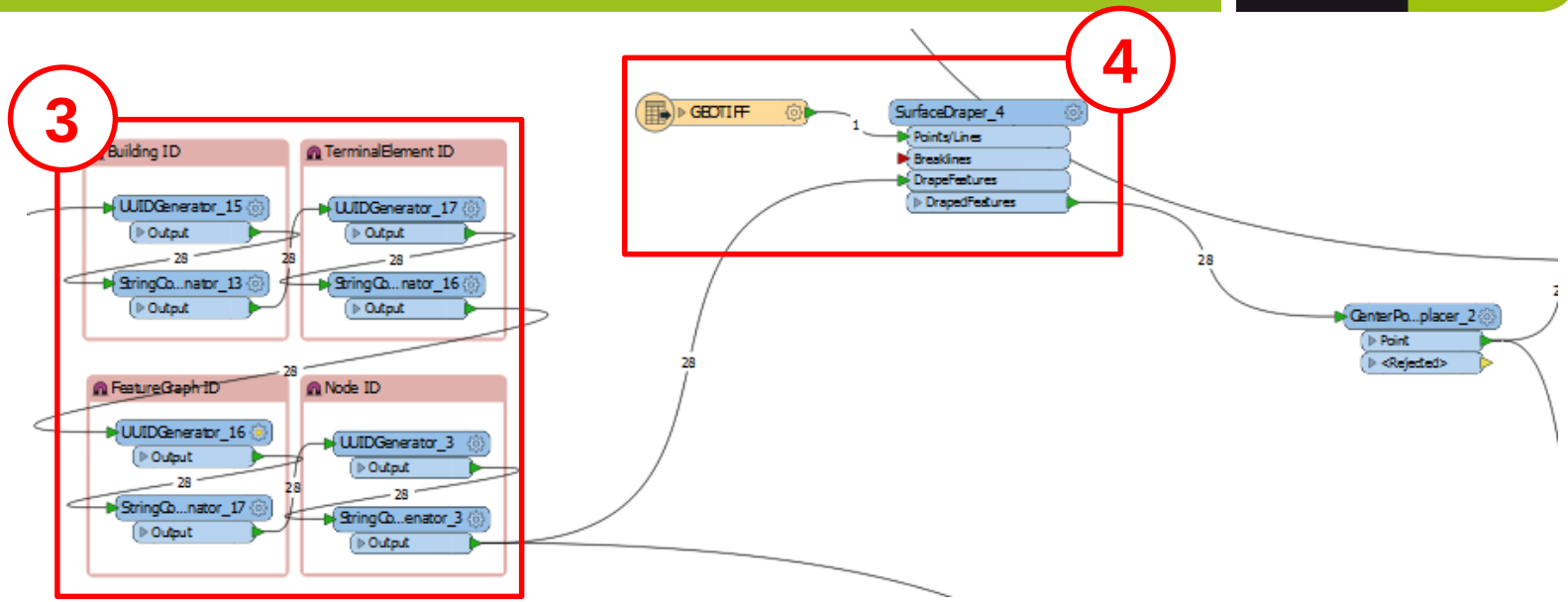
Buildings



1 The buildings are passed (optionally through the SpatialFilter to transform only a subset of features) to a FeatureMerger, which merges them 1:1 with the NetworkID and NetworkGraphID (we will need these for determining ownership of TerminalElements, FeatureGraphs, etc.

2 The buildings are then passed to a SpatialRelator, which relates them to the output of the Parcels bookmark using the "Requestor Intersects Supplier„ spatial predicate.

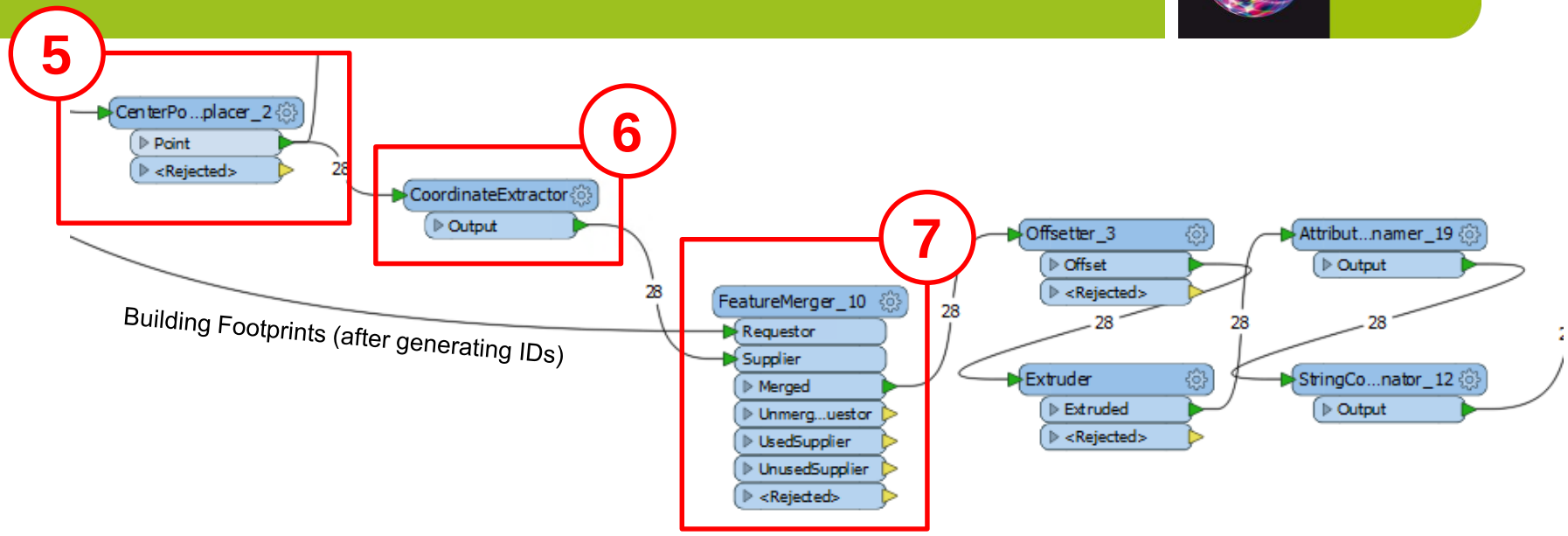
Buildings



3 The buildings are then passed to a series of UUIDGenerator + StringConcatenator combos to generate all the IDs we will need for the remainer of their transformation (Building, TerminalElement, FeatureGraph & Node)

4 These building footprints are then passed to a SurfaceDraper which drapes them onto the DEM, giving them their elevation.

Buildings

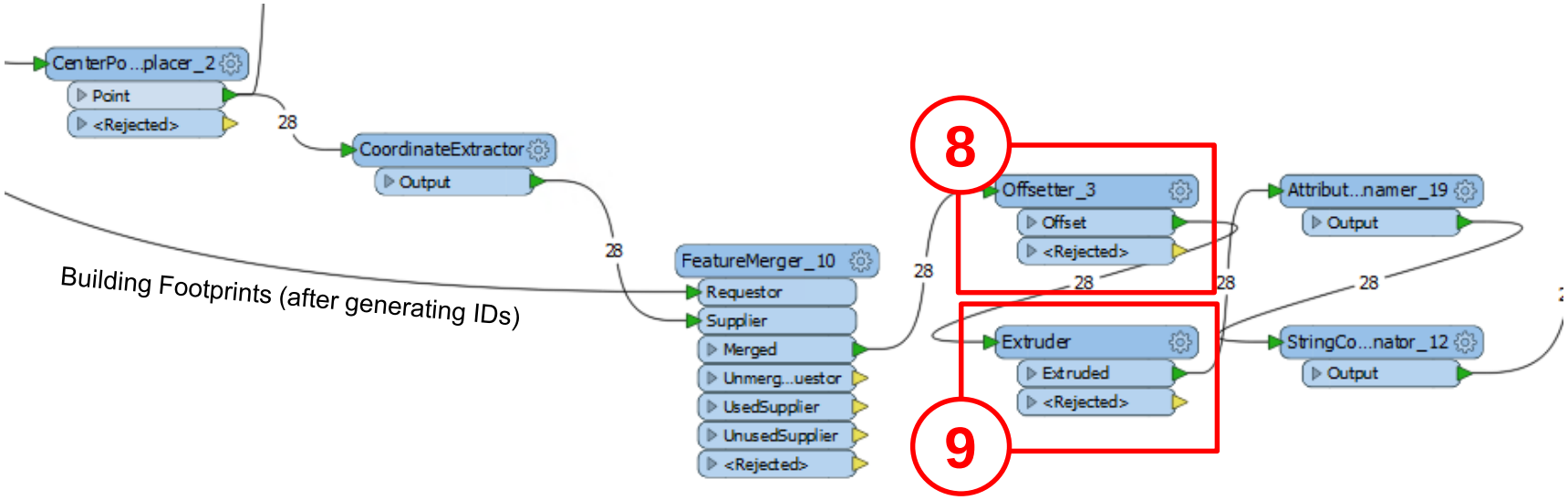


5 The draped buildings are then passed to a CenterPointReplacer, which creates a point at the 3D centre of the draped footprint.

6 These points are then passed to a CoordinateExtractor, which stores each point‘s XYZ coordinates as attributes.

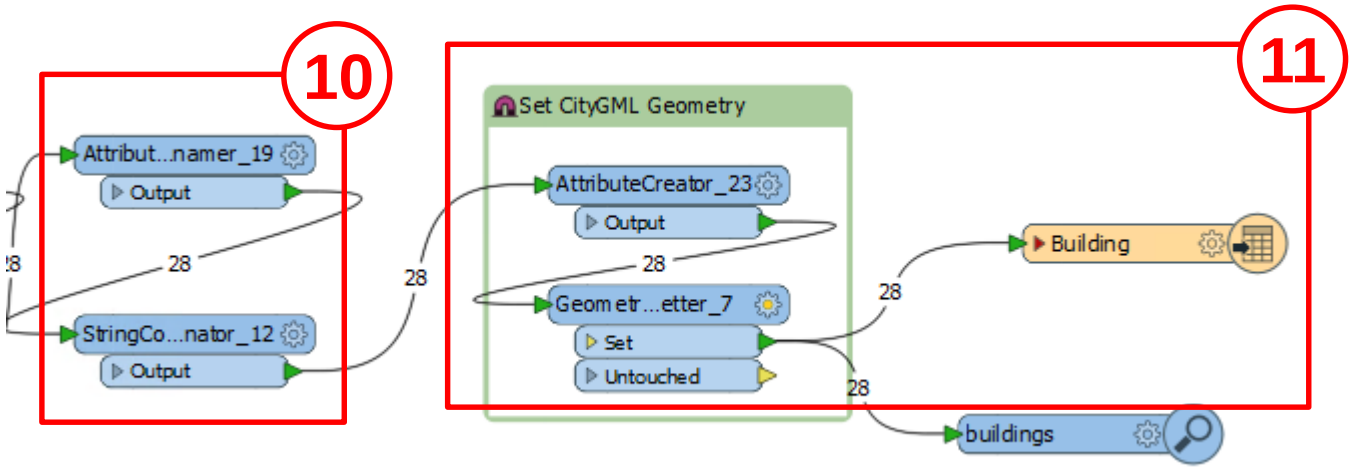
7 They are then passed to a FeatureMerger (BuildingID = BuildingID), which gives the point‘s XYZ coordinates to each original Building footprint.

Buildings



8 The footprints (now with XYZ point coordinates) are then passed to an Offsetter, which raises the buildings to the Z-coordinate of the point. The reason this is done is because if we had just raised the draped footprints directly, it would have warped the polygons to fit the terrain exactly, resulting in rounded, unnatural shapes for the buildings.

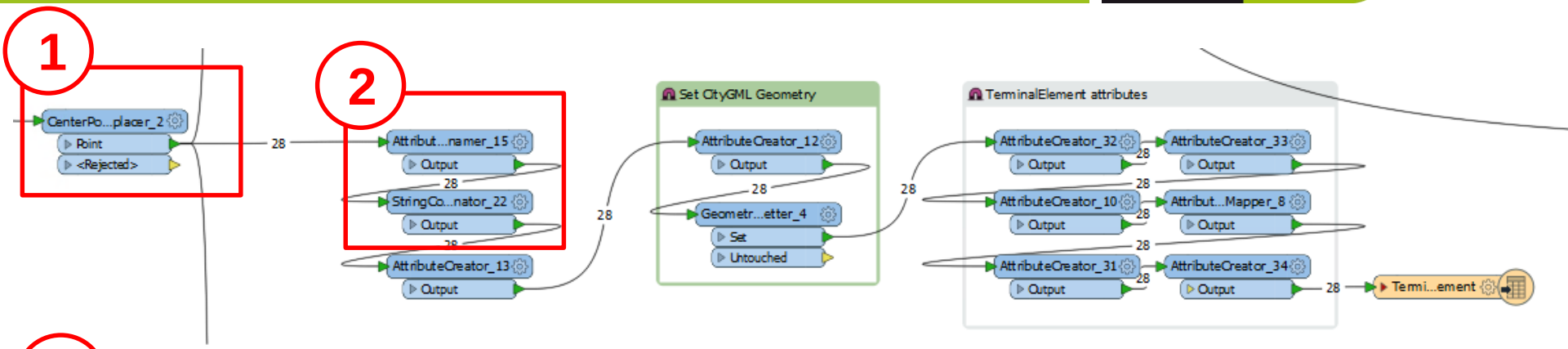
9 The raised footprints are then passed to an Extruder, which extrudes the them to the number of floors * 3m. Footprints without floor information are assumed to be 2 floors tall.



10 The extruded footprints are then passed to an AttributeRenamer & StringConcatenator, which set the Building ID to be the gml_id, and create a name for the Building that is a concatenation of its street address & postal code, respectively.

11 The CityGML geometry is then set by creating an attribute called citygml_lod_name and setting it to lod1Solid, then setting this attribute using the GeometryPropertySetter. The buldings are then written to the Building Writer.

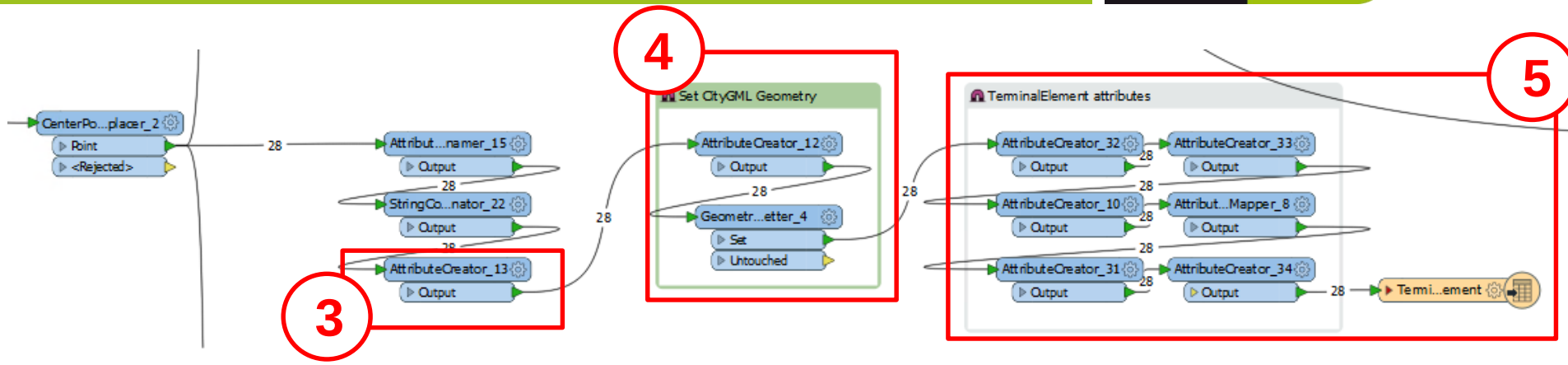
TerminalElements



1 To show how the TerminalElements are made, we need to travel back along the transformer chain. We go back to the CenterPointReplacer where we made the 3D points out of the draped building footprints. Since these points were derived from the building footprints, we have already assigned them all necessary Ids, including the TerminalElement ID (see Buildings step 3).

2 The points are then passed to an AttributeRenamer, which renamed all the Ids to what they need to be from the TerminalElement‘s perspective (TerminalElement_ID = gml_id, Network_ID = gml_parent_id, Building_ID = utility_connected_city_object, FeatureGraph_ID = utility_topo_graph_xlink). The StringConcatenor right afterwards appends a # symbol to the utility_topo_graph_xlink attribute.

TerminalElements

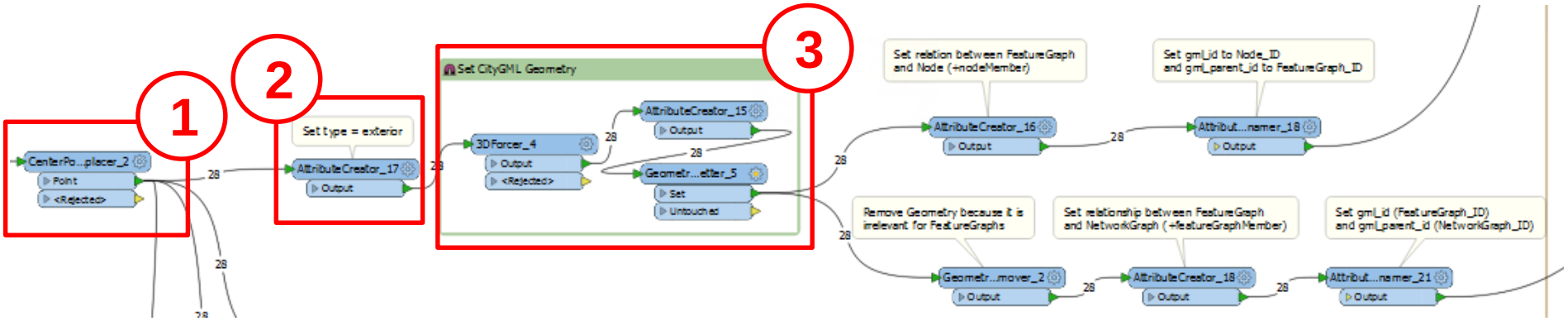


3 An AttributeCreator is used to set the points‘ „citygml_feature_role“ attribute to „component.“

4 The points then have their CityGML geometry set by creating an attribute called „citygml_lod_name“ and setting it to „lod1Geometry“. This attribute is then set as the CityGML geometry attribute.

5 The points then have various semantic attributes set, according to attributes from the source data, and are written to the TerminalElement Writer.

TerminalElement FeatureGraphs & Nodes

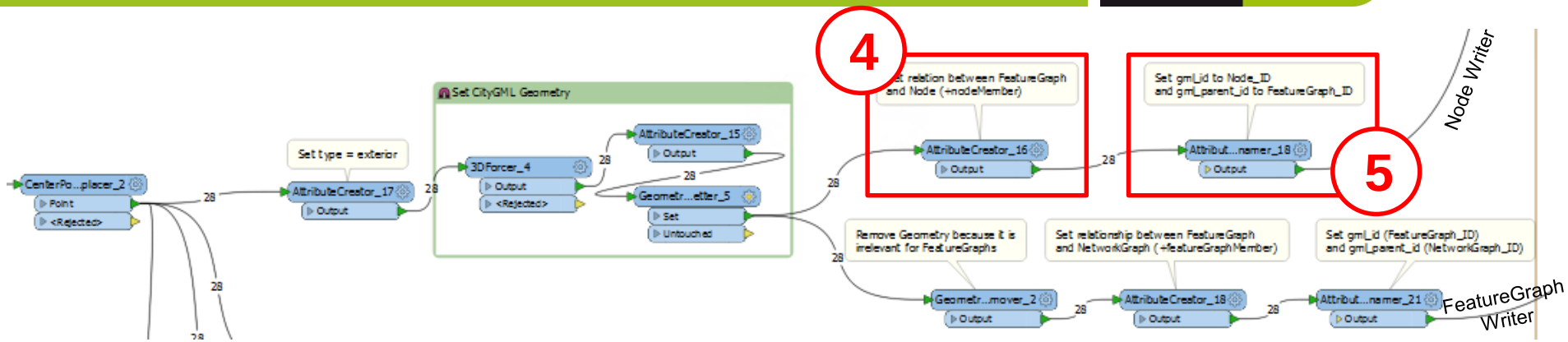


1 To show the process of creating the TerminalElements‘ Featuregraphs, we must once again return to the CenterPointReplacer (see Buildings step 3).

2 Since the TerminalElement‘s FeatureGraph will consist only of a single Node, that Node must be of “exterior“ type. We set this with an AttributeCreator.

3 The points then have their CityGML geometry set by creating an attribute called „citygml_lod_name“ and setting it to „realization“. This attribute is then set as the CityGML geometry attribute. A 3DForcer is used to force the point back to 2D (z=0), since FeatureGraphs are always 2D.

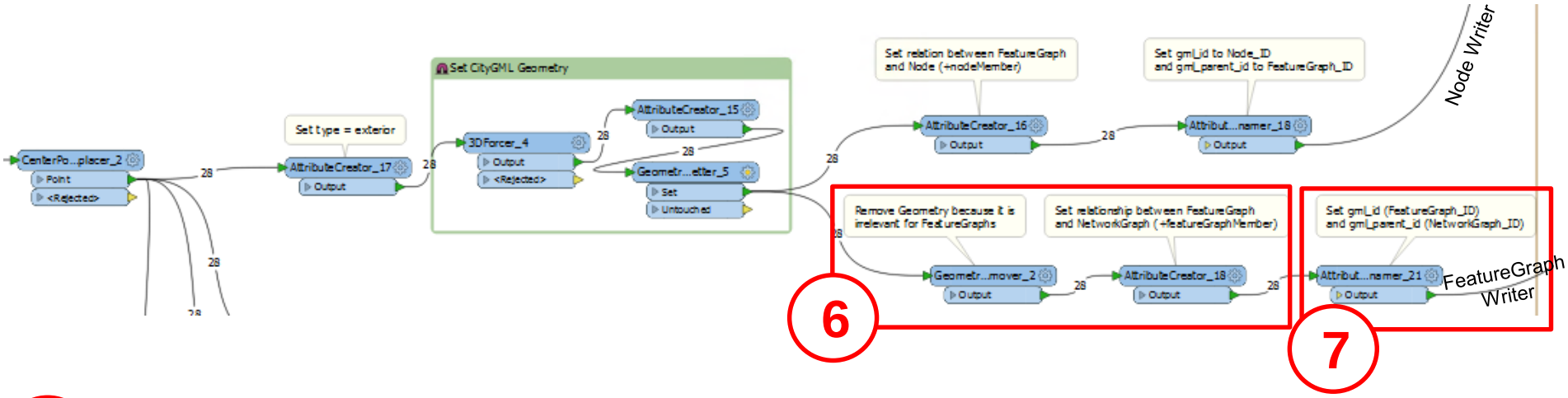
TerminalElement FeatureGraphs & Nodes



4 The points are then passed to an AttributeCreator which creates a „citygml_feature_role“ attribute to „nodeMember“, indicating that it is a Node that belongs to a FeatureGraph.

5 The points are then passed to an AttributeRenamer which sets the Node ID as the „gml_id“ attribute and the FeatureGraph ID to the „gml_parent_id“ attribute. Thez are then written to the Node Writer..

TerminalElement FeatureGraphs & Nodes

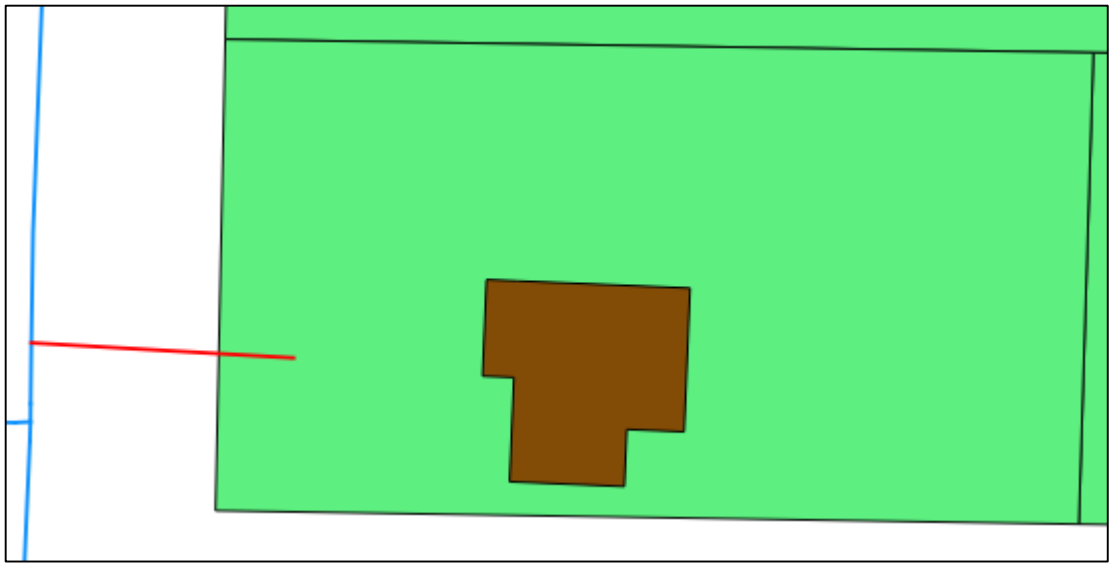


6 Going back to when the CityGML geometry was set, the points take a different branch to a GeometryRemover, because they are goin to the FeatureGraph transformer and FeatureGraphs have no inherent Geometry. They also have a „featureGraphMember“ relationship set with the NetworkGraph ID.

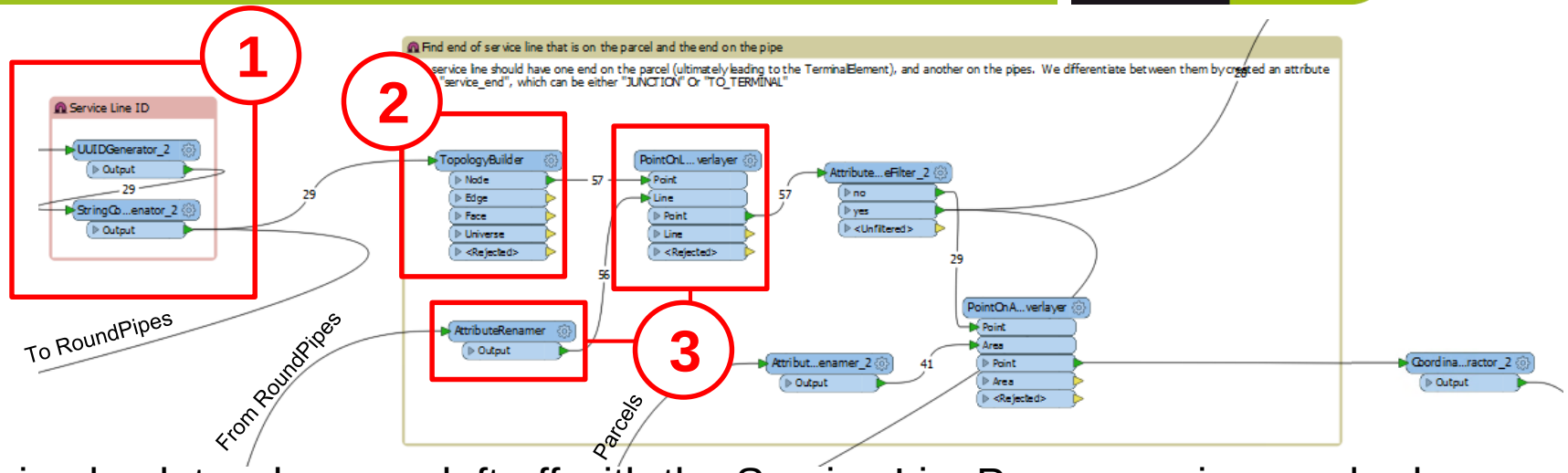
7 The „FeatureGraph_ID“ atribute is then renamed to „gml_id“ and „NetworkGraph_ID“ is renamed to „gml_parent_id“. The features are then sent to the FeatureGraph Writer.

Now that we have created the Buildings, along with their TerminalElements and the FeatureGraphs thereof, it is time to make the topological connection between the service line pipes and the TerminalElement‘s FeatureGraph‘s Nodes.

As a reminder, the only way we know how to make this relationship is by bridging the gap from the service line pipe to the building via the parcel:



InterFeatureLinks

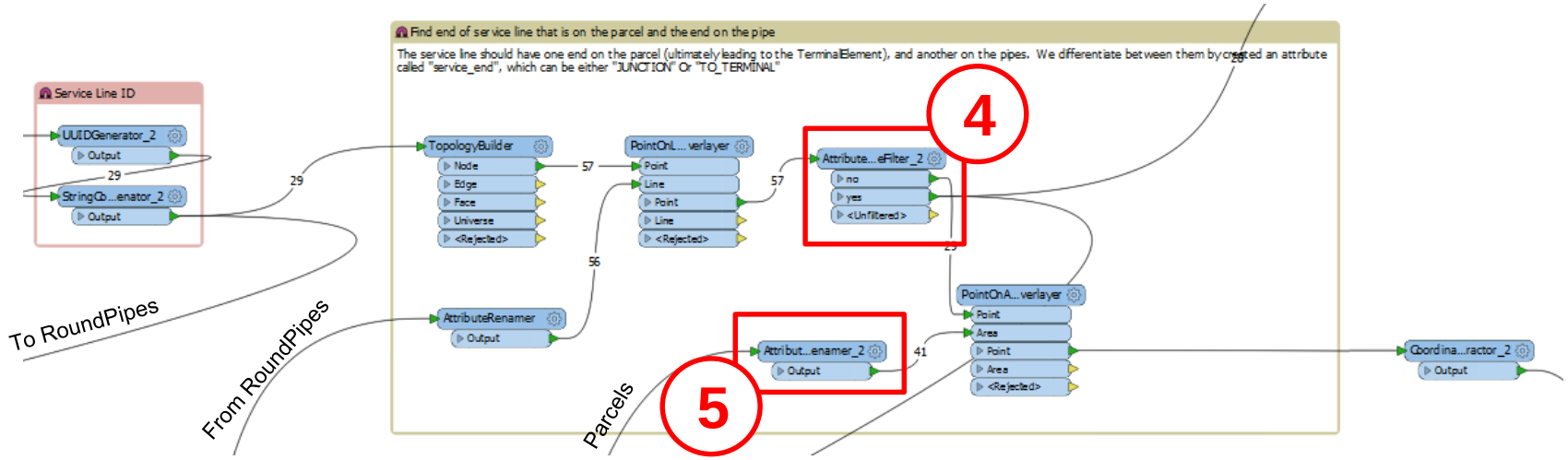


1 Going back to where we left off with the Service LinePreprocessing, we had just assigned each (non-orphaned) Service Line a Service Line ID. The ServiceLines were also passed to the RoundPipe production chain. (See Service Line Preprocessing Step 3)

2 The Service Lines are then passed to a TopologyBuilder, which breaks them into their node points.

3 The node points are then passed to a PointOnAreaOverlayer transformer, which overlays them with the RoundPipe Features, which have had their „RoundPipe_ID“ attribute renamed to „Parent_RoundPipe“.

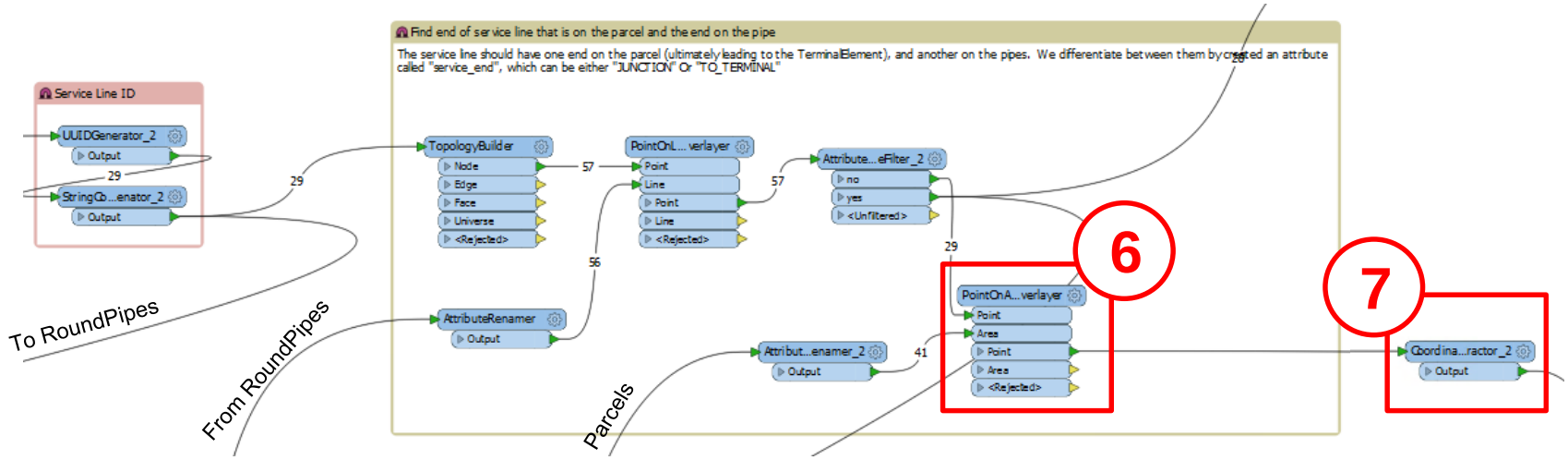
InterFeatureLinks



4 The Service Lines are then passed to an AttributeRangeFilter. By counting the number of connections determined by the PointOnAreaOverLayer attribute, we can separate the nodes that are leading towards the building and those that are touching the main pipes.

5 Meanwhile, the previously Parcels are passed to an AttributeRenamer that changes the „Parcel_ID“ attribute to „Parent_Parcel_ID“.

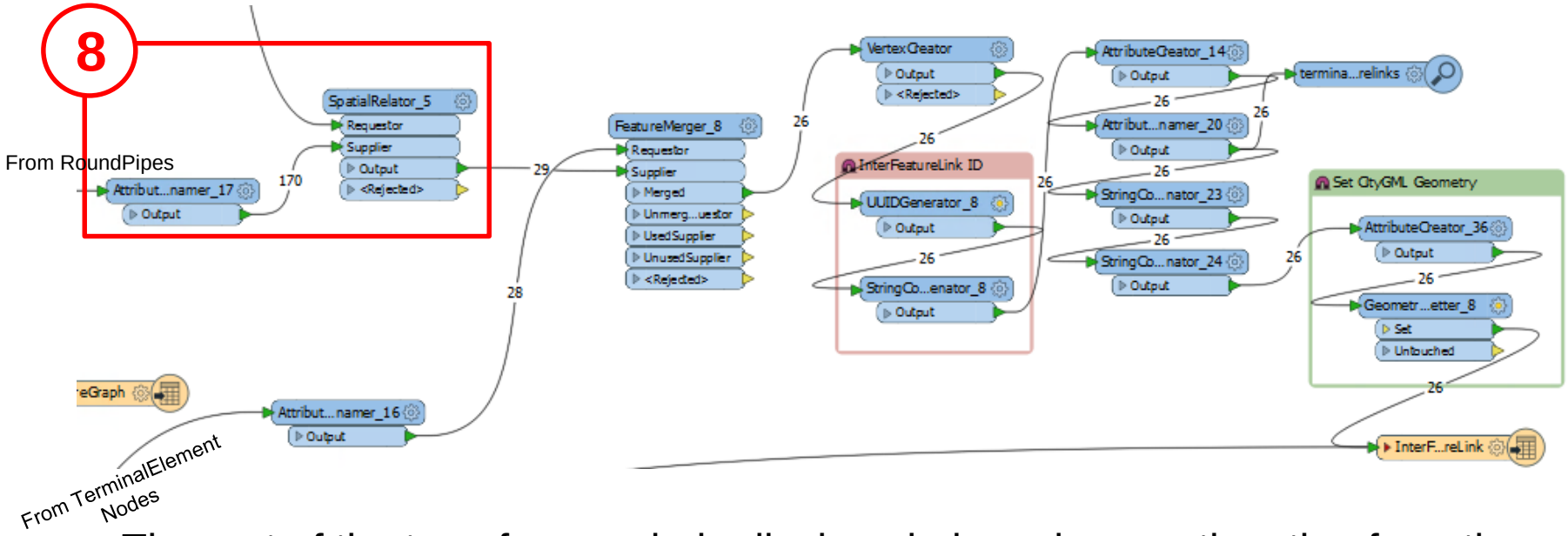
InterFeatureLinks



6 The points are then passed to a PointOnAreaOverlayer, which gives this „terminal service line node“ the „Parent_Parcel_ID“ attribute.

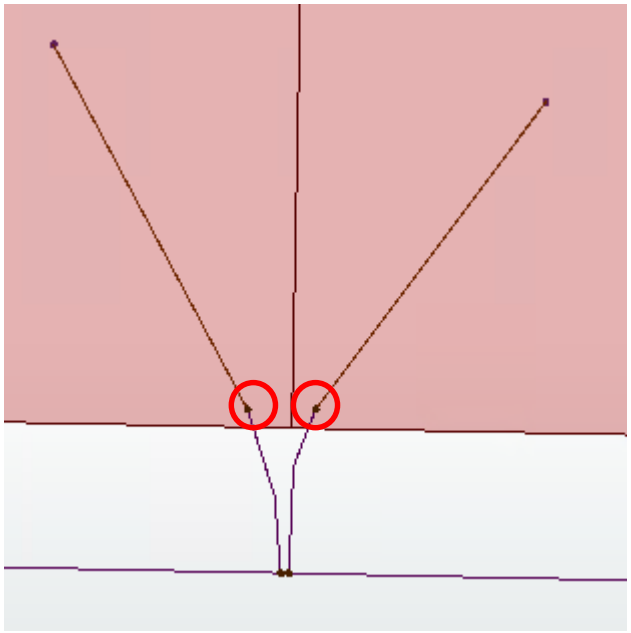
7 The coordinates of this point are extracted and stored in an attribute called „pipe_x“ and „pipe_y“.

InterFeatureLinks



The part of the transformer chain displayed above is a continuation from the last image, it is just found in a different location in the workbench, on the center-right edge.

8 The points are passed to a SpatialRelator, white checks for an intersection between the point and the nodes coming from the RoundPipe production chain. This is how we know which Node from the RoundPipes' Featuregraphs we have to connect to.



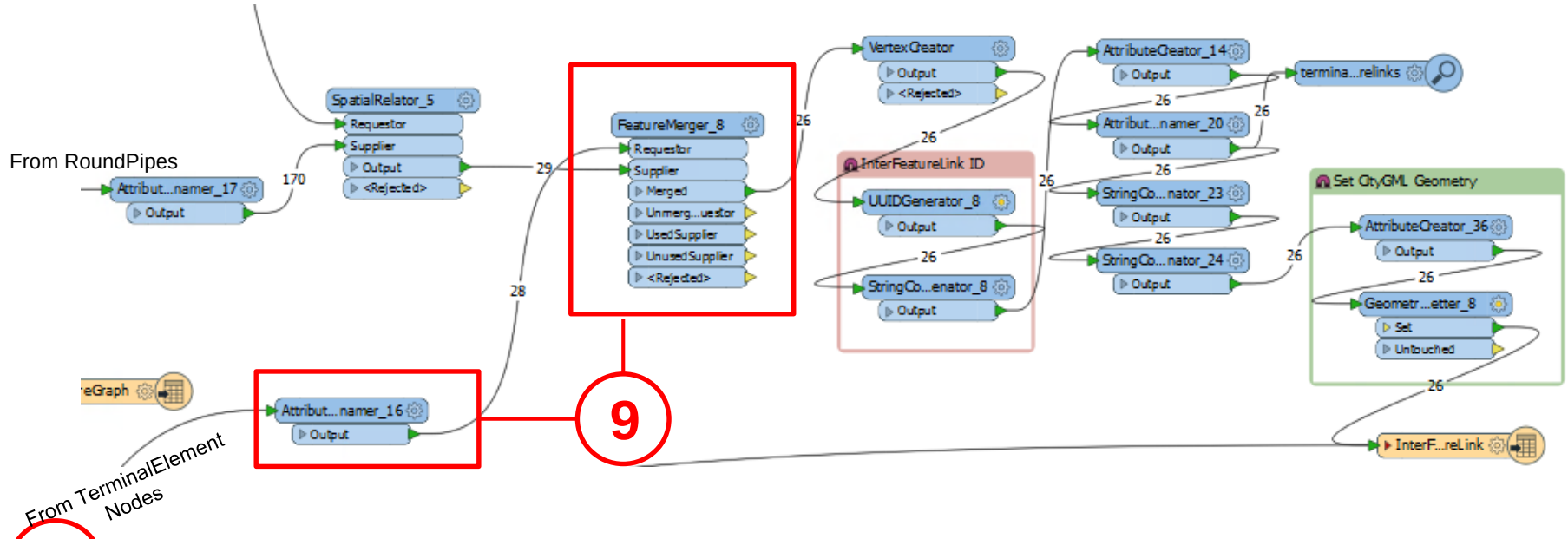
When we pre-processed the Service Lines, we forwarded them to the main pipe RoundPipe production chain, so that they would be included in the network as physical pipes.

In this chain, they had FeatureGraphs created, with associated Nodes. These nodes also represent the point at which our InterFeatureLinks to the TerminalElements originate from the main pipe Network.

They are shown here, circled in red.

We need the NodeID of these nodes, so that our InterFeatureLinks can reference them as their Start Node ID.

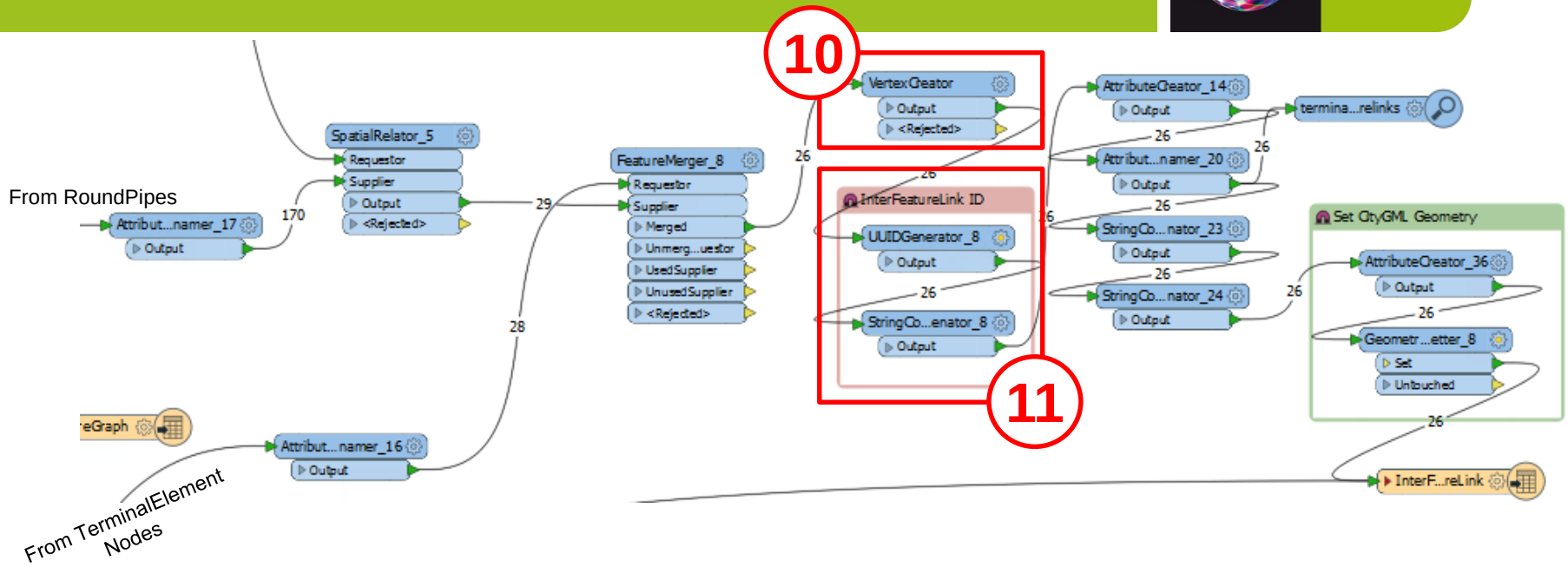
InterFeatureLinks



9 The points are then passed to a FeatureMerger, whichgives their attributes to the TerminalElements of the Buildings, using their „Parent_ParcelID“ attribute and the „Parcel_ID“ attribute of the TerminalElement Nodes. This allows for every TerminalElement on a given Parcel to be connected to the appropriate service line, so that parcels with multiple buildings are all connected to the same service line.

Note that the TerminalElement Node points are actually the requestor in this case, the attributes of the service line endpoint are being added to them.

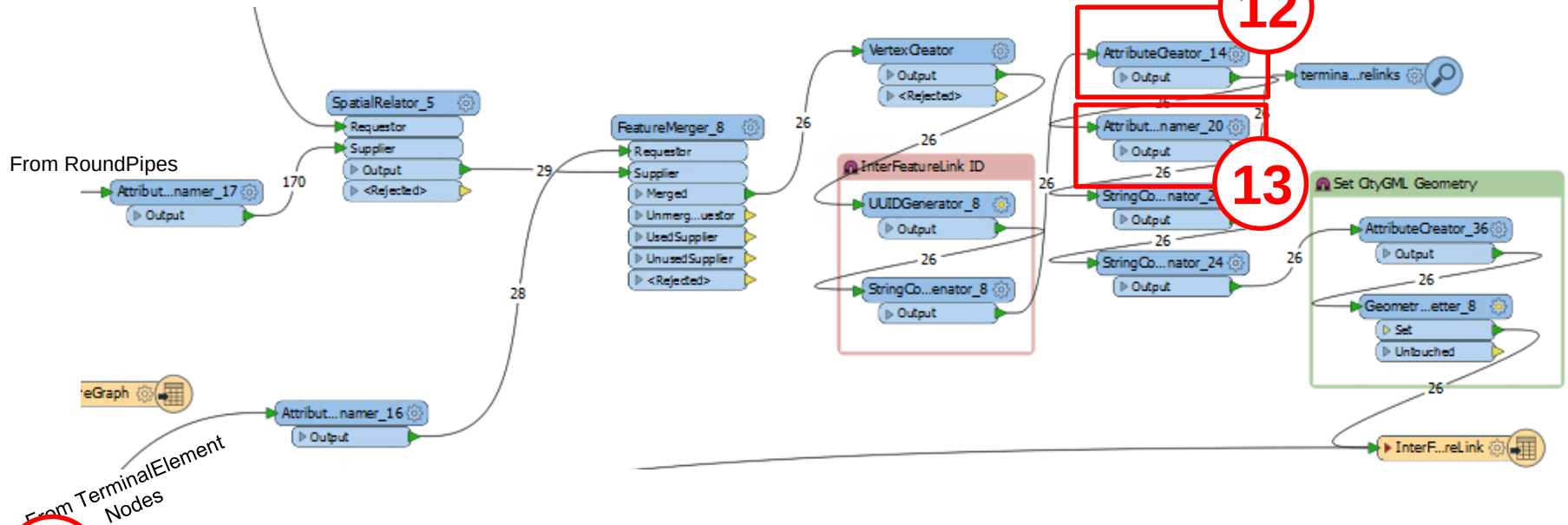
InterFeatureLinks



10 The points are then passed to a VertexCreator, which draws lines starting at each TerminalElement node point to its respectve service line node point.

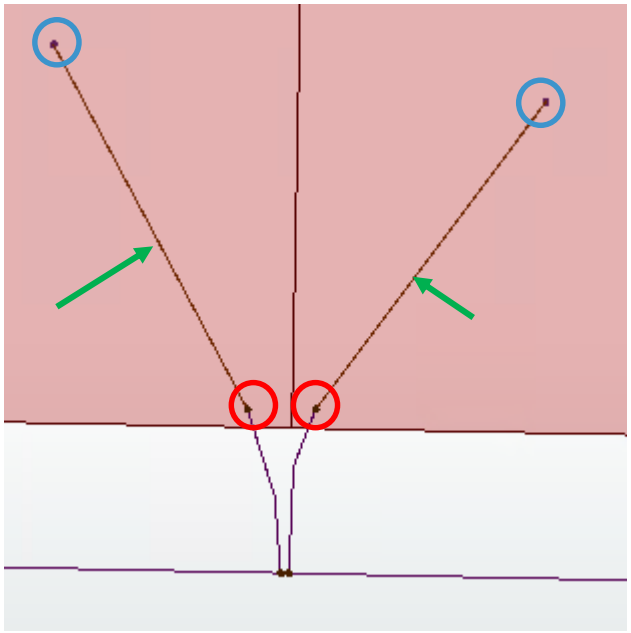
11 The newly-created lines are then assigned a UUID and prefixed with „TerminalElementID_“.

InterFeatureLinks



12 The lines then have two new attributes created, „citygml_fetaure_role“, which is set to „linkMember“. This defines the feature as being a member of the NetworkGraph. „utility_type“, is set to „connects“, which indicates that it is a connection between two nodes of different FeatureGraphs.

13 Attributes are then renamed to establish the correct final attribution for the features. „Service_Node_ID“ becomes „utility_start_xlink_href“, „Terminal_Node_ID“ becomes „utility_end_xlink_href“, „InterFeatureLink_ID“ bcomes „gml_id“ and „NetworkGraph_ID“ becomes „gml_parent_id“



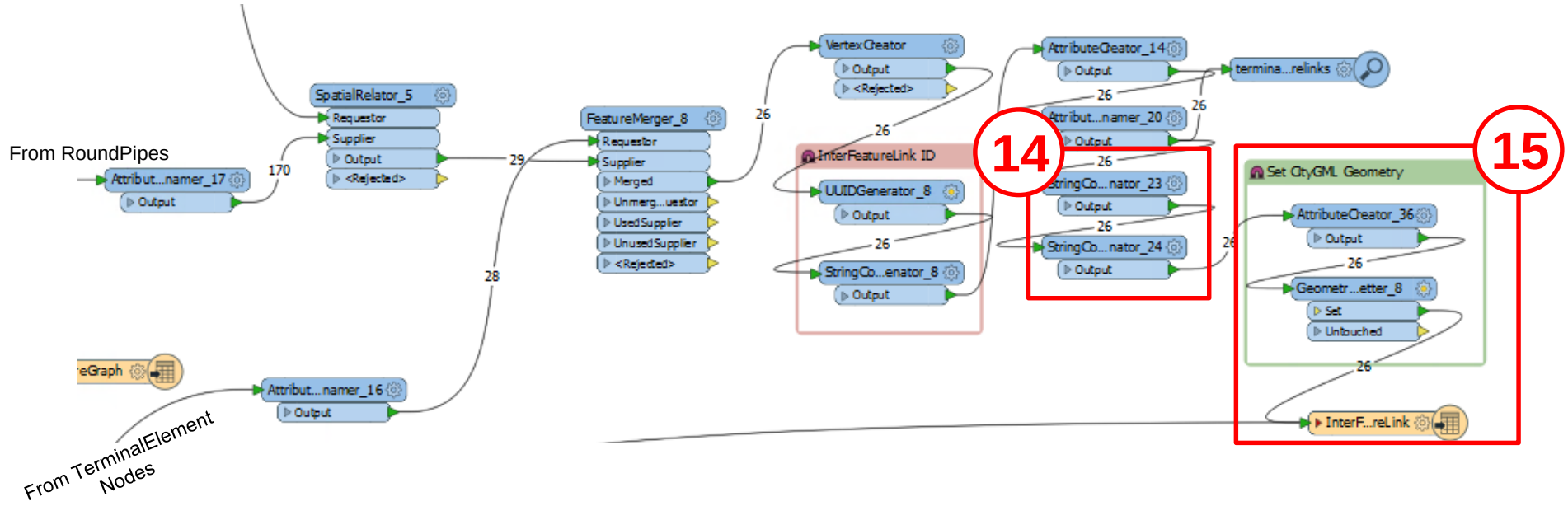
To describe visually where we are:

Red circled points = utility_start_xlink_href (formerly Service_Node_ID)

Blue circled points = utility_end_xlink_href (formerly Terminal_Node_ID)

Green arrows = InterFeatureLink, created from VertexCreator transformer

InterFeatureLinks



14 Two StringConcatenators are used to append a „#“ symbol to the „utility_start_xlink_href“ and „utility_end_xlink_href“ attributes, as it is a CityGML convention to refer to xlinked IDs in the same document in this way.

15 The geometry is then specified by creating an attributed called „citygml_lod_name“ and setting it to „realization“, then setting the Geometry with a GeometrySetter on the same attribute.